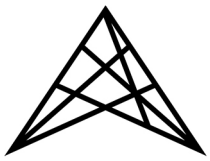


AIMMS

Function Reference

AIMMS

August, 2025

CONTENTS

1 Elementary Computational Operations 1

2 Algorithmic Capabilities 263

3 Model Handling 573

4 Data Management 621

5 User Interface Related Functions 687

6 Development Support 719

7 System Interaction 727

8 Predefined Identifiers 779

9 Suffices 863

10 Deprecated 913

Index 965

CHAPTER

ONE

ELEMENTARY COMPUTATIONAL OPERATIONS

1.1 Arithmetic Functions

AIMMS supports the following arithmetic functions:

1.1.1 Abs

```
Abs(
    x              ! (input) numerical expression
    )
```

Arguments

x A scalar numerical expression.

Return Value

The function *Abs* (page 1) returns the absolute value of *x*.

Example

```
_p_returnM1 := Abs(-1); ! returns 1. Abs of negative value is the negated value.
_p_returnZ  := Abs( 0); ! returns 0. Abs of 0.0 is 0.0
_p_returnP1 := Abs( 1); ! returns 1. Abs of positive value is the same value.
```

Note: The function *Abs* (page 1) can be used in constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems if the argument assumes values around 0.

See also:

- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.2 ArcCos

```
ArcCos(
      x              ! (input) numerical expression
      )
```

Arguments

x A scalar numerical expression in the range $[-1, 1]$.

Return Value

The *ArcCos* (page 1) function returns the arccosine of x in the range 0 to π radians.

Graph

Example

```
_p_returnM1 := arcCos(-1); ! returns pi
_p_returnZ  := arcCos( 0); ! returns half pi
_p_returnP1 := arcCos( 1); ! returns 0
```

Note:

- A run-time error results if x is outside the range $[-1, 1]$.
- The function *ArcCos* (page 1) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *ArcSin* (page 4), *ArcTan* (page 7), *Cos* (page 11).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Inverse Trigonometric Functions (Wikipedia).

1.1.3 ArcCosh

```
ArcCosh(
       x             ! (input) numerical expression
       )
```

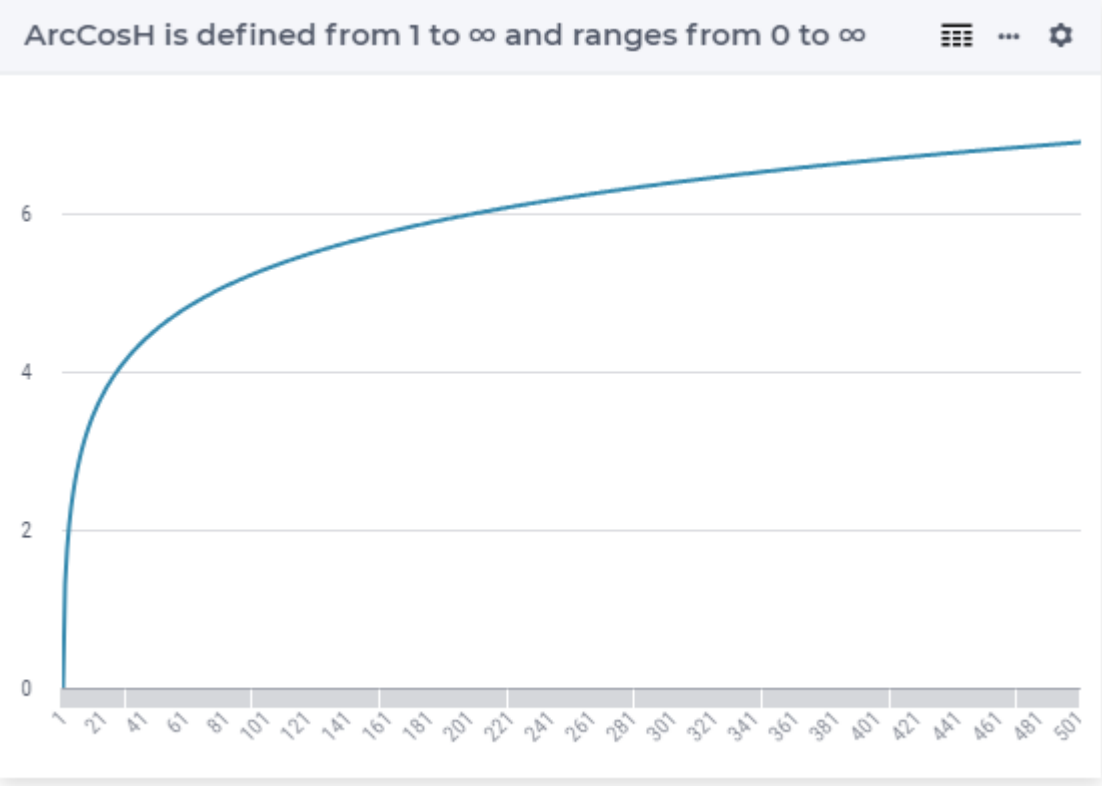
Arguments

x A scalar numerical expression in the range $[1, \infty)$.

Return Value

The *ArcCosh* (page 3) function returns the inverse hyperbolic cosine of x in the range from 0 to ∞.

Graph



Example

```
_p_return1   := arcCosH(1);   ! returns 0
_p_return1e3 := arcCosH(1e3); ! returns 7.60
_p_return1e6 := arcCosH(1e6); ! returns 14.51
_p_return1e9 := arcCosH(1e9); ! returns 21.42
_p_return0   := arcCosH(0);   ! Runtime error
```

Note:

- A run-time error results if x is outside the range $[1, \infty]$.
- The function *ArcCosh* (page 3) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *ArcSinh* (page 6), *ArcTanh* (page 9), *Cosh* (page 12).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Inverse Hyperbolic Functions (Wikipedia).

1.1.4 ArcSin

```
ArcSin(
      x              ! (input) numerical expression
      )
```

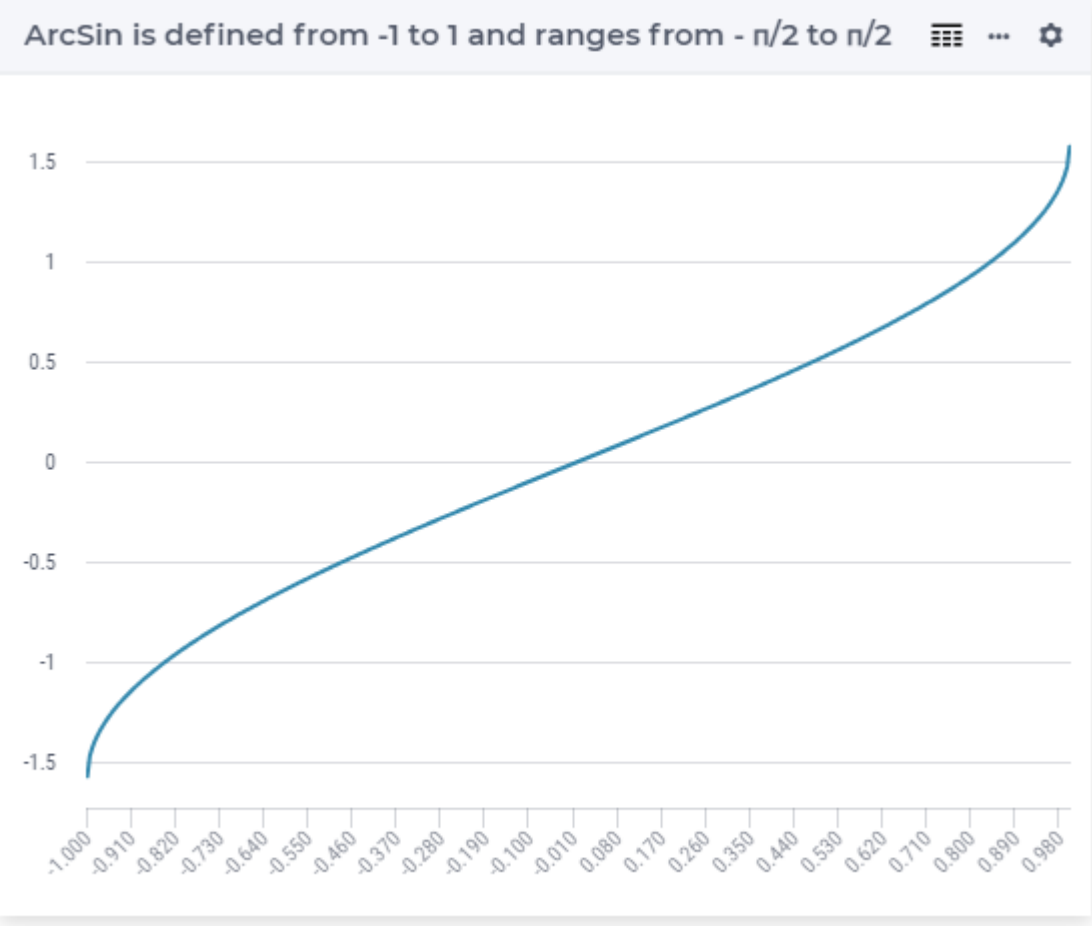
Arguments

x A scalar numerical expression in the range $[-1, 1]$.

Return Value

The *ArcSin* (page 4) function returns the arcsine of x in the range $-\pi/2$ to $\pi/2$ radians.

Graph



Example

```
_p_returnM1 := arcSin(-1); ! returns -pi/2
_p_returnZ  := arcSin( 0); ! returns 0
_p_returnP1 := arcsin( 1); ! returns pi/2
```

Note:

- A run-time error results if x is outside the range $[-1, 1]$.
- The function *ArcSin* (page 4) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *ArcCos* (page 1), *ArcTan* (page 7), *Sin* (page 34).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Inverse Trigonometric Functions (Wikipedia)

1.1.5 ArcSinh

```
ArcSinh(
       x             ! (input) numerical expression
       )
```

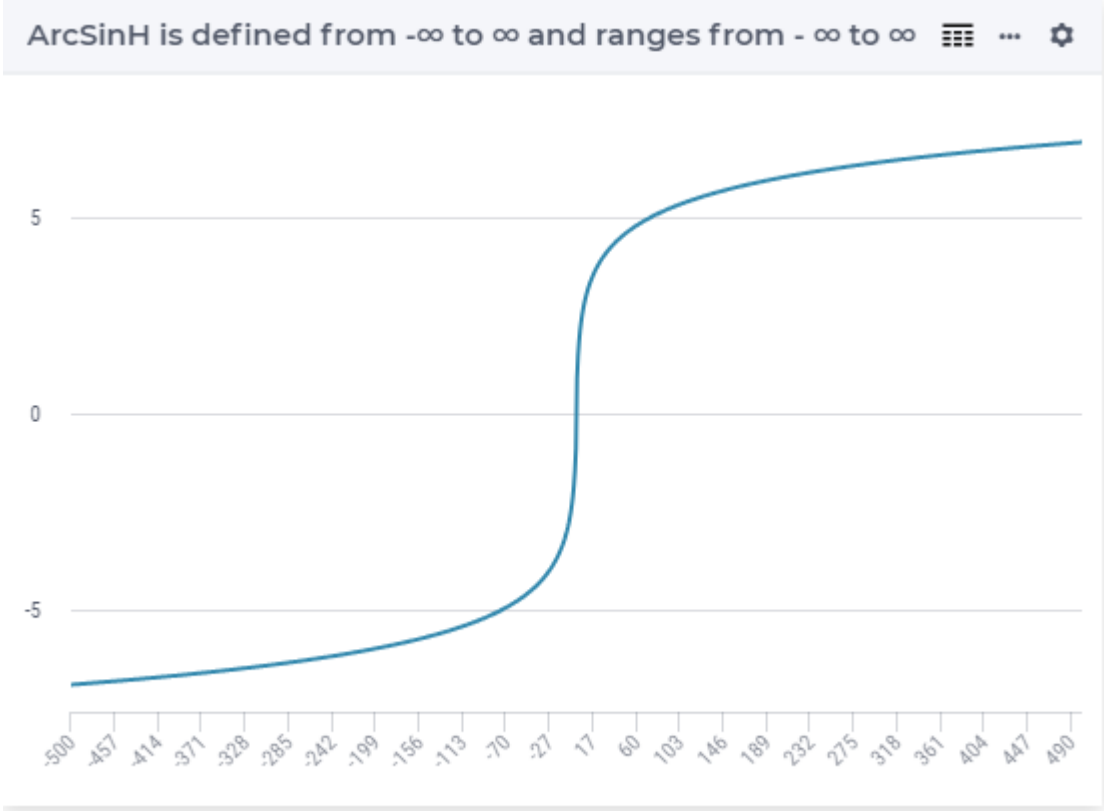
Arguments

x A scalar numerical expression.

Return Value

The *ArcSinh* (page 6) function returns the inverse hyperbolic sine of x in the range from $-\infty$ to ∞.

Graph



Example

```
_p_returnA := arcSinH(-1e9); ! returns -21.416413017506
_p_returnB := arcSinH(-1e3); ! returns -7.600902709542
_p_returnC := arcsinH( 0.0); ! returns 0
_p_returnD := arcSinH( 1e3); ! returns 7.600902709542
_p_returnE := arcSinH( 1e9); ! returns 21.416413017506
```

Note: The function *ArcSinh* (page 6) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *ArcCosh* (page 3), *ArcTanh* (page 9), *Sinh* (page 36).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Inverse Hyperbolic Functions (Wikipedia).

1.1.6 ArcTan

```
ArcTan(
      x              ! (input) numerical expression
      )
```

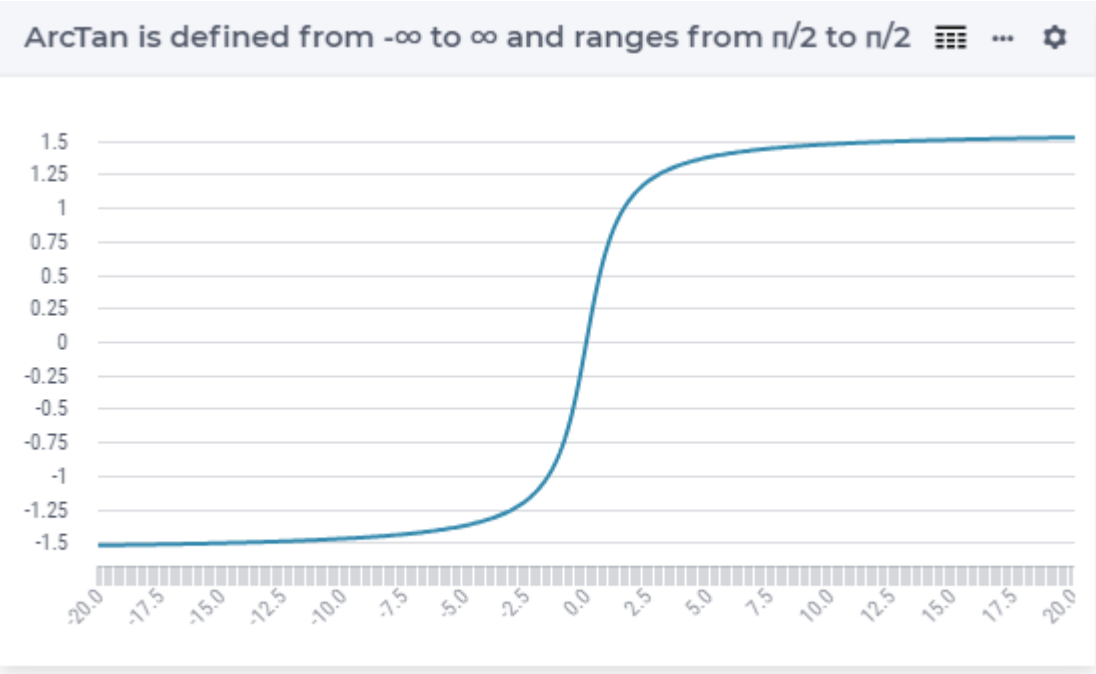
Arguments

x A scalar numerical expression.

Return Value

The *ArcTan* (page 7) function returns the arctangent of x in the range $-\pi/2$ to $\pi/2$ radians.

Graph



Example

```
_p_returnM1 := arcTan(-1e9); ! returns almost -pi/2
_p_returnZ  := arcTan( 0); ! returns 0
_p_returnP1 := arcTan( 1e9); ! returns almost pi/2
```

Note: The function *ArcTan* (page 7) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *ArcSin* (page 4), *ArcCos* (page 1), *Tan* (page 40).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Inverse Trigonometric Functions (Wikipedia).

1.1.7 ArcTanh

```
ArcTanh(
       x              ! (input) numerical expression
       )
```

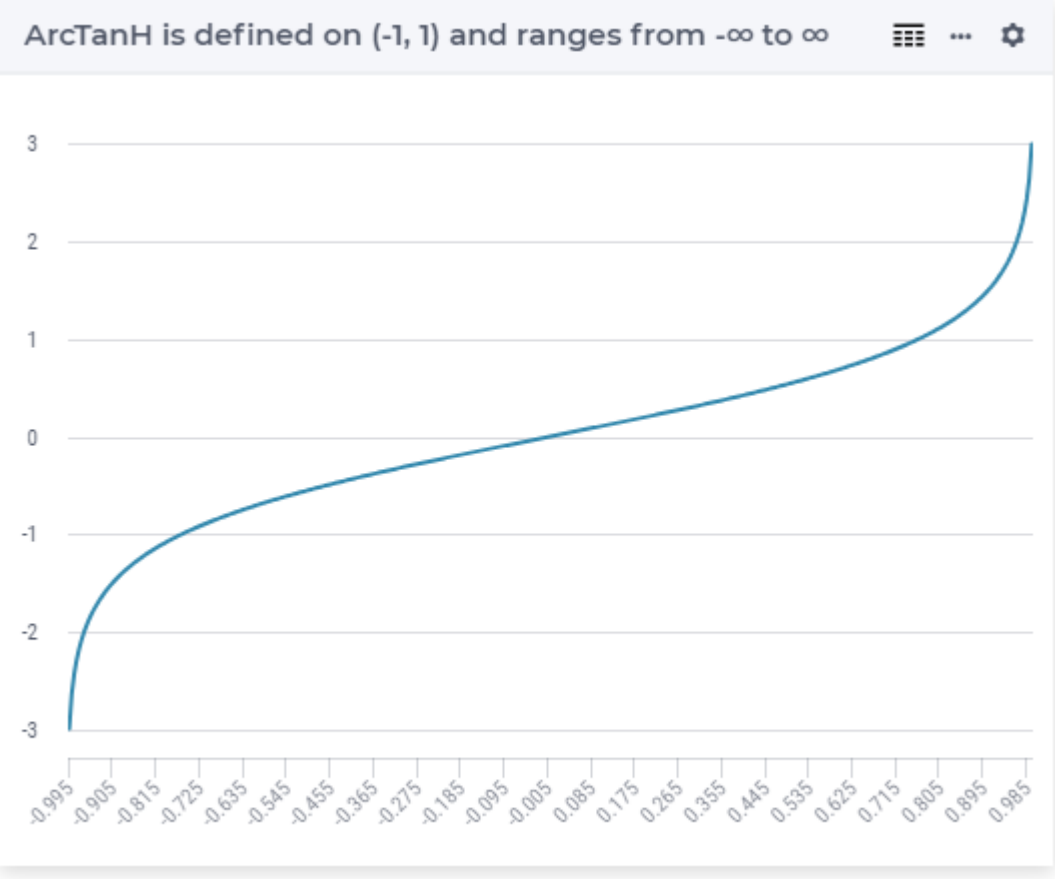
Arguments

x A scalar numerical expression in the range $(-1, 1)$.

Return Value

The *ArcTanh* (page 9) function returns the inverse hyperbolic tangent of x.

Graph



Example

```
_p_returnA := arcTanH(-0.99); ! returns -2.646652412362
_p_returnB := arcTanH( 0   ); ! returns  0
_p_returnC := arcTanH( 0.99); ! returns  2.646652412362
```

Note:

- A run-time error results if x is outside the range $(-1, 1)$.
- The function *ArcTanh* (page 9) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *ArcCosh* (page 3), *ArcSinh* (page 6), *Tanh* (page 42).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Inverse Hyperbolic Functions (Wikipedia).

1.1.8 Ceil

```
Ceil(
    x              ! (input) numerical expression
    )
```

Arguments

x A scalar numerical expression.

Return Value

The function *Ceil* (page 10) returns the smallest integer value $\geq x$.

Example

```
_p_returnA := ceil(-1.00); ! returns -1
_p_returnB := ceil(-0.99); ! returns  0
_p_returnC := ceil( 0   ); ! returns  0
_p_returnD := ceil( 0.99); ! returns  1
```

Note:

- The function *Ceil* (page 10) will round to the nearest integer, if it lies within the equality tolerances `equality_absolute_tolerance` and `equality_relative_tolerance`.
- The function *Ceil* (page 10) can be used in the constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems around integer values.
- When the numerical expression contains a unit, the function *Ceil* (page 10) will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

- The functions *Floor* (page 20), *Round* (page 30), *Precision* (page 28), *Trunc* (page 43).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Numeric tolerances are discussed in Arithmetic Functions of the Language Reference.

1.1.9 Cos

```
Cos(
   x              ! (input) numerical expression
   )
```

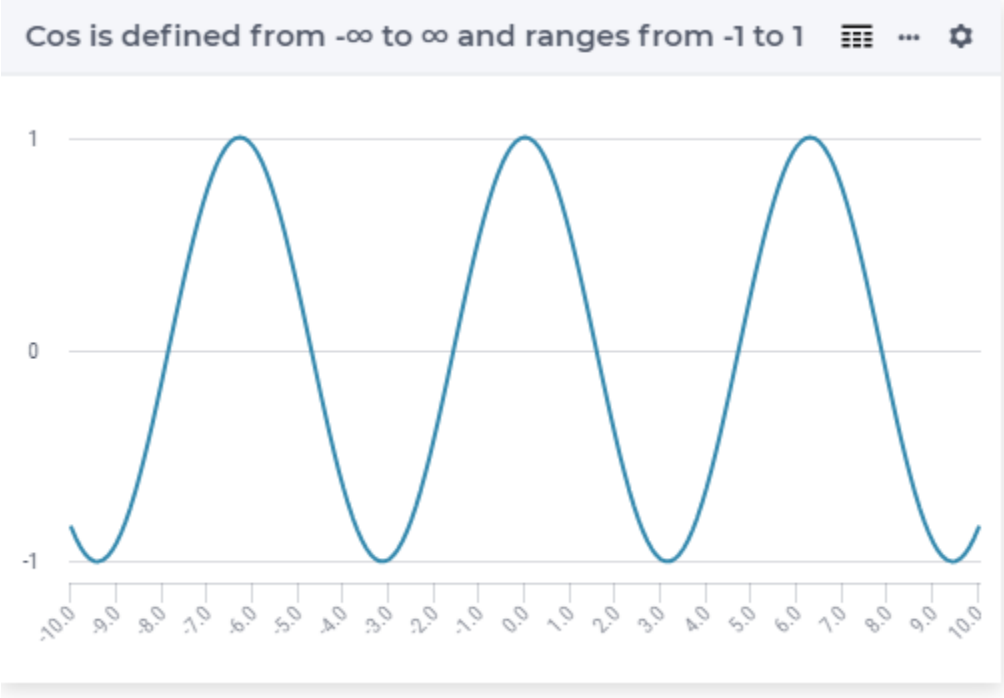
Arguments

x A scalar numerical expression in radians.

Return Value

The *Cos* (page 11) function returns the cosine of *x* in the range −1 to 1.

Graph



Example

```
_p_returnM1 := Cos(-p_pi); ! cos(-pi) = -1
_p_returnZ  := Cos( 0   ); ! cos( 0 ) =  1
_p_returnP1 := Cos( p_pi); ! cos( pi) = -1
```

Note: The function *Cos* (page 11) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Sin* (page 34), *Tan* (page 40), *ArcCos* (page 1).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Sine and Cosine (Wikipedia).

1.1.10 Cosh

```
Cosh(
    x              ! (input) numerical expression
    )
```

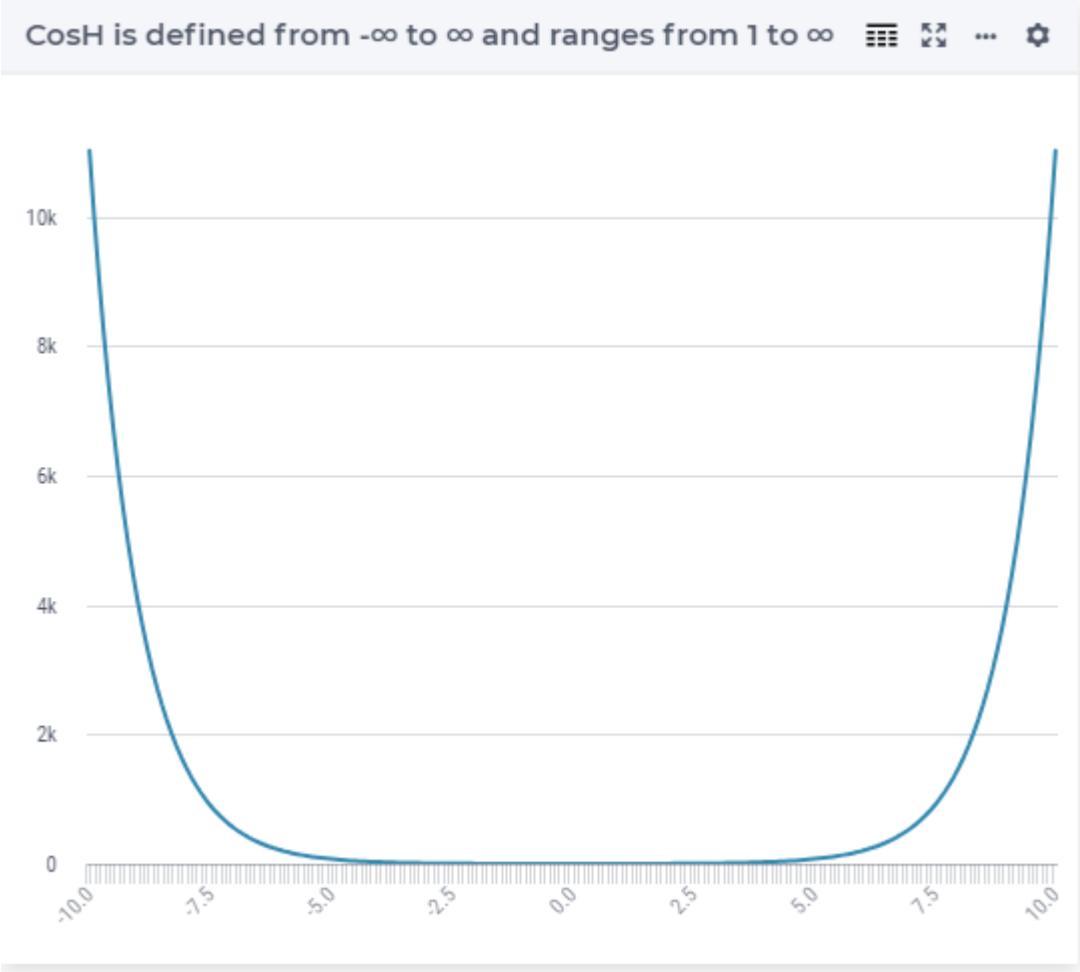
Arguments

x A scalar numerical expression.

Return Value

The *Cosh* (page 12) function returns the hyperbolic cosine of x in the range 1 to ∞.

Graph



Example

```
_p_returnM1 := Cosh(-10); ! cosh(-10) = 11013.232920103323
_p_returnZ  := Cosh( 0 ); ! cosh(  0) =     1
_p_returnP1 := Cosh( 10); ! cosh( 10) = 11013.232920103323
```

Note: The function *Cosh* (page 12) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Sinh* (page 36), *Tanh* (page 42), *ArcCosh* (page 3).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Hyperbolic Functions (Wikipedia)

1.1.11 Cube

```
Cube(
    x             ! (input) numerical expression
    )
```

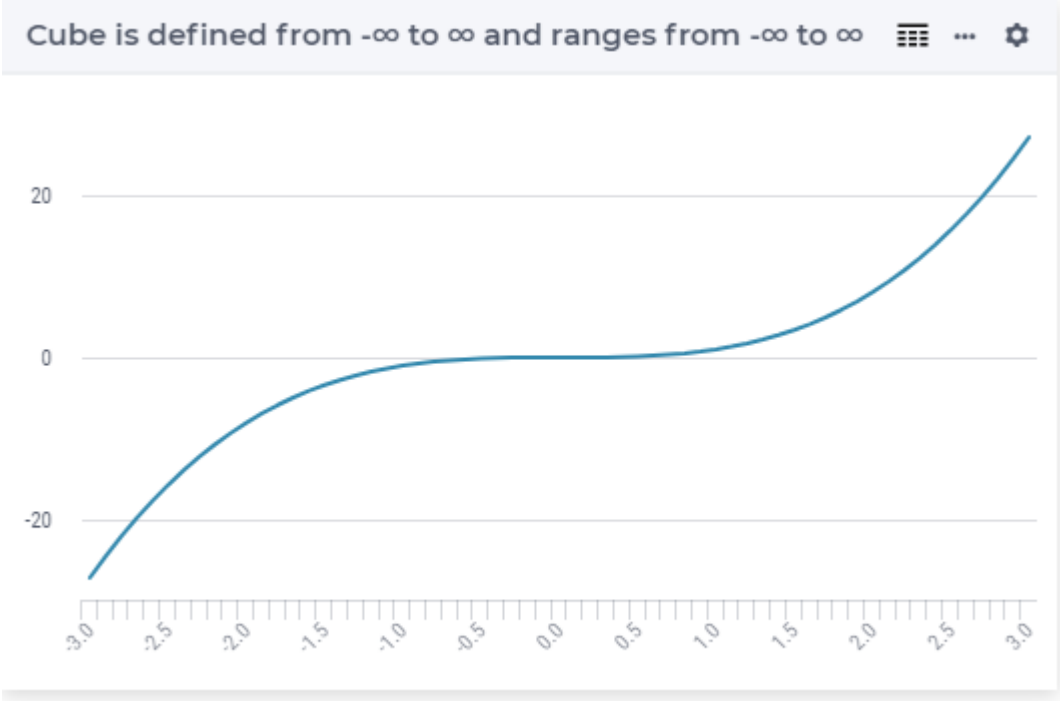
Arguments

x A scalar numerical expression.

Return Value

The function *Cube* (page 14) returns x^3.

Graph



Example

```
_p_returnM1 := Cube(-1); ! Cube(-1) = -1
_p_returnZ  := Cube( 0); ! Cube( 0) =  0
_p_returnP1 := Cube( 1); ! Cube( 1) =  1
_p_returnP2 := Cube( 2); ! Cube( 2) =  8
```

Note: The function *Cube* (page 14) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Power* (page 27), *Sqr* (page 37), and *Sqrt* (page 39).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.12 Degrees

```
Degrees(
       x             ! (input) numerical expression
       )
```

Arguments

x A scalar numerical expression.

Return Value

The function *Degrees* (page 15) returns the value of *x* converted from radians to degrees.

Example

```
_p_returnA := degrees(p_pi  ); ! degrees(pi)   = 180
_p_returnB := degrees(p_pi/2); ! degrees(pi/2) =  90
_p_returnC := degrees(   0  ); ! degrees(0)    =   0
_p_returnD := degrees(p_pi/4); ! degrees(pi/4) =  45  1
```

Note: The function *Degrees* (page 15) can be used in constraints of linear and nonlinear mathematical programs.

See also:

- The function *Radians* (page 29).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.13 Div

```
Div(
    x,      ! (input) numerical expression
    y       ! (input) numerical expression
    )
```

Arguments

x A scalar numerical expression.

y A scalar numerical expression unequal to 0.

Return Value

The function *Div* (page 16) returns x divided by y rounded down to an integer.

Example

```
_p_returnA := div( 8, 3 ); ! div( 8, 3 ) =   2
_p_returnB := div( 8,-3 ); ! div( 8,-3 ) =  -3
_p_returnC := div(-8, 3 ); ! div(-8, 3 ) =  -3
_p_returnD := div(-8,-3 ); ! div(-8,-3 ) =   2
```

Note: A run-time error results if y equals 0.

See also:

- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Function *Mod* (page 26).

1.1.14 ErrorF

```
ErrorF(
       x             ! (input) numerical expression
       )
```

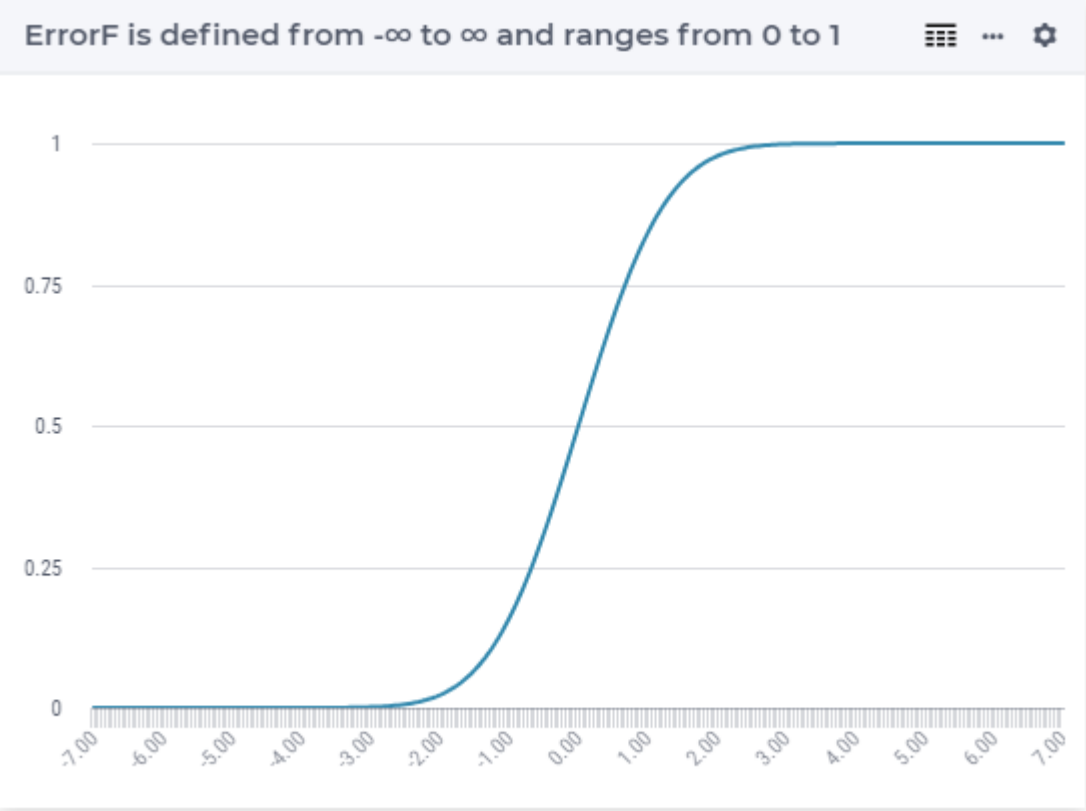
Arguments

x A scalar numerical expression.

Return Value

The function *ErrorF* (page 17) returns the error function value $\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}}\, dt$.

Graph



Example

```
_p_returnM1 := ErrorF(-1); ! ErrorF(-1) = 0.158655253931
_p_returnZ  := ErrorF( 0); ! ErrorF( 0) = 0.5
_p_returnP1 := ErrorF( 1); ! ErrorF( 1) = 0.841344746069
```

Note: The function *ErrorF* (page 17) can be used in constraints of nonlinear mathematical programs.

See also:

- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Error Function (Wikipedia).

1.1.15 Exp

```
Exp(
    x              ! (input) numerical expression
    )
```

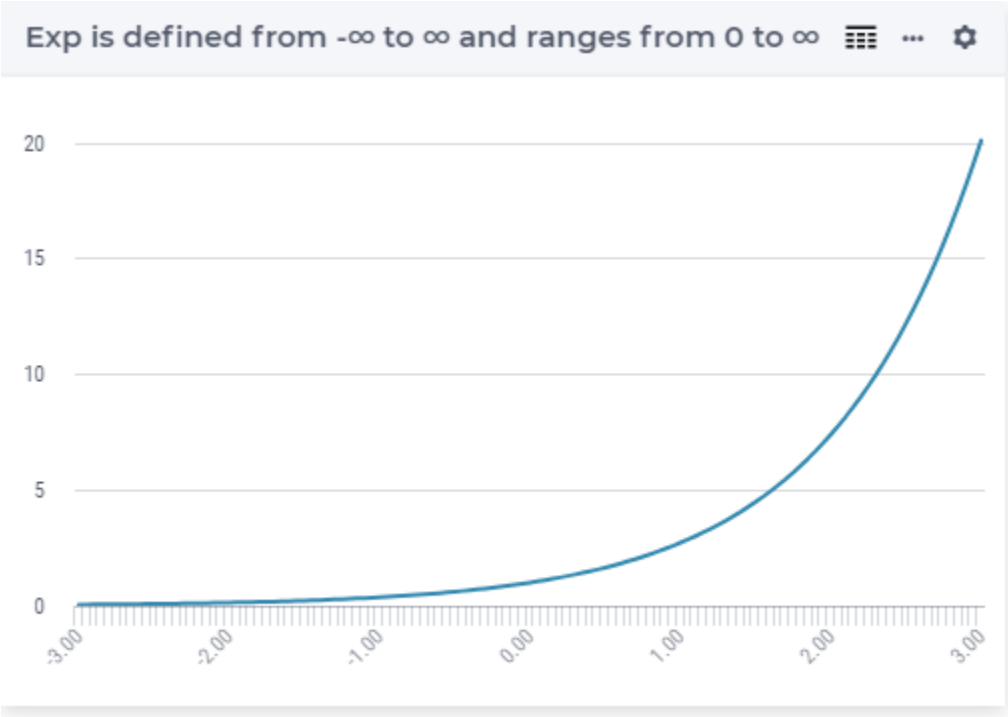
Arguments

x A scalar numerical expression.

Return Value

The function *Exp* (page 18) returns the exponential value e^x.

Graph



Example

```
_p_returnM1 := Exp(-1); ! Exp(-1) = 0.367879441171
_p_returnZ  := Exp( 0); ! Exp( 0) = 1
_p_returnP1 := Exp( 1); ! Exp( 1) = 2.718281828459
```

Note: The function *Exp* (page 18) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Log* (page 21), *Log10* (page 22).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Exponential Function (Wikipedia).

1.1.16 Floor

```
Floor(
     x             ! (input) numerical expression
     )
```

Arguments

x A scalar numerical expression.

Return Value

The function *Floor* (page 20) returns the largest integer value $\leq x$.

Example

```
_p_returnA := Floor(-1.00); ! returns -1
_p_returnB := Floor(-0.99); ! returns -1
_p_returnC := Floor( 0   ); ! returns  0
_p_returnD := Floor( 0.99); ! returns  0
```

Note:

- The function *Floor* (page 20) will round to the nearest integer, if it lies within the equality tolerances `equality_absolute_tolerance` and `equality_relative_tolerance`.
- The function *Floor* (page 20) can be used in the constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems around integer values.
- When the numerical expression contains a unit, the function *Floor* (page 20) will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

- The functions *Ceil* (page 10), *Round* (page 30), *Precision* (page 28), *Trunc* (page 43).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Numeric tolerances are discussed in Arithmetic Functions of the Language Reference.

1.1.17 Log

```
Log(
    x              ! (input) numerical expression
    )
```

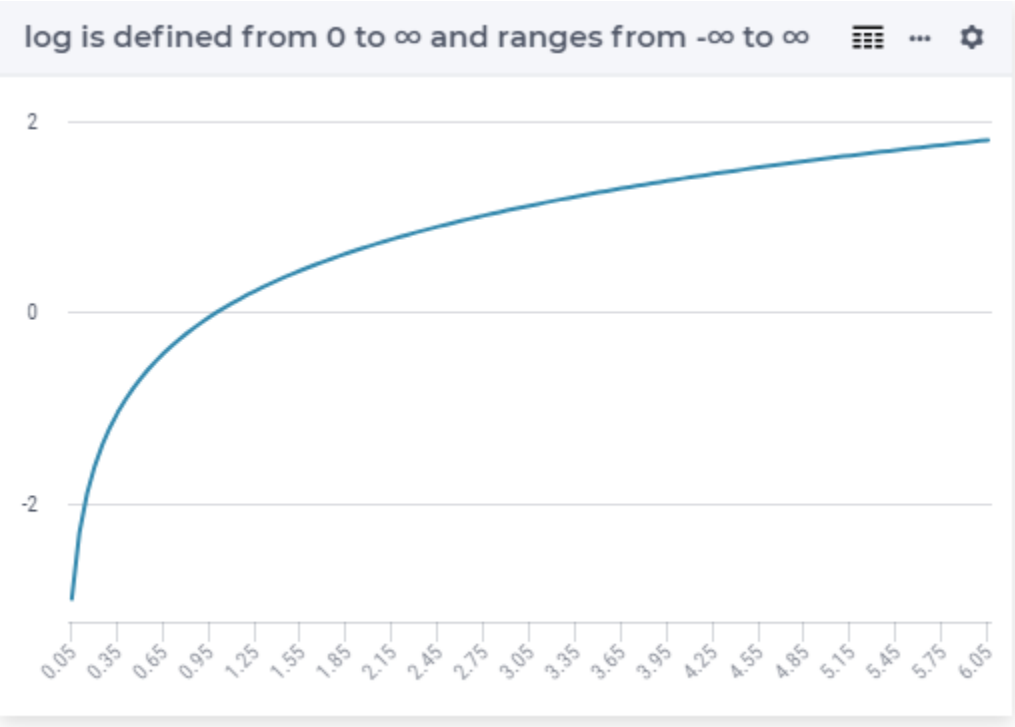
Arguments

x A scalar numerical expression in the range $(0, \infty)$.

Return Value

The function *Log* (page 21) returns the natural logarithm $\ln(x)$.

Graph



Example

```
p_return := Log(10);
! Result is 2.302585093
```

Note:

- A run-time error results if x is outside the range $(0, \infty)$.
- The function *Log* (page 21) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Exp* (page 18), *Log10* (page 22).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.18 Log10

```
Log10(
     x              ! (input) numerical expression
     )
```

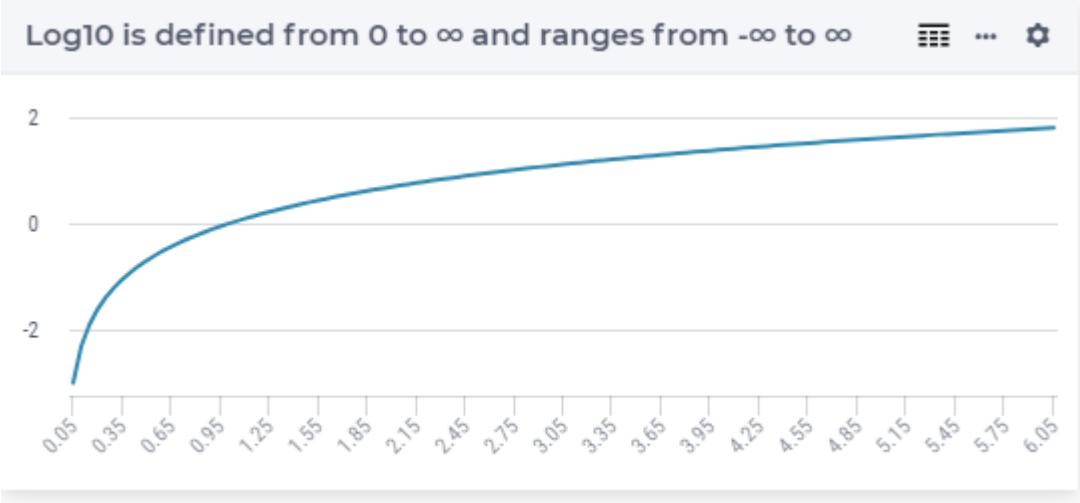
Arguments

x A scalar numerical expression in the range $(0, \infty)$.

Return Value

The function *Log10* (page 22) returns the base-10 logarithm of x.

Graph



Example

```
p_return := Log(10);
! Result is 1
```

Note:

- A run-time error results if x is outside the range $(0, \infty)$.
- The function *Log10* (page 22) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Exp* (page 18), *Log* (page 21).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.19 MapVal

```
MapVal(
      x              ! (input) numerical expression
      )
```

Arguments

x A scalar numerical expression.

Return Value

The function *MapVal* (page 23) returns the (integer) mapping value of any real or special number *x*, according to the following table.

Value x	Description	MapVal value
number	any valid real number	0
UNDF	undefined (result of an arithmetic error)	4
NA	not available	5
INF	$+\infty$	6
-INF	$-\infty$	7
ZERO	numerically indistinguishable from zero, but has the logical value of one.	8

Example

```
_p_val := inf ;
_p_return := mapval(_p_val); ! mapval(inf) returns 6.
```

See also:

- Special numbers in AIMMS and the *MapVal* (page 23) function are discussed in full detail in Real Values and Arithmetic Extensions of the Language Reference.

1.1.20 Max

```
Max(
    x1,      ! (input) numerical, string or element expression
    x2,      ! (input) numerical, string or element expression
    ...
    )
```

- The **Function** Max, returns the maximum of its arguments.
- Not to be confused with the **iterative operator**, Max, which returns the maximum of all values iterated over.

Arguments

x1,x2,... Multiple numerical, string or element expressions.

Return Value

The function *Max* (page 24) returns the largest number, the string highest in the lexicographical ordering, or the element value with the highest ordinal value, among $x1, x2, \ldots$

Example

```
_s_names := data { John, Jack } ;
_p_values(_i_name) := data { John : 3, Jack : 4 } ;
_ep_name1 := 'John' ;
_ep_name2 := 'Jack' ;

_sp_name1 := formatString("%e", _ep_name1);
_sp_name2 := formatString("%e", _ep_name2);
_sp_names(_i_name) := formatString("%e", _i_name);

_p_maxFunc := max( _p_values( 'John' ), _p_values( 'Jack' ) ); ! returns 4
_p_maxIter := max( _i_name, _p_values( _i_name ) ); ! returns 4

! For elements, ordering is based on their positions in the set.
_ep_maxFunc := max( _ep_name1, _ep_name2 ); ! returns 'Jack'.
_ep_maxIter := max( _i_name, _i_name ); ! returns 'Jack'.

! For strings, ordering is the lexicographic ordering.
_sp_maxFunc := max( _sp_name1, _sp_name2 ); ! returns "John"
_sp_maxIter := max( _i_name, _sp_names(_i_name) );  ! returns "John"
```

Note: The function *Max* (page 24) can be used in constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems if the first order derivatives of two arguments between which the *Max* (page 24) function switches are discontinous.

See also:

- The function *Min* (page 25).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.21 Min

```
Min(
    x1,      ! (input) numerical, string or element expression
    x2,      ! (input) numerical, string or element expression
    ..
    )
```

- The **Function** `Min`, returns the minimum of its arguments.
- Not to be confused with the **iterative operator**, `Min`, which returns the minimum of all values iterated over.

Arguments

x1,x2,... Multiple numerical, string or element expressions.

Return Value

The function *Min* (page 25) returns the smallest number, the string lowest in the lexicographical ordering, or the element value with the lowest ordinal value, among $x1, x2, \ldots$

Example

```
_s_names := data { John, Jack } ;
_p_values(_i_name) := data { John : 3, Jack : 4 } ;
_ep_name1 := 'John' ;
_ep_name2 := 'Jack' ;

_sp_name1 := formatString("%e", _ep_name1);
_sp_name2 := formatString("%e", _ep_name2);
_sp_names(_i_name) := formatString("%e", _i_name);

! For elements, ordering is based on their positions in the set.
_p_minFunc := min( _p_values( 'John' ), _p_values( 'Jack' ) ); ! returns 3
_p_minIter := min( _i_name, _p_values( _i_name ) ); ! returns 3

! For strings, ordering is the lexicographic ordering.
_ep_minFunc := min( _ep_name1, _ep_name2 ); ! returns 'John'.
_ep_minIter := min( _i_name, _i_name ); ! returns 'John'.

_sp_minFunc := min( _sp_name1, _sp_name2 ); ! returns "Jack"
_sp_minIter := min( _i_name, _sp_names(_i_name) );  ! returns "Jack"
```

Note: The function *Min* (page 25) can be used in constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems if the first order derivatives of two arguments between which the *Min* (page 25) function switches are discontinous.

See also:

- The function *Max* (page 24).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.22 Mod

```
Mod(
    x,      ! (input) numerical expression
    y       ! (input) numerical expression
    )
```

Arguments

x A scalar numerical expression.

y A scalar numerical expression unequal to 0.

Return Value

The function *Mod* (page 26) returns the remainder of x after division by $|y|$.

- For $y > 0$, the result is an integer in the range $0, \ldots, y-1$ if both x and y are integers, or in the interval $[0, y)$ otherwise.
- For $y < 0$, the result is an integer in the range $y-1, \ldots, 0$ if both x and y are integers, or in the interval $(y, 0]$ otherwise.

Example

```
_p_returnA := mod( 8, 3 ); ! mod( 8, 3 ) =   2
_p_returnB := mod( 8,-3 ); ! mod( 8,-3 ) =  -1
_p_returnC := mod(-8, 3 ); ! mod(-8, 3 ) =   1
_p_returnD := mod(-8,-3 ); ! mod(-8,-3 ) =  -2
```

Note:

- A run-time error results if y equals 0.
- The function *Mod* (page 26) can be used in constraints of mathematical programs. However, nonlinear solver may experience convergence problems if x assumes values around multiples of y.

See also:

- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Function *Div* (page 16).

1.1.23 Power

```
Power(
     x,       ! (input) numerical expression
     y        ! (input) numerical expression
     )
```

Arguments

x A scalar numerical expression.

y A scalar numerical expression.

Return Value

The function *Power* (page 27) returns x raised to the power y.

Example

```
_p_returnA := power( 8, 3 ); ! power( 8, 3 ) =  512
_p_returnB := power( 8,-3 ); ! power( 8,-3 ) =  0.001953125000
_p_returnC := power(-8, 3 ); ! power(-8, 3 ) =  -512
_p_returnD := power(-8,-3 ); ! power(-8,-3 ) =  -0.001953125000
```

Note:

- The following combination of arguments is allowed:
  - $x > 0$
  - $x = 0$ and $y > 0$
  - $x < 0$ and y integer

  In all other cases a run-time error will result.
- The function can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Cube* (page 14), *Sqr* (page 37), and *Sqrt* (page 39).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.24 Precision

```
Precision(
     x,            ! (input) numerical expression
     y             ! (input) integer expression
     )
```

Arguments

x A scalar numerical expression.

y An integer expression.

Return Value

The function *Precision* (page 28) returns *x* rounded to *y* significant digits.

Example

```
_p_val := 123.456789 ;
_p_returnA := precision( _p_val, 1 ); ! precision( _p_val, 1 ) =  100
_p_returnB := precision( _p_val, 3 ); ! precision( _p_val, 3 ) =  123
_p_returnC := precision( _p_val, 5 ); ! precision( _p_val, 5 ) =  123.46
_p_returnD := precision( _p_val, 7 ); ! precision( _p_val, 7 ) =  123.4568
```

Note:

- The function *Precision* (page 28) can be used in constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems around the discontinuities of the *Precision* (page 28) function.
- When the numerical expression contains a unit, the function *Precision* (page 28) will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

- The functions *Round* (page 30), *Ceil* (page 10), *Floor* (page 20), *Trunc* (page 43).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.25 Radians

```
Radians(
       x             ! (input) numerical expression
       )
```

Arguments

x A scalar numerical expression.

Example

```
_p_returnA := radians(180); ! radians(180) = pi
_p_returnB := radians( 90); ! radians( 90) = pi/2
_p_returnC := radians(  0); ! radians(  0) = 0
_p_returnD := radians( 45); ! radians( 45) = pi/4
```

Return Value

The function *Radians* (page 29) returns the value of *x* converted from degrees to radians.

Note: The function *Radians* (page 29) can be used in constraints of linear and nonlinear mathematical programs.

See also:

- The function *Degrees* (page 15).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.26 Round

```
Round(
     x,            ! (input) numerical expression
     decimals      ! (optional) integer expression
     )
```

Arguments

x A scalar numerical expression.

decimals (optional) An integer expression.

Return Value

The function *Round* (page 30) returns the integer value nearest to *x*. In the presence of the optional argument *n* the function *Round* (page 30) returns the value of *x* rounded to *n* decimal places left ($decimals < 0$) or right ($decimals > 0$) of the decimal point.

Note:

- The function *Round* (page 30) can be used in constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems around the discontinuities of the *Round* (page 30) function.

- When the numerical expression contains a unit (has a magnitude), the function *Round* (page 30) will first convert the expression to that unit, before evaluating the function itself. See also the option `rounding compatibility` in the option category `backward compatibility`.

Example

Basic

```
_p_val := 123.456789 ;
_p_returnA := round( _p_val, -1 ); ! round( _p_val, -1 ) =  120
_p_returnB := round( _p_val,  0 ); ! round( _p_val,  0 ) =  123
_p_returnC := round( _p_val,  2 ); ! round( _p_val,  2 ) =  123.46
_p_returnD := round( _p_val,  5 ); ! round( _p_val,  5 ) =  123.45679
```

The above illustrates how to round up to a number of decimals, either positive or negative.

Arguments with a Magnitude

Let the magnitudes of interest be specified in a quantity, for instance the unitless quantity:

```
Quantity SI_Unitless {
    BaseUnit: -;
    Conversions: {
        %   -> - : #-># / 100,
        k1  -> - : #-># * 1000,
        M1  -> - : #-># * 1000000
    }
    Comment: "Expresses a dimensionless value.""}
```

then these units can be used to specify the magnitudes of the following identifiers:

```
Parameter _p_inpSmall;
Parameter _p_pct {
    Unit: %;
}
Parameter _p_rndSmall;
Parameter _p_rndpct {
    Unit: %;
}
Parameter _p_inpLarge;
Parameter _p_kilo {
    Unit: k1;
}
Parameter _p_mega {
    Unit: M1;
}
Parameter _p_rndLarge;
Parameter _p_rndKilo {
    Unit: k1;
}
```

(continues on next page)

(continued from previous page)

```
Parameter _p_rndMega {
    Unit: M1;
}
```

And the following code will assign some values:

```
_p_inpSmall := 0.12345 ;
_p_pct := _p_inpSmall ;

_p_rndSmall := round( _p_inpSmall );
_p_rndPct := round( _p_pct );

_p_inpLarge := 123456789 ;
_p_kilo := _p_inpLarge ;
_p_mega := _p_inpLarge ;

_p_rndLarge := round( _p_inpLarge );
_p_rndKilo := round( _p_kilo );
_p_rndMega := round( _p_mega );

block where listing_number_precision := 6 ;
    display { _p_inpSmall, _p_pct, _p_rndSmall, _p_rndPct },
        { _p_inpLarge, _p_kilo, _p_mega,
        _p_rndLarge, _p_rndKilo, _p_rndMega } ;
endblock ;
```

The code will produce the following overview in the listing file:

```
_p_inpSmall :=  0.123450      ;
_p_pct      := 12.345000  [%] ;
_p_rndSmall :=         0      ;
_p_rndpct   :=        12  [%] ;

_p_inpLarge :=     123456789       ;
_p_kilo     := 123456.789000  [k1] ;
_p_mega     :=    123.456789  [M1] ;
_p_rndLarge :=     123456789       ;
_p_rndKilo  :=        123457  [k1] ;
_p_rndMega  :=           123  [M1] ;
```

As you can see, `_p_rndpct` is a rounded multiple of a percentage. Note that in the computation of this value, round uses the unit of its argument, so the unit of `_p_pct`.

Similarly, `_p_rndKilo`, and `_p_rndMega` are rounded multiples of 1.000 and 1.000.000 respectively by using the units of `_p_kilo` and `_p_mega`.

See also:

- *Precision* (page 28), *Ceil* (page 10), *Floor* (page 20), *Trunc* (page 43).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.27 ScalarValue

```
ScalarValue(
    identifier,    ! (input) element expression into AllIdentifiers
    suffix         ! (optional) element expression into AllSuffixNames
   )
```

Arguments

identifier A scalar element expression into *AllIdentifiers* (page 821).

suffix A scalar element expression into *AllSuffixNames* (page 809).

Return Value

The function *ScalarValue* (page 32) returns the value contained in the scalar identifier *identifier* or scalar reference *identifier.suffix*.

Example

Given the declarations inside module `elementary::arithmetic::funcScalarValue`:

```
DeclarationSection scalar_value_global_declarations {
    Parameter p_rev;
    Parameter p_cst;
    Parameter p_pct;
    Parameter p_cap;
}
```

Then the code:

```
p_rev := 12 ;
p_cst := 34 ;
p_pct := 56 ;
p_cap := 78 ;
_s_kpis := scalar_value_global_declarations ;
_p_kpiVal(_i_kpi) := ScalarValue( _i_kpi );

block where single_column_display := 1;
    display _p_kpiVal ;
endblock ;
```

Will put the following table in the listing file:

```
_p_kpiVal := data
{ 'elementary::arithmetic::funcScalarValue::p_rev' : 12,
  'elementary::arithmetic::funcScalarValue::p_cst' : 34,
  'elementary::arithmetic::funcScalarValue::p_pct' : 56,
  'elementary::arithmetic::funcScalarValue::p_cap' : 78 } ;
```

Note: When *identifier* or *identifier.suffix* is not a scalar numerical valued reference, the function *ScalarValue* (page 32) returns 0.0.

See also:

- The function *Val* (page 44).
- The *ScalarValue* (page 32) function is a function that operates on subsets of *AllIdentifiers* (page 821).
- Other functions that operate on subsets of *AllIdentifiers* (page 821) are referenced in Working with the Set AllIdentifiers of the Language Reference.

1.1.28 Sign

```
Sign(
    x              ! (input) numerical expression
    )
```

Arguments

x A scalar numerical expression.

Return Value

The function *Sign* (page 34) returns $+1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$.

Example

```
_p_returnM1 := Sign(-3); ! returns -1. Sign of negative value is -1.
_p_returnZ  := Sign( 0); ! returns  0. Sign of 0.0 is 0.0
_p_returnP1 := Sign( 3); ! returns  1. Sign of positive value is 1.
```

Note: The function *Sign* (page 34) can be used in constraints of nonlinear mathematical programs. However, nonlinear solver may experience convergence problems round 0.

See also:

- The function *Abs* (page 1).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.29 Sin

```
Sin(
   x              ! (input) numerical expression
   )
```

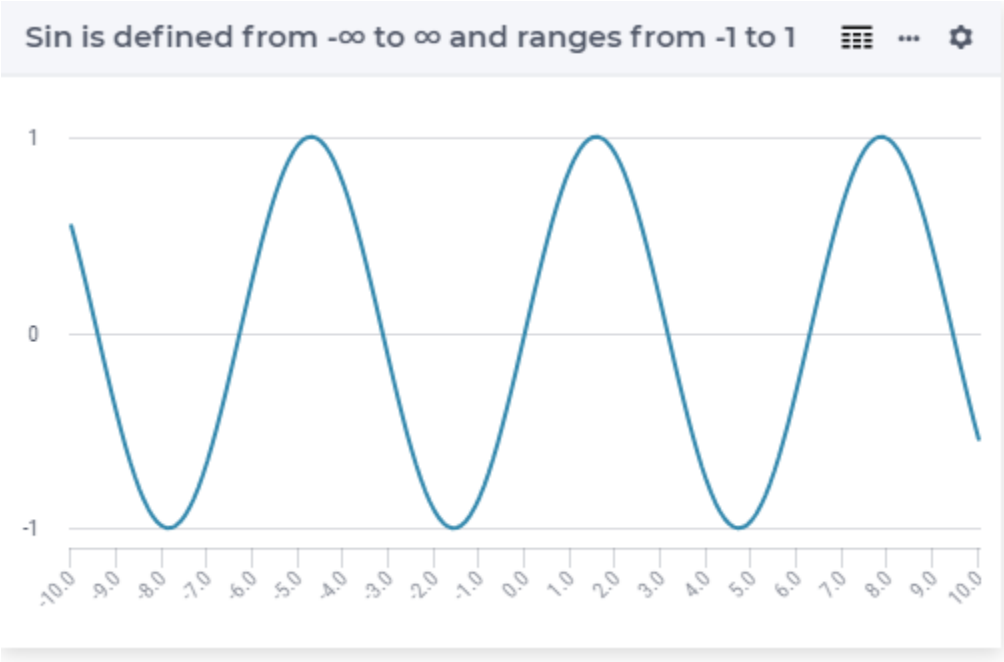
Arguments

x A scalar numerical expression in radians.

Return Value

The *Sin* (page 34) function returns the sine of x in the range -1 to 1.

Graph



Example

```
_p_returnM1 := Sin(-p_pi); ! Sin(-pi) = 0
_p_returnZ  := Sin( 0   ); ! Sin( 0 ) = 0
_p_returnP1 := Sin( p_pi); ! Sin( pi) = 0
```

Note: The function *Sin* (page 34) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Cos* (page 11), *Tan* (page 40), *ArcSin* (page 4).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Sine and Cosine (Wikipedia).

1.1.30 Sinh

```
Sinh(
    x              ! (input) numerical expression
    )
```

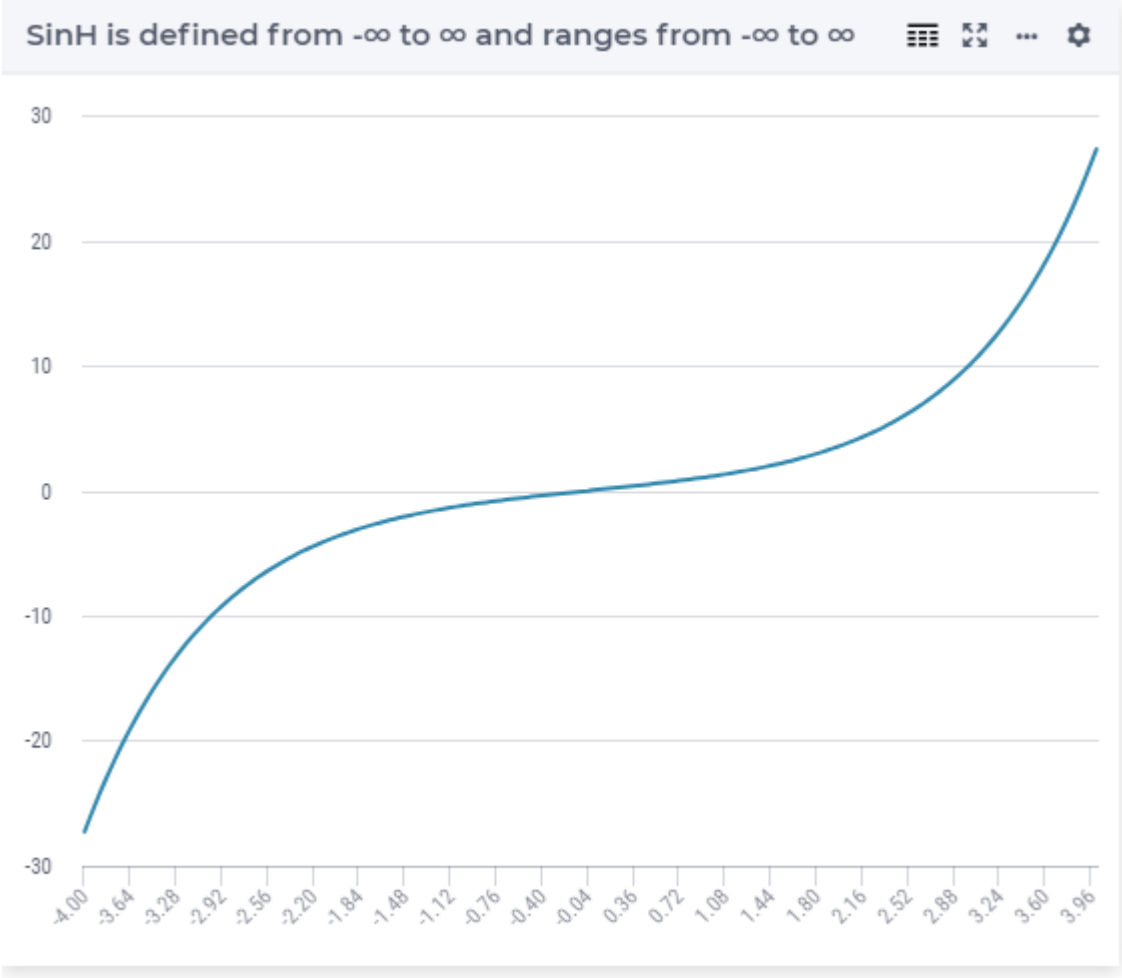
Arguments

x A scalar numerical expression.

Return Value

The *Sinh* (page 36) function returns the hyperbolic sine of x in the range $-\infty$ to ∞.

Graph



Example

```
_p_returnA := SinH( 0    ); ! SinH( 0   ) = 0
_p_returnB := SinH( 1    ); ! SinH( 1   ) = 1.175201193644
_p_returnC := SinH( p_pi ); ! SinH( pi  ) = 11.548739357258
_p_returnD := SinH( 10   ); ! SinH( 1e3 ) = 11013.232874703393
```

Note: The function *Sinh* (page 36) can be used in the constraints of nonlinear mathematical programs.

See also:

- The functions *Cosh* (page 12), *Tanh* (page 42), *ArcSinh* (page 6).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.31 Sqr

```
Sqr(
    x              ! (input) numerical expression
    )
```

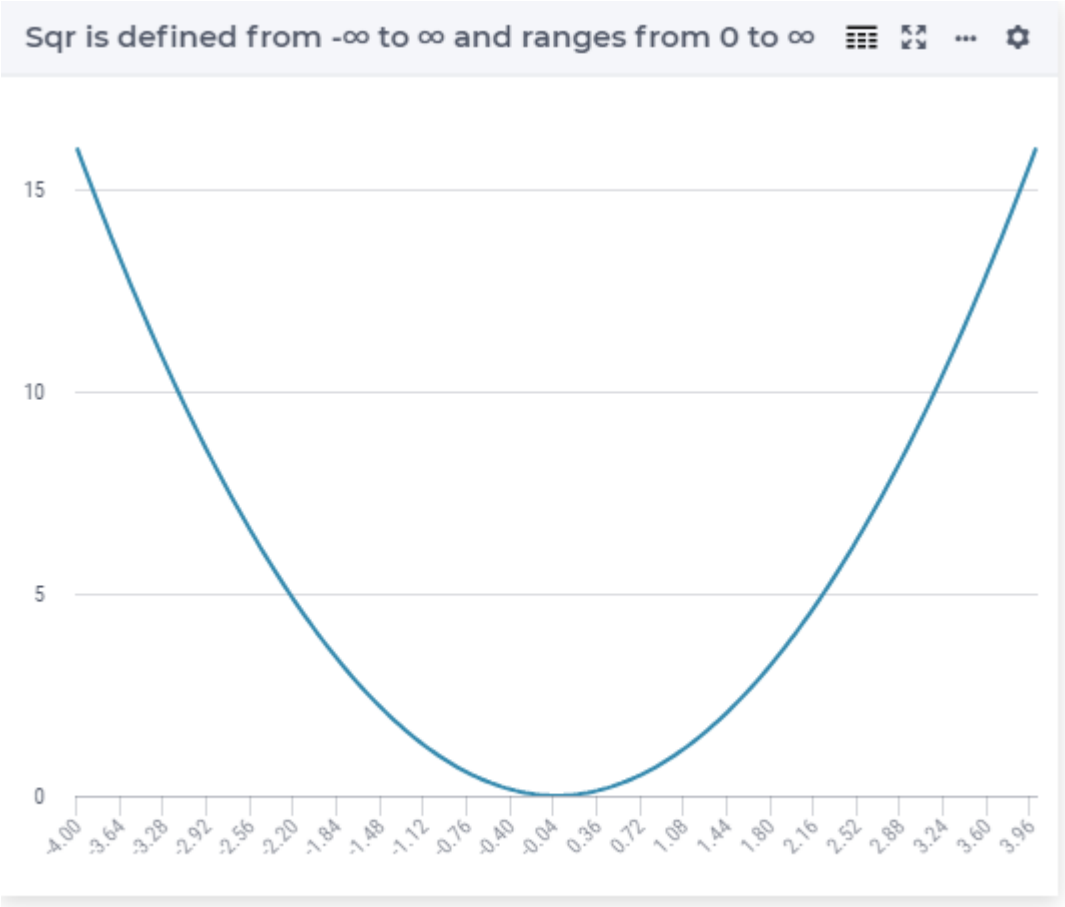
Arguments

x A scalar numerical expression.

Return Value

The function *Sqr* (page 37) returns x^2.

Graph



Example

```
_p_returnA := Sqr( 0    ); ! Sqr(  0 ) = 0
_p_returnB := Sqr( 1    ); ! Sqr(  1 ) = 1
_p_returnC := Sqr( p_pi ); ! Sqr( pi ) = 9.869604401089
_p_returnD := Sqr( 10   ); ! Sqr( 10 ) = 100
```

Note: The function *Sqr* (page 37) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Power* (page 27), *Cube* (page 14), and *Sqrt* (page 39).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.

1.1.32 Sqrt

```
Sqrt(
    x              ! (input) numerical expression
    )
```

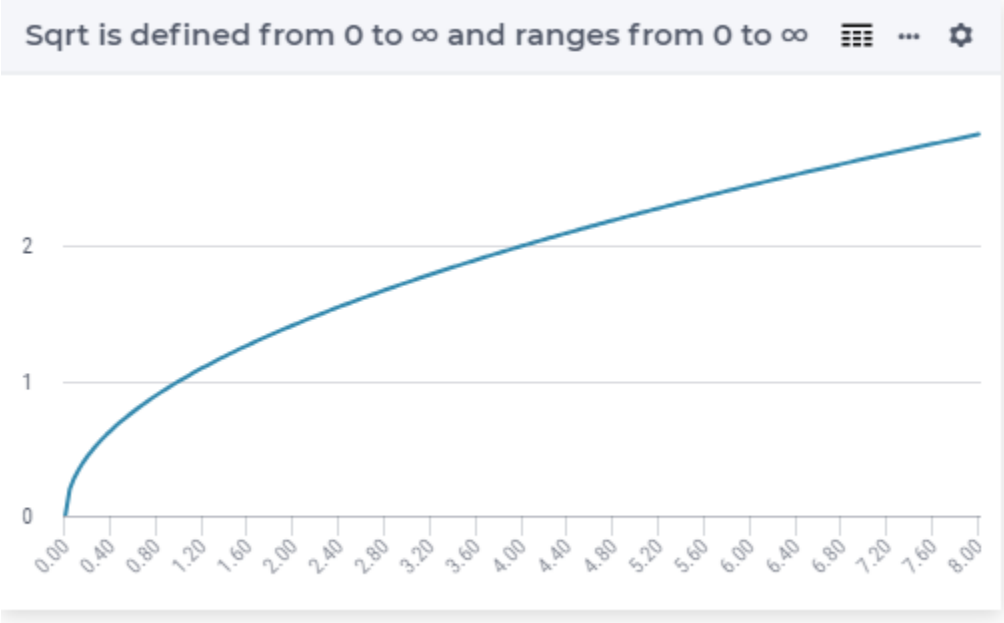
Arguments

x A scalar numerical expression in the range $[0, \infty)$.

Return Value

The function *Sqrt* (page 39) returns the $\sqrt{x}$.

Graph



Example

```
_p_returnA := Sqrt( 0    ); ! Sqrt(  0 ) = 0
_p_returnB := Sqrt( 1    ); ! Sqrt(  1 ) = 1
_p_returnC := Sqrt( p_pi ); ! Sqrt( pi ) = 1.772453850906
_p_returnD := Sqrt( 10   ); ! Sqrt( 10 ) = 3.162277660168
```

Note:

- A run-time error results if x is outside the range $[0, \infty)$.
- The function *Sqrt* (page 39) can be used in the constraints of nonlinear mathematical programs.

See also:

- The functions *Power* (page 27), *Cube* (page 14), and *Sqr* (page 37).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Square Root (Wikipedia).

1.1.33 Tan

```
Tan(
   x              ! (input) numerical expression
   )
```

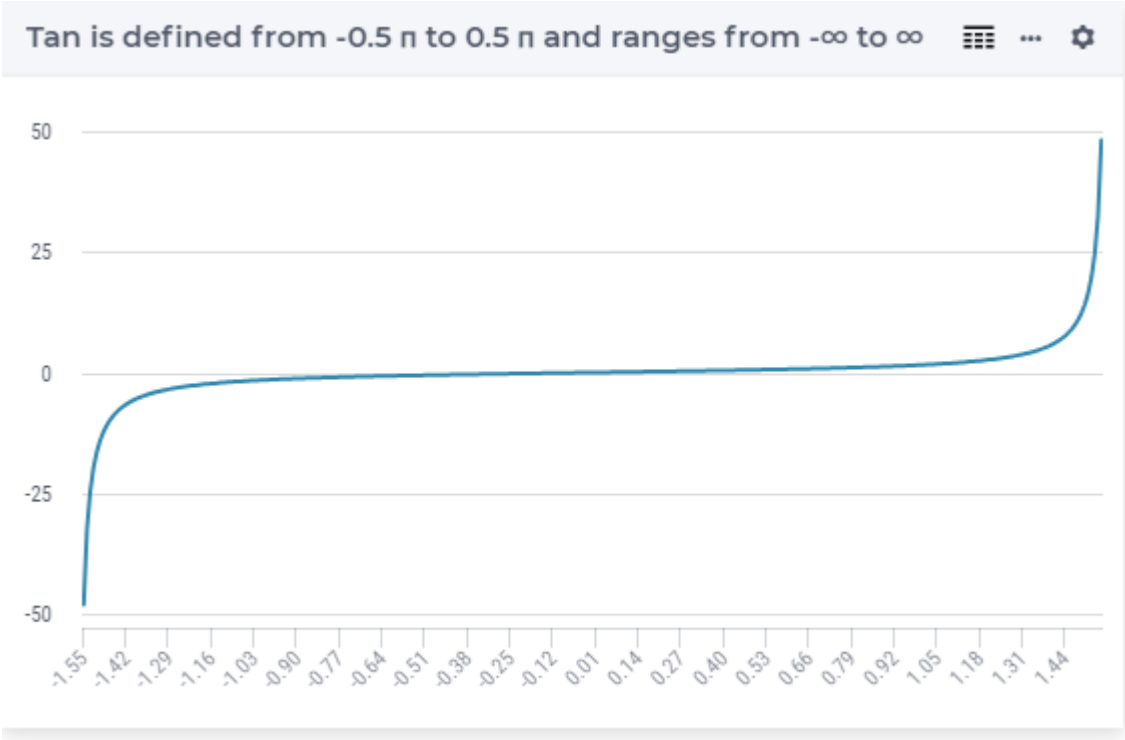
Arguments

x A scalar numerical expression in radians.

Return Value

The *Tan* (page 40) function returns the tangent of x in the range $-\infty$ to ∞.

Graph



Example

```
_p_returnM1 := Tan(-1.56); ! Tan(-1.56) = -92.620496316705
_p_returnZ  := Tan( 0   ); ! Tan( 0) = 0
_p_returnP1 := Tan( 1.56); ! Tan( 1.56) = 92.620496316705
```

Note: The function *Tan* (page 40) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Cos* (page 11), *Sin* (page 34), *ArcTan* (page 7).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Sine and Cosine (Wikipedia).

1.1.34 Tanh

```
Tanh(
    x             ! (input) numerical expression
    )
```

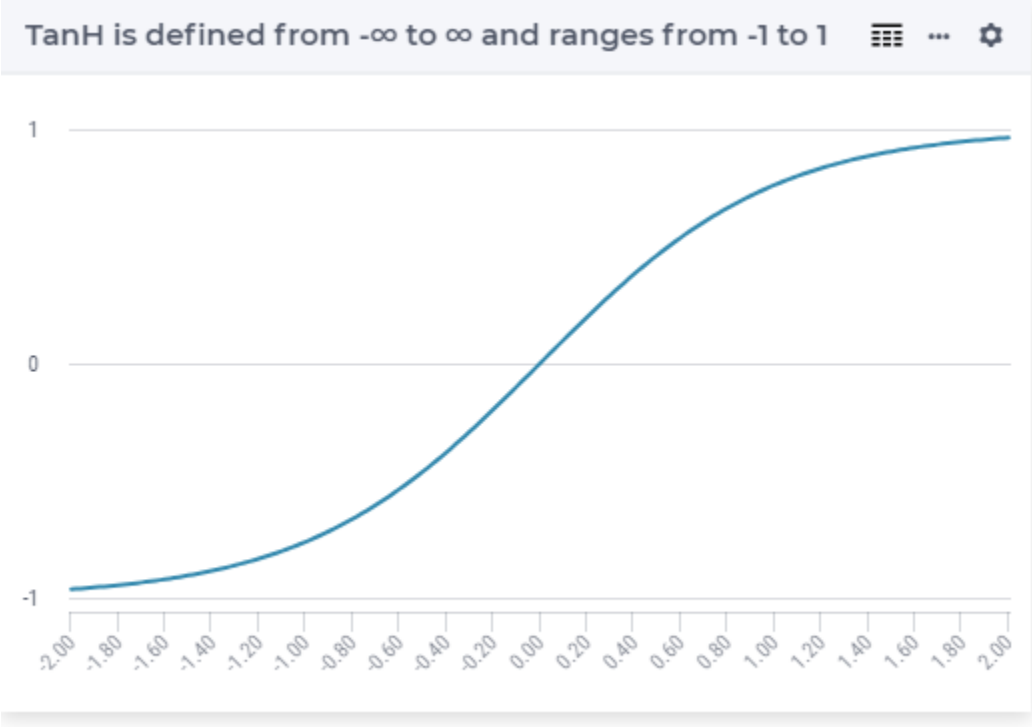
Arguments

x A scalar numerical expression.

Return Value

The *Tanh* (page 42) function returns the hyperbolic tangent of x in the range -1 to 1.

Graph



Example

```
_p_returnA := TanH( 0    ); ! TanH( 0   ) = 0
_p_returnB := TanH( 1    ); ! TanH( 1   ) = 0.761594155956
_p_returnC := TanH( p_pi ); ! TanH( pi  ) = 0.996272076221
_p_returnD := TanH( 1e9  ); ! TanH( 1e3 ) = 1
```

Note: The function *Tanh* (page 42) can be used in constraints of nonlinear mathematical programs.

See also:

- The functions *Cosh* (page 12), *Sinh* (page 36), *ArcTanh* (page 9).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Hyperbolic Functions (Wikipedia).

1.1.35 Trunc

```
Trunc(
     x       ! (input) numerical expression
     )
```

Arguments

x A scalar numerical expression.

Return Value

The function *Trunc* (page 43) returns the truncated value of x: $\text{sgn}\,(x) \cdot \lfloor |\ x\ | \rfloor$.

Example

```
_p_returnA := Trunc(-1.00); ! returns -1
_p_returnB := Trunc(-0.99); ! returns  0
_p_returnC := Trunc( 0   ); ! returns  0
_p_returnD := Trunc( 0.99); ! returns  0
```

Note:

- The function *Trunc* (page 43) will round to the nearest integer, if it lies within the equality tolerances `equality_absolute_tolerance` and `equality_relative_ tolerance`.
- The function *Trunc* (page 43) can be used in the constraints of nonlinear mathematical programs. However, nonlinear solver may experience convergence problems around integer argument values.
- When the numerical expression contains a unit, the function *Trunc* (page 43) will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

- The functions *Ceil* (page 10), *Floor* (page 20), *Round* (page 30), *Precision* (page 28).
- Arithmetic functions are discussed in full detail in Arithmetic Functions of the Language Reference.
- Numeric tolerances are discussed in Arithmetic Functions of the Language Reference.

1.1.36 Val

```
Val(
    str    ! (input) string or element expression
   )
```

Arguments

str A scalar string or element expression.

Return Value

The function *Val* (page 44) returns the numerical value represented by the string or element *str*.

Example

```
_s_nums := data { 1, '3.14' } ;
_p_returnA := Val( first( _s_nums ) ); ! returns 1
_p_returnB := Val( last( _s_nums ) ); ! returns  3.14
_p_returnC := Val( "33"   ); ! returns  33
_p_returnD := Val( "1.0e3"); ! returns  1000
```

Note: If *str* cannot be interpreted as a numerical value, a runtime error may occur, see option `suppress error messages of val function`.

See also:

- The *Val* (page 44) function is discussed in full detail in Intrinsic Functions for Sets and Set Elements of the Language Reference.

1.2 Set Related Functions

AIMMS supports the following set related functions:

1.2.1 ActiveCard

The function *ActiveCard* (page 45) returns the cardinality of active elements in its identifier argument, or the cardinality of active elements of a suffix of that identifier.

```
ActiveCard(
    Identifier,     ! (input) identifier reference
    [Suffix]        ! (optional) element in the set AllSuffixNames
    )
```

An element in identifier P is active if its tuple components are within the domain sets of P and within the index domain condition of P (if any). Otherwise it is inactive.

Arguments

Identifier A reference to a set or an indexed identifier.

Suffix An element in the predefined set *AllSuffixNames* (page 809).

Return Value

1. If *Identifier* is a set, the function ActiveCard returns the number of active elements in *Identifier*.
2. If *Identifier* is an indexed identifier, the function ActiveCard returns the number of active elements in *Identifier*.
3. If *Suffix* is given, the number of active elements for the given suffix of *Identifier*.

Note: The *ActiveCard* (page 45) function cannot be applied to slices of indexed identifiers. In such a case, you can use the Count operator to count the number of active elements.

Example

```
! Small data set: keeping stock of two fruits.
_s_fruits := data { apple, pear } ;
_p_stock(_i_fruit) := data { apple : 4, pear : 5 } ;

! Removing one fruit from our data set.
_s_fruits -= data { pear };
! There is now only one fruit left in _p_stock - _p_stock has one active and one␣
↪inactive element.

display _p_stock ; ! Writes "_p_stock(_i_fruit) := data { apple : 4 };" to listing file.

_p_activeCard := ActiveCard( _p_stock ); ! 1: ActiveCard returns the number of active␣
↪elements.
_p_card := Card( _p_stock ); ! 2; Card returns the number of elements, both active and␣
↪inactive.

! The difference between Card and ActiveCard is the number of inactive elements.
```

See also:

- The function *Card* (page 46) and Count operator (see also Numerical Iterative Operators of the Language Reference).
- The procedure *RestoreInactiveElements* (page 60).
- The explanation of inactive elements in the Language Reference at Inactive data.
- The AIMMS Developer tool Identifier Cardinalities, see Observing identifier cardinalities.

1.2.2 Card

The function *Card* (page 46) returns the cardinality of an identifier, or the cardinality of a suffix of that identifier. To support the various usages there are three different flavors of *Card* (page 46):

```
Card(
    Identifier    ! (input) a set expression
    )
```

```
Card(
    Identifier    ! (input) an identifier reference
    )
```

```
Card(
    Identifier,    ! (input) element in the set AllIdentifiers
    [Suffix]       ! (optional) element in the set AllSuffixNames
    )
```

Arguments

Identifier A reference to an identifier that may contain data, or a simple set or a set expression.

Suffix An element in the predefined set of *AllSuffixNames* (page 809).

Return Value

If the first argument is a set (expression), the function will return the number of elements in the set. In all other cases the function returns the number of nondefault values stored in the data of the given identifier, or in the data of the suffix of that identifier.

Example

```
! Some data
_s_genders := data { male, female };
_s_names := data { John, Jill, Jack, Joan } ;
_s_nameByGender(_i_gender) := data {
        male : { John, Jack },
        female : { Jill, Joan }
} ;

! Getting the cardinality of a set
_p_noGenders := card(_s_genders); ! Returns 2
_p_noNames := card( _s_names );  ! Returns 4

! Getting the cardinality of sets in an indexed set
_p_noNamesByGender(_i_gender) :=
        card( _s_nameByGender( _i_gender ) ); ! Returns 2, twice.
```

Note:

- The *Card* (page 46) function cannot be applied to slices of indexed identifiers. In such a case, you can use the Count operator to count the number of nondefault elements.
- When the *Card* (page 46) function is used inside the definition of a parameter or a set and the first argument is an index or element parameter into the set *AllIdentifiers* (page 821) (using the third prototype above) then the definition depends on all identifiers that can appear on the left hand side of an assignment (sets without a definition, parameters without a definition, variables and constraints). The cardinality will be computed for all identifiers, including those with a definition. These definitions will not be made up to date, however. This is illustrated in the following example.

```
Parameter A;
Parameter B {
    Definition   :  A + 1;
}
Parameter TheCards {
    IndexDomain  :  IndexIdentifiers;
    Definition   :  Card( IndexIdentifiers, 'Level' );
}
Body:
    A := 1;
    display TheCards;
```

Here TheCards is computed in the display statement because A just changed. The definition of TheCards, that is made up to date by the display statement, will, however, not invoke the computation of B, although it is not up to date. This is done in order to avoid circular references while making set and parameter definitions up to date. In order to make B up to date consider using the Update statement, see also Nonprocedural Execution of the Language Reference.

See also:

- The function *ActiveCard* (page 45) and the Count operator.
- Nonprocedural Execution of the Language Reference.

1.2.3 CloneElement

The procedure *CloneElement* (page 48) copies the data associated with a particular element to another element.

```
CloneElement(
  updateSet,              ! (input, output) a set identifier
  originalElement,        ! (input) an element in the set
  cloneName,              ! (input) a string that is the name of the clone
  cloneElement,           ! (output) an element parameter
  includeDefinedSubsets ) ! (optional) an integer, default 0.
```

The procedure *CloneElement* (page 48) performs the following actions:

1. It creates or finds an element with name cloneName: cloneElement. The element cloneElement is inserted into updateSet if it is not already there. This insertion is only permitted if updateSet does not have a definition.
2. For each domain set of updateSet, say insertDomainSet, the element cloneElement is inserted into insertDomainSet if it is not already there. Such an insertion is only permitted if insertDomainSet does not have a definition.

3. For each subset of `updateSet`, say `insertSubset` in which `originalElement` is an element, `cloneElement` is also inserted into `insertSubset`. If `includeDefinedSubsets` is 0, then `insertSubset` is skipped if it is a defined subset.
4. The domain sets of steps 1 and 2, and the sets modified in step 3 form a set, say `modifiedSets`.
5. Identifiers declared over a set in `modifiedSets` that meet one of the following criteria, are selected:
   - It is a non-local multi-dimensional set without a definition.
   - It is a non-local parameter without a definition.
   - It is a variable.
   - It is a constraint.

   These identifiers form the set `modifiedIdentifiers`.
6. For each identifier in the set `modifiedIdentifiers`, and all suffixes of this identifier, the data associated with element `originalElement` is copied to `cloneElement`.

Arguments

updateSet A one-dimensional set.

originalElement An element valued expression that should result in an element in `updateSet`.

cloneName A string expression that should result in a name that is in the set `updateSet` or can be added to that set.

cloneElement An element parameter, in which the resulting element is stored.

includeDefinedSubsets When non-zero, defined subsets are included in the `modifiedSets` as well. When these defined subsets are evaluated thereafter again, this may result in the creation of inactive data. Inactive data can be removed by a `CLEANUP` or `CLEANDEPENDENTS` statement, see Data Control of the Language Reference. Defined subsets that are defined as an enumeration are never included.

Return Value

The procedure returns 1 if successful and 0 otherwise. Possible reasons for returning 0 are:

- `originalElement` is not in `updateSet`.
- `cloneName` equals name of `originalElement`.
- There are no identifiers modified.

Note: If you want to make sure that the string `cloneName` is not yet an element in `updateSet`, use a statement like:

```
if ( not ( cloneName in updateSet ) ) then
    CloneElement( ... );
endif ;
```

Example

With the following declarations (and initial data):

```
Set S {
    Index        :  i, j;
    Parameter    :  ep;
    InitialData  :  data { a };
}
Parameter P {
    IndexDomain  :  i;
    InitialData  :  data { a : 1 };
}
Parameter Q {
    IndexDomain  :  (i,j);
    InitialData  :  data { ( a, a ) : 1 };
}
```

the statement

```
CloneElement( S, 'a', "b", ep );
```

results in S, P, Q and ep having the following data:

```
S := data { a, b } ;
P := data { a : 1,  b : 1 } ;
Q := data { ( a, a ) : 1,  ( a, b ) : 1,  ( b, a ) : 1,  ( b, b ) : 1 } ;
ep := 'b' ;
```

See also:

- The function *StringToElement* (page 66), the procedure *FindUsedElements* (page 53) and the procedure *RestoreInactiveElements* (page 60).

1.2.4 Element

With the function *Element* (page 50) you can retrieve the n-th element from a set.

```
Element(
       Set,           ! (input) set reference
       n              ! (input) integer expression
       )
```

Arguments

Set The set from which an element is to be returned.

n An integer expression indicating the ordinal number of the element to be returned.

Return Value

The function Element returns the n-th element of set *Set*.

Example

```
_s_names := data { Jack, Jill, John, Joan } ;
_s_positions := ElementRange(1,4);

! Relate the positions to the names.
_ep_name( _i_pos ) := element( _s_names, ord( _i_pos ) ) ;
```

Note: If there is no n-th element in *Set*, the function returns the empty element `''` instead.

1.2.5 ElementCast

With the function *ElementCast* (page 51) you can cast an element of one set to an (existing) element with the same name in a set with a different root set.

```
ElementCast(
     set,              ! (input) a set expression
     element,          ! (input) a scalar element expression
     [create]          ! (optional) 0 or 1
     )
```

Arguments

set A set in which you want to find a specific element name.

element A scalar element expression, representing the element that you want to convert to a different root set hierarchy.

create (optional) An indicator whether or not a nonexisting element are added to the set during the call.

Return Value

The function returns the existing element or, if the element cannot be converted to an existing element and the argument *create* is not set to 1, the function returns the empty element. If *create* is set to 1, nonexisting elements will be created on the fly.

Example

The code:

```
s_Jnames := data { Jack, Jill, John, Joan } ;
s_bnames := data { Brian, Brooke, Benjamin, Bianca } ;

for i_jname do
    s_localNames += ElementCast( s_localNames, i_jname, create: 1 );
endfor ;
for i_bname do
    s_localNames += ElementCast( s_localNames, i_bname, create: 1 );
endfor ;
display s_localNames ;
```

Will produce:

```
elementary::setop::funcElementCast::s_localNames :=
data { Jack, Jill, John, Joan, Brian, Brooke, Benjamin, Bianca } ;
```

in the listing file.

See also:

- The procedure *SetElementAdd* (page 64).

1.2.6 ElementRange

With the function *ElementRange* (page 52) you can create a set with elements in which each element can be constructed using a prefix string, a postfix string, and a a sequential number.

```
ElementRange(
     from,            ! (input) integer expression
     to,              ! (input) integer expression
     [incr,]          ! (optional) integer expression
     [prefix,]        ! (optional) string expression
     [postfix,]       ! (optional) string expression
     [fill]           ! (optional) 0 or 1
     )
```

Arguments

from The integer value for which the first element must be created.

to The integer value for which the last element must be created.

incr (optional) The integer-valued interval length between two consecutive elements. If omitted, then the default interval length of 1 is used.

prefix (optional) The prefix string for every element. If omitted, then the elements have no prefix (and thus start with the number).

postfix (optional) The postfix string for every element. If omitted, then the elements have no postfix (and thus end with the number).

fill (optional) This logical indicator specifies whether the numbers must be padded with leading zeroes. If omitted, then the default value 1 is used.

Return Value

The function returns a set containing the created elements.

Example

The code:

```
_s_ints     := ElementRange(1,12);
_s_regions  := ElementRange(1,12,prefix:"reg",fill:0);
_s_products := ElementRange(1,12,prefix:"prd",fill:1);
display _s_ints, _s_regions, _s_products ;
```

produces

```
_s_ints := data { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;

_s_regions := data { reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8, reg9, reg10, reg11,␣
↪reg12 } ;

_s_products := data { prd01, prd02, prd03, prd04, prd05, prd06, prd07, prd08, prd09,␣
↪prd10, prd11, prd12 } ;
```

in the listing file.

1.2.7 FindUsedElements

The procedure *FindUsedElements* (page 53) finds all elements of a particular set that are in use in a given collection of indexed model identifiers.

```
FindUsedElements(
     SearchSet,           ! (input) a set
     SearchIdentifiers,   ! (input) a subset of AllIdentifiers
     UsedElements         ! (output) a subset
     )
```

Arguments

SearchSet The set for which you want to find the used elements.

SearchIdentifiers A subset of `AllIdentifiers` (page 821), holding identifiers that are indexed over SearchSet.

UsedElements A subset of *SearchSet*. On return this subset will contain the elements that are currently used (i.e. have corresponding nondefault values) in the identifiers contained in *SearchIdentifiers*.

Example

Given declarations:

```
Set s_time {
        Index: i_t;
}
Set s_products {
        Index: i_prod;
}
Set s_productsAvailable {
        SubsetOf: s_products;
}
Set s_productsRequired {
        SubsetOf: s_products;
}
Set s_productsRequiredNotAvailable {
        SubsetOf: s_products;
}
Set s_supplyIds {
        SubsetOf: AllIdentifiers;
        Definition: Supply_data;
        Comment: "The identifiers stocking or supplying products.""}
Set s_demandIds {
        SubsetOf: AllIdentifiers;
        Definition: Demand_data;
        Comment: "The identifiers requiring products""}
DeclarationSection Supply_data {
        Parameter p_stock {
                IndexDomain: i_prod;
        }
        Parameter p_production {
                IndexDomain: (i_prod,i_t);
        }
}
DeclarationSection Demand_data {
        Parameter p_demand {
                IndexDomain: i_prod;
        }
}
```

Specifying a bit of data:

```
! Get some sample data
s_time := elementRange(1,3, prefix: "t" );
s_products := ElementRange( 1, 5, prefix: "p" );
p_stock(i_prod) := data { p2 : 3, p3 : 5 } ;
p_production(i_prod,i_t) := data {
    ( p1, t1 ) : 4,
    ( p2, t2 ) : 3 } ;
p_demand(i_prod) := data { p1 : 3, p2 : 3, p4 : 3, p5 : 3 } ;
```

Then the code:

```
! Find the products that are on stock or can be produced:
FindUsedElements(
    SearchSet         :  s_products,
    SearchIdentifiers :  s_supplyIds,
    UsedElements      :  s_productsAvailable);

! Find the products that are in demand:
FindUsedElements(
    SearchSet         :  s_products,
    SearchIdentifiers :  s_demandIds,
    UsedElements      :  s_productsRequired);

display s_productsAvailable, s_productsRequired ;
```

Will produce:

```
s_productsAvailable := data { p1, p2, p3 } ;

s_productsRequired := data { p1, p2, p4, p5 } ;
```

in the listing file.

And the code

```
! Check if we require products that are not available.
s_productsRequiredNotAvailable :=
    s_productsRequired - s_productsAvailable ;
if card( s_productsRequiredNotAvailable ) then
    raise error formatString(
        "The products %s are required, but cannot be made available",
        setAsString( s_productsRequiredNotAvailable ) );
endif ;
```

will raise the error:

```
The products { p4, p5 } are required, but cannot be made available.
```

See also:

- Data control function

1.2.8 First

With the function *First* (page 55) you can retrieve the first element from a set.

```
First(
       Set,             ! (input) set reference
     )
```

Arguments

Set The set from which the first element is to be returned.

Return Value

The function First returns the first element of set *Set*.

Note: If there is no element in *Set*, the function returns the empty element '' instead.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
        Definition: ElementRange(0,9,prefix:"p");
}
Set s_topProducts {
        SubsetOf: s_products;
        Index: i_topProd;
        OrderBy: user;
}
Parameter p_rev {
        IndexDomain: i_prod;
}
ElementParameter ep_firstProd {
        Range: s_products;
}
ElementParameter ep_theTopProd {
        Range: s_products;
}
```

The code:

```
! a bit of data.
p_rev(i_prod) := data {
        p0 : 3,  p1 : 7,  p2 : 9,  p3 : 9,  p4 : 3,
        p5 : 4,  p6 : 8,  p7 : 6,  p8 : 8,  p9 : 7 } ;

! Sorting on revenue, note that s_topProducts has order by: user.
s_topProducts := sort( i_prod, -p_rev(i_prod) );
```

(continues on next page)

(continued from previous page)

```
! Getting the first of both sets.
ep_firstProd := first( s_products );
ep_theTopProd := first( s_topProducts );

! Printing to listing file.
display s_topProducts, ep_firstProd, ep_theTopProd ;
```

produces

```
s_topProducts := data { p2, p3, p6, p8, p1, p9, p7, p5, p0, p4 } ;

ep_firstProd := 'p0' ;

ep_theTopProd := 'p2' ;
```

in the listing file.

1.2.9 Last

With the function *Last* (page 57) you can retrieve the last element from a set.

```
Last(
        Set,              ! (input) set reference
    )
```

Arguments

Set The set from which the last element is to be returned.

Return Value

The function Last returns the last element of set *Set*.

Note: If there is no element in *Set*, the function returns the empty element '' instead.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
        Definition: ElementRange(0,9,prefix:"p");
}
Set s_topProducts {
        SubsetOf: s_products;
```

(continues on next page)

(continued from previous page)

```
        Index: i_topProd;
        OrderBy: user;
}
Parameter p_rev {
        IndexDomain: i_prod;
}
ElementParameter ep_lastProd {
        Range: s_products;
}
ElementParameter ep_theBottomProd {
        Range: s_products;
}
```

The code:

```
! a bit of data.
p_rev(i_prod) := data {
        p0 : 3,  p1 : 7,  p2 : 9,  p3 : 9,  p4 : 3,
        p5 : 4,  p6 : 8,  p7 : 6,  p8 : 8,  p9 : 7 } ;

! Sorting on revenue, note that s_topProducts has order by: user.
s_topProducts := sort( i_prod, -p_rev(i_prod) );

! Getting the first of both sets.
ep_lastProd := last( s_products );
ep_theBottomProd := last( s_topProducts );

! Printing to listing file.
display s_topProducts, ep_lastProd, ep_theBottomProd ;
```

produces

```
s_topProducts := data { p2, p3, p6, p8, p1, p9, p7, p5, p0, p4 } ;

ep_lastProd := 'p9' ;

ep_theBottomProd := 'p4' ;
```

in the listing file.

1.2.10 Ord

The function *Ord* (page 58) returns the ordinal number of a set element relative to a set.

```
Ord(
   index,        ! (input) element expression
   [set]         ! (optional) set reference
   )
```

Arguments

index An element expression for which you want to obtain the ordinal number.

set (optional) The set with respect to which you want the ordinal number to be taken. If omitted, *set* is assumed to be the range of the argument *index*.

Return Value

The function *Ord* (page 58) returns the ordinal number of *index* in set *set*.

Note: A compile time error occurs if the argument *set* is not present, and AIMMS is unable to determine the range of *index*.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
        Definition: ElementRange(0,9,prefix:"p");
}
Set s_topProducts {
        SubsetOf: s_products;
        Index: i_topProd;
        OrderBy: user;
}
Parameter p_rev {
        IndexDomain: i_prod;
}
ElementParameter ep_p5 {
        Range: s_products;
}
Parameter p_ordP5;
Parameter p_ordP5top;
```

The code:

```
! a bit of data.
p_rev(i_prod) := data {
        p0 : 3,  p1 : 7,  p2 : 9,  p3 : 9,  p4 : 3,
        p5 : 4,  p6 : 8,  p7 : 6,  p8 : 8,  p9 : 7 } ;

! Sorting on revenue, note that s_topProducts has order by: user.
s_topProducts := sort(i_prod, -p_rev(i_prod) );

! Getting the position in the set of element p5
ep_p5 := 'p5' ;
p_ordP5 := ord( ep_p5 ); ! The position of 'p5' in set s_products is 6.
p_ordP5top := ord( ep_p5, s_topProducts ); ! The position of 'p5' in set s_topProducts is␣
↪8.
```

Shows the use of adding a set to the function *Ord*.

1.2.11 RestoreInactiveElements

The procedure *RestoreInactiveElements* (page 60) finds and restores all elements that were previously removed from a particular set, but for which inactive data still exists in a given collection of indexed model identifiers.

```
RestoreInactiveElements(
     SearchSet,          ! (input/output) a set
     SearchIdentifiers,  ! (input) a subset of AllIdentifiers
     UsedElements        ! (output) a subset
     )
```

Arguments

SearchSet The set for which you want to find the inactive elements.

SearchIdentifiers A subset of *AllIdentifiers* (page 821), holding identifiers that are indexed over *SearchSet*.

UsedElements A subset of *SearchSet*. On return this subset will contain all the inactive elements that are currently used (i.e. have corresponding nondefault values) in the identifiers contained in *SearchIdentifiers*.

Note: The inactive elements found are placed in the *result-set*, but are also automatically added to the *search-set*.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
}
Parameter p_rev {
        IndexDomain: i_prod;
}
Set s_searchIds {
        SubsetOf: AllIdentifiers;
}
Set s_restoredProducts {
        SubsetOf: s_products;
}
```

The code:

```
! a bit of data.
s_products := ElementRange(0,9,prefix:"p");
p_rev(i_prod) := data {
        p1 : 7,  p2 : 9,  p3 : 9,
```

(continues on next page)

(continued from previous page)

```
        p6 : 8,  p7 : 6,  p8 : 8 } ;

! Remove some elements from root set.
s_products -= data { p3, p4, p8, p9 } ;

! Display resulting data
display s_products ; ! s_products := data { p0, p1, p2, p5, p6, p7 } ;
display p_rev ; ! p_rev := data { p1 : 7,  p2 : 9,  p6 : 8,  p7 : 6 } ;
! p_rev only has 4 elements left; some elements of p_rev seem to have disappeared;
! These elements are called the inactive elements of p_rev.

! Search for inactive products that have a revenue, and restore
! the corresponding elements to active.
s_searchIds := data { p_rev } ;
RestoreInactiveElements(
        SearchSet         :  s_products,
        SearchIdentifiers :  s_searchIds,
        UsedElements      :  s_restoredProducts);

display s_restoredProducts ; ! s_restoredProducts := data { p3, p8 } ;
display s_products ; ! s_products := data { p0, p1, p2, p3, p5, p6, p7, p8 } ;
! The previously inactive data of p_rev is now active again:
display p_rev ; ! p_rev := data { p1 : 7,  p2 : 9,  p3 : 9,  p6 : 8,  p7 : 6,  p8 : 8 } ;
```

See also:

- The function *ActiveCard* (page 45).
- The explanation of inactive elements in the Language Reference at Inactive data.
- The AIMMS Developer tool Identifier Cardinalities, see Observing identifier cardinalities.

1.2.12 SetAddRecursive

With the procedure *SetAddRecursive* (page 61) you can merge the elements of one set into another set.

```
SetAddRecursive(
     toSet,              ! (input/output) a set
     fromSet             ! (input) a set
     )
```

Arguments

toSet The set into which the elements of *fromSet* are merged.

fromSet The set that you want to merge in *toSet*.

Note:

- The sets *toSet* and *fromSet* should have the same root set.

- The difference between this function and a regular set assignment is that in case *fromSet* is not the domain of *toSet* all elements added to *toSet* will also be added to the domain set of *toSet*.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
}
Set s_allActiveProducts {
        SubsetOf: s_products;
}
Set s_fastMovingProducts {
        SubsetOf: s_allActiveProducts;
}
Set s_newFastMovingProducts {
        SubsetOf: s_products;
}
```

and a bit of initial data

```
! A bit of data
s_products := ElementRange(0,9,prefix:"p");
s_allActiveProducts := subrange( s_products, 'p2', 'p7' );
s_fastMovingProducts := subrange( s_allActiveProducts, 'p4', 'p7' );
s_newFastMovingProducts := subrange( s_products, 'p6', 'p9' );
```

the display statement

```
display s_products, s_allActiveProducts, s_fastMovingProducts, s_newFastMovingProducts ;
```

produces:

```
s_products := data { p0, p1, p2, p3, p4, p5, p6, p7, p8, p9 } ;
s_allActiveProducts := data { p2, p3, p4, p5, p6, p7 } ;
s_fastMovingProducts := data { p4, p5, p6, p7 } ;
s_newFastMovingProducts := data { p6, p7, p8, p9 } ;
```

After applying `SetAddRecursive`:

```
SetAddRecursive(
        toSet   :  s_fastMovingProducts,
        fromSet :  s_newFastMovingProducts);
```

The display statement

```
display s_products, s_allActiveProducts, s_fastMovingProducts, s_newFastMovingProducts ;
```

produces:

```
s_products := data { p0, p1, p2, p3, p4, p5, p6, p7, p8, p9 } ;
s_allActiveProducts := data { p2, p3, p4, p5, p6, p7, p8, p9 } ;
```

(continues on next page)

(continued from previous page)

```
s_fastMovingProducts := data { p4, p5, p6, p7, p8, p9 } ;
s_newFastMovingProducts := data { p6, p7, p8, p9 } ;
```

As you can see, the elements p8 and p9, are added both to s_fastMovingProducts and its domain set s_allActiveProducts.

1.2.13 SetAsString

With the function *SetAsString* (page 63) you can cast a set expression to a string.

```
SetAsString(
    set             ! (input) a set expression
    )
```

Arguments

set A set for which you want to get a string representation.

Return Value

The function returns the string representation of the given set, being elements of the set separated by commas and enclosed in curly brackets. If the set is empty then the strings is just curly brackets, like so '{}'.

Note: This function is available since AIMMS version 4.89.

Implicit cast from a set expression to a string is decommissioned since AIMMS version 4.89 and will result in a compile error in a future AIMMS version. *SetAsString* (page 63) function is intended to be used instead.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
}
Set s_productGroups {
        Index: i_pg;
        Definition: data { fast, slow };
}
Set s_productsByGroup {
        IndexDomain: i_pg;
        SubsetOf: s_products;
}
StringParameter sp_productGroup {
        IndexDomain: i_pg;
}
```

With the data:

```
! A bit of data
s_products := elementRange(1,4,prefix:"p");
s_productsByGroup(i_pg) := data {
        fast : { p3, p4 },
        slow : { p1, p2 }
} ;
```

The code:

```
sp_productGroup(i_pg) := SetAsSTring( s_productsByGroup(i_pg) );
block where single_column_display := 1;
        display sp_productGroup ;
endblock ;
```

produces

```
elementary::setop::funcSetAsString::sp_productGroup := data
{ fast : "{ p3, p4 }",
  slow : "{ p1, p2 }" } ;
```

in the listing file.

1.2.14 SetElementAdd

With the procedure *SetElementAdd* (page 64) you can add new elements to a one-dimensional set. When you apply *SetElementAdd* (page 64) to a root set, the element will be added to that root set. When you apply it to a subset, the element will be added to the subset as well as to all its supersets, up to and including its associated root set.

```
SetElementAdd(
     Setname,    ! (input/output) a reference to a simple set
     Elempar,    ! (output) an element parameter
     Newname     ! (input) a scalar string expression
     )
```

Arguments

Setname The root set or subset to which you want to add the element. This set cannot be an indexed set, nor a set that is an element in an indexed set.

Elempar An element parameter into *Setname*, that on return will point to the newly added element.

Newname A string holding the name of the element to be added.

Note: If the element already exists in the set, the procedure does not make any changes to the set, and on return the element parameter *Elempar* will point to the existing element.

Example

Given the declarations:

```
Set s_products;
Set s_fastMovingProducts {
    SubsetOf: s_products;
}
ElementParameter ep_newProd {
    Range: s_products;
}
```

The code

```
SetElementAdd(
    Setname :  s_fastMovingProducts,
    Elempar :  ep_newProd,
    Newname :  "p0");

display s_products, s_fastMovingProducts, ep_newProd ;
```

will produce in the listing file:

```
s_products := data { p0 } ;
s_fastMovingProducts := data { p0 } ;
ep_newProd := 'p0' ;
```

As you can see, the element `p0` is added to both `s_fastMovingProducts` and `s_products`.

See also:

- The function *ElementCast* (page 51) and the procedures *SetElementRename* (page 65) and *StringToElement* (page 66).
- The lexical conventions for set elements in Lexical Conventions.

1.2.15 SetElementRename

With the procedure *SetElementRename* (page 65) you can rename an element in a set.

```
SetElementRename(
     Setname,         ! (input) a set
     Element,         ! (input) an element parameter
     Newname          ! (input) a scalar string expression
     )
```

Arguments

Setname The root set or subset in which you want to rename an element.

Element The element that you want to rename.

Newname A string holding the new name of the element.

Note:

- If the new name for the element already exists in the set, the procedure will generate an execution error.
- AIMMS will refuse to rename a set element, if an explicit reference to such an element exists in the model source.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
}
Parameter p_rev {
        IndexDomain: i_prod;
}
ElementParameter ep_prod {
        Range: s_products;
}
```

And a bit of data:

```
setElementAdd( s_products, ep_prod, "p0" );
p_rev( ep_prod ) := 1 ;
display p_rev ; ! p_rev := data { p0 : 1 } ;
```

The statement

```
SetElementRename(
        Setname :  s_products,
        Element :  ep_prod,
        Newname :  "p1");
```

renames `p0` to `p1`, which can be seen when displaying p_rev again.

```
display p_rev  ; ! p_rev := data { p1 : 1 } ;
```

See also:

- The procedure *SetElementAdd* (page 64), and the function *StringToElement* (page 66).
- The lexical conventions for set elements in Lexical Conventions.

1.2.16 StringToElement

With the function *StringToElement* (page 66) you can convert a string into an (existing) element of a set.

```
StringToElement(
     Set,              ! (input/output) a reference to a simple set
     Name,             ! (input) a scalar string
     [create]          ! (optional) 0 or 1, default 0
     )
```

Arguments

Set A set in which you want to find (or create) a specific element.

Name A scalar string expression, representing the string that you want to convert.

create (optional) An indicator whether or not a nonexisting element are added to the set during the call.

Return Value

The function returns the existing element or, if the string cannot be converted to an existing element and the argument *create* is not set to 1, the function return the empty element. If *create* is set to 1, nonexisting elements will be created on the fly.

Example

Given the declarations:

```
Set s_products {
        Index: i_prod;
        Parameter: ep_p0, ep_p1, ep_p2;
}
```

The statements

```
ep_p0 := StringToElement( s_products, "p0", create: 1);
ep_p1 := StringToElement( s_products, "p1", create: 1);
ep_p2 := StringToElement( s_products, "p2", create: 0);

display s_products, ep_p0, ep_p1, ep_p2 ;
```

will produce the following in the listing file:

```
s_products := data { p0, p1 } ;
ep_p0 := 'p0' ;
ep_p1 := 'p1' ;
ep_p2 := '' ;
```

As you can see, an element for `p2` was not created as the corresponding argument `create` was set to 0.

See also:

- The function *ElementCast* (page 51) and the procedure *SetElementAdd* (page 64).
- The lexical conventions for set elements in Lexical Conventions.

1.2.17 SubRange

The function *SubRange* (page 67) extracts a subrange of consecutive elements from an existing set.

```
SubRange(
     Superset,      ! (input) a simple set
     First,         ! (input) an element
     Last           ! (input) an element
     )
```

Arguments

Superset The set containing the subrange of elements that you want to extract.

First An element in *Superset* representing the first element of the subrange.

Last An element in *Superset* representing the last element of the subrange.

Return Value

The function returns a set containing the subrange of elements extracted from *Superset*. If the element *First* is positioned after *Last*, then the empty set is returned.

Example

Given the declarations:

```
Set s_products;
Set s_youngProducts {
        SubsetOf: s_products;
}
```

The statements

```
sp_products := SetAsString( s_products );
sp_youngProducts := SetAsString( s_youngProducts );

display sp_products, sp_youngProducts ;
```

will produce the following in the listing file:

```
s_products := data { p0, p1, p2, p3, p4 } ;
s_youngProducts := data { p2, p3, p4 } ;
```

1.3 String Manipulation Functions

AIMMS supports the following functions for manipulating strings:

1.3.1 Character

The function *Character* (page 69) returns the string consisting of a single character whose ordinal number is the value of the argument.

```
Character(
    n              ! (input) a numeric expression
    )
```

Arguments

n A numeric expression in the range $\{0..55295\} \cup \{57344..\}$.

Return Value

The function Character returns a string of length 1. Exception: when the value 0 is passed it returns the empty string.

Example

The code:

```
sp_pct := character( 37 );
display sp_pct ;
```

will produce the following in the listing file:

```
sp_pct := "%" ;
```

References

- The function *CharacterNumber* (page 69).
- Wikipedia

1.3.2 CharacterNumber

The function *CharacterNumber* (page 69) returns the character number of the first character in a string. It returns 0 for the empty string.

```
CharacterNumber(
     text            ! (input) a scalar string expression
     )
```

Arguments

text The string for which you want to have the value of the first character.

Return Value

The function CharacterNumber returns a value in the range { 0 .. }.

Example

The code:

```
p_charNoPct := characterNumber( "%" );
display p_charNoPct  ;
```

will produce the following in the listing file:

```
p_charNoPct := 37 ;
```

References

- The function *Character* (page 69).
- Wikipedia

1.3.3 FindNthString

The function *FindNthString* (page 70) searches for the n-th occurrence of a substring (a key) within a search string.

```
FindNthString(
     SearchString,    ! (input) a scalar string expression
     Key,             ! (input) a scalar string expression
     Nth,             ! (input) an integer expression
     [CaseSensitive], ! (optional) binary
     [WordOnly],      ! (optional) binary
     [IgnoreWhite]    ! (optional) binary
     )
```

Arguments

SearchString The string in which you want to find the substring *Key*.

Key The substring to search for.

Nth The function will search for the *Nth* occurrence of the substring. If this number is negative, then the function will search backwards starting from the right.

CaseSensitive The search will be case sensitive when the value is 1. The default depends on the setting of the option `Case_sensitive_string_comparison`, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option `Case_sensitive_string_comparison` is 'On'.

WordOnly It is a word only search when this option is set to 1. The default is 0.

IgnoreWhite The search ignores whites if this option is set to 1. The default is 0.

Note: As with all string comparisons within AIMMS, the function *FindNthString* (page 70) is case sensitive by default. You can modify this behavior through the option `Case_Sensitive_String_Comparison`.

Return Value

The function returns the start position of the n-th occurrence of the substring starting from the left (or right). If the substring does not exist within the string, or does not occur *Nth* times then the function returns 0. When the argument `Nth` is 0, then this function will always return 0.

Example

Given the declarations:

```
StringParameter _sp_str;
StringParameter _sp_key1;
StringParameter _sp_key2;
Parameter _p_pos1;
Parameter _p_pos2;
```

And a bit of data:

```
_sp_str := "Nice weather today";
_sp_key1 := "r t";
_sp_key2 := "tomorrow";
```

The code:

```
_p_pos1 := FindNthString(
        SearchString  :  _sp_str,
        Key           :  _sp_key1,
        Nth           :  1,
        CaseSensitive :  1,
        WordOnly      :  0,
        IgnoreWhite   :  0);
_p_pos2 := FindNthString(
        SearchString  :  _sp_str,
```

(continues on next page)

(continued from previous page)

```
        Key           :  _sp_key2,
        Nth           :  1,
        CaseSensitive :  1,
        WordOnly      :  0,
        IgnoreWhite   :  0);
display _p_pos1, _p_pos2 ;
```

will produce the following in the listing file:

```
_p_pos1 := 12 ;
_p_pos2 := 0 ;
```

Indicating that the string "r t" was found, but the string "tomorrow" was not found.

See also:

The functions *FindString* (page 74), *StringOccurrences* (page 81), *RegexSearch* (page 77).

1.3.4 FindReplaceNthString

The function *FindReplaceNthString* (page 72) constructs a string by searching for the Nth occurrence of a substring (a key) within a search string and replacing this occurrence with another string. It returns the constructed string.

```
FindReplaceNthString(
     SearchString,    ! (input) a scalar string expression
     Key,             ! (input) a scalar string expression
     Replacement,     ! (input) a scalar string expression
     Nth,             ! (input) an integer expression
     [CaseSensitive], ! (optional) binary
     [WordOnly]       ! (optional) binary
     )
```

Arguments

SearchString The string in which you want to find the substring *key*.

Key The substring to search for.

Replacement The string used to replace *Key*.

Nth The function will search for the *Nth* occurrence of the substring. If this number is negative, then the function will search backwards starting from the right.

CaseSensitive The search will be case sensitive when the value is 1. The default depends on the setting of the option Case_sensitive_string_comparison, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option Case_sensitive_string_comparison is 'On'.

WordOnly It is a word only search when this option is set to 1. The default is 0.

Note: As with all string comparisons within AIMMS, the function *FindReplaceNthString* (page 72) is case sensitive by default. You can modify this behavior through the option Case_Sensitive_String_Comparison.

Return Value

The function returns the resulting string. If the *Nth* occurrence of *Key* is not found, the original string is returned.

Example

Given the declarations:

```
StringParameter sp_str;
StringParameter sp_key;
StringParameter sp_rep;
StringParameter sp_res;
```

And a bit of data:

```
sp_str := "Hello Hello";
sp_key := "Hello";
sp_rep := "mr";
```

The code:

```
sp_res := FindReplaceNthString(
        SearchString  :  sp_str,
        Key           :  sp_key,
        Replacement   :  sp_rep,
        Nth           :  2,
        CaseSensitive :  1,
        WordOnly      :  0);
display sp_res  ;
```

will produce the following in the listing file:

```
sp_res := "Hello mr" ;
```

Indicating that the second “Hello” is replaced by the string “mr” ;

See also:

The functions *FindNthString* (page 70), *StringOccurrences* (page 81) and *FindReplaceStrings* (page 73).

1.3.5 FindReplaceStrings

The function *FindReplaceStrings* (page 73) constructs a string by searching for every occurrence of a substring (a key) within a search string and replaces it with another string. It returns the constructed string.

```
FindReplaceStrings(
     SearchString,    ! (input) a scalar string expression
     Key,             ! (input) a scalar string expression
     Replacement,     ! (input) a scalar string expression
     [CaseSensitive], ! (optional) binary
     [WordOnly]       ! (optional) binary
     )
```

Arguments

SearchString The string in which you want to find the substring *key*.

Key The substring to search for.

Replacement The string used to replace *Key*.

CaseSensitive The search will be case sensitive when the value is 1. The default depends on the setting of the option `Case_sensitive_string_comparison`, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option `Case_sensitive_string_comparison` is 'On'.

WordOnly It is a word only search when this option is set to 1. The default is 0.

Note: As with all string comparisons within AIMMS, the function *FindReplaceStrings* (page 73) is case sensitive by default. You can modify this behavior through the option `Case_Sensitive_String_Comparison`.

Return Value

The function returns the resulting string. If *Key* is not found, the original string is returned.

Example

Given the declarations:

```
StringParameter sp_str;
StringParameter sp_key;
StringParameter sp_rep;
StringParameter sp_res;
```

And a bit of data:

```
sp_str := "Hello Hello";
sp_key := "Hello";
sp_rep := "'Allo";
```

The code:

```
sp_res := FindReplaceStrings(
        SearchString  :  sp_str,
        Key           :  sp_key,
        Replacement   :  sp_rep,
        CaseSensitive :  1,
        WordOnly      :  0);
display sp_res  ;
```

will produce the following in the listing file:

```
sp_res := "'Allo 'Allo" ;
```

Indicating that both "Hello" words are replaced by the string "'Allo" ;

See also:

The functions *FindString* (page 74), *StringOccurrences* (page 81) and *FindReplaceNthString* (page 72).

1.3.6 FindString

The function *FindString* (page 74) searches for the occurrence of a substring (a key) within a search string.

```
FindString(
     SearchString,     ! (input) a scalar string expression
     Key,              ! (input) a scalar string expression
     [CaseSensitive], ! (optional) binary
     [WordOnly],       ! (optional) binary
     [IgnoreWhite]     ! (optional) binary
     )
```

Arguments

SearchString The string in which you want to find the substring *key*.

Key The substring to search for.

CaseSensitive The search will be case sensitive when the value is 1. The default depends on the setting of the option `Case_sensitive_string_comparison`, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option `Case_sensitive_string_comparison` is 'On'.

WordOnly It is a word only search when this option is set to 1. The default is 0.

IgnoreWhite The search ignores whites if this option is set to 1. The default is 0.

Note: As with all string comparisons within AIMMS, the function *FindString* (page 74) is case sensitive by default. You can modify this behavior through the option `Case_Sensitive_String_Comparison`.

Return Value

The function returns the start position of the first occurrence of the substring. If the substring does not exist, then the function returns 0.

Example

Given the declarations:

```
StringParameter _sp_str;
StringParameter _sp_key1;
StringParameter _sp_key2;
Parameter _p_pos1;
Parameter _p_pos2;
```

And a bit of data:

```
_sp_str := "Nice weather today";
_sp_key1 := "r t";
_sp_key2 := "tomorrow";
```

The code:

```
_p_pos1 := FindString(
        SearchString  :  _sp_str,
        Key           :  _sp_key1,
        CaseSensitive :  1,
        WordOnly      :  0,
        IgnoreWhite   :  0);
_p_pos2 := FindString(
        SearchString  :  _sp_str,
        Key           :  _sp_key2,
        CaseSensitive :  1,
        WordOnly      :  0,
        IgnoreWhite   :  0);
display _p_pos1, _p_pos2 ;
```

will produce the following in the listing file:

```
_p_pos1 := 12 ;
_p_pos2 := 0 ;
```

Indicating that the string "r t" was found, but the string "tomorrow" was not found.

See also:

The functions *FindNthString* (page 70), *RegexSearch* (page 77).

1.3.7 FormatString

With the *FormatString* (page 76) function you can compose a string that is built up from combinations of numbers, strings and set elements. The *FormatString* (page 76) function accepts a varying number of arguments, defined by the conversion specifiers in the format string.

```
FormatString(
     formatstring,    ! (input) a literal double quoted string
     arguments,       ! (input) a list of numbers, strings, and set elements
     ...
     )
```

Arguments

formatstring A format string that specifies how the returned string is composed. The string should contain the proper conversion specifier for each following argument.

arguments,... One or more arguments of type number, string or element. The order of these arguments must coincide with the order of the conversion specifiers in *formatstring*.

Return Value

The function returns the formatted string.

Example

The code:

```
p_million := 1e6 ;
sp_millions := formatString("%i %6.3n %.0g", p_million, p_million, p_million );
display sp_millions  ; ! sp_millions := "1000000 1.000e+06 1e+06" ;
```

Is a first illustration of various ways of formatting a number.

See also:

For a detailed description of the conversion specifiers in AIMMS see Formatting Strings of the Language Reference.

1.3.8 GarbageCollectStrings

The procedure *GarbageCollectStrings* (page 77) removes any unused strings in the internal data structures of AIMMS. If you do not call this procedure explicitly, AIMMS performs an automatic garbage collect at certain places during execution. For example as part of the Empty statement when recently a lot of string valued expressions have been executed.

```
GarbageCollectStrings()
```

Note: Use this procedure only when you notice that AIMMS uses a lot of memory that might be related to having many strings in the model. It is a rather expensive procedure in terms of execution time, because it needs to enumerate all the individual entries of all string parameters in the model. After runnig it you might see a drop in the memory that is in use by AIMMS, but be aware that because of the internal memory model of AIMMS, some memory is not given back to the operating system directly, but has only been marked for re-use in subsequent memory requests.

1.3.9 RegexSearch

The function *RegexSearch* (page 77) tells if there is a substring in the search string that matches the regex pattern.

```
RegexSearch(
     SearchString,    ! (input) a scalar string expression
     Pattern,         ! (input) a scalar string expression
     [CaseSensitive]  ! (optional) binary
     )
```

Arguments

SearchString The string in which you want to find a substring matching the regex *pattern*.

Pattern The regular expressions pattern to match. Multilines are not supported.

CaseSensitive (optional) The search will be case sensitive when the value is 1. The default depends on the setting of the option `Case_sensitive_string_comparison`, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option `Case_sensitive_string_comparison` is 'On'.

Note:

- The used regular expressions grammar follows the implementation of the modified ECMAScript regular expression grammar in the C++ Standard Library. It follows ECMA-262 grammar and POSIX grammar, with some modifications.
- To include a special character in a string, it should be escaped by the backslash character \ (for more information on special characters see also Formatting Strings of the Language Reference). In regular expressions special characters also have to be escaped in order to be included in a pattern. So, for example, in order to match a backslash character the pattern should contain four backslashes (see the example below).

Return Value

The function returns 1 if a substring that matches the regex pattern exists in the search string. When the pattern is an empty string, the function returns 1. In all other cases, the function returns 0.

Example

The following example checks if the path contains the specified folder name on disk C. With the following declarations (and initial data):

```
Parameter P;
StringParameter path {
    InitialData: "C:\\ProgramFiles\\Folder\\SubFolder";
}
StringParameter regexPattern {
    InitialData: "c:.*\\\\ProgramFiles(\\\\|$)";
}
```

the statement

```
P := regexsearch(path, regexPattern, 0);
```

results in P being 1.

The used regular expression pattern specifies that the path starts with `c:`, followed by zero or more characters (regular expression: `.*`), followed by `\ProgramFiles` (regular expression: `\\\\ProgramFiles`), and ends with a backslash or the end of a line (regular expression: `(\\\\|$)`).

See also:

The functions *FindString* (page 74), *FindNthString* (page 70).

1.3.10 RegexReplace

The function *RegexReplace* (page 78) finds all matches with the given regular expression and replaces them with the specified replacement. The modified string is returned.

```
RegexReplace(
     SearchString,     ! (input) a scalar string expression
     Pattern,          ! (input) a scalar string expression
     Replacement,      ! (input) a scalar string expression
     [CaseSensitive]   ! (optional) binary
     )
```

Arguments

SearchString The string in which you want to find a substring matching the regex *pattern*.

Pattern The regular expressions pattern to match. Multilines are not supported.

Replacement The pattern to use as replacement of each found match.

CaseSensitive (optional) The search will be case sensitive when the value is 1. The default depends on the setting of the option `Case_sensitive_string_comparison`, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option `Case_sensitive_string_comparison` is 'On'.

Note:

- The used regular expressions grammar follows the implementation of the modified ECMAScript regular expression grammar in the C++ Standard Library. It follows ECMA-262 grammar and POSIX grammar, with some modifications.
- To include a special character in a string, it should be escaped by the backslash character \ (for more information on special characters see also Formatting Strings. In regular expressions, special characters also have to be escaped in order to be included in a pattern. For example, in order to match a backslash character the pattern should contain four backslashes (see the example below).

Return Value

The function returns a new string where all replacements are applied. If the pattern was not found in the input string the returned strings is the same as the input string.

Example

The following example will replace all vowels with a *

```
str := regexreplace("The quick brown fox", "a|e|i|o|u", "*" 0);
```

results in `str` being "Q**ck br*wn f*x".

And in the following example all vowels will be replaced by the same vowel in between brackets:

```
str := regexreplace("The quick brown fox", "a|e|i|o|u", "[$&]" 0);
```

results in `str` being "Q[u][i]ck br[o]wn f[o]x".

See also:

The functions *RegexSearch* (page 77), *FindReplaceStrings* (page 73).

1.3.11 StringCapitalize

The function *StringCapitalize* (page 80) converts the first character of a string to upper case, and all other characters to lower case.

```
StringCapitalize(
     text            ! (input) a scalar string expression
     )
```

Arguments

text The string that you want to capitalize.

Return Value

The function returns the capitalized string.

Example

The code:

```
sp_str := "hello HELLO";
sp_res := StringCapitalize( sp_str );
display sp_res ; ! sp_res := "Hello hello" ;
```

See also:

The functions *StringToLower* (page 82), *StringToUpper* (page 82).

1.3.12 StringLength

The function *StringLength* (page 80) returns the number of characters in a string.

```
StringLength(
     text            ! (input) a scalar string expression
     )
```

Arguments

text The string for which you want to retrieve the length.

Return Value

The function returns the number of characters in the string.

Example

```
sp_str := "hello HELLO";
p_res := StringLength( sp_str );
display p_res ; ! p_res := 11 ;
```

1.3.13 StringOccurrences

The function *StringOccurrences* (page 81) counts the number of occurrences of a particular substring in a string.

```
StringOccurrences(
     SearchString,     ! (input) a string expression
     Key,              ! (input) a string expression
     [CaseSensitive], ! (optional) binary
     [WordOnly],       ! (optional) binary
     [IgnoreWhite]     ! (optional) binary
     )
```

Arguments

SearchString A string in which you want to find the substring(s).

Key The substring.

CaseSensitive The search will be case sensitive when the value is 1. The default depends on the setting of the option `Case_sensitive_string_comparison`, and is 1 if this option is 'On' and 0 if this option is 'Off'. The default of the option `Case_sensitive_string_comparison` is 'On'.

WordOnly It is a word only search when this option is set to 1. The default is 0.

IgnoreWhite The search ignores whites if this option is set to 1. The default is 0.

Return Value

The function returns how many occurrences of the substring *Key* exist in the string *SearchString*.

Example

The code:

```
sp_str := "Hello Hello";
sp_key := "Hello";
p_res := StringOccurrences(
        SearchString  :  sp_str,
        Key           :  sp_key,
        CaseSensitive :  1,
        WordOnly      :  0,
        IgnoreWhite   :  0);
display p_res ; ! p_res := 2 ;
```

See also:

The functions *FindString* (page 74), *FindNthString* (page 70).

1.3.14 StringToLower

The function *StringToLower* (page 82) converts all characters of a string to lower case.

```
StringToLower(
     text            ! (input) a scalar string expression
     )
```

Arguments

text The string that you want to convert to lower case characters.

Return Value

The function returns the lower case string.

Example

The code:

```
sp_str := "hello HELLO";
sp_res := StringToLower( sp_str );
display sp_res ; ! sp_res := "hello hello" ;
```

See also:

The functions *StringToUpper* (page 82), *StringCapitalize* (page 80).

1.3.15 StringToUpper

The function *StringToUpper* (page 82) converts all characters of a string to upper case.

```
StringToUpper(
     text            ! (input) a scalar string expression
     )
```

Arguments

text The string that you want to convert to upper case characters.

Return Value

The function returns the upper case string.

Example

The code:

```
sp_str := "hello HELLO";
sp_res := StringToUpper( sp_str );
display sp_res ; ! sp_res := "HELLO HELLO" ;
```

See also:

The functions *StringToLower* (page 82), *StringCapitalize* (page 80).

1.3.16 SubString

The function *SubString* (page 83) retrieves a substring from a specific string, based on the start and end position of this substring within this string.

```
SubString(
     str,       ! (input) a scalar string expression
     from,      ! (input) an integer value
     to         ! (input) an integer value
     )
```

Arguments

str The string from which you want to retrieve the substring.

from The start position of the substring within *str*.

to The end position of the substring within *str*.

Return Value

The function returns the requested substring.

Example

The code:

```
sp_str := "hello HELLO";
sp_res := subString( sp_str, 4, 5 );
display sp_res ; ! sp_res := "lo" ;
```

Note: If the arguments *from* and *to* are positive, then the position is calculated from the start of the string (i.e. the first character is on position 1). If the arguments *from* and *to* are negative, then the position is calculated from the end of the string (i.e. the last character is on position −1). *from* must be less than or equal to *to*, and if either of the values exceeds the length of the string, they are automatically set within the proper range.

1.4 Unit Functions

AIMMS supports the following functions for unit related functions:

1.4.1 AtomicUnit

With the function *AtomicUnit* (page 84) you can retrieve the atomic unit expression corresponding to the unit expression passed as the argument to the function.

```
AtomicUnit(
     unit                ! (input) scalar unit expression
     )
```

Arguments

unit A unit expression of which the associated atomic unit expression must be computed

Return Value

The function returns the atomic unit expression corresponding to *unit*.

Note: The atomic unit expression associated with a given unit is the unit expression solely in terms of atomic unit symbols by which the given unit differs a constant scale factor only.

Example

Given the unit of measurement declarations:

```
Quantity SI_Time_Duration {
        BaseUnit: s;
        Conversions: {
                day->s : #-># * 86400,
                hour->s : #-># * 3600,
                minute->s : #-># * 60
        }
        Comment: "Expresses the value for the duration of periods.""}
Quantity SI_Velocity {
        BaseUnit: m/s;
        Comment: "Expresses the value for the change in distance per time unit.""}
UnitParameter up_baseVelocity {
        Definition: unit(m/s);
}
Quantity SI_Length {
        BaseUnit: m;
        Conversions: km->m : #-># * 1000;
        Comment: "Expresses the value of a distance.""}
UnitParameter _up_atomic;
```

The code:

```
_up_atomic := AtomicUnit( km/hour );
display _up_atomic ; ! _up_atomic := [m/s] ;
```

Returns the atomic units for speed.

And with the procedure:

```
Procedure pr_isCommensurate {
        Arguments: (up_one,up_two);
        Body: {
                _bp_isCommensurate := AtomicUnit( up_one ) = AtomicUnit( up_two );

                return _bp_isCommensurate ;
        }
        Comment: "Returns 1 iff units up_one and up_two measure the same quantity.""
↪    UnitParameter up_one {
                Property: Input;
        }
        UnitParameter up_two {
                Property: Input;
        }
        Parameter _bp_isCommensurate {
                Range: binary;
        }
}
```

we can verify whether [km/hour] is a velocity by:

```
_bp_isVelocity := pr_isCommensurate( unit(km/hour), up_baseVelocity );
```

See also:

Unit expressions are discussed in full detail in Units of Measurement of the Language Reference.

1.4.2 ConvertUnit

With the function *ConvertUnit* (page 86) you can compute the associated unit value of a unit expression with respect to a given convention.

```
ConvertUnit(
     unit,                ! (input) scalar unit expression
     convention           ! (input) element expression
     )
```

Arguments

unit A unit expression of which the associated unit value in the given convention must be computed

convention An element expression in to *AllConventions* (page 817), representing the convention with respect to which a unit value must be computed.

Return Value

The function returns the associated unit value of *unit* with respect to *convention*.

Example

Given the unit of measurement declarations:

```
Quantity SI_Time_Duration {
        BaseUnit: s;
        Conversions: {
                day->s : #-># * 86400,
                hour->s : #-># * 3600,
                minute->s : #-># * 60
        }
        Comment: "Expresses the value for the duration of periods.""}
Quantity SI_Velocity {
        BaseUnit: m/s;
        Comment: "Expresses the value for the change in distance per time unit.""}
Quantity SI_Length {
        BaseUnit: m;
        Conversions: {
                km->m : #-># * 1000,
                mile->m : #-># * 1609.344
        }
        Comment: "Expresses the value of a distance.""}
```

(continues on next page)

(continued from previous page)

```
Convention cnv_metric {
        PerQuantity: {
                SI_Length        : m,
                SI_Time_Duration : s,
                SI_Velocity      : m/s
        }
}
Convention cnv_imperial {
        PerQuantity: SI_Length : mile;
}
```

The code:

```
_up_cnv := ConvertUnit( unit(km/hour), cnv_imperial );
display _up_cnv ; ! _up_cnv := [mile/hour] ;
```

Returns the speed according to the imperial convention.

See also:

Unit expressions and conventions are discussed in full detail in Units of Measurement of the Language Reference.

1.4.3 EvaluateUnit

With the function *EvaluateUnit* (page 87) you can compute the numerical value (with associated unit) of a given unit expression.

```
EvaluateUnit(
     unit                ! (input) scalar unit expression
     )
```

Arguments

unit A unit expression of which the numerical value (with associated unit) must be computed

Return Value

The function returns the numerical value (with associated unit), corresponding to one unit of *unit*.

Example

Given the unit of measurement declarations:

```
Quantity SI_Time_Duration {
        BaseUnit: s;
        Conversions: {
                day->s : #-># * 86400,
                hour->s : #-># * 3600,
                minute->s : #-># * 60
```

(continues on next page)

(continued from previous page)

```
        }
        Comment: "Expresses the value for the duration of periods."";
Quantity SI_Velocity {
        BaseUnit: m/s;
        Comment: "Expresses the value for the change in distance per time unit."";
Quantity SI_Length {
        BaseUnit: m;
        Conversions: {
                km->m : #-># * 1000,
                mile->m : #-># * 1609.344
        }
        Comment: "Expresses the value of a distance."";
Parameter _p_uv {
        Unit: m/s;
}
```

The code:

```
_p_uv := EvaluateUnit( unit(km/hour) );
display _p_uv ; ! 0.278 ; ;
```

Returns the conversion factor in [km/hour].

Note: The function *EvaluateUnit* (page 87) is an extension of AIMMS' local unit override capabilities which allows computed unit expressions.

See also:

Unit expressions are discussed in full detail in Units of Measurement of the Language Reference.

1.4.4 StringToUnit

With the function *StringToUnit* (page 88) you can compute a unit value corresponding to a given string expression.

```
StringToUnit(
     str          ! (input) scalar string expression
     )
```

Arguments

str A string expression of which the associated unit value must be computed

Return Value

The function returns the associated unit value of *str*, or fails if the given string does not correspond to a string constant.

Example

Given the unit of measurement declarations:

```
Quantity SI_Time_Duration {
        BaseUnit: s;
        Conversions: {
                day->s : #-># * 86400,
                hour->s : #-># * 3600,
                minute->s : #-># * 60
        }
        Comment: "Expresses the value for the duration of periods.""}
Quantity SI_Velocity {
        BaseUnit: m/s;
        Comment: "Expresses the value for the change in distance per time unit.""}
Quantity SI_Length {
        BaseUnit: m;
        Conversions: {
                km->m : #-># * 1000,
                mile->m : #-># * 1609.344
        }
        Comment: "Expresses the value of a distance.""}
```

The code:

```
_sp_speed := "km/hour" ;
_up_speed := StringToUnit( _sp_speed );
display _up_speed ; ! _up_speed := [km/hour] ;
```

Shows the coversion of string "km/hour" to unit [km/hour].

See also:

Unit expressions discussed in full detail in Units of Measurement of the Language Reference.

1.4.5 Unit

The function *Unit* (page 89) returns the unit value of a given unit constant.

```
Unit(
     unit          ! (input) scalar unit constant
     )
```

Arguments

unit A unit constant of which the associated unit value must be computed

Return Value

The function returns the unit value of a unit constant *unit*.

Example

Given the unit of measurement declarations:

```
Quantity SI_Time_Duration {
        BaseUnit: s;
        Conversions: {
                day->s : #-># * 86400,
                hour->s : #-># * 3600,
                minute->s : #-># * 60
        }
        Comment: "Expresses the value for the duration of periods.""}
Quantity SI_Velocity {
        BaseUnit: m/s;
        Comment: "Expresses the value for the change in distance per time unit.""}
Quantity SI_Length {
        BaseUnit: m;
        Conversions: {
                km->m : #-># * 1000,
                mile->m : #-># * 1609.344
        }
        Comment: "Expresses the value of a distance.""}
```

The code:

```
_up_speed := Unit( km/hour );
display _up_speed ; ! _up_speed := [km/hour] ;
```

illustrates how to assign a literal unit to a unit parameter.

Note: The function *Unit* (page 89) simply returns its argument. It exists to allow the use of numeric constants in computed unit expressions.

See also:

Unit expressions discussed in full detail in Units of Measurement of the Language Reference.

1.5 Time Functions

AIMMS supports the following time-related functions:

1.5.1 Aggregate

With the procedure *Aggregate* (page 91) you can aggregate time-dependent data from a calendar time scale (time slots) to a horizon time scale (periods).

```
Aggregate(
     TimeslotData,        ! (input) an indexed identifier over a calendar
     PeriodData,          ! (output) an indexed identifier over a horizon
     TimeTable,           ! (input) an AIMMS time table
     Type,                ! (input) an element in the set AggregationTypes
     [Locus]              ! (optional) a value between 0 and 1
     )
```

Arguments

TimeslotData An identifier (slice) containing the data to be aggregated. The domain sets in the index domain of this identifier should at least contain a calendar set, and all other sets should coincide with the domain of *PeriodData*.

PeriodData An identifier (slice) that on return will contain the aggregated data. The domain sets in the index domain of this identifier should at least contain a horizon set, and all other sets should coincide with the domain of *TimeslotData*.

TimeTable An indexed set in a calendar and defined over a horizon. This horizon and calendar should match with the index domains of *TimeslotData* and *PeriodData*.

Type An element of the pre-defined set *AggregationTypes* (page 792) (`summation`, `average`, `maximum`, `minimum`, or `interpolation`).

Locus (only for* `interpolation` *type) A number between 0 and 1, that indicates at which moment in a period the quantity is to be measured.

Example

Given the timetable declarations in the example of CreateTimeTable and the following declarations:

```
Parameter p_datCal {
        IndexDomain: i_day;
        Definition: {
                data
                { 2024-01-01 : 3,
                  2024-01-02 : 4,
                  2024-01-03 : 5,
                  2024-01-04 : 3,
                  2024-01-05 : 4,
                  2024-01-06 : 5,
```

(continues on next page)

(continued from previous page)

```
                    2024-01-07 : 3,
                    2024-01-08 : 4,
                    2024-01-09 : 5,
                    2024-01-10 : 3,
                    2024-01-11 : 4,
                    2024-01-12 : 5,
                    2024-01-13 : 3,
                    2024-01-14 : 4 }
        }
}
Parameter p_datHor {
        IndexDomain: i_hor;
}
```

The code

```
Aggregate(
        TimeSlotData :  p_datCal,
        PeriodData   :  p_datHor,
        TimeTable    :  s_timetable,
        Type         :  'summation' );
```

Produces:

```
p_datHor := data {
        p0 :  3,
        p1 :  9,
        p2 : 12,
        p3 : 15,
        p4 : 16 } ;
```

See also:

The procedure *DisAggregate* (page 101). Time-dependent aggregation and disaggregation is discussed in full detail in Data Conversion of Time-Dependent Identifiers of the Language Reference.

1.5.2 ConvertReferenceDate

The function *ConvertReferenceDate* (page 92) converts a reference date from one timezone to the other.

```
ConvertReferenceDate(
     ReferenceDate,          ! (input) a string expression
     FromTimezone,           ! (input) an element expression
     ToTimezone,             ! (input) an element expression
     IgnoreDST               ! (optional) a numerical expression (default 0)
     )
```

Arguments

ReferenceDate A string that holds a reference date in *FromTimezone*.

FromTimezone An element of `AllTimeZones` (page 853) with respect to which *ReferenceDate* is expressed.

ToTimezone An element of `AllTimeZones` (page 853) with respect to which the resulting reference date must be expressed.

IgnoreDST A numerical expression indicating whether daylight saving time must be ignored in the conversion.

Return Value

The result of `ConvertReferenceDate` (page 92) is a reference date in *ToTimezone* corresponding to the reference date *ReferenceDate* in *FromTimezone*.

Example

Given a selection of timezones

```
! Picking a few timezones relative to me.
_ep_CETTZ    := 'Central European Standard Time' ; ! Where I live.
_ep_BrazilTZ := 'Central Brazilian Standard Time' ;
_ep_IndianTZ := 'India Standard Time' ;
```

Wondering what time it is in the other timezones when I wake up:

```
_sp_wakeup := "2020-04-01 06:00" ; ! My alarm rings at 6 AM
_sp_refUTC := ConvertReferenceDate(
        ReferenceDate :  _sp_wakeup,
        FromTZ        :  _ep_CETTZ,
        ToTZ          :  'UTC',
        IgnoreDST     :  1);
_sp_refBrazil := ConvertReferenceDate(
        ReferenceDate :  _sp_wakeup,
        FromTZ        :  _ep_CETTZ,
        ToTZ          :  _ep_BrazilTZ,
        IgnoreDST     :  1);
_sp_refInda := ConvertReferenceDate(
        ReferenceDate :  _sp_wakeup,
        FromTZ        :  _ep_CETTZ,
        ToTZ          :  _ep_IndianTZ,
        IgnoreDST     :  1);
display _sp_refUTC, _sp_refBrazil, _sp_refInda ;
```

Without considering daylight saving time, the result is:

```
_sp_refUTC    := "2020-04-01 05:00" ;
_sp_refBrazil := "2020-04-01 01:00" ;
_sp_refInda   := "2020-04-01 10:30" ;
```

See also:

AIMMS support for time zones is discussed in full detail in Support for Time Zones and Daylight Saving Time and Working in Multiple Time Zones of the Language Reference.

1.5.3 CreateTimeTable

With the procedure *CreateTimeTable* (page 94) you can create a timetable in AIMMS.

```
CreateTimeTable(
     Timetable,              ! (output) an indexed set
     CurrentTimeslot,        ! (input) an element in a calendar
     CurrentPeriod,          ! (input) an element in a horizon
     PeriodLength,           ! (input) one-dimensional integer parameter
     LengthDominates,        ! (input) one-dimensional binary parameter
     InactiveTimeSlots,      ! (input) a subset of a calendar
     DelimiterSlots          ! (input) a subset of a calendar
     )
```

Arguments

Timetable An indexed set in a calendar and defined over the horizon to be linked to the calendar. This argument implicitly sets the calendar and horizon used for the creation of the timetable. The other arguments of the procedure should match with this calendar and horizon.

CurrentTimeslot An element of a calendar (a time slot) that should be aligned with the *CurrentPeriod* in the horizon.

CurrentPeriod An element of a horizon (a period) that should be aligned with the *timeslot* in the calendar.

PeriodLength A one-dimensional integer parameter, specifying the desired length of each period in the horizon in terms of the number of time slots to be contained in it.

LengthDominates A one-dimensional binary parameter, indicating whether reaching the specified *PeriodLength* dominates over the presence of any delimiter slot for every period in the horizon.

InactiveTimeSlots A subset of the calendar, indicating the time slots that must be excluded from the timetable.

DelimiterSlots A subset of the calendar, indicating the time slots that will (usually) result in starting a new period in the horizon.

Example

Given the declarations:

```
Calendar cal_days {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2024-01-14";
        TimeslotFormat: "%c%y-%m-%d";
}
```

(continues on next page)

(continued from previous page)

```
Set s_horizon {
        Index: i_hor;
        Definition: ElementRange(0,4,prefix:"p");
}
ElementParameter ep_currentTimeSlot {
        Range: cal_days;
}
ElementParameter ep_currentPeriod {
        Range: s_horizon;
}
Parameter p_periodLength {
        IndexDomain: i_hor;
        Definition: data { p0 : 1,  p1 : 2,  p2 : 3,  p3 : 4,  p4 : 4 };
}
Parameter bp_lengthDominates {
        IndexDomain: i_hor;
}
Set s_inactiveTimeSlots {
        SubsetOf: cal_days;
}
Set s_delimiterSlots {
        SubsetOf: cal_days;
}
Set s_timetable {
        IndexDomain: i_hor;
        SubsetOf: cal_days;
}
```

The code:

```
ep_currentTimeSlot := element( cal_days, 2 );
ep_currentPeriod := element( s_horizon, 2 );
CreateTimeTable(
        TimeTable         :  s_timetable,         ! Timetable constructed
        CurrentTimeSlot   :  ep_currentTimeSlot,  ! Timeslot 2024-01-02 is linked to
        CurrentPeriod     :  ep_currentPeriod,    ! start of period p1.
        PeriodLength      :  p_periodLength,      ! Lengths are 1, 2, 3, 4, and 4 days
        LengthDominates   :  bp_lengthDominates,  ! No length dominates
        InactiveTimeSlots :  s_inactiveTimeSlots, ! No inactive periods
        DelimiterSlots    :  s_delimiterSlots);   ! And no delimiter time slots.
display s_timetable ;
```

creates the timetable:

```
s_timetable := data
{ p0 : { 2024-01-01 } ,
  p1 : { 2024-01-02, 2024-01-03 } ,
  p2 : { 2024-01-04, 2024-01-05, 2024-01-06 } ,
  p3 : { 2024-01-07, 2024-01-08, 2024-01-09, 2024-01-10 } ,
  p4 : { 2024-01-11, 2024-01-12, 2024-01-13, 2024-01-14 } } ;
```

See also:

The procedures *Aggregate* (page 91), *DisAggregate* (page 101). For a more detailed description of the creation of

timetables, see Creating Timetables of the Language Reference.

1.5.4 CurrentToMoment

The function *CurrentToMoment* (page 96) converts the current time to the elapsed time with respect to a specific reference date.

```
CurrentToMoment(
     Unit,                    ! (input) a time unit
     ReferenceDate            ! (input) a string expression
     )
```

Arguments

Unit The time unit that is used to return the elapsed time.

ReferenceDate A string that holds the begin date using the fixed format for date and time, see paragraph Reference date format of the Language Reference.

Return Value

The result of *CurrentToMoment* (page 96) is the elapsed time in *Unit* since *ReferenceDate*.

Example

Given the declarations:

```
Parameter _p_secs {
        Unit: s;
}
Parameter _p_mins {
        Unit: minute;
}
```

The code:

```
_p_secs := CurrentToMoment(
        Unit          :  [s],
        Referencedate :  "2020-01-01 00:00:00");
_p_mins := CurrentToMoment(
        Unit          :  [minute],
        Referencedate :  "2020-01-01 00:00:00");
display _p_secs, _p_mins ;
```

Resulted in:

```
_p_secs := 142691236 [s] ;
_p_mins := 2378187.267 [minute] ;
```

When this example was created.

See also:

- The function *StringToMoment* (page 106).
- Measure Execution Time illustrates the use of some time functions. The purpose of *CurrentToMoment* (page 96) in that post is to compute the time since a starting point.

1.5.5 CurrentToString

The function *CurrentToString* (page 97) creates a string representation of the current time in the a specified format.

```
CurrentToString(
     Format           ! (input) a string expression
     )
```

Arguments

Format A string that holds the date and time format used in the returned string. Valid format strings are described in Format of Time Slots and Periods

Return Value

The result of *CurrentToString* (page 97) is a description of the current time according to *Format*.

Example

The code:

```
_sp_today := CurrentToString("%d-%Am|AllAbbrMonths|-%c%y");
_sp_nowAsReferenceDate :=  CurrentToString("%c%y-%m-%d %H:%M:%S");

display _sp_today, _sp_nowAsReferenceDate ;
```

Resulted in:

```
_sp_today := "09-Jul-2024" ;
_sp_nowAsReferenceDate := "2024-07-09 14:25:50" ;
```

When this example was created.

Note: There is an option `Current_Time_in_LocalDST` that specifies whether this function takes into account the effects of daylight savings time.

See also:

- The functions *MomentToString* (page 102), *CurrentToMoment* (page 96).
- Measure Execution Time illustrates the use of some time functions. The purpose of *CurrentToString* (page 97) in that post is to mark the starting point.

1.5.6 CurrentToTimeSlot

The function *CurrentToTimeSlot* (page 97) determines the time slot in a calendar that corresponds with the current time.

```
CurrentToTimeSlot(
     Calendar           ! (input) a calendar
     )
```

Arguments

Calendar An identifier of type calendar.

Return Value

The function *CurrentToTimeSlot* (page 97) returns the time slot in the calendar that contains the current moment.

Note: There is an option `Current_Time_in_LocalDST` that specifies whether this function takes into account the effects of daylight savings time.

Example

Given the following declarations:

```
Calendar cal_DaysLongGone {
        Index: i_dayGone;
        Unit: day;
        BeginDate: "2000-01-01";
        EndDate: "2000-12-31";
        TimeslotFormat: "%c%y-%m-%d";
}
Calendar cal_ForALongTime {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2064-12-31";
        TimeslotFormat: "%c%y-%m-%d";
}
```

The code:

```
ep_past := CurrentToTimeSlot( cal_DaysLongGone );
ep_curr := CurrentToTimeSlot( cal_ForALongTime );
display ep_past, ep_curr ;
```

Results in:

```
ep_past := '' ;
ep_curr := '2024-07-15' ;
```

when this example was created.

See also:

The functions *StringToTimeSlot* (page 108), *MomentToTimeSlot* (page 103).

1.5.7 DaylightSavingEndDate

The function *DaylightSavingEndDate* (page 99) computes the end date of daylight saving time for a particular year in a particular time zone.

```
DaylightSavingEndDate(
     Year,                    ! (input) an element expression
     Timezone                 ! (input) an element expression
     )
```

Arguments

Year An element of a yearly calendar for the end date of daylight saving time must be computed.

Timezone An element in the predefined set *AllTimeZones* (page 853).

Return Value

The result of *DaylightSavingEndDate* (page 99) is the end date of daylight saving time, as a reference date, for the time zone *Timezone* in the year *Year*.

Example

Given the declarations:

```
Calendar cal_years {
    Index: i_year;
    Unit: year;
    BeginDate: "2024";
    EndDate: "2024";
    TimeslotFormat: "%c%y";
}
ElementParameter ep_year {
    Range: cal_years;
}
```

Then the code:

```
ep_year := first( cal_years );
_sp_startDaylightSaving2024 := DaylightSavingStartDate( ep_year,
    'Central European Standard Time' );
display _sp_startDaylightSaving2024 ;
```

results in:

```
_sp_startDaylightSaving2024 := "2024-03-31 02" ;
```

See also:

AIMMS support for time zones is discussed in full detail in Support for Time Zones and Daylight Saving Time and Working in Multiple Time Zones of the Language Reference.

1.5.8 DaylightSavingStartDate

The function *DaylightSavingStartDate* (page 100) computes the start date of daylight saving time for a particular year in a particular time zone.

```
DaylightSavingStartDate(
    Year,                    ! (input) an element expression
    Timezone                 ! (input) an element expression
    )
```

Arguments

Year An element of a yearly calendar for the end date of daylight saving time must be computed.

Timezone An element in the predefined set *AllTimeZones* (page 853).

Return Value

The result of *DaylightSavingStartDate* (page 100) is the start date of daylight saving time, as a reference date, for the time zone *Timezone* in the year *Year*.

Example

Given the declarations:

```
Calendar cal_years {
    Index: i_year;
    Unit: year;
    BeginDate: "2024";
    EndDate: "2024";
    TimeslotFormat: "%c%y";
}
ElementParameter ep_year {
    Range: cal_years;
}
```

Then the code:

```
ep_year := first( cal_years );
_sp_startDaylightSaving2024 := DaylightSavingStartDate( ep_year,
    'Central European Standard Time' );
display _sp_startDaylightSaving2024 ;
```

results in:

```
_sp_startDaylightSaving2024 := "2024-03-31 02" ;
```

See also:

AIMMS support for time zones is discussed in full detail in Support for Time Zones and Daylight Saving Time and Working in Multiple Time Zones of the Language Reference.

1.5.9 DisAggregate

With the procedure *DisAggregate* (page 101) you can disaggregate time-dependent data from a horizon time scale (periods) to a calendar time scale (time slots).

```
DisAggregate(
     PeriodData,          ! (input) an indexed identifier over a horizon
     TimeslotData,        ! (output) an indexed identifier over a calendar
     Timetable,           ! (input) an AIMMS time table
     Type,                ! (input) an element in the set AggregationTypes
     [Locus]              ! (optional) a value between 0 and 1
     )
```

Arguments

PeriodData An identifier (slice) containing the data to be disaggregated. The domain sets in the index domain of this identifier should at least contain a horizon set, and all other sets should coincide with the domain of *TimeslotData*.

TimeslotData An identifier (slice) that on returns will contain the disaggregated data. The domain sets in the index domain of this identifier should at least contain a calendar set, and all other sets should coincide with the domain of *PeriodData*.

Timetable An indexed set in a calendar and defined over a horizon. This horizon and calendar should match with the index domains of `TimeslotData` and `PeriodData`.

Type An element of the pre-defined set *AggregationTypes* (page 792) (`summation`, `average`, `maximum`, `minimum`, or `interpolation`).

Locus (only for ``interpolation`` type) A number between 0 and 1, that indicates at which moment in a period the quantity is to be measured.

Example

Given the timetable declarations in the example of *CreateTimeTable* (page 94) and the following declarations:

```
Parameter p_datCal {
        IndexDomain: i_day;
}
Parameter p_datHor {
        IndexDomain: i_hor;
        Definition: {
                data
```

(continues on next page)

(continued from previous page)

```
                { p0 : 20,
                  p1 : 30,
                  p2 : 40,
                  p3 : 50,
                  p4 : 60
                 }
        }
}
```

The code

```
DisAggregate(
        PeriodData   :  p_datHor,
        TimeSlotData :  p_datCal,
        TimeTable    :  s_timetable,
        Type         :  'summation' );
```

Produces:

```
p_datCal(i_day) := data
{ 2024-01-01 : 20.000,
  2024-01-02 : 15.000,
  2024-01-03 : 15.000,
  2024-01-04 : 13.333,
  2024-01-05 : 13.333,
  2024-01-06 : 13.333,
  2024-01-07 : 12.500,
  2024-01-08 : 12.500,
  2024-01-09 : 12.500,
  2024-01-10 : 12.500,
  2024-01-11 : 15.000,
  2024-01-12 : 15.000,
  2024-01-13 : 15.000,
  2024-01-14 : 15.000 } ;
```

See also:

The procedure *Aggregate* (page 91). Time-dependent aggregation and disaggregation is discussed in full detail in Data Conversion of Time-Dependent Identifiers of the Language Reference.

1.5.10 MomentToString

The function *MomentToString* (page 102) creates a string representation of a moment, that is calculated from a given amount of elapsed time since a specific reference date.

```
MomentToString(
     Format,                 ! (input) a string expression
     unit,                   ! (input) a time unit
     ReferenceDate,          ! (input) a string expression
     Elapsed                 ! (input) a numerical expression
     )
```

Arguments

Format A string that holds the date and time format used in the returned string. Valid format strings are described in Format of Time Slots and Periods. Note that the format uses the local timezone by default. Thus the UTC timezone should be specified if the intent is to use the result as a reference date.

unit The time unit that is used in the argument *Elapsed*.

ReferenceDate A string that holds the begin date using the fixed format for date and time, see paragraph Reference date format of the Language Reference.

Elapsed A numerical value of the time elapsed since *ReferenceDate*.

Return Value

The result of *MomentToString* (page 102) is a string describing the corresponding moment according to *Format*.

Example

The code

```
sp_nextYear := MomentToString(
    Format        :  "%c%y-%m-%d",
    unit          :  day,
    ReferenceDate :  "2021-01-01",
    Elapsed       :  365[day]);
display sp_nextYear ;
```

gives the result:

```
sp_nextYear := "2022-01-01" ;
```

References

- The function *StringToMoment* (page 106).

1.5.11 MomentToTimeSlot

The function *MomentToTimeSlot* (page 103) determines the time slot in a calendar that corresponds with the a moment that is specified as the elapsed time since a specific reference date.

```
MomentToTimeSlot(
     Calendar,           ! (input) a calendar
     ReferenceDate,      ! (input) an element (time-slot) in the calendar
     Elapsed             ! (input) a numerical value
     )
```

Arguments

Calendar An identifier of type calendar.

ReferenceDate A specific time-slot in *Calendar* holding the reference time.

Elapsed The elapsed time since *ReferenceDate*. This should be an integral multiple of the calendar's time unit in order to select the time slot that is the return value of this function.

Return Value

The function *MomentToTimeSlot* (page 103) returns the time slot in the calendar that contains the given moment. When the time slot is outside the calendar the empty element is returned.

Example

The code

```
ep_nextYear := MomentToTimeSlot(
        Calendar      :  cal_days,
        ReferenceDate :  '2021-01-01',
        Elapsed       :  365[day]);
display ep_nextYear ;
```

gives the result:

```
ep_nextYear := '2022-01-01' ;
```

See also:

The functions *TimeSlotToMoment* (page 111), *CurrentToTimeSlot* (page 97), *StringToTimeSlot* (page 108).

1.5.12 PeriodToString

With the function *PeriodToString* (page 104) you can obtain a description of a period in a timetable that consists of multiple calendar slots.

```
PeriodToString(
     Format,              ! (input) a string expression
     Timetable,           ! (input) an AIMMS time table
     Period               ! (input) an element in a horizon
     )
```

Arguments

Format A string that holds the date and time format used in the returned string. This format string can contain period specific conversion specifiers to generate a description referring to both the beginning and end of the period, see Format of Time Slots and Periods

Timetable An indexed set in a calendar and defined over a horizon.

Period An element in the horizon that is defined by *Timetable*.

Return Value

The result of *PeriodToString* (page 104) is a string describing the corresponding moment according to *Format*.

Example

Given the declarations:

```
Calendar cal_days {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2024-01-14";
        TimeslotFormat: "%c%y-%m-%d";
}
Horizon s_horizon {
        Index: i_hor;
        CurrentPeriod: ep_currentPeriod;
        Definition: ElementRange(0,4,prefix:"p");
}
ElementParameter ep_currentTimeSlot {
        Range: cal_days;
}
ElementParameter ep_currentPeriod {
        Range: s_horizon;
}
Parameter p_periodLength {
        IndexDomain: i_hor;
        Definition: data { p0 : 1,  p1 : 2,  p2 : 3,  p3 : 4,  p4 : 4 };
}
Parameter bp_lengthDominates {
        IndexDomain: i_hor;
}
Set s_inactiveTimeSlots {
        SubsetOf: cal_days;
}
Set s_delimiterSlots {
        SubsetOf: cal_days;
}
Set s_timetable {
        IndexDomain: i_hor;
        SubsetOf: cal_days;
```

(continues on next page)

(continued from previous page)

```
}
StringParameter sp_periodName {
        IndexDomain: i_hor;
}
```

And the data preparation:

```
ep_currentTimeSlot := element( cal_days, 2 );
ep_currentPeriod := element( s_horizon, 2 );
CreateTimeTable(
        TimeTable         :  s_timetable,
        CurrentTimeSlot   :  ep_currentTimeSlot,
        CurrentPeriod     :  ep_currentPeriod,
        PeriodLength      :  p_periodLength,
        LengthDominates   :  bp_lengthDominates,
        InactiveTimeSlots :  s_inactiveTimeSlots,
        DelimiterSlots    :  s_delimiterSlots);
```

The code

```
sp_periodName( i_hor in s_horizon ) :=
        PeriodToString(
                Format    :  "%B%Aw|AllAbbrWeekdays| %Am|AllAbbrMonths| %d, %c%y - %I
↪%Aw|AllAbbrWeekdays| %Am|AllAbbrMonths| %d, %c%y",
                Timetable :  s_timetable,
                Period    :  i_hor);
block where single_column_display := 1;
        display s_timetable ;
        display sp_periodName ;
endblock ;
```

Prints the following in the listing file:

```
s_timetable := data
{ p0 : { 2024-01-01 } ,
  p1 : { 2024-01-02, 2024-01-03 } ,
  p2 : { 2024-01-04, 2024-01-05, 2024-01-06 } ,
  p3 : { 2024-01-07, 2024-01-08, 2024-01-09, 2024-01-10 } ,
  p4 : { 2024-01-11, 2024-01-12, 2024-01-13, 2024-01-14 } } ;


sp_periodName := data
{ p0 : "Mon Jan 01, 2024 - Mon Jan 01, 2024",
  p1 : "Tue Jan 02, 2024 - Wed Jan 03, 2024",
  p2 : "Thu Jan 04, 2024 - Sat Jan 06, 2024",
  p3 : "Sun Jan 07, 2024 - Wed Jan 10, 2024",
  p4 : "Thu Jan 11, 2024 - Sun Jan 14, 2024" } ;
```

See also:

The procedure *CreateTimeTable* (page 94).

1.5.13 StringToMoment

The function *StringToMoment* (page 106) converts a given time string (in a free time format) to the elapsed time with a respect to a specific reference date.

```
StringToMoment(
     Format,              ! (input) a string expression
     Unit,                ! (input) a time unit
     ReferenceDate,       ! (input) a string expression
     Timeslot             ! (input) a string expression
     )
```

Arguments

Format A string that holds the date and time format used in the fourth argument *Timeslot*. Valid format strings are described in Format of Time Slots and Periods

Unit The time unit that is used to return the elapsed time.

ReferenceDate A string that holds the begin date using the fixed format for date and time, see paragraph Reference date format of the Language Reference.

Note: Depending on the option `use UTC forCaseAndStartEndDate`, reference dates are with respect to

- value `on` : timezone `'UTC'` or
- value `off`: timezone `'local'`.

Timeslot A string representing a specific date and time moment using the format specified in the first argument *Format*.

Return Value

The result of *StringToMoment* (page 106) is the elapsed time in *unit* between *reference-date* and *date*.

Example

The code:

```
block where use_UTC_forCaseAndStartEndDate := 'on' ;
    _p_noSec1 := StringToMoment(
        Format        :  "%c%y-%m-%d %H:%M:%S%TZ('UTC')",
        Unit          :  [s],
        ReferenceDate :  "2020-01-01 09:30:30",
        Timeslot      :  "2020-01-01 09:30:58");
    display _p_noSec1 ;
endblock ;
block where use_UTC_forCaseAndStartEndDate := 'off' ;
    _p_noSec2 := StringToMoment(
        Format        :  "%c%y-%m-%d %H:%M:%S%TZ('local')",
        Unit          :  [s],
```

(continues on next page)

(continued from previous page)

```
        ReferenceDate :  "2020-01-01 09:30:30",
        Timeslot      :  "2020-01-01 09:30:58");
    display _p_noSec2 ;
endblock ;
```

Will result in:

```
_p_noSec1 := 28 [s] ;

_p_noSec2 := 28 [s] ;
```

See also:

The functions *MomentToString* (page 102), *CurrentToMoment* (page 96).

1.5.14 StringToTimeSlot

The function *StringToTimeSlot* (page 108) determines the time slot in a calendar that corresponds with the a moment that is specified using a free format string.

```
StringToTimeSlot(
     Format,          ! (input) a string expression
     Calendar,        ! (input) a calendar
     MomentString     ! (input) a string expression
     )
```

Arguments

Format A string that holds the date and time format used in the third argument *MomentString*. Valid format strings are described in Format of Time Slots and Periods

Calendar An identifier of type calendar.

MomentString A string expression of the moment (using the format given in `Format`) that should be matched with the time slots in the calendar.

Return Value

The function *StringToTimeSlot* (page 108) returns the time slot in the calendar that contains the given moment.

Example

Given the declarations:

```
Calendar cal_days {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2024-01-14";
        TimeslotFormat: "%c%y-%m-%d";
}
ElementParameter _ep_day {
        Range: cal_days;
}
ElementParameter _ep_tooLate {
        Range: cal_days;
}
```

The code

```
_ep_day := StringToTimeSlot(
                Format       :  "%m-%d",
                Calendar     :  cal_days,
                MomentString :  "01-03");
_ep_tooLate := StringToTimeSlot(
                Format       :  "%m-%d",
                Calendar     :  cal_days,
                MomentString :  "01-20");
display _ep_day, _ep_tooLate ;
```

results in:

```
_ep_day := '2024-01-03' ;
_ep_tooLate := '' ;
```

See also:

The functions *CurrentToTimeSlot* (page 97), *MomentToTimeSlot* (page 103).

1.5.15 TestDate

The function *TestDate* (page 109) tests whether or not a particular date is according to given format.

```
TestDate(
     Format,         ! (input) a string expression
     Date,           ! (input) a string expression
     requireUnique   ! (optional) default 1.
     )
```

Arguments

Format A string that holds the date and time format used in the returned string. Valid format strings are described in Format of Time Slots and Periods

Date It is tested whether or not this string is according to format *Format*.

requireUnique When 1, it requires the year number to be present in the date.

Return Value

The result of *TestDate* (page 109) is 1 if *Date* is according to format *Format* and an existing data, and 0 otherwise. If the result is 0, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Example

```
_bp_ok1 := TestDate( "%c%y-%m-%d", "2015-xx-xx" ); ! ok becomes 0; Not numeric.
_bp_ok2 := TestDate( "%c%y-%m-%d", "2015-02-29" ); ! ok becomes 0; Feb 2015 has only 28␣
↪days.
_bp_ok3 := TestDate( "%c%y-%m-%d", "2016-02-29" ); ! ok becomes 1; Feb 29, 2016 exists.
_bp_ok4 := TestDate( "%c%y-%m-%d", "2015-04-31" ); ! ok becomes 0; April 31 does not␣
↪exist.
_bp_ok5 := TestDate( "%c%y-%m-%d", "2015-04-01" ); ! ok becomes 1; April 01 does exist (-
↪;
_bp_ok6 := TestDate( "%m-%d", "03-03", requireUnique:1 ); ! Not unique, ok becomes 0.
_bp_ok7 := TestDate( "%m-%d", "03-03", requireUnique:0 ); ! Uniqueness not required; ok␣
↪becomes 1.
_bp_ok8 := TestDate( "%Aw|AllAbbrWeekdays| %Am|AllAbbrMonths| %d, %c%y", "Mon Jul 22,␣
↪2024" ); ! ok becomes 1; Jul 24, 2024 is a Monday.
_bp_ok9 := TestDate( "%Aw|AllAbbrWeekdays| %Am|AllAbbrMonths| %d, %c%y", "Tue Jul 22,␣
↪2024" ); ! ok becomes 0; Jul 24, 2024 is not a Tuesday.
```

See also:

The function *CurrentToString* (page 97).

1.5.16 TimeSlotCharacteristic

The function *TimeSlotCharacteristic* (page 110) obtains a numeric value which characterizes the time slot, in terms of its day of the week, its day in the year, etc.

```
TimeSlotCharacteristic(
     Timeslot,       ! (input) an element (time-slot) in a calendar
     Characteristic, ! (input) an element in TimeslotCharacteristics
     Timezone,       ! (optional) an element in AllTimeZones, default Local.
     IgnoreDST       ! (optional) 0-1 expression, default 0.
     )
```

Arguments

Timeslot A element refering to a time-slot in a calendar.

Characteristic An element in the predefined set *TimeSlotCharacteristics* (page 815), each element in this set refers to a specific value that can be retrieved for a time slot.

Timezone A time zone from the predefined set *AllTimeZones* (page 853).

IgnoreDST A 0-1 expression indicating whether or not to ignore daylight savings time.

Return Value

The function *TimeSlotCharacteristic* (page 110) returns a numerical value for the requested time slot characteristic.

Example

Given the declarations:

```
Calendar cal_days {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2024-01-14";
        TimeslotFormat: "%c%y-%m-%d";
}
ElementParameter _ep_firstDay {
        Range: cal_days;
}
Parameter _p_dayNoInWeek;
```

The code:

```
_ep_firstDay := first( cal_days );
_p_dayNoInWeek := TimeslotCharacteristic(
        Timeslot       :  _ep_firstDay,
        Characteristic :  'weekday' );
display _p_dayNoInWeek ;
```

Results in:

```
_p_dayNoInWeek := 1 ;
```

Indicating that the first day of the calendar is a Monday.

See also:

The function *TimeSlotCharacteristic* (page 110) is discussed in full detail in Creating Timetables of the Language Reference.

1.5.17 TimeSlotToMoment

The function *TimeSlotToMoment* (page 111) calculates the elapsed time since a specific reference date for a given time slot in a calendar.

```
TimeSlotToMoment(
     Calendar,         ! (input) a calendar
     ReferenceDate,    ! (input) an element (time-slot) in the calendar
     Timeslot          ! (input) an element (time-slot) in the calendar
     )
```

Arguments

Calendar An identifier of type calendar.

ReferenceDate A specific time-slot in *Calendar* holding the reference time.

Timeslot A specific time slot in the calendar.

Return Value

The function *TimeSlotToMoment* (page 111) returns the elapsed time since the reference date for the given time slot (measured in the calendar's unit).

Example

Given the declarations:

```
Calendar cal_days {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2024-01-14";
        TimeslotFormat: "%c%y-%m-%d";
}
Parameter _p_noDaysToSecond {
        Unit: day;
}
Parameter _p_noDays {
        Unit: day;
}
ElementParameter _ep_first {
        Range: cal_days;
}
ElementParameter _ep_second {
        Range: cal_days;
}
ElementParameter _ep_last {
        Range: cal_days;
}
```

The code:

```
_ep_first := first( cal_days );
_ep_second := element( cal_days, 2 );
_ep_last := last( cal_days );
_p_noDaysToSecond := TimeslotToMoment(
        Calendar      :  cal_days,
        ReferenceDate :  _ep_first,
        Timeslot      :  _ep_second);
_p_noDays := TimeslotToMoment(
        Calendar      :  cal_days,
        ReferenceDate :  _ep_first,
        Timeslot      :  _ep_last);
display _ep_first, _ep_last, _p_noDaysToSecond, _p_noDays ;
```

Results in:

```
_ep_first := '2024-01-01' ;
_ep_last := '2024-01-14' ;
_p_noDaysToSecond := 1 [day] ;
_p_noDays := 13 [day] ;
```

See also:

The functions *MomentToTimeSlot* (page 103), *CurrentToTimeSlot* (page 97), *StringToTimeSlot* (page 108).

1.5.18 TimeSlotToString

The function *TimeSlotToString* (page 113) creates a string representation of a specific time slot in a calendar.

```
TimeSlotToString(
     Format,          ! (input) a string expression
     Calendar,        ! (input) a calendar
     Timeslot         ! (input) an element (timeslot) in the calendar
     )
```

Arguments

Format A string that holds the date and time format used in the returned string. Valid format strings are described in Format of Time Slots and Periods

Calendar An identifier of type calendar.

Timeslot A specific time-slot in the calendar.

Return Value

The function *TimeSlotToString* (page 113) returns a string representation of the time slot.

Example

Given the declarations:

```
Calendar cal_days {
        Index: i_day;
        Unit: day;
        BeginDate: "2024-01-01";
        EndDate: "2024-01-14";
        TimeslotFormat: "%c%y-%m-%d";
}
```

The code:

```
_sp_day1 := TimeslotToString(
        Format   :  "%Aw|AllAbbrWeekDays| %Am|AllAbbrMonths| %d, %c%y",
        Calendar :  cal_days,
        Timeslot :  first(cal_days));
display _sp_day1 ;
```

Results in:

```
_sp_day1 := "Mon Jan 01, 2024" ;
```

See also:

The functions *MomentToString* (page 102), *CurrentToTimeSlot* (page 97), *StringToTimeSlot* (page 108).

1.5.19 TimeZoneOffSet

The function *TimeZoneOffSet* (page 114) computes, in minutes, the offset between two time zones.

```
TimeZoneOffSet(
     FromTZ,      ! (input) an element expression
     ToTZ         ! (input) an element expression
     [UseDST]     ! (optional) 0 or 1
     )
```

Arguments

FromTZ An element from the set *AllTimeZones* (page 853).

ToTZ An element from the set *AllTimeZones* (page 853).

UseDST (optional) A scalar expression specifying whether or not the current setting for daylight saving time (DST) in both time zones should be taken into account. The default is 0, indicating DST is not used.

Return Value

The result of *TimeZoneOffSet* (page 114) is the offset, in minutes, between *FromTZ* and *ToTZ*.

Note: The result of the function has an associated unit, namely minutes. If *FromTZ* is UTC, the offset of *ToTZ* is the usual offset with respect to UTC (or GMT).

Example

Given the declarations:

```
Parameter p_tzOffset {
        IndexDomain: IndexTimezones;
        Unit: minute;
        Definition: TimezoneOffset('UTC',IndexTimeZones,0);
}
Set s_orderedTimeZones {
        SubsetOf: AllTimeZones;
        Index: i_otz;
        OrderBy: -p_tzOffset(IndexTimezones);
        Definition: Alltimezones-data{local,localdst};
}
```

In these declarations/definitions, first the offset with respect to UTC is computed. Based on that offset, the timezones are ordered.

The code:

```
block where single_column_display := 1;
        display p_tzOffset(i_otz) ;
endblock ;
```

This display statement uses the ordering of timezones above, and the first few lines are:

```
( p_tzOffset(i_otz) ) [minute] := data
{ 'Line Islands Standard Time'      :  840,
  'Samoa Standard Time'             :  780,
  'Tonga Standard Time'             :  780,
  UTC+13                            :  780,
  'Chatham Islands Standard Time'   :  765,
  'Fiji Standard Time'              :  720,
  'Kamchatka Standard Time'         :  720,
```

(continues on next page)

(continued from previous page)

```
  'New Zealand Standard Time'       :  720,
...
```

See also:

AIMMS support for time zones is discussed in full detail in Support for Time Zones and Daylight Saving Time and Working in Multiple Time Zones of the Language Reference.

1.6 Financial Functions

Financial functions can be of great use in modeling financial optimization models. They perform common business calculations, such as determining

- the depreciation of an asset,
- the payments for a loan,
- the future value or net present value of an investment, and
- the values of bonds, coupons or other securities.

Having these functions available in AIMMS prevents you from having to implement such functionality into your models yourself. Common arguments for the financial functions include:

Values the value of an investment, security or cash flow at a specific time. For example, the amount paid periodically to an investment or loan.

Rates the interest rate or discount rate for an investment or security. For example, the desired internal return on investment could be 8 percent.

Dates the date of measurements, payments or other events. For example, the date of settlement of a security. AIMMS's financial functions always expect dates to be provided in the format `"ccyy-mm-dd"`.

Interval lengths (in time periods) the number of periods that has to be analyzed. For example, the useful life of an asset or the number of payments or periods of an investment

Type the time when payments are made during the period. For example, at the beginning of a month or the end of the month.

The financial functions supported by AIMMS can be divided into separate categories. Each of these categories will be shortly introduced (including the mathematical equations underlying the functions in a category) and each of the available functions will be described in full detail. The following categories can be distinguished:

1.6.1 General Conversions

Prices (such as security prices) are often provided as a fractional price, whereas the financial functions in AIMMS always expect decimal prices. AIMMS supports the following conversion functions between fractional and decimal prices:

- *PriceDecimal* (page 117)
- *PriceFractional* (page 118)

Annual interest rates can be given as a nominal rate (just the sum of interest rates over the number of compounding periods) or in the form of an effective rate (including the effects of interest over interest for all compounding periods). AIMMS supports the following interest rate conversion functions:

- *RateEffective* (page 119)

- *RateNominal* (page 120)

PriceDecimal

The function *PriceDecimal* (page 117) converts a price expressed as a fractional number to a price expressed as a decimal number depending on the input parameter *FractionBase*.

```
PriceDecimal(
    FractionalPrice,            ! (input) numerical expression
    FractionBase                ! (input) numerical expression
    )
```

Arguments

FractionalPrice The price expressed as a fractional number. *FractionalPrice* can be any real number.

FractionBase The base used as the denominator of the fraction. *FractionBase* must be a positive integer.

Return Value

The function *PriceDecimal* (page 117) returns the *FractionalPrice* expressed as a decimal number.

Equation

The conversion between decimal and fractional prices is based on the system of equations

$$\begin{cases} \lfloor p_f \rfloor = \lfloor p_d \rfloor & \text{(integer parts)} \\ p_f - \lfloor p_f \rfloor = \frac{b}{10^{\lceil \log b \rceil}} \left(p_d - \lfloor p_d \rfloor \right) & \text{(fractional parts)} \end{cases}$$

where p_d is the decimal price, p_f the fractional price and b the base.

Note:

- For bases which are a power of 10, the decimal and fractional prices coincide. In all other cases, the fractional price is smaller than the decimal price.
- The function *PriceDecimal* (page 117) is similar to the Excel function DOLLARDE .

Example

```
_p_r1 := PriceDecimal( 1.02, 16 );
_p_r2 := PriceDecimal( 1.1 , 32 );
```

results in:

```
_p_r1 := 1.125000 ;
_p_r2 := 1.312500 ;
```

References

- The function *PriceFractional* (page 118).
- Microsoft support DOLLARDE
- EXCELJET example DOLLARDE

PriceFractional

The function *PriceFractional* (page 118) converts a price expressed as a decimal number to a price expressed as a fractional number depending on the input parameter *FractionBase*.

```
PriceFractional(
    DecimalPrice,              ! (input) numerical expression
    FractionBase               ! (input) numerical expression
    )
```

Arguments

DecimalPrice The price expressed as a decimal number. *DecimalPrice* can be any real number.

FractionBase The base to be used as the denominator of the fraction. *FractionBase* must be a positive integer.

Return Value

The function *PriceFractional* (page 118) returns the *DecimalPrice* expressed as a fractional number.

Note:

- The system of equations on which the conversion between decimal and fractional prices is based, is explained for the function *PriceDecimal* (page 117) (the inverse of *PriceFractional* (page 118)).
- The function `FractionalDecimal` is similar to the Excel function `DOLLARFR`.

Example

```
_p_r1 := PriceFractional( 1.125000, 16 );
_p_r2 := PriceFractional( 1.312500, 32 );

block where listing_number_precision := 6 ;
        display _p_r1, _p_r2 ;
endblock ;
```

results in:

```
_p_r1 := 1.020000 ;
_p_r2 := 1.100000 ;
```

References

- The function *PriceDecimal* (page 117).
- Microsoft support DOLLARDE

RateEffective

The function *RateEffective* (page 119) returns the effective annual interest rate, expressed as a fraction, on the basis of a nominal interest rate plus the number of compounding periods per year.

```
RateEffective(
    NominalRate,              ! (input) numerical expression
    NumberPeriods             ! (input) numerical expression
    )
```

Arguments

NominalRate The nominal annual interest rate expressed as a fraction. *NominalRate* must be a nonnegative decimal number.

NumberPeriods The number of compounding periods per year. *NumberPeriods* must be a positive integer.

Return Value

The function *RateEffective* (page 119) returns the effective annual interest rate expressed as a fraction.

Equation

The conversion between nominal and effective rates is based on the equation

$$r_{eff} = \left(1 + \frac{r_{nom}}{n}\right)^n - 1$$

where r_{eff} is the effective annual rate, r_{nom} the nominal annual rate and n the number of compounding periods.

Note:

- This function can be used in an objective function or constraint, and the input parameter *NominalRate* can be used as a variable.
- The function *RateEffective* (page 119) is similar to the Excel function `EFFECT`.

Example

```
! Each month 1.5 % interest, corresponds to 18 % per year
! but if it accumulates, it becomes:
_p_r1 := RateEffective( 12 * 0.015 , 12 );
! 19.6 %

! Each year 4 % interest, corresponds to 40 % per decade
! but if it accumulates, it becomes:
_p_r2 := RateEffective( 10 * 0.04, 10 );
! 48 %

block where listing_number_precision := 6 ;
    display _p_r1, _p_r2 ;
endblock ;
```

results in:

```
_p_r1 := 0.195618 ;
_p_r2 := 0.480244 ;
```

References

- The function *RateNominal* (page 120).
- Microsoft Support EXCEL Effect

RateNominal

The function *RateNominal* (page 120) returns the nominal annual interest rate, expressed as a fraction, on the basis of an effective annual interest rate plus the number of compounding periods per year.

```
RateNominal(
    EffectiveRate,              ! (input) numerical expression
    NumberPeriods               ! (input) numerical expression
    )
```

Arguments

EffectiveRate The effective annual interest rate expressed as a fraction. *EffectiveRate* must be a nonnegative decimal number.

NumberPeriods The number of compounding periods per year. *NumberPeriods* must be a positive integer.

Return Value

The function *RateNominal* (page 120) returns the nominal annual interest rate expressed as a fraction.

Note:

- The equation on which the conversion between nominal and effective rates is based, is explained for the function *RateEffective* (page 119) (the inverse of *RateNominal* (page 120)).
- This function can be used in an objective function or constraint, and the input parameter *EffectiveRate* can be used as a variable.
- The function *RateNominal* (page 120) is similar to the Excel function NOMINAL.

Example

```
_p_r1 := RateNominal( 2, 12 );
_p_r2 := RateNominal( 5, 12 );

block where listing_number_precision := 6 ;
        display _p_r1, _p_r2 ;
endblock ;
```

results in:

```
_p_r1 := 1.150472 ;
_p_r2 := 1.932440 ;
```

References

- The function *RateEffective* (page 119).
- Microsoft Support EXCEL Nominal

1.6.2 Day Count Bases and Dates

Many financial functions require date arguments, and depend on differences between two dates, either as a number of days or as a fraction of a year. This chapter discusses the date format expected by AIMMS's financial functions and the different methods to compute date differences used from which you can choose in many functions.

Format of Date Arguments

All date arguments in AIMMS's financial functions should be provided in the fixed string date format "ccyy-mm-dd". So, 15 August, 2000 should be passed to a financial function as the string "2000-08-15". If you want to pass an element from a daily calendar as a date argument, you should convert it to the fixed string date format using the function *TimeSlotToString* (page 113)

Day Count Bases

The result of many financial functions depends on the way with which differences between two dates are dealt with. Such functions have a *day count basis* argument, which determines how the difference between two dates is calculated, either in days or as a fraction of a year. AIMMS supports 5 different day count basis methods, each of which is commonly used in the financial markets. Each of these methods is specified by a way to count days and a way to determine how many days are in a year.

Method 1 - NASD Method / 360 Days Calculating with day count basis method 1 means that a year is assumed to consist of 12 periods of 30 days. A year consists of 360 days. The difference between this method and method 5 is the way the last day of a month is handled.

Method 2 - Actual / Actual Calculating with day count basis method 2 means that both the number of days between two dates and the number of dates in a year are actual.

Method 3 - Actual / 360 Days Calculating with day count basis method 3 means that the number of days between two dates is actual and that the number of days in a year is 360. When using this method, you should note that the year fraction of two dates that are one year apart is larger than 1 (365/360) and that this may lead to unwanted results.

Method 4 - Actual / 365 Days Calculating with day count basis method 4 means that the number of days between two dates is actual and that the number of days in a year is 365.

Method 5 - European Method / 360 Days Calculating with day count basis method 5 means that a year is assumed to consist of 12 periods of 30 days. A year consists of 360 days. The difference between this method and method 1 is the way the last day of a month is handled.

When the day count basis argument is optional, AIMMS assumes the NASD method 1 by default.

Date Differences

AIMMS supports the following functions for computing differences between two dates:

- *DateDifferenceDays* (page 122)
- *DateDifferenceYearFraction* (page 124)

DateDifferenceDays

The function *DateDifferenceDays* (page 122) calculates the number of days between two dates based on the specified day count basis.

```
DateDifferenceDays(
    FirstDate,              ! (input) scalar string expression
    SecondDate,             ! (input) scalar string expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

FirstDate The first date must be in date format.

SecondDate The second date must be in date format, and later than *FirstDate*.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *DateDifferenceDays* (page 122) returns the number of days between the two dates.

Note: The function *DateDifferenceDays* (page 122) is similar to the Excel function `DAYS300`.

Example

The code

```
_p_r1 := DateDifferenceDays( "2024-02-01", "2024-03-01", 1 );
_p_r2 := DateDifferenceDays( "2024-02-01", "2024-03-01", 2 );
_p_r3 := StringToMoment(
    Format        :  "%c%y-%m-%d",
    Unit          :  [day],
    ReferenceDate :  "2024-02-01",
    Timeslot      :  "2024-03-01");

block where listing_number_precision := 6 ;
    display _p_r1, _p_r2, _p_r3  ;
endblock ;
```

results in:

```
_p_r1 := 30 ;
_p_r2 := 29 ;
_p_r3 := 29 [day] ;
```

Remarks:

1. The default method of `DateDifferenceDays` uses 30 days per month, so one month results in 30.
2. The more advanced method, method 2, uses the actual number of days per month.
3. *MomentToString* (page 102) can do the same, but also supports other time units including `second`. Note that the result of this function has a time unit.

References

- Day count basis *methods* (page 122).
- *MomentToString* (page 102)

DateDifferenceYearFraction

The function *DateDifferenceYearFraction* (page 124) calculates the year fraction between two dates based on the specified day count basis.

```
DateDifferenceYearFraction(
    FirstDate,            ! (input) scalar string expression
    SecondDate,           ! (input) scalar string expression
    [Basis]               ! (optional) numerical expression
    )
```

Arguments

FirstDate The first date must be in date format.

SecondDate The second date must be in date format, and later than *FirstDate*.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *DateDifferenceYearFraction* (page 124) returns the difference between *FirstDate* and *SecondDate* in fractions of a year.

Note: The function *DateDifferenceYearFraction* (page 124) is similar to the Excel function YEARFRAC.

Example

The following code illustrates the differences between the various methods supported.

```
! Method 1: each month is always 1/12 of a year.
_p_r1 := DateDifferenceYearFraction( "2024-02-01", "2024-03-01", 1 );
_p_r1e := 1 / 12 ;

! Method 2: use actual number of days per month and per year (here leap year).
_p_r2 := DateDifferenceYearFraction( "2024-02-01", "2024-03-01", 2 );
_p_r2e := 29 / 366 ;
_p_r2t := StringToMoment(
        Format        :  "%c%y-%m-%d",
        Unit          :  [day],
        ReferenceDate :  "2024-02-01",
```

(continues on next page)

(continued from previous page)

```
        Timeslot      :  "2024-03-01") /
       StringToMoment(
        Format        :  "%c%y-%m-%d",
        Unit          :  [day],
        ReferenceDate :  "2024-01-01",
        Timeslot      :  "2025-01-01") ;

! Method 3: use actual number of days per month, but always 360 days per year.
_p_r3 := DateDifferenceYearFraction( "2024-02-01", "2024-03-01", 3 );
_p_r3e := 29 / 360 ;

! Method 4: use actual number of days per month, but always 365 days per year.
_p_r4 := DateDifferenceYearFraction( "2024-02-01", "2024-03-01", 4 );
_p_r4e := 29 / 365 ;

! Method 5: the European variation of method 1.
_p_r5 := DateDifferenceYearFraction( "2024-02-01", "2024-03-01", 5 );
_p_r5e := 1 / 12 ;

block where listing_number_precision := 6 ;
    display { _p_r1, _p_r1e }, { _p_r2, _p_r2e, _p_r2t }, { _p_r3, _p_r3e }, { _p_r4, _p_
↪r4e }, { _p_r5, _p_r5e } ;
endblock ;
```

and the results illustrate that there are small but significant differences:

```
_p_r1  := 0.083333 ;
_p_r1e := 0.083333 ;

_p_r2  := 0.079235 ;
_p_r2e := 0.079235 ;
_p_r2t := 0.079235 ;

_p_r3  := 0.080556 ;
_p_r3e := 0.080556 ;

_p_r4  := 0.079452 ;
_p_r4e := 0.079452 ;

_p_r5  := 0.083333 ;
_p_r5e := 0.083333 ;
```

References

- Day count basis *methods* (page 122).

1.6.3 Depreciations

This chapter discusses the functions available in AIMMS for the depreciation of an asset. Depreciation can be performed in many ways, for example by a fixed amount in every period, or by depreciation amounts that decrease over time. An asset is characterized by its purchase (or initial) cost c and its salvage value s (the value at the end of the useful life of the asset).

Useful Life

The accounting periods for depreciating the asset have a length of one year, but do not necessarily have to start at January 1. The useful life of the asset is either given as a fixed amount of L years, or is computed dynamically on the basis of the characteristics of the depreciation. The first period is the period from the purchase date until the beginning of the next regular accounting period. If the purchase date does not coincide with the beginning of an accounting period, the depreciations take place in $L + 1$ accounting periods.

General Equations

The following system of equations are true for all types of depreciations supported by AIMMS, where d_i is the actual depreciation in period i, $\tilde{d}_i$ is the generic depreciation computed in a method-dependent manner, and v_i the value of the asset at the beginning of period i.

$$d_i = \max(0, \min(\tilde{d}_i, v_i - s))$$
$$v_i = c - \sum_{j=1}^{i-1} d_j$$

The equations express that generic method-dependent depreciation method will be adapted to yield the actual depreciation value to make sure that the value of an asset v_i can never drop below its salvage value s.

Method-Dependent Equations

For each depreciation method available in AIMMS, the equations used to compute the generic method-dependent depreciation amount $\tilde{d}_i$ will be listed in the description of the depreciation function. In most occasions these equations use the fraction f_{PN}, which expresses the year fraction from the purchase date until the beginning of the next regular accounting period. Its value depends on the selected *day-count basis* (page 122) method.

AIMMS supports the following linear depreciation by constant amounts functions:

- `DepreciationLinearLife` (page 127)
- `DepreciationLinearRate` (page 129)

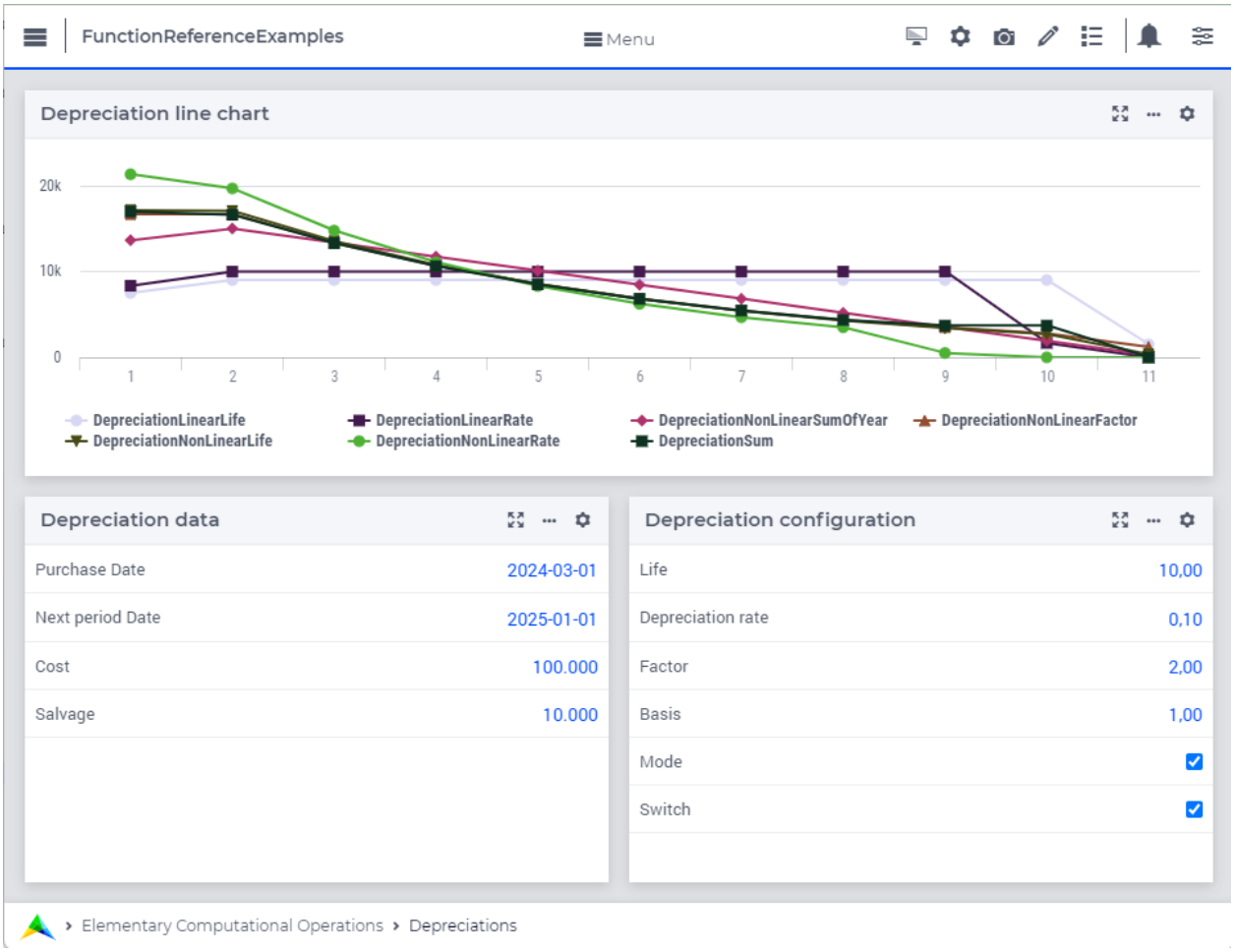
AIMMS supports the following non-linear depreciation by linear declining amounts functions:

- `DepreciationNonLinearSumOfYear` (page 131)

AIMMS supports the following non-linear depreciation by non-linear declining amounts functions:

- `DepreciationNonLinearLife` (page 135)
- `DepreciationNonLinearFactor` (page 133)

- *DepreciationNonLinearRate* (page 137)
- *DepreciationSum* (page 140)



DepreciationLinearLife

The function *DepreciationLinearLife* (page 127) returns the depreciation of an asset for the specified period, using straight-line depreciation. The accounting periods have a length of one year, but they don't necessary need to start January 1. The depreciation amounts are equal for every period. In case of partial periods, a relatively equal part must be depreciated.

```
DepreciationLinearLife(
    PurchaseDate,           ! (input) scalar string expression
    NextPeriodDate,         ! (input) scalar string expression
    Cost,                   ! (input) numerical expression
    Salvage,                ! (input) numerical expression
    Life,                   ! (input) numerical expression
    Period,                 ! (input) numerical expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Life The number of periods until the asset will be fully depreciated, also called the useful life of the asset. *Life* must be a positive integer.

Period The period for which you want to compute the depreciation. *Period* an integer in the range $\{1, Life + 1\}$. Period 1 is the (partial) period from *PurchaseDate* until *NextPeriodDate*.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *DepreciationLinearLife* (page 127) returns the depreciation of an asset for the specified period.

Equation

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equation

$$\tilde{d}_1 = f_{PN} \frac{c - s}{L}$$
$$\tilde{d}_i = \frac{c - s}{L} \qquad (i \neq 1).$$

Note: The function *DepreciationLinearLife* (page 127) is similar to the Excel function SLN.

Example

The following code illustrates how to compute the linear depreciation for each period of an investment initially costing 100.000, at the end of its useful life having value of 10.000, over a period of 10 years.

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life +1 );
_p_deprec( _i_per ) := DepreciationLinearLife(
        PurchaseDate   :  "2024-03-01",
        NextPeriodDate :  "2025-01-01",
        Cost           :  1e5,
        Salvage        :  1e4,
        Life           :  10,
```

(continues on next page)

(continued from previous page)

```
        Period          :  _i_per,
        Basis           :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );

block where single_column_display := 1 ;
        display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{  1 : 7500,
   2 : 9000,
   3 : 9000,
   4 : 9000,
   5 : 9000,
   6 : 9000,
   7 : 9000,
   8 : 9000,
   9 : 9000,
  10 : 9000,
  11 : 1500 } ;
```

As you can see, the depreciation of the first and last period add up to the depreciation for a single year.

References

- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

DepreciationLinearRate

The function *DepreciationLinearRate* (page 129) returns the depreciation of an asset for the specified period, using linear depreciation. The accounting periods have a length of one year, but they don't necessary need to start January 1. The sum of the depreciation amounts of all periods cannot be higher than the difference between the cost and the salvage.

```
DepreciationLinearRate(
    PurchaseDate,           ! (input) scalar string expression
    NextPeriodDate,         ! (input) scalar string expression
    Cost,                   ! (input) numerical expression
    Salvage,                ! (input) numerical expression
    Period,                 ! (input) numerical expression
    DepreciationRate,       ! (input) numerical expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Period The period for which you want to compute the depreciation. *Period* must be a positive integer. Period 1 is the (partial) period from *PurchaseDate* until *NextPeriodDate*.

DepreciationRate The value of the asset declines every period by an amount equal to the depreciation rate times the *Cost*. *DepreciationRate* must be a numerical expression in the range $[0, \frac{1}{2})$.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function `DepreciationLinearRate` (page 129) returns the depreciation of an asset for the specified period.

Equation

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equation

$$\tilde{d}_1 = f_{PN} rc$$
$$\tilde{d}_i = rc \qquad (i \neq 1)$$

where r is the depreciation rate.

Note:

- The useful life of the asset is determined by the depreciation rate, and the requirement that the value of the asset can never drop below its salvage value.
- The function `DepreciationLinearRate` (page 129) is similar to the Excel function AMORLINC.

Example

The following code illustrates how to compute the linear depreciation for each period of an investment initially costing 100.000, at the end of its useful life having value of 10.000, with a constant rate of 10%.

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life  );
_p_deprec( _i_per ) := DepreciationLinearRate(
```

(continues on next page)

(continued from previous page)

```
    PurchaseDate     :  "2024-03-01",
    NextPeriodDate   :  "2025-01-01",
    Cost             :  1e5,
    Salvage          :  1e4,
    Period           :  _i_per,
    DepreciationRate :  0.1,
    Basis            :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{  1 :  8333.333333,
   2 : 10000.000000,
   3 : 10000.000000,
   4 : 10000.000000,
   5 : 10000.000000,
   6 : 10000.000000,
   7 : 10000.000000,
   8 : 10000.000000,
   9 : 10000.000000,
  10 :  1666.666667 } ;
```

References

- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

DepreciationNonLinearSumOfYear

The function *DepreciationNonLinearSumOfYear* (page 131) returns the depreciation of an asset for the specified period, using sum of years' digits depreciation. The accounting periods have a length of one year, but they don't necessary need to start January 1. The depreciation amounts decline linear for every following period until the value reaches the salvage.

```
DepreciationNonLinearSumOfYear(
    PurchaseDate,             ! (input) scalar string expression
    NextPeriodDate,           ! (input) scalar string expression
    Cost,                     ! (input) numerical expression
    Salvage,                  ! (input) numerical expression
    Life,                     ! (input) numerical expression
    Period,                   ! (input) numerical expression
    [Basis]                   ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Life The number of periods until the asset will be fully depreciated, also called the useful life of the asset. *Life* must be a positive integer.

Period The period for which you want to compute the depreciation. *Period* an integer in the range $\{1, Life + 1\}$. Period 1 is the (partial) period from *PurchaseDate* until *NextPeriodDate*.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *DepreciationNonLinearSumOfYear* (page 131) returns the depreciation of an asset for the specified period.

Equation

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equation

$$\tilde{d}_1 = \frac{c - s}{\frac{1}{2}L(L+1)} L f_{PN}$$
$$\tilde{d}_i = \frac{c - s}{\frac{1}{2}L(L+1)} (L + 2 - i - f_{PN}) \qquad (i \neq 1).$$

Note: The function *DepreciationNonLinearSumOfYear* (page 131) is similar to the Excel function SYD.

Example

Using `DepreciationNonLinearSumOfYear` for declining depreciation over a period of 10 years:

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life+1  );
_p_deprec( _i_per ) := DepreciationNonLinearSumOfYear(
        PurchaseDate     :  "2024-03-01",
        NextPeriodDate   :  "2025-01-01",
        Cost             :  1e5,
        Salvage          :  1e4,
        Life             :  _p_life,
```

(continues on next page)

(continued from previous page)

```
        Period          :  _i_per,
        Basis           :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{  1 : 13636.363636,
   2 : 15000.000000,
   3 : 13363.636364,
   4 : 11727.272727,
   5 : 10090.909091,
   6 :  8454.545455,
   7 :  6818.181818,
   8 :  5181.818182,
   9 :  3545.454545,
  10 :  1909.090909,
  11 :   272.727273 } ;
```

References

- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

DepreciationNonLinearFactor

The function *DepreciationNonLinearFactor* (page 133) returns the depreciation of an asset for the specified period, using double-declining balance depreciation or some other method you specify. The accounting periods have a length of one year, but they don't necessary need to start January 1. The depreciation amounts decline by the factor times a fixed rate for every succeeding period. The higher the used factor, the sooner the asset is totally depreciated.

```
DepreciationNonLinearFactor(
    PurchaseDate,             ! (input) scalar string expression
    NextPeriodDate,           ! (input) scalar string expression
    Cost,                     ! (input) numerical expression
    Salvage,                  ! (input) numerical expression
    Life,                     ! (input) numerical expression
    Period,                   ! (input) numerical expression
    Factor                    ! (input) numerical expression
    [Basis,]                  ! (optional) numerical expression
    [Mode]                    ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Life The number of periods until the asset will be fully depreciated, also called the useful life of the asset. *Life* must be a positive integer.

Period The period for which you want to compute the depreciation. *Period* an integer in the range $\{1, Life + 1\}$. Period 1 is the (partial) period from *PurchaseDate* until *NextPeriodDate*.

Factor The rate by which the depreciation declines is $\frac{Factor}{Life}$. *Factor* must be a numerical expression in the range $[1, \infty)$. In case $Factor = 2$ we define this method as double declining depreciation.

Basis The day-count basis method to be used. The default is 1.

Mode Specifies how partial periods will be handled. *Mode* must be binary. $Mode = 0$: we just take a relatively equal part of the depreciation for a full year. This is mathematically incorrect, but is rather common in the financial world. $Mode = 1$: the depreciation for the partial periods is calculated so that the asset exactly equals its Salvage after its useful life. The default is 0.

Return Value

The function `DepreciationNonLinearFactor` (page 133) returns the depreciation of an asset for the specified period.

Equation

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equations

$$\tilde{d}_1 = \begin{cases} f_{PN} r c & \text{for } \textit{Mode} = 0 \\ \left(1 - (1-r)^{f_{PN}}\right) c & \text{for } \textit{Mode} = 1 \end{cases}$$

$$\tilde{d}_i = (c - d_1) r (1-r)^{i-2} \qquad (i \neq 1)$$

where the depreciation rate r equals

$$r = \frac{f}{L}$$

with f the *Factor* argument.

Note:

- The useful life of the asset is determined by the *Factor* and *Life* arguments, and the requirement that the value of the asset can never drop below its salvage value.
- The function `DepreciationLinearNonFactor` is similar to the Excel function DDB.

Example

Using DepreciationNonLinearFactor for declining depreciation over a period of 10 years:

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life+1  );
_p_deprec( _i_per ) := DepreciationNonLinearFactor(
    PurchaseDate     :  "2024-03-01",
    NextPeriodDate   :  "2025-01-01",
    Cost             :  1e5,
    Salvage          :  1e4,
    Life             :  _p_life,
    Period           :  _i_per,
    Factor           :  2,
    Basis            :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{  1 : 16666.666667,
   2 : 16666.666667,
   3 : 13333.333333,
   4 : 10666.666667,
   5 :  8533.333333,
   6 :  6826.666667,
   7 :  5461.333333,
   8 :  4369.066667,
   9 :  3495.253333,
  10 :  2796.202667,
  11 :  1184.810667 } ;
```

References

- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

DepreciationNonLinearLife

The function *DepreciationNonLinearLife* (page 135) returns the depreciation of an asset for the specified period, using fixed declining balance depreciation. The accounting periods have a length of one year, but they don't necessary need to start January 1. The depreciation amounts decline by a fixed rate for every succeeding period.

```
DepreciationNonLinearLife(
    PurchaseDate,             ! (input) scalar string expression
    NextPeriodDate,           ! (input) scalar string expression
```

(continues on next page)

(continued from previous page)

```
    Cost,                   ! (input) numerical expression
    Salvage,                ! (input) numerical expression
    Life,                   ! (input) numerical expression
    Period,                 ! (input) numerical expression
    [Basis,]                ! (optional) numerical expression
    [Mode]                  ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Life The number of periods until the asset will be fully depreciated, also called the useful life of the asset. *Life* must be a positive integer.

Period The period for which you want to compute the depreciation. *Period* an integer in the range $\{1, Life + 1\}$. Period 1 is the (partial) period from *PurchaseDate* until *NextPeriodDate*.

Basis The day-count basis method to be used. The default is 1.

Mode Specifies how partial periods will be handled. *Mode* must be binary. $Mode = 0$: we just take a relatively equal part of the depreciation for a full year. This is mathematically incorrect, but is rather common in the financial world. $Mode = 1$: the depreciation for the partial periods is calculated so that the asset exactly equals its Salvage after its useful life. The default is 0.

Return Value

The function `DepreciationNonLinearLife` (page 135) returns the depreciation of an asset for the specified period.

Equation

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equations

$$\tilde{d}_1 = \begin{cases} f_{PN} r v_1 & \text{for } \textit{Mode} = 0 \\ \left(1 - (1-r)^{f_{PN}}\right) v_1 & \text{for } \textit{Mode} = 1 \end{cases}$$

$$\tilde{d}_i = r v_i \qquad (i \neq 1)$$

where the depreciation rate r equals

$$r = 1 - \left(\frac{s}{c}\right)^{1/L}$$

Note: The function `DepreciationLinearNonLife` is similar to the Excel function DB.

Example

Using `DepreciationNonLinearLife` for declining depreciation over a period of 10 years:

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life+1  );
_p_deprec( _i_per ) := DepreciationNonLinearLife(
    PurchaseDate     :  "2024-03-01",
    NextPeriodDate   :  "2025-01-01",
    Cost             :  1e5,
    Salvage          :  1e4,
    Life             :  _p_life,
    Period           :  _i_per,
    Basis            :  1,
    Mode             :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{  1 : 17459.581473,
   2 : 16976.233585,
   3 : 13484.701656,
   4 : 10711.279262,
   5 :  8508.271548,
   6 :  6758.360319,
   7 :  5368.356422,
   8 :  4264.237080,
   9 :  3387.203912,
  10 :  2690.551704,
  11 :   391.223038 } ;
```

References

- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

DepreciationNonLinearRate

The function DepreciationNonLinearRate (page 137) returns the depreciation of an asset for the specified period, using factor-declining depreciation. The *DepreciationRate* determines the factor. The accounting periods have a length of one year, but they don't necessary need to start January 1.

```
DepreciationNonLinearRate(
    PurchaseDate,              ! (input) scalar string expression
    NextPeriodDate,            ! (input) scalar string expression
    Cost,                      ! (input) numerical expression
    Salvage,                   ! (input) numerical expression
    Period,                    ! (input) numerical expression
    DepreciationRate,          ! (input) numerical expression
    [Basis,]                   ! (optional) numerical expression
    [Mode]                     ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Period The period for which you want to compute the depreciation. *Period* an integer in the range $\{1, Life + 1\}$. Period 1 is the (partial) period from *PurchaseDate* until *NextPeriodDate*.

DepreciationRate The value of the asset declines every period by an amount equal to the depreciation rate times the *Cost*. *DepreciationRate* must be a numerical expression in the range $[0, \frac{1}{2})$.

Basis The day-count basis method to be used. The default is 1.

Mode Specifies how partial periods will be handled. *Mode* must be binary. $Mode = 0$: we just take a relatively equal part of the depreciation for a full year. This is mathematically incorrect, but is rather common in the financial world. $Mode = 1$: the depreciation for the partial periods is calculated so that the asset exactly equals its Salvage after its useful life. The default is 0.

Return Value

The function DepreciationNonLinearRate (page 137) returns the depreciation of an asset for the specified period.

Equation

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equations

$$\tilde{d}_1 = \begin{cases} f_{PN} r f c & \text{for } \mathit{Mode} = 0 \\ \left(1 - (1 - rf)^{f_{PN}}\right) c & \text{for } \mathit{Mode} = 1 \end{cases}$$

$$\tilde{d}_i = \begin{cases} r f v_i & (1 < i < \tilde{L} - 1) \\ \frac{1}{2} v_i & (i = \tilde{L} - 1) \\ v_i - s & (i = \tilde{L}) \end{cases}$$

where r is the *DepreciationRate*, $\tilde{L} = \lceil 1/r \rceil$ the useful life of the asset, and the depreciation coefficient f is determined by

$$f = \begin{cases} 1.5 & \text{for } \frac{1}{4} \leq r < \frac{1}{2} \\ 2.0 & \text{for } \frac{1}{6} \leq r < \frac{1}{4} \\ 2.5 & \text{for } r < \frac{1}{6} \end{cases}$$

Note: The function `DepreciationLinearNonRate` is similar to the Excel function AMORDEGRC.

Example

Using `pr_testDepreciationNonLinearRate` for declining depreciation over a period of 10 years:

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life  );
_p_deprec( _i_per ) := DepreciationNonLinearRate(
        PurchaseDate     :  "2024-03-01",
        NextPeriodDate   :  "2025-01-01",
        Cost             :  1e5,
        Salvage          :  1e4,
        Period           :  _i_per,
        DepreciationRate :  0.1,
        Basis            :  1,
        Mode             :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{ 1 : 21316.370243,
  2 : 19670.907439,
  3 : 14753.180579,
  4 : 11064.885435,
  5 :  8298.664076,
  6 :  6223.998057,
  7 :  4667.998543,
```

(continues on next page)

(continued from previous page)

```
 8 :  3500.998907,
 9 :   502.996721 } ;
```

References

- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

DepreciationSum

The function `DepreciationSum` (page 140) returns the depreciation of an asset for the specified interval, using factor-declining depreciation. The accounting periods have a length of one year, but they don't necessary need to start January 1. A parameter Switch is used to indicated that, when straight-line depreciation results in greater depreciation than factor-declining depreciation, the calculation of the depreciation has to be based on that method.

```
DepreciationSum(
    PurchaseDate,           ! (input) scalar string expression
    NextPeriodDate,         ! (input) scalar string expression
    Cost,                   ! (input) numerical expression
    Salvage,                ! (input) numerical expression
    Life,                   ! (input) numerical expression
    StartPeriod,            ! (input) numerical expression
    EndPeriod,              ! (input) numerical expression
    Factor,                 ! (input) numerical expression
    [Basis,]                ! (optional) numerical expression
    [Mode,]                 ! (optional) numerical expression
    [Switch]                ! (optional) numerical expression
    )
```

Arguments

PurchaseDate The date of purchase of the asset. *PurchaseDate* must be given in a date format. This is the first day that there will be depreciated.

NextPeriodDate The next date after the balance is drawn up. *NextPeriodDate* must also be in date format. *NextPeriodDate* is the first day of a new period and must be further in time than *PurchaseDate*, but not more than one year after *PurchaseDate*. When *NextPeriodDate* is an empty string, it will get the default value of January 1st of the next year after purchase.

Cost The purchase or initial cost of the asset. *Cost* must be a positive number.

Salvage The value of the asset at the end of its useful life. *Salvage* must be a scalar numerical expression in the range $[0, Cost)$.

Life The number of periods until the asset will be fully depreciated, also called the useful life of the asset. *Life* must be a positive integer.

StartPeriod The starting period of the interval, for which you want to compute the sum of depreciation, this may also indicate a partial period. *StartPeriod* must be an integer in the range $\{1, Life\}$. *StartPeriod* must have the same unit as *Life*.

EndPeriod The last period of the interval, for which you want to compute the sum of depreciation. *EndPeriod* must be an integer in the range $\{StartPeriod, Life\}$. *EndPeriod* must have the same unit as *Life*.

Factor The rate by which the depreciation declines is $\frac{Factor}{Life}$. *Factor* must be a numerical expression in the range $[1, \infty)$. In case $Factor = 2$ we define this method as double declining depreciation.

Basis The day-count basis method to be used. The default is 1.

Mode Specifies how partial periods will be handled. *Mode* must be binary. $Mode = 0$: we just take a relatively equal part of the depreciation for a full year. This is mathematically incorrect, but is rather common in the financial world. $Mode = 1$: the depreciation for the partial periods is calculated so that the asset exactly equals its Salvage after its useful life. The default is 0.

Switch Indicates whether to switch to straight-line depreciation when the depreciation amounts will be higher applying that method, or not to switch. *Switch* must be binary. If $Switch = 0$: do not switch, if $Switch = 1$: switch. The default is 1.

Return Value

The function *DepreciationSum* (page 140) returns the depreciation of an asset for the specified period.

Note: The function *DepreciationSum* (page 140) is similar to the Excel function VDB.

Example

Using `DepreciationSum` for declining depreciation over a period of 10 years:

```
_p_life := 10 ;
_s_periods := ElementRange(1,_p_life  );
_p_deprec( _i_per ) := DepreciationSum(
    PurchaseDate   :  "2024-03-01",
    NextPeriodDate :  "2025-01-01",
    Cost           :  1e5,
    Salvage        :  1e4,
    Life           :  _p_life,
    StartPeriod    :  _i_per,
    EndPeriod      :  _i_per,
    Factor         :  2,
    Basis          :  1,
    Mode           :  1,
    Switch         :  1);
_p_totDeprec := sum( _i_per, _p_deprec( _i_per ) );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_deprec( _i_per ) ;
endblock ;
```

The actual values computed are:

```
_p_deprec(_i_per) := data
{  1 : 16968.734756,
   2 : 16606.253049,
```

(continues on next page)

(continued from previous page)

```
  3 : 13285.002439,
  4 : 10628.001951,
  5 :  8502.401561,
  6 :  6801.921249,
  7 :  5441.536999,
  8 :  4353.229599,
  9 :  3706.459199,
 10 :  3706.459199 } ;
```

References

- The functions *DepreciationNonLinearFactor* (page 133), *DepreciationLinearLife* (page 127).
- Day count basis *methods* (page 122).
- General equations for computing *depreciations* (page 126).

1.6.4 Investments

Investments and Loans

When dealing with investments or loans, several cash flows are scheduled within a certain time frame, such as the

- present value (the value at the beginning of the scheduled time frame),
- future value (the value at the end of the scheduled time frame), and
- periodic payments during the scheduled time frame.

AIMMS provides several functions to calculate each of these cash flows (or the interest rate used) in the presence of all others.

Constant Payments

Investments and loans with constant, periodic payments and a constant interest rate are special. When the payments are annual, such an investment is called an annuity. The constant payments of these investments consist of a principal and an interest payment. The principal payment will generally increase in time whereas the interest payment will decrease in time. Two different types of investments with constant payments and interest rates can be distinguished:

- The first type, also referred as type 0, has payments that are made at the end of each period.
- The second type, type 1, has payments that are made at the beginning of each period. This type has no interest payment at the beginning of the first period, but does have an extra period, after the last periodic payment, with an interest payment over the last period and an inverse principal payment.

Equations

Cash flows can be either positive or negative, where a positive payment indicates that you are receiving this payment. Taking the interest into account, the total value of an investment must be equal to zero after all cash flows have occurred. For example, a positive present value and positive payments will lead to a negative future value: your debt has grown. The following equation expresses the relation between all the cash flows that take place

$$v_p(1+r)^N + p\sum_{i=1}^{N}(1+r)^{i-1+T} + v_f = 0$$

where v_p is the present value, v_f is the future value, p is the constant periodic payment, r is the constant interest rate and T is the investment type as discussed above.

AIMMS supports the following investment functions with constant, periodic payments:

- *InvestmentConstantPresentValue* (page 147)
- *InvestmentConstantFutureValue* (page 143)
- *InvestmentConstantPeriodicPayment* (page 145)
- *InvestmentConstantInterestPayment* (page 148)
- *InvestmentConstantPrincipalPayment* (page 150)
- *InvestmentConstantCumulativeInterestPayment* (page 152)
- *InvestmentConstantCumulativePrincipalPayment* (page 154)
- *InvestmentConstantNumberPeriods* (page 155)
- *InvestmentConstantRateAll* (page 159)
- *InvestmentConstantRate* (page 157)

Variable Payments

When the cash flows are variable (i.e. not constant), take place at irregular intervals, or when the interest rate varies over time, it still possible to compute present values, future values, and the internal rate of return, i.e. the rate received for an investment consisting of payments and income.

AIMMS supports the following investment functions for variable cash flows:

- *InvestmentVariablePresentValue* (page 161)
- *InvestmentVariablePresentValueInPeriodic* (page 162)
- *InvestmentSingleFutureValue* (page 165)
- *InvestmentVariableInternalRateReturnAll* (page 168)
- *InvestmentVariableInternalRateReturn* (page 166)
- *InvestmentVariableInternalRateReturnInPeriodicAll* (page 172)
- *InvestmentVariableInternalRateReturnInPeriodic* (page 170)
- *InvestmentVariableInternalRateReturnModified* (page 174)

InvestmentConstantFutureValue

The function *InvestmentConstantFutureValue* (page 143) returns the future value of an investment based on periodic, constant payments and a constant interest rate.

```
InvestmentConstantFutureValue(
    PresentValue,           ! (input) numerical expression
    Payment,                ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    InterestRate,           ! (input) numerical expression
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

Payment The periodic payment for the investment. *Payment* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantFutureValue* (page 143) returns the cash balance you want to attain after the last payment is made.

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue*, *Payment* and *InterestRate* can be used as a variable.
- The function *InvestmentConstantFutureValue* (page 143) is similar to the Excel function FV.

Example

Receiving regularly money over time, leads to a debt:

```
! Receiving 10 per period,
! for 10 periods, and
! an interest of 4%,
! results in a debt of 120.06
_p_constFutureValue := InvestmentConstantFutureValue(
    PresentValue  :  0,
```

(continues on next page)

(continued from previous page)

```
    Payment       :  10,
    NumberPeriods :  10,
    InterestRate  :  0.04,
    type          :  0);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_constFutureValue;
endblock ;
```

The future value computed is negative, indicating a debt:

```
_p_constFutureValue := -120.061071 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantPeriodicPayment

The function *InvestmentConstantPeriodicPayment* (page 145) returns the periodic payment for an investment based on periodic, constant payments and a constant interest rate.

```
InvestmentConstantPeriodicPayment(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    InterestRate,           ! (input) numerical expression
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantPeriodicPayment* (page 145) returns the periodic payment for the investment.

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue*, *FutureValue* and *InterestRate* can be used as a variable.
- The function *InvestmentConstantPeriodicPayment* (page 145) is similar to the Excel function PMT.

Example

Perhaps the following will encourage you to save up first:

```
! How much needs to be paid per period,
! for 10 periods, and an interest rate of 4%
! to attain a capital of 100 in the end?
_p_constPeriodicPaymentSavingup := InvestmentConstantPeriodicPayment(
    PresentValue  :  0,
    FutureValue   :  100,
    NumberPeriods :  10,
    InterestRate  :  0.04,
    type          :  0);

! Spending a 100 now, how much do I have to
! reimburse/compensate over each of the 10 periods,
! whereby the interest rate is 4%
_p_constPeriodicPaymentReimbursing := InvestmentConstantPeriodicPayment(
    PresentValue  :  100,
    FutureValue   :  0,
    NumberPeriods :  10,
    InterestRate  :  0.04,
    type          :  0);

block where single_column_display := 1, listing_number_precision := 6 ;
    display { _p_constPeriodicPaymentSavingup, _p_constPeriodicPaymentReimbursing } ;
endblock ;
```

The result of this example shows there is a 50% difference in amount to be saved per period, if you spent it at the beginning or at the end:

```
_p_constPeriodicPaymentSavingup    :=  -8.329094 ;
_p_constPeriodicPaymentReimbursing := -12.329094 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantPresentValue

The function *InvestmentConstantPresentValue* (page 147) returns the present value of an investment based on periodic, constant payments and a constant interest rate.

```
InvestmentConstantPresentValue(
    FutureValue,              ! (input) numerical expression
    Payment,                  ! (input) numerical expression
    NumberPeriods,            ! (input) numerical expression
    InterestRate,             ! (input) numerical expression
    Type                      ! (input) numerical expression
    )
```

Arguments

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

Payment The periodic payment for the investment. *Payment* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantPresentValue* (page 147) returns the total amount that a series of future payments is worth at this moment.

Note:

- This function can be used in an objective function or constraint and the input parameters *FutureValue*, *Payment* and *InterestRate* can be used as a variable.
- The function *InvestmentConstantPresentValue* (page 147) is similar to the Excel function PV.

Example

Paying regularly money over time, resolves a debt:

```
! Paying 10 per period,
! for 10 periods, and
! an interest of 4%,
! resolves a debt of 81
_p_constPresentValue := InvestmentConstantPresentValue(
    FutureValue   :  0,
    Payment       :  10,
    NumberPeriods :  10,
    InterestRate  :  0.04,
    type          :  0);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_constPresentValue;
endblock ;
```

The present value computed is negative, indicating a debt:

```
_p_constPresentValue := -81.108958 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantInterestPayment

The function *InvestmentConstantInterestPayment* (page 148) returns the interest payment of the specified period for an investment based on periodic, constant payments and a constant interest rate. Every periodic payment can be divided in two parts: an interest payment and a principal repayment.

```
InvestmentConstantInterestPayment(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    Period                  ! (input) numerical expression
    InterestRate,           ! (input) numerical expression
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

Period The period for which you want to compute the interest payment. *Period* must be an integer in the range $\{1, NumberPeriods + Type\}$. When $Type = 1$, the extra period is to account the interest over the former period.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantInterestPayment* (page 148) returns the interest payment for the specified period.

Equation

The interest payment i_i in period i is computed through the equation

$$i_i = -v_p r(1+r)^{i-1-T} - p\left(\left((1+r)^{i-1-T} - 1\right)(1+r)^T + rT\right)$$

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue*, *FutureValue* and *InterestRate* can be used as a variable.
- The function *InvestmentConstantInterestPayment* (page 148) is similar to the Excel function `IPMT`.

Example

How much interest is paid for a loan of ten years and an interest of 4% in each period:

```
_p_life := 10 ;
_bp_type := 1;
_s_periods := ElementRange(1, _p_life + _bp_type);
_p_InterestPayment(_i_per) :=
    InvestmentConstantInterestPayment(
        PresentValue  :  100,
        FutureValue   :  0,
        NumberPeriods :  10,
```

(continues on next page)

(continued from previous page)

```
        Period        :  _i_per,
        InterestRate  :  0.04,
        type          :  _bp_type);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_InterestPayment;
endblock ;
```

The following table shows that the interest portion of each anuity becomes less:

```
_p_InterestPayment := data
{  2 : -3.525804e+00,
   3 : -3.192640e+00,
   4 : -2.846150e+00,
   5 : -2.485800e+00,
   6 : -2.111036e+00,
   7 : -1.721282e+00,
   8 : -1.315937e+00,
   9 : -8.943784e-01,
  10 : -4.559576e-01,
  11 :  4.440892e-15 } ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantPrincipalPayment

The function *InvestmentConstantPrincipalPayment* (page 150) returns the principal payment of the specified period for an investment based on periodic, constant payments and a constant interest rate. Every periodic payment can be divided in two parts: an interest payment and a principal payment.

```
InvestmentConstantPrincipalPayment(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    Period                  ! (input) numerical expression
    InterestRate,           ! (input) numerical expression
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

Period The period for which you want to compute the interest payment. *Period* must be an integer in the range $\{1, NumberPeriods + Type\}$. When $Type = 1$, the extra period is to account the interest over the former period.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantPrincipalPayment* (page 150) returns the principal payment for the specified period.

Equation

The principal payment p_i in period i follows from the relation

$$p_i = p - i_i$$

where i_i is the interest payment in period i.

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue*, *FutureValue* and *InterestRate* can be used as a variable.
- The function *InvestmentConstantPrincipalPayment* (page 150) is similar to the Excel function PPMT.

Example

How much is paid off for a loan of ten years and an interest of 4% in each period:

```
_p_life := 10 ;
_bp_type := 1;
_s_periods := ElementRange(1, _p_life + _bp_type);
_p_PrincipalPayment(_i_per) :=
        InvestmentConstantPrincipalPayment(
                PresentValue  :  100,
                FutureValue   :  0,
```

(continues on next page)

(continued from previous page)

```
                NumberPeriods :  10,
                Period        :  _i_per,
                InterestRate  :  0.04,
                type          :  _bp_type);
block where single_column_display := 1, listing_number_precision := 8 ;
        display _p_PrincipalPayment;
endblock ;
```

The following table that the pay off portion of each anuity increases:

```
_p_PrincipalPayment := data
{  1 : -1.18548985e+01,
   2 : -8.32909443e+00,
   3 : -8.66225821e+00,
   4 : -9.00874854e+00,
   5 : -9.36909848e+00,
   6 : -9.74386242e+00,
   7 : -1.01336169e+01,
   8 : -1.05389616e+01,
   9 : -1.09605201e+01,
  10 : -1.13989409e+01,
  11 : -4.44089210e-15 } ;
```

Except for the first period; there is no interest to be paid when no time is passed.

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantCumulativeInterestPayment

The function *InvestmentConstantCumulativeInterestPayment* (page 152) returns the cumulative interest payment for the specified interval for an investment based on periodic, constant payments and a constant interest rate. Every periodic payment can be divided in two parts: an interest payment and a principal payment.

```
InvestmentConstantCumulativeInterestPayment(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    StartPeriod,            ! (input) numerical expression
    EndPeriod,              ! (input) numerical expression
    InterestRate,           ! (input) numerical expression
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

StartPeriod The starting period of the interval for which you want to compute the cumulative interest payment. *StartPeriod* must be an integer in the range $\{1, NumberPeriods\}$.

EndPeriod The ending period of the interval for which you want to compute the cumulative interest payment. *EndPeriod* must be an integer in the range $\{StartPeriod, NumberPeriods\}$.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function `InvestmentConstantCumulativeInterestPayment` (page 152) returns the sum of the interest payments for the periods in the specified interval.

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue*, *FutureValue* and *InterestRate* can be used as a variable.
- The function `InvestmentConstantCumulativeInterestPayment` (page 152) is similar to the Excel function `CUMIPMT`.

Example

How much of the payments during the first half of the loan is interest?

```
_p_interestFirstFiveYears :=
    InvestmentConstantCumulativeInterestPayment(
        PresentValue  :  10,
        FutureValue   :  0,
        NumberPeriods :  10,
        StartPeriod   :  1,
        EndPeriod     :  5,
        InterestRate  :  0.1,
        type          :  0);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_interestFirstFiveYears ;
endblock ;
```

That is significant:

```
_p_interestFirstFiveYears := -4.306601 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantCumulativePrincipalPayment

The function *InvestmentConstantCumulativePrincipalPayment* (page 154) returns the cumulative principal payment for the specified interval for an investment based on periodic, constant payments and a constant interest rate. Every periodic payment can be divided in two parts: an interest payment and a principal payment.

```
InvestmentConstantCumulativePrincipalPayment(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    StartPeriod,            ! (input) numerical expression
    EndPeriod,              ! (input) numerical expression
    InterestRate,           ! (input) numerical expression
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

StartPeriod The starting period of the interval for which you want to compute the cumulative interest payment. *StartPeriod* must be an integer in the range $\{1, NumberPeriods\}$.

EndPeriod The ending period of the interval for which you want to compute the cumulative interest payment. *EndPeriod* must be an integer in the range $\{StartPeriod, NumberPeriods\}$.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantCumulativePrincipalPayment* (page 154) returns the sum of the principal payments for the periods in the specified interval.

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue*, *FutureValue* and *InterestRate* can be used as a variable.
- The function *InvestmentConstantCumulativePrincipalPayment* (page 154) is similar to the Excel function CUMPRINC.

Example

How much of the payments during the first half of the loan is pay off?

```
_p_payoffFirstFiveYears :=
    InvestmentConstantCumulativePrincipalPayment(
        PresentValue  :  10,
        FutureValue   :  0,
        NumberPeriods :  10,
        StartPeriod   :  1,
        EndPeriod     :  5,
        InterestRate  :  0.1,
        type          :  0);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_payoffFirstFiveYears ;
endblock ;
```

That is actually less than the interest pay!

```
_p_payoffFirstFiveYears := -3.830669 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantNumberPeriods

The function *InvestmentConstantNumberPeriods* (page 155) returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

```
InvestmentConstantNumberPeriods(
    PresentValue,            ! (input) numerical expression
    FutureValue,             ! (input) numerical expression
    Payment,                 ! (input) numerical expression
    InterestRate,            ! (input) numerical expression
```

(continues on next page)

(continued from previous page)

```
    Type                    ! (input) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

Payment The value of the periodic payment for the investment. Payment must be a real number. *Payment* and *InterestRate* cannot both be 0.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Return Value

The function *InvestmentConstantNumberPeriods* (page 155) returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

Note:

- The value returned may be fractional.
- The function *InvestmentConstantNumberPeriods* (page 155) is similar to the Excel function NPER.

Example

`InvestmentConstantNumberPeriods` is used here to determine how long it takes to save up for an investment, or pay back on a loan, given a certain amount of money that can be reserved per period.

```
! How long does it take to save up 100
! when paying 10 each period and an interest rate of 4%?
_p_nperSavingUp :=
    InvestmentConstantNumberPeriods(
        PresentValue :  0,
        FutureValue  :  -100,
        Payment      :  10,
        InterestRate :  0.04,
        type         :  0);

! How long does it take to reimburse 100
! when paying 10 each period and an interest rate of 4%?
_p_nperReimbursing :=
    InvestmentConstantNumberPeriods(
```

(continues on next page)

(continued from previous page)

```
        PresentValue :  -100,
        FutureValue  :  0,
        Payment      :  10,
        InterestRate :  0.04,
        type         :  0);
block where single_column_display := 1, listing_number_precision := 6 ;
    display { _p_nperSavingUp, _p_nperReimbursing};
endblock ;
```

The results in the listing file state that saving up requires less time than paying back.

```
_p_nperSavingUp     :=  8.578942 ;
_p_nperReimbursing  := 13.024384 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantRate

The function *InvestmentConstantRate* (page 157) returns the interest rate for an investment based on periodic, constant payments and a constant interest rate. This function uses the procedure `InvestmentConstantRateAll` to determine all possible interest rates and returns the interest rate that is within the specified bounds.

```
InvestmentConstantRate(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    Payment,                ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    Type,                   ! (input) numerical expression
    [LowerBound,]           ! (optional) numerical expression
    [UpperBound,]           ! (optional) numerical expression
    [Error]                 ! (optional) numerical expression
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

Payment The periodic payment for the investment. *Payment* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

LowerBound Indicates a minimum for the interest rate to be accepted by this function. The default is -1.

UpperBound Indicates a maximum for the interest rate to be accepted by this function. The default is 5.

Error Indicates whether AIMMS should give an error if multiple solutions are found that satisfy the bounds. $Error = 0$: if multiple solutions are found, return the solution with the smallest absolute value. $Error = 1$: if multiple solutions are found, return an error message. The default is 0.

Return Value

The function *InvestmentConstantRate* (page 157) returns the interest rate for an investment based on periodic, constant payments and a constant interest rate.

Note:

- The function *InvestmentConstantRate* (page 157) can be used in an objective function or constraint. The input parameters *PresentValue*, *FutureValue* and *Payment* can be used as variables.
- The function *InvestmentConstantRate* (page 157) is similar to the Excel function RATE.

Example

InvestmentConstantRate is used when there is a fixed series of equal payments to determine the interest percentage needed to make up for a loan or to save up for an investment.

```
! Which interest rate do I need to save up for a given investment
! with a given series of fixed savings?
_p_pctSavingUp :=
        InvestmentConstantRate(
                PresentValue  :  0,
                FutureValue   :  -100,
                Payment       :  10,
                NumberPeriods :  8,
                type          :  0,
                LowerBound    :  -1,
                UpperBound    :  5,
                Error         :  1);

! With which percentage can I pay off a loan,
! given a fixed series of equal payments?
_p_pctReimburse :=
        InvestmentConstantRate(
                PresentValue  :  -100,
                FutureValue   :  0,
                Payment       :  10,
                NumberPeriods :  13,
                type          :  0,
                LowerBound    :  -1,
                UpperBound    :  5,
                Error         :  1);

block where single_column_display := 1, listing_number_precision := 6 ;
```

(continues on next page)

(continued from previous page)

```
        display { _p_pctSavingUp, _p_pctReimburse};
endblock ;
```

As the number of periods used are not equal, it is hard to compare the results.

```
_p_pctSavingUp  := 0.062870 ;
_p_pctReimburse := 0.039769 ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentConstantRateAll

The procedure *InvestmentConstantRateAll* (page 159) returns the interest rate(s) for an investment based on periodic, constant payments and a constant interest rate.

```
InvestmentConstantRateAll(
    PresentValue,           ! (input) numerical expression
    FutureValue,            ! (input) numerical expression
    Payment,                ! (input) numerical expression
    NumberPeriods,          ! (input) numerical expression
    Type,                   ! (input) numerical expression
    Mode,                   ! (input) numerical expression
    NumberSolutions,        ! (output) numerical expression
    Solutions               ! (output) one-dimensional parameter
    )
```

Arguments

PresentValue The total amount that a series of future payments is worth at this moment. *PresentValue* must be a real number.

FutureValue The cash balance you want to attain after the last payment is made. *FutureValue* must be a real number.

Payment The periodic payment for the investment. *Payment* must be a real number.

NumberPeriods The total number of payment periods for the investment. *NumberPeriods* must be a positive integer.

Type Indicates when payments are due. $Type = 0$: Payments are due at the end of each period. $Type = 1$: Payments are due at the beginning of each period.

Mode Indicates whether all the solutions need to be found or just one. $Mode = 0$: the search for solutions stops after one solution is found. $Mode = 1$: the search for solutions continues till all solutions are found.

NumberSolutions The number of solutions found. If $Mode = 0$ *NumberSolutions* will always be 1.

Solutions There is not always a unique solution for *InterestRate*. Dependent on *Mode* one solution or all the solutions will be given. Solutions smaller than -1 are not supposed to be relevant, so the search for solutions is limited to the area greater than -1.

Note:

- When you want to use this procedure in an objective function or constraint you have to use `InvestmentConstantRate`.
- The function *InvestmentConstantRateAll* (page 159) is similar to the Excel function `RATE`.

Example

Given the local declarations:

```
Parameter _p_noSol;
Set _s_solutionNumbers {
    SubsetOf: Integers;
    Index: _i_solNo;
}
Parameter _p_sols {
    IndexDomain: _i_solNo;
}
```

All solutions to `InvestmentConstantRate` can be obtained using

```
_s_solutionNumbers := ElementRange(1,100);
InvestmentConstantRateAll(
    PresentValue    :  -100,
    FutureValue     :  0,
    Payment         :  10,
    NumberPeriods   :  13,
    type            :  0,
    Mode            :  1,
    NumberSolutions :  _p_noSol,
    Solutions       :  _p_sols );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_noSol, _p_sols ;
endblock ;
```

In this particular example there is only one solution:

```
_p_noSol := 1 ;


_p_sols := data
{ 1 : 0.039769 } ;
```

References

- General *equations* (page 143) for investments with constant, periodic payments.

InvestmentVariablePresentValue

The function *InvestmentVariablePresentValue* (page 161) returns the net present value for an investment based on a series of periodic cash flows at the end of the periods and a constant interest rate.

```
InvestmentVariablePresentValue(
    Value,                  ! (input) one-dimensional numerical parameter
    InterestRate            ! (input) numerical expression
    )
```

Arguments

Value The periodic payments (positive or negative), which must be equally spaced in time and occur at the end of each period. The order of the payments in *Value* must be the same as the order in which the cash flows occur. *Value* is an one dimensional parameter of real numbers. *Value* should contain at least one nonzero number. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Return Value

The function *InvestmentVariablePresentValue* (page 161) returns the net present value of an investment, which is the total value of all the future cash flows at the beginning of the first period.

Equation

The net present value v_p is computed through the equation

$$v_p = \sum_{i=1}^{n} \frac{p_i}{(1+r)^i}$$

where p_i are the (variable) periodic payments, and r is the (constant) interest rate.

Note:

- When all payments are constant, the net present value computed here is equal to the negative value of the present value computed by the function *InvestmentConstantPresentValue* (page 147) with the future value set to 0.0.
- This function can be used in an objective function or constraint and the input parameters *Value* and *InterestRate* can be used as a variable.
- The function *InvestmentVariablePresentValue* (page 161) is similar to the Excel function `NPV`.

Example

Given the local declarations:

```
Set _s_years {
    SubsetOf: Integers;
    Index: _i_year;
}
Parameter _p_value {
    IndexDomain: _i_year;
}
Parameter _p_npv;
```

Net present value can be computed as follows:

```
_s_years := ElementRange(2005,2008);
_p_value(_i_year) := ord(_i_year) * 100 + 50 ;
_p_npv := InvestmentVariablePresentValue( _p_value, 0.07 );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_npv ;
endblock ;
```

With the following result in the listing file:

```
_p_npv := 987.553700 ;
```

References

- The function *InvestmentConstantPresentValue* (page 147).

InvestmentVariablePresentValueInPeriodic

The function *InvestmentVariablePresentValueInPeriodic* (page 162) returns the net present value on the date of the first cash flow for an investment based on a series of in-periodic cash flows and a constant interest rate.

```
InvestmentVariablePresentValueInPeriodic(
    Value,                  ! (input) one-dimensional numerical expression
    Date,                   ! (input) one-dimensional string expression
    InterestRate,           ! (input) numerical expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

Value The payments (positive or negative). *Value* is an one-dimensional parameter of real numbers. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts. *Value* must contain at least one positive and at least one negative number.

Date The dates on which the payments occur. *Date* and *Value* must have the same order. *Date* is an one-dimensional parameter of dates given in a date format. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they may occur in any order. *Date* should contain as many dates as the number of values given by *Value*.

InterestRate The interest rate per period for the investment. *InterestRate* must be a numerical expression in the range $(-1, 1)$.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *InvestmentVariablePresentValueInPeriodic* (page 162) returns the net present value of an investment, which is the total value of all the future cash flows at this moment.

Equation

The net present value v_p is computed through the equation

$$v_p = \sum_{i=1}^{n} \frac{p_i}{(1+r)^{f_i}}$$

where p_i are the periodic payments, r is the (constant) interest rate, and f_i is the difference between date i and the first date (so, $f_1 = 0$), according to the selected day-count basis method.

Note:

- This function can be used in an objective function or constraint and the input parameters *Value* and *InterestRate* can be used as a variable.
- The function *InvestmentVariablePresentValueInPeriodic* (page 162) is similar to the Excel function `XNPV`.

Example

Given the local declarations:

```
Set _s_qs {
    SubsetOf: Integers;
    Index: _i_q;
}
Parameter _p_val {
    IndexDomain: _i_q;
}
StringParameter _sp_startQuarter {
```

(continues on next page)

(continued from previous page)

```
    IndexDomain: _i_q;
}
Parameter _p_npv;
```

Net present value, with contributions made at the start of each quarter, can be computed as follows:

```
! Prepping some data:
_s_qs := ElementRange(1,4);
_p_val(_i_q) := 3 + 2 * ord(_i_q)  ;
_sp_startQuarter(_i_q) := MomentToString(
    Format        :  "%c%y-%m-%d",
    unit          :  [month],
    ReferenceDate :  "2024-01-01",
    Elapsed       :  ((ord(_i_q)-1)*3)[month]);

! NPV computation
_p_npv := InvestmentVariablePresentValueInperiodic(
    value        :  _p_val,
    date         :  _sp_startQuarter,
    InterestRate :  0.07,
    Basis        :  1);

! Putting input and result in listing file
block where single_column_display := 1, listing_number_precision := 6 ;
    display { _p_val, _sp_startQuarter }, _p_npv ;
endblock ;
```

With the following result in the listing file:

```
Composite table:
    _i_q    _p_val  _sp_startQuarter
!   ----    ------  ----------------
       1         5  "2024-01-01"
       2         7  "2024-04-01"
       3         9  "2024-07-01"
       4        11  "2024-10-01"
    ;

    _p_npv := 31.038963 ;
```

References

- Day count basis *methods* (page 122).

InvestmentSingleFutureValue

The function *InvestmentSingleFutureValue* (page 165) returns the future value, the cash balance, of a payment made at this moment, present value, with periodic interest rates.

```
InvestmentSingleFutureValue(
    PresentValue,            ! (input) numerical expression
    PeriodicRate             ! (input) one-dimensional numerical expression
    )
```

Arguments

PresentValue Payment made at the start of the first period. *PresentValue* must be a real number. If *PresentValue* is a negative number it represents an outgoing amount and when it is a positive number it represents an incoming amount.

PeriodicRate Interest rates which differ per period. *PeriodicRate* is a one-dimensional parameter, which should contain at least one nonzero number. The periods must be equally spaced in time and the interest rates must be ordered.

Return Value

The function *InvestmentSingleFutureValue* (page 165) returns the future value of the present value, using the periodic interest rates.

Equation

The future value v_f is computed through the equation

$$v_f = v_p \prod_{i=1}^{n} (1 + r_i)$$

where v_p is the present value, and r_i the variable, periodic interest rates.

Note:

- This function can be used in an objective function or constraint and the input parameters *PresentValue* and *PeriodicRate* can be used as a variable.
- The function *InvestmentSingleFutureValue* (page 165) is similar to the Excel function `FVSCHEDULE`.

Example

Given the local declarations:

```
Set _s_years {
    SubsetOf: Integers;
    Index: _i_year;
}
Parameter _p_value {
    IndexDomain: _i_year;
}
Parameter _p_npv;
```

Net present value can be computed as follows:

```
_s_years := ElementRange(2005,2008);
_p_value(_i_year) := ord(_i_year) * 100 + 50 ;
_p_npv := InvestmentVariablePresentValue( _p_value, 0.07 );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_npv ;
endblock ;
```

With the following result in the listing file:

```
_p_npv := 987.553700 ;
```

References

- Day count basis *methods* (page 122).

InvestmentVariableInternalRateReturn

The function *InvestmentVariableInternalRateReturn* (page 166) returns the internal rate of return for an investment based on a series of periodic cash flows. The internal rate of return is the rate received for an investment. This function uses the procedure *InvestmentVariableInternalRateReturnAll* to determine all possible internal rates and returns the internal rate that is within the specified bounds.

```
InvestmentVariableInternalRateReturn(
    Value,                  ! (input) one-dimensional numerical expression
    [LowerBound,]           ! (optional) numerical expression
    [UpperBound,]           ! (optional) numerical expression
    [Error]                 ! (optional) numerical expression
    )
```

Arguments

Value The periodic payments (positive or negative), which must be equally spaced in time. The order of the payments in *Value* must be the same as the order in which the cash flows occur. *Value* is an one dimensional parameter of real numbers. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts. em Value must contain at least one positive and at least one negative number.

LowerBound Indicates a minimum for the internal rate to be accepted by this function. The default is -1.

UpperBound Indicates a maximum for the internal rate to be accepted by this function. The default is 5.

Error Indicates whether AIMMS should give an error if multiple solutions are found that satisfy the bounds. $Error = 0$: if multiple solutions are found, return the solution with the smallest absolute value. $Error = 1$: if multiple solutions are found, return an error message. The default is 0.

Return Value

The function *InvestmentVariableInternalRateReturn* (page 166) returns the internal rate of return for an investment based on a series of periodic cash flows. The internal rate of return is the rate received for an investment.

Note:

- The function *InvestmentVariableInternalRateReturn* (page 166) can be used in an objective function or constraint. The input parameter *Value* can be used as a variable.
- The function *InvestmentVariableInternalRateReturn* (page 166) is similar to the Excel function IRR.

Example

Often, internal rate of return is an investment first, (period 0, negative value), and then return cashflows in following periods.

```
_s_periods := ElementRange(0,4);
_p_val('0') := -100 ;
_p_val('1') := 50 ;
_p_val(_i_per | _i_per > '1') := _p_val(_i_per-1) * 1.5 ;
_p_irr := InvestmentVariableInternalRateReturn(
        value      :  _p_val,
        LowerBound :  -1,
        UpperBound :  5,
        Error      :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_val, _p_irr ;
endblock ;
```

This results in the following IRR:

```
_p_val := data
{ 0 : -100.000000,
  1 :   50.000000,
```

(continues on next page)

(continued from previous page)

```
  2 :   75.000000,
  3 :  112.500000,
  4 :  168.750000 } ;

_p_irr := 0.688847 ;
```

References

- The functions *InvestmentVariableInternalRateReturnAll* (page 168),
  - *InvestmentVariableInternalRateReturnInPeriodic* (page 170).

InvestmentVariableInternalRateReturnAll

The procedure *InvestmentVariableInternalRateReturnAll* (page 168) returns the internal rate of return for an investment based on a series of periodic cash flows. The internal rate of return is the rate received for an investment consisting of payments (negative values) and income (positive values).

```
InvestmentVariableInternalRateReturnAll(
    Value,                  ! (input) one-dimensional numerical expression
    Mode,                   ! (input) numerical expression
    NumberSolutions,        ! (output) numerical expression
    IRR                     ! (output) one-dimensional numerical expression
    )
```

Arguments

Value The periodic payments (positive or negative), which must be equally spaced in time. The order of the payments in *Value* must be the same as the order in which the cash flows occur. *Value* is an one dimensional parameter of real numbers. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts. em Value must contain at least one positive and at least one negative number.

Mode Indicates whether all the solutions need to be found or just one. $Mode = 0$: the search for solutions stops after one solution is found. $Mode = 1$: the search for solutions continues till all solutions are found.

NumberSolutions The number of solutions found. When $Mode = 0$ the *NumberSolutions* will be 1.

IRR The internal rate of return for the investment. There is not always a unique solution for *IRR*. Dependent on *Mode* one solution or all the solutions will be given. Solutions smaller than -1 are not supposed to be relevant, so the search for solutions is limited to the area greater than -1.

Equation

The internal rate of return r is a solution of the equation

$$\sum_{i=1}^{n} \frac{p_i}{(1+r)^i} = 0$$

where p_i are the periodic payments.

Note:

- The internal rate of return is the interest rate at which the investment has a zero net present value.
- When you want to use this procedure in an objective function or constraint you have to use *InvestmentVariableInternalRateReturn*.
- The function *InvestmentVariableInternalRateReturnAll* (page 168) is similar to the Excel function IRR.

Example

Often, internal rate of return is an investment first, (period 0, negative value), and then return cashflows in following periods.

```
_s_periods := ElementRange(0,4);
_p_val('0') := -100 ;
_p_val('1') := 50 ;
_p_val(_i_per | _i_per > '1') := _p_val(_i_per-1) * 1.5 ;
_s_sols := ElementRange(1,5);
_p_sols := 0 ;
InvestmentVariableInternalRateReturnAll(
    value           :  _p_val,
    Mode            :  1,
    NumberSolutions :  _p_sols,
    IRR             :  _p_irr(_i_sol) );

block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_val, _p_sols, _p_irr ;
endblock ;
```

This results in the following IRR:

```
_p_val := data
{ 0 : -100.000000,
  1 :   50.000000,
  2 :   75.000000,
  3 :  112.500000,
  4 :  168.750000 } ;

_p_sols := 1 ;


_p_irr := data
{ 1 : 0.688847 } ;
```

References

- The functions *InvestmentVariableInternalRateReturn* (page 166),
- *InvestmentVariableInternalRateReturnInPeriodic* (page 170).

InvestmentVariableInternalRateReturnInPeriodic

The function *InvestmentVariableInternalRateReturnInPeriodic* (page 170) returns the internal rate of return for an investment based on a series of in-periodic cash flows. The internal rate of return is the interest rate received for an investment. This function uses the procedure InvestmentVariableInternalRateReturnInPeriodicAll to determine all possible internal rates and returns the internal rate that is within the specified bounds.

```
InvestmentVariableInternalRateReturnInPeriodic(
    Value,                    ! (input) one-dimensional numerical expression
    Date,                     ! (input) one-dimensional string expression
    [Basis,]                  ! (optional) numerical expression
    [LowerBound,]             ! (optional) numerical expression
    [UpperBound,]             ! (optional) numerical expression
    [Error]                   ! (optional) numerical expression
    )
```

Arguments

Value The periodic payments (positive or negative), which must be equally spaced in time. The order of the payments in *Value* must be the same as the order in which the cash flows occur. *Value* is an one dimensional parameter of real numbers. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts. em Value must contain at least one positive and at least one negative number.

Date The dates on which the payments occur. *Date* and *Value* must have the same order. *Date* is an one-dimensional parameter of dates given in a date format. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they may occur in any order. *Date* should contain as many dates as the number of values given by *Value*.

Basis The day-count basis method to be used. The default is 1.

LowerBound Indicates a minimum for the internal rate to be accepted by this function. The default is -1.

UpperBound Indicates a maximum for the internal rate to be accepted by this function. The default is 5.

Error Indicates whether AIMMS should give an error if multiple solutions are found that satisfy the bounds. $Error = 0$: if multiple solutions are found, return the solution with the smallest absolute value. $Error = 1$: if multiple solutions are found, return an error message. The default is 0.

Return Value

The function *InvestmentVariableInternalRateReturnInPeriodic* (page 170) returns the internal rate of return for an investment based on a series of in-periodic cash flows. The internal rate of return is the interest rate received for an investment.

Note:

- The function *InvestmentVariableInternalRateReturnInPeriodic* (page 170) can be used in an objective function or constraint. The input parameter *Value* can be used as a variable.
- The function *InvestmentVariableInternalRateReturnInPeriodic* (page 170) is similar to the Excel function XIRR.

Example

Often, internal rate of return is an investment first, (period 0, negative value), and then return cashflows in following periods.

```
_s_periods := ElementRange(0,4);
_p_val('0') := -100 ;
_p_val('1') := 25 ;
_p_val(_i_per | _i_per > '1') := _p_val(_i_per-1) * 1.1 ;
_sp_startQuarter(_i_per) := MomentToString(
    Format        :  "%c%y-%m-%d",
    unit          :  [month],
    ReferenceDate :  "2024-01-01",
    Elapsed       :  ((ord(_i_per)-1)*3)[month]);

_p_irr := InvestmentVariableInternalRateReturnInPeriodic(
    value      :  _p_val,
    date       :  _sp_startQuarter,
    LowerBound :  -1,
    UpperBound :  5,
    Error      :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display { _p_val, _sp_startQuarter }, _p_irr ;
endblock ;
```

This results in the following IRR:

```
Composite table:
    _i_per         _p_val  _sp_startQuarter
!   ------    -----------  ----------------
          0    -100.000000  "2024-01-01"
          1      25.000000  "2024-04-01"
          2      27.500000  "2024-07-01"
          3      30.250000  "2024-10-01"
          4      33.275000  "2025-01-01"
    ;

    _p_irr := 0.258815 ;
```

References

- The functions *InvestmentVariableInternalRateReturn* (page 166),
- *InvestmentVariableInternalRateReturnInPeriodicAll* (page 172).
- Day count basis *methods* (page 122).

InvestmentVariableInternalRateReturnInPeriodicAll

The procedure *InvestmentVariableInternalRateReturnInPeriodicAll* (page 172) returns the internal rate of return for an investment based on a series of in-periodic cash flows. The internal rate of return is the interest rate received for an investment.

```
InvestmentVariableInternalRateReturnInPeriodicAll(
    Value,                  ! (input) one-dimensional numerical expression
    Date,                   ! (input) one-dimensional string expression
    Mode,                   ! (input) numerical expression
    IRR,                    ! (output) one-dimensional numerical expression
    NumberOfSolutions,      ! (output) numerical expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

Value The payments (positive or negative). *Value* is an one-dimensional parameter of real numbers. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts. *Value* must contain at least one positive and at least one negative number.

Date The dates on which the payments occur. *Date* and *Value* must have the same order. *Date* is an one-dimensional parameter of dates given in a date format. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they may occur in any order. *Date* should contain as many dates as the number of values given by *Value*.

Mode Indicates whether all the solutions need to be found or just one. $Mode = 0$: the search for solutions stops after one solution is found. $Mode = 1$: the search for solutions continues till all solutions are found.

IRR The internal rate of return for the investment. There is not always a unique solution for *IRR*. Dependent on *Mode* one solution or all the solutions will be given. Solutions smaller than -1 are not supposed to be relevant, so the search for solutions is limited to the area greater than -1.

NumberSolutions The number of solutions found. When $Mode = 0$ the *NumberSolutions* will be 1.

Basis The day-count basis method to be used. The default is 1.

Equation

The internal rate of return r is a solution of the equation

$$\sum_{i=1}^{n} \frac{p_i}{(1+r)^{f_i}} = 0$$

where p_i are the periodic payments, and f_i is the difference between date i and the first date (so, $f_1 = 0$), according to the selected day-count basis method.

Note:

- When you want to use the procedure in an objective function or constraint you have to use `InvestmentVariableInternalRateReturnInPeriodic`.
- The procedure *InvestmentVariableInternalRateReturnInPeriodicAll* (page 172) is similar to the Excel function XIRR.

Example

Often, internal rate of return is an investment first, (period 0, negative value), and then return cashflows in following periods.

```
_s_periods := ElementRange(0,4);
_p_val('0') := -100 ;
_p_val('1') := 25 ;
_p_val(_i_per | _i_per > '1') := _p_val(_i_per-1) * 1.1 ;
_sp_startQuarter(_i_per) := MomentToString(
    Format        :  "%c%y-%m-%d",
    unit          :  [month],
    ReferenceDate :  "2024-01-01",
    Elapsed       :  ((ord(_i_per)-1)*3)[month]);
_s_sols := ElementRange(1,5);

InvestmentVariableInternalRateReturnInPeriodicAll(
    value             :  _p_val,
    date              :  _sp_startQuarter,
    Mode              :  1,
    NumberOfSolutions :  _p_sols,
    IRR               :  _p_irr(_i_sol) );
block where single_column_display := 1, listing_number_precision := 6 ;
    display { _p_val, _sp_startQuarter }, _p_sols, _p_irr ;
endblock ;
```

This results in the following IRR:

```
Composite table:
    _i_per         _p_val  _sp_startQuarter
!   ------    -----------  ----------------
         0    -100.000000  "2024-01-01"
         1      25.000000  "2024-04-01"
         2      27.500000  "2024-07-01"
```

(continues on next page)

(continued from previous page)

```
        3       30.250000  "2024-10-01"
        4       33.275000  "2025-01-01"
    ;

    _p_sols := 1 ;


    _p_irr := data
    { 1 : 0.258815 } ;
```

References

- The functions *InvestmentVariableInternalRateReturn* (page 166),
- *InvestmentVariableInternalRateReturnInPeriodic* (page 170).
- Day count basis *methods* (page 122).

InvestmentVariableInternalRateReturnModified

The function *InvestmentVariableInternalRateReturnModified* (page 174) returns the modified internal rate of return for an investment based on a series of periodic cash flows. It considers both the cost made for the investment and the interest received on the reinvestment of cash flows.

```
InvestmentVariableInternalRateReturnModified(
    Value,                  ! (input) one-dimensional numerical expression
    FinanceRate,            ! (input) numerical expression
    ReinvestRate            ! (input) numerical expression
    )
```

Arguments

Value The periodic payments (positive or negative), which must be equally spaced in time. The order of the payments in *Value* must be the same as the order in which the cash flows occur. *Value* is an one dimensional parameter of real numbers. *Value* given by positive numbers represent incoming amounts and *Value* given by negative numbers represent outgoing amounts. *Value* must contain at least one positive and at least one negative number.

FinanceRate Interest rate you pay on money used in negative cash flows. *FinanceRate* must be a numerical expression in the range $[-1, \infty)$.

ReinvestRate Interest rate you receive on the positive cash flows as you reinvest them. *ReinvestRate* must be a numerical expression in the range $[-1, \infty)$.

Return Value

The function *InvestmentVariableInternalRateReturnModified* (page 174) returns the modified internal rate of return for the investment.

Equation

The internal rate of return r is the solution of the equation

$$(1+r)^{n-1} = -\frac{\text{NPV}(v^+, r_r)(1+r_r)^n}{\text{NPV}(v^-, r_f)(1+r_f)}$$

where n is the number of periods considered, $v_i = v_i^+ - v_i^-$ (with $v_i^+, v_i^- \geq 0$), r_f the finance rate, r_r the reinvestment rate, and NPV the function *InvestmentVariablePresentValue* (page 161).

Note:

- This function can be used in an objective function or constraint and the input parameters *Value*, *FinanceRate* and *ReinvestRate* can be used as a variable.
- There should be at least one negative and one positive *Value*.
- The function *InvestmentVariableInternalRateReturnModified* (page 174) is similar to the Excel function *MIRR <https://support.microsoft.com/en-us/office/mirr-function-28b62fff-b057-47ee-9ff9-13ea2628a007>_*.

Example

Often, internal rate of return is an investment first, (period 0, negative value), and then return cashflows in following periods.

```
_s_periods := ElementRange(0,4);
_p_val('0') := -100 ;
_p_val('1') := 50 ;
_p_val(_i_per | _i_per > '1') := _p_val(_i_per-1) * 1.5 ;
_p_irr := InvestmentVariableInternalRateReturnModified(
    value        :  _p_val,
    FinanceRate  :  0.07,
    ReinvestRate :  0.04 );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_val, _p_irr ;
endblock ;
```

This results in the following IRR:

```
_p_val := data
{ 0 : -100.000000,
  1 :   50.000000,
  2 :   75.000000,
  3 :  112.500000,
  4 :  168.750000 } ;

_p_irr := 0.434215 ;
```

References

- The function *InvestmentVariableInternalRateReturn* (page 166).

1.6.5 Securities

There are several types of securities, each with its own features and scheduled cash flows. Cash flows can be scheduled at the end of every coupon period or just at the end of the security's life. If we see a security as an investment, its yield can be viewed as the internal rate of return. The cash flows of a security can consists of periodic payments (equal to a certain percentage of the par value), the coupons, and the future value of the security. In general, the general cash flow equation

$$v_p(1+r)^N + p\sum_{i=1}^{N}(1+r)^{i-1} + v_f = 0$$

where v_p is the present value, v_f is the future value, N the number of periods, p is a constant periodic payment and r is the constant interest rate, holds. AIMMS provides functions the most common types of securities like treasury bills and bonds. However, the present value, future value, periodic payments, number of periods and interest rate are different for each specify security type.

Security Types

We distinguish three main types of securities:

- securities with zero coupon periods (discounted securities),
- securities with one coupon period (at maturity), and
- securities with multiple coupon periods

Discounted Securities

In the case of discounted (or zero coupon) securities such as treasury bills, there are no periodical payments. The only positive cash flow is a fixed redemption at the end of the security's life. Therefore, only the value of this redemption and the investment made for the security determine its yield. In this case, the present value is equal to the price $-P$, the price at which the security is bought at the settlement date, there 0 periods (so no periodic payments), and the future value at the maturity date is equal to the redemption R. Thus the general cash flow equation reduces to

$$-P(1 + r_y f_{SM}) + R = 0$$

where r_y is the annual yield of the security, and f_{SM} is the difference (in fractions of years) between the settlement and maturity date, computed with respect to the specified day count basis *method* (page 122).

Discount Rate

Commonly with discounted securities, the yield is not expressed in terms of the price, but in terms of the fixed redemption. The discount rate is the increase in value per year as a percentage of the redemption. The relationship between the yield r_y and the discount rate r_d is given by

$$1 + r_y f_{SM} = \frac{1}{1 - r_d f_{SM}}$$

which leads to the following equivalent relation between price and redemption

$$-P + R(1 - r_d f_{SM}) = 0$$

Treasury Bills

A treasury bill is a discounted security with less than one year from settlement until maturity, the number of days in one year is fixed at 360 and redemption is fixed at 100.

Functions for Discounted Securities

AIMMS supports the following functions for securities with zero coupon periods:

- *SecurityDiscountedPrice* (page 179)
- *SecurityDiscountedRedemption* (page 180)
- *SecurityDiscountedYield* (page 182)
- *SecurityDiscountedRate* (page 183)
- *TreasuryBillPrice* (page 184)
- *TreasuryBillYield* (page 185)
- *TreasuryBillBondEquivalent* (page 187)

One-Coupon Securities

Securities that only pay interest at maturity can be seen as securities with only one coupon period, where the accrued interest increases linearly in time until it is paid (when the security expires), and the redemption equals the par value of the security. In the general cash flow equation,

- the present value

  $$v_p = -P - v_{par} r_c f_{IS},$$

  where P is the price of the account at settlement and f_{IS} is the difference between the issue and settlement date (in fraction of years) with respect to the specified day count basis *method* (page 122), to account for the accrued interest from the issue date until settlement,

- the periodic payment

  $$p = v_{par} r_y f_{IM},$$

  where r_y is the annual yield and f_{IM} is the difference between the issue and maturity date (in fraction of years) with respect to the specified day count basis *method* (page 122), and

- the interest rate

  $$r = r_y f_{SM},$$

  where f_{SM} is the difference between the settlement and maturity date (in fraction of years) with respect to the specified day count basis *method* (page 122).

This results in the following equation for securities with one coupon period:

$$(-P - v_{par} r_c f_{IS})(1 + r_y f_{SM}) + v_{par} r_y f_{IM} + v_{par} = 0$$

Functions for One-Coupon Securities

AIMMS supports the following functions for securities with one coupon period:

- *SecurityMaturityPrice* (page 188)
- *SecurityMaturityCouponRate* (page 189)
- *SecurityMaturityYield* (page 191)
- *SecurityMaturityAccruedInterest* (page 192)

Multi-Coupon Securities

For securities with multiple coupon periods, interest will be accrued linearly during and paid at the end of each coupon period (i.e. at the coupon date). In the general cash flow equation

- the number of periods

  $$N = \lceil f f_{SM} \rceil,$$

  where f is the coupon frequency (number of coupon periods per year), and f_{SM} the difference between settlement and maturity date (in fraction of years) with respect to the specified day count basis *method* (page 122),

- the present value

  $$v_p = -P - v_{par} \frac{r_c}{f} \frac{f_{PS}}{f_{PN}},$$

  where P is the price of the security at settlement, v_{par} the par value of the security, r_c the annual coupon rate, f_{PS} the difference (in fraction of years) between the previous coupon and settlement date, and f_{PN} the difference between the previous and next coupon date, both with respect to the specified day count basis *method* (page 122),

- the periodic payment

  $$p = v_{par} \frac{r_c}{f}$$

- the interest rate

  $$r = \frac{r_y}{f},$$

  where r_y is the annual yield.

This results in the following equation for securities with multiple coupon periods:

$$\left(-P - v_{par} \frac{r_c}{f} \frac{f_{PS}}{f_{PN}}\right)^{N-1+\frac{f_{SN}}{f_{PN}}} + \sum_{i=1}^{N} v_{par} \frac{r_c}{f} \left(1 + \frac{r_y}{f}\right)^{N-i} + R = 0$$

Functions for Multi-Coupon Securities

AIMMS supports the following functions for securities with multiple coupon periods:

- *SecurityCouponNumber* (page 193)
- *SecurityCouponPreviousDate* (page 195)
- *SecurityCouponNextDate* (page 196)

- *SecurityCouponDays* (page 197)
- *SecurityCouponDaysPreSettlement* (page 198)
- *SecurityCouponDaysPostSettlement* (page 199)
- *SecurityPeriodicPrice* (page 200)
- *SecurityPeriodicRedemption* (page 202)
- *SecurityPeriodicCouponRate* (page 204)
- *SecurityPeriodicYieldAll* (page 205)
- *SecurityPeriodicYield* (page 207)
- *SecurityPeriodicAccruedInterest* (page 209)
- *SecurityPeriodicDuration* (page 210)
- *SecurityPeriodicDurationModified* (page 212)

SecurityDiscountedPrice

The function *SecurityDiscountedPrice* (page 179) returns the price of a discounted security at settlement date.

```
SecurityDiscountedPrice(
    SettlementDate,   ! (input) scalar string expression
    MaturityDate,     ! (input) scalar string expression
    Redemption,       ! (input) numerical expression
    DiscountRate,     ! (input) numerical expression
    [Basis]           ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Redemption The amount repaid at maturity date. *Redemption* must be a positive real number.

DiscountRate The rate the security's value increases per year as a fraction of the redemption value. *DiscountRate* must be a positive real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityDiscountedPrice* (page 179) returns the price of the security at settlement date.

Note:

- This function can be used in an objective function or constraint and the input parameters *Redemption* and *DiscountRate* can be used as a variable.
- The function *SecurityDiscountedPrice* (page 179) is similar to the Excel function *PRICEDISC <https://support.microsoft.com/en-us/office/pricedisc-function-d06ad7c1-380e-4be7-9fd9-75e3079acfd3>_*.

Example

Selling a security one year before maturity with a discount rate of 10%:

```
_p_sdp := SecurityDiscountedPrice(
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2025-01-01",
    Redemption     :  100,
    DiscountRate   :  0.1,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_sdp ;
endblock ;
```

This results in the following discounted price:

```
_p_sdp := 90 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 176) for discounted securities.

SecurityDiscountedRedemption

The function *SecurityDiscountedRedemption* (page 180) returns the repayment at maturity date of a discounted security.

```
SecurityDiscountedRedemption(
    SettlementDate,    ! (input) scalar string expression
    MaturityDate,      ! (input) scalar string expression
    Price,             ! (input) numerical expression
    DiscountRate,      ! (input) numerical expression
    [Basis]            ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Price The price of the security at settlement date. *Price* must be a positive real number.

DiscountRate The rate the security's value increases per year as a fraction of the redemption value. *DiscountRate* must be a positive real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityDiscountedRedemption* (page 180) returns the amount paid at maturity date.

Note:

- This function can be used in an objective function or constraint and the input parameters *Price* and *DiscountRate* can be used as a variable.
- The function *SecurityDiscountedRedemption* (page 180) is similar to the Excel function RECEIVED.

Example

Buying a security now for a given price, what will be the maturity value next year with a discount rate of 10%:

```
_p_sdr := SecurityDiscountedRedemption(
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2025-01-01",
    Price          :  90,
    DiscountRate   :  0.1,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_sdr ;
endblock ;
```

This results in the following redemption value:

```
_p_sdr := 100 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 176) for discounted securities.

SecurityDiscountedYield

The function *SecurityDiscountedYield* (page 182) returns the yield of a discounted security at maturity date.

```
SecurityDiscountedYield(
    SettlementDate,  ! (input) scalar string expression
    MaturityDate,    ! (input) scalar string expression
    Price,           ! (input) numerical expression
    Redemption,      ! (input) numerical expression
    [Basis]          ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Price The price of the security at settlement date. *Price* must be a positive real number.

Redemption The amount repaid at maturity date. *Redemption* must be a positive real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityDiscountedYield* (page 182) returns the annual rate the security's value increases as a fraction of the price.

Note:

- This function can be used in an objective function or constraint and the input parameters *Price* and *Redemption* can be used as a variable.
- The function *SecurityDiscountedYield* (page 182) is similar to the Excel function YIELDDISC.

Example

Knowing price, redemption, and length period; what was the interest (with respect to price)?

```
_p_sdf := SecurityDiscountedYield(
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2025-01-01",
    Price          :  90,
    Redemption     :  100,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_sdf ;
endblock ;
```

This results in the following interest:

```
_p_sdf := 0.111111 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 176) for discounted securities.

SecurityDiscountedRate

The function *SecurityDiscountedRate* (page 183) returns the discount rate of a discounted security.

```
SecurityDiscountedRate(
    SettlementDate,  ! (input) scalar string expression
    MaturityDate,    ! (input) scalar string expression
    Price,           ! (input) numerical expression
    Redemption,      ! (input) numerical expression
    [Basis]          ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Price The price of the security at settlement date. *Price* must be a positive real number.

Redemption The amount repaid at maturity date. *Redemption* must be a positive real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityDiscountedRate* (page 183) returns the annual rate the security's value increases as a fraction of the redemption value.

Note:

- This function can be used in an objective function or constraint and the input parameters *Price* and *Redemption* can be used as a variable.
- The function *SecurityDiscountedRate* (page 183) is similar to the Excel function DISC.

Example

Knowing price, redemption, and length period; what was the interest (with respect to redemption)?

```
_p_sdr := SecurityDiscountedRate(
        SettlementDate :  "2024-01-01",
        MaturityDate   :  "2025-01-01",
        Price          :  90,
        Redemption     :  100,
        Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_sdr ;
endblock ;
```

This results in the following interest:

```
_p_sdr := 0.100000 ;
```

References

- Day count basis *methods* (page 122).
  - General *equations* (page 176) for discounted securities.

TreasuryBillPrice

The function *TreasuryBillPrice* (page 184) returns the price of a Treasury bill at settlement date. A Treasury bill is a discounted security with less than one year from settlement until maturity, the number of days in one year is fixed at 360 and redemption is fixed at 100.

```
TreasuryBillPrice(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    DiscountRate             ! (input) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

DiscountRate The discount rate of the security as a fraction of the redemption. *DiscountRate* must be a positive real number.

Return Value

The function *TreasuryBillPrice* (page 184) returns the price of a Treasury bill at settlement date.

Note:

- This function can be used in an objective function or constraint and the input parameter *DiscountRate* can be used as a variable.
- The function *TreasuryBillPrice* (page 184) is similar to the Excel function TBILLPRICE.

Example

Half a year, 10%, what does this do?

```
_p_tbp := TreasuryBillPrice(
        SettlementDate :  "2024-07-01",
        MaturityDate   :  "2025-01-01",
        DiscountRate   :  0.1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_tbp ;
endblock ;
```

Well:

```
_p_tbp := 94.888889 ;
```

References

- General *equations* (page 176) for discounted securities.

TreasuryBillYield

The function *TreasuryBillYield* (page 185) returns the yield of a Treasury bill at settlement date. A Treasury bill is a discounted security with less than one year from settlement until maturity, the number of days in one year is fixed at 360 and redemption is fixed at 100.

```
TreasuryBillYield(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Price                    ! (input) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Price The price the security is worth at this moment. *Price* must be a positive real number.

Return Value

The function *TreasuryBillYield* (page 185) returns the annual rate the Treasury bill's value increases as a percentage of the price.

Note:

- This function can be used in an objective function or constraint and the input parameter *Price* can be used as a variable.
- The function *TreasuryBillYield* (page 185) is similar to the Excel function TBILLYIELD.

Example

What is the yield if a bond of 100 is bought for 95 and matures in half a year?

```
_p_tby := TreasuryBillYield(
        SettlementDate :  "2024-07-01",
        MaturityDate   :  "2025-01-01",
        Price          :  95);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_tby ;
endblock ;
```

Well:

```
_p_tby := 0.102975 ;
```

References

- General *equations* (page 176) for discounted securities.

TreasuryBillBondEquivalent

The function *TreasuryBillBondEquivalent* (page 187) returns the bond equivalent yield of a treasury bill. A Treasury bill is a discounted security with less than one year from settlement until maturity, the number of days in one year is fixed at 360 and redemption is fixed at 100.

```
TreasuryBillBondEquivalent(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    DiscountRate             ! (input) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be given in a date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

DiscountRate The discount rate of the security as a percentage of the redemption. *DiscountRate* must be a positive real number.

Return Value

The function *TreasuryBillBondEquivalent* (page 187) returns the bond equivalent yield of a Treasury bill.

Note:

- This function can be used in an objective function or constraint and the input parameter *DiscountRate* can be used as a variable.
- The function *TreasuryBillBondEquivalent* (page 187) is similar to the Excel function *TBILLEQ <https://support.microsoft.com/en-us/office/tbilleq-function-2ab72d90-9b4d-4efe-9fc2-0f81f2c19c8c>*.

Example

Bond equivalent yield for a treasury bill, half a year out, 10%:

```
_p_tbbe := TreasuryBillBondEquivalent(
        SettlementDate :  "2024-07-01",
        MaturityDate   :  "2025-01-01",
        DiscountRate   :  0.1);
block where single_column_display := 1, listing_number_precision := 6 ;
```

(continues on next page)

(continued from previous page)

```
        display _p_tbbe ;
endblock ;
```

Well:

```
_p_tbbe := 0.106850 ;
```

References

- General *equations* (page 176) for discounted securities.

SecurityMaturityPrice

The function *SecurityMaturityPrice* (page 188) returns the price at settlement date of a security that pays interest at maturity.

```
SecurityMaturityPrice(
    IssueDate,                ! (input) scalar string expression
    SettlementDate,           ! (input) scalar string expression
    MaturityDate,             ! (input) scalar string expression
    ParValue,                 ! (input) numerical expression
    CouponRate,               ! (input) numerical expression
    Yield,                    ! (input) numerical expression
    [Basis]                   ! (optional) numerical expression
    )
```

Arguments

IssueDate The date of issue of the security. *IssueDate* must be given in date format.

SettlementDate The date of settlement of the security. *SettlementDate* must also be in date format and must be a date after *IssueDate*.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityMaturityPrice* (page 188) returns the price of the security at settlement date.

Note:

- This function can be used in an objective function or constraint and the input parameters *ParValue*, *CouponRate*, and *Yield* can be used as a variable.
- The function *SecurityMaturityPrice* (page 188) is similar to the Excel function *PRICEMAT <https://support.microsoft.com/en-us/office/pricemat-function-52c3b4da-bc7e-476a-989f-a95f675cae77>_*.

Example

The code

```
_p_smp := SecurityMaturityPrice(
    IssueDate      :  "2020-01-01",
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2025-01-01",
    ParValue       :  100,
    CouponRate     :  0.059,
    Yield          :  0.061,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_smp ;
endblock ;
```

This results in:

```
_p_smp := 98.454665 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 177) for securities with one coupon.

SecurityMaturityCouponRate

The function *SecurityMaturityCouponRate* (page 189) returns the coupon rate of a security that pays interest at maturity.

```
SecurityMaturityCouponRate(
    IssueDate,                ! (input) scalar string expression
    SettlementDate,           ! (input) scalar string expression
    MaturityDate,             ! (input) scalar string expression
    ParValue,                 ! (input) numerical expression
    Price,                    ! (input) numerical expression
```

(continues on next page)

(continued from previous page)

```
    Yield,                  ! (input) numerical expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

IssueDate The date of issue of the security. *IssueDate* must be given in date format.

SettlementDate The date of settlement of the security. *SettlementDate* must also be in date format and must be a date after *IssueDate*.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Price The price of the security at settlement date. *Price* must be a positive real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityMaturityCouponRate* (page 189) returns the annual interest rate of the security as a percentage of the par value.

Note: This function can be used in an objective function or constraint and the input parameters *ParValue*, *Price*, and *Yield* can be used as a variable.

Example

The code:

```
_p_smcr := SecurityMaturityCouponRate(
        IssueDate      :  "2020-01-01",
        SettlementDate :  "2024-01-01",
        MaturityDate   :  "2025-01-01",
        ParValue       :  100,
        Price          :  99,
        Yield          :  0.02,
        Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_smcr ;
endblock ;
```

Produces in the listing file:

```
_p_smcr := 0.010652 ;
```

References

- Day count basis *methods* (page 122).
  - General *equations* (page 177) for securities with one coupon.

SecurityMaturityYield

The function `SecurityMaturityYield` (page 191) returns the yield of a security that pays interest at maturity.

```
SecurityMaturityYield(
    IssueDate,              ! (input) scalar string expression
    SettlementDate,         ! (input) scalar string expression
    MaturityDate,           ! (input) scalar string expression
    ParValue,               ! (input) numerical expression
    Price,                  ! (input) numerical expression
    CouponRate,             ! (input) numerical expression
    [Basis]                 ! (optional) numerical expression
    )
```

Arguments

IssueDate The date of issue of the security. *IssueDate* must be given in date format.

SettlementDate The date of settlement of the security. *SettlementDate* must also be in date format and must be a date after *IssueDate*.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Price The price of the security at settlement date. *Price* must be a positive real number.

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function `SecurityMaturityYield` (page 191) returns the annual rate the security's value increases as a percentage of the price.

Note:

- This function can be used in an objective function or constraint and the input parameters *ParValue*, *Price*, and *CouponRate* can be used as a variable.
- The function `SecurityMaturityYield` (page 191) is similar to the Excel function YIELDMAT.

Example

The code:

```
_p_smy := SecurityMaturityYield(
    IssueDate      :  "2020-01-01",
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2025-01-01",
    ParValue       :  100,
    Price          :  99,
    CouponRate     :  0.05,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_smy ;
endblock ;
```

Produces in the listing file:

```
_p_smy := 0.050420 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 177) for securities with one coupon.

SecurityMaturityAccruedInterest

The function *SecurityMaturityAccruedInterest* (page 192) returns the accrued interest for a security that pays interest at maturity.

```
SecurityMaturityAccruedInterest(
    IssueDate,       ! (input) scalar string expression
    SettlementDate,  ! (input) scalar string expression
    ParValue,        ! (input) numerical expression
    CouponRate,      ! (input) numerical expression
    [Basis]          ! (optional) numerical expression
    )
```

Arguments

IssueDate The date of issue of the security. *IssueDate* must be given in date format.

SettlementDate The date of settlement of the security. *SettlementDate* must also be in date format and must be a date after *IssueDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

CouponRate The annual interest rate of the security as a fraction of the par value. *CouponRate* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityMaturityAccruedInterest* (page 192) returns the interest accrued from issue date until settlement date.

Note:

- This function can be used in an objective function or constraint and the input parameters *CouponRate* and *ParValue* can be used as a variable.
- The function *SecurityMaturityAccruedInterest* (page 192) is similar to the Excel function *ACCRINTM <https://support.microsoft.com/en-us/office/accrintm-function-f62f01f9-5754-4cc4-805b-0e70199328a7>_*.

Example

How much interest was paid for a security

- over a period of five years,
- with an interest rate of 7%,
- and bought for a 100 at the time?

```
_p_smai := SecurityMaturityAccruedInterest(
    IssueDate      :  "2020-01-01",
    SettlementDate :  "2025-01-01",
    ParValue       :  100,
    CouponRate     :  0.07,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_smai ;
endblock ;
```

No interest on interest, so:

```
_p_smai := 35 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 177) for securities with one coupon.

SecurityCouponNumber

The function *SecurityCouponNumber* (page 193) returns the number of coupons from settlement date and maturity date of a security that pays interest at the end of each coupon period.

```
SecurityCouponNumber(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Frequency,               ! (input) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

Return Value

The function *SecurityCouponNumber* (page 193) returns the number of coupon payments from the settlement date until the maturity date.

Note: The function *SecurityCouponNumber* (page 193) is similar to the Excel function *COUPNUM <https://support.microsoft.com/en-us/office/coupnum-function-a90af57b-de53-4969-9c99-dd6139db2522>_*.

Example

How many quarterly coupons are paid out over a period of five years?

```
_p_scn := SecurityCouponNumber(
        SettlementDate :  "2025-01-01",
        MaturityDate   :  "2030-01-01",
        Frequency      :  4);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_scn ;
endblock ;
```

Twenty:

```
_p_scn := 20 ;
```

References

- Day count basis *methods* (page 122).
  - General *equations* (page 178) for securities with multiple coupons.

SecurityCouponPreviousDate

The function *SecurityCouponPreviousDate* (page 195) returns the last coupon-date previous to settlement date of a security that pays interest at the end of each coupon period.

```
SecurityCouponPreviousDate(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Frequency                ! (input) numerical expression
    PreviousDate             ! (output) string parameter
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

PreviousDate The date on which the coupon period, in which the settlement date falls, starts and on which the previous coupon period ends.

Note: The function *SecurityCouponPreviousDate* (page 195) is similar to the Excel function *COUPPCD <https://support.microsoft.com/en-us/office/couppcd-function-2eb50473-6ee9-4052-a206-77a9a385d5b3>_*.

Example

When was the last coupon issued, when the settlement date is Feb 1, and the coupons are issued quarterly until Jan 1?

```
SecurityCouponPreviousDate(
        SettlementDate :  "2025-02-01",
        MaturityDate   :  "2030-01-01",
        Frequency      :  4,
        PreviousDate   :  _sp_tcpd);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _sp_tcpd ;
endblock ;
```

Here it coincides with the start of every quarter, which is also Jan 1:

```
_sp_tcpd := "2025-01-01" ;
```

References

- General *equations* (page 178) for securities with multiple coupons.

SecurityCouponNextDate

The function *SecurityCouponNextDate* (page 196) returns the first coupon-date next to settlement date of a security that pays interest at the end of each coupon period.

```
SecurityCouponNextDate(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Frequency                ! (input) numerical expression
    NextDate                 ! (output) string parameter
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

NextDate The date on which the coupon period ends and on which the next coupon period starts.

Note: The function *SecurityCouponNextDate* (page 196) is similar to the Excel function *COUPNCD <https://support.microsoft.com/en-us/office/coupncd-function-fd962fef-506b-4d9d-8590-16df5393691f>_*.

Example

When will the next coupon be issued, when the settlement date is Feb 1, and the coupons are issued quarterly until Jan 1?

```
SecurityCouponNextDate(
        SettlementDate :  "2025-02-01",
        MaturityDate   :  "2030-01-01",
        Frequency      :  4,
        NextDate       :  _sp_tcnd);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _sp_tcnd ;
endblock ;
```

Here it coincides with the start of every quarter, which is also Jan 1:

```
_sp_tcnd := "2025-04-01" ;
```

References

- General *equations* (page 178) for securities with multiple coupons.

SecurityCouponDays

The function *SecurityCouponDays* (page 197) returns the number of days of the coupon period in which settlement date falls. In other words the number of days from the last coupon-date previous to settlement date until the first coupon-date next to settlement date of a security that pays interest at the end of each coupon period.

```
SecurityCouponDays(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Frequency,               ! (input) numerical expression
    [Basis]                  ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityCouponDays* (page 197) returns the number of days of the coupon period in which the settlement date falls.

Note: The function *SecurityCouponDays* (page 197) is similar to the Excel function *COUPDAYS <https://support.microsoft.com/en-us/office/coupdays-function-cc64380b-315b-4e7b-950c-b30b0a76f671>_*.

Example

The length of the coupon period in which settlement day is:

```
_p_scd := SecurityCouponDays(
    SettlementDate :  "2025-02-01",
    MaturityDate   :  "2030-01-01",
    Frequency      :  4,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_scd ;
endblock ;
```

Day basis 1, just three months, each of 30 days:

```
_p_scd := 90 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityCouponDaysPreSettlement

The function *SecurityCouponDaysPreSettlement* (page 198) returns the number of days from the last coupon-date previous to settlement date until settlement date of a security that pays interest at the end of each coupon period.

```
SecurityCouponDaysPreSettlement(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Frequency,               ! (input) numerical expression
    [Basis]                  ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityCouponDaysPreSettlement* (page 198) returns the number of days from the previous coupon-date until the settlement date, using the specified day-count basis.

Note: The function *SecurityCouponDaysPreSettlement* (page 198) is similar to the Excel function *COUPDAYBS <https://support.microsoft.com/en-us/office/coupdaybs-function-eb9a8dfb-2fb2-4c61-8e5d-690b320cf872>*.

Example

The number of days since the last coupon issued up to the settlement date?

```
_p_scdps := SecurityCouponDaysPreSettlement(
    SettlementDate :  "2025-02-01",
    MaturityDate   :  "2030-01-01",
    Frequency      :  4,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_scdps ;
endblock ;
```

Day basis 1, just one months, so that is 30 days:

```
_p_scdps := 30 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityCouponDaysPostSettlement

The function *SecurityCouponDaysPostSettlement* (page 199) returns the number of days from the first coupon-date next to settlement date until settlement date of a security that pays interest at the end of each coupon period.

```
SecurityCouponDaysPostSettlement(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    Frequency,               ! (input) numerical expression
    [Basis]                  ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityCouponDaysPostSettlement* (page 199) returns the number of days from the first coupon-date next to settlement date until settlement date.

Note: The function *SecurityCouponDaysPostSettlement* (page 199) is similar to the Excel function *COUPDAYSNC <https://support.microsoft.com/en-us/office/coupdaysnc-function-5ab3f0b2-029f-4a8b-bb65-47d525eea547>*.

Example

The number of days from the settlement date up to the next coupon date:

```
_p_scdps := SecurityCouponDaysPostSettlement(
    SettlementDate :  "2025-02-01",
    MaturityDate   :  "2030-01-01",
    Frequency      :  4,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_scdps ;
endblock ;
```

Day basis 1, just two months, so that is 60 days:

```
_p_scdps := 60 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicPrice

The function *SecurityPeriodicPrice* (page 200) returns the price at settlement date of a security that pays interest at the end of each coupon period.

```
SecurityPeriodicPrice(
    SettlementDate,  ! (input) scalar string expression
    MaturityDate,    ! (input) scalar string expression
    ParValue,        ! (input) numerical expression
    Redemption,      ! (input) numerical expression
    Frequency,       ! (input) numerical expression
    CouponRate,      ! (input) numerical expression
    Yield,           ! (input) numerical expression
    [Basis]          ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Redemption The amount repaid for the security at the maturity date. *Redemption* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityPeriodicPrice* (page 200) returns the price of the security at settlement date.

Note:

- This function can be used in an objective function or constraint and the input parameters *ParValue*, *Redemption*, *CouponRate*, and *Yield* can be used as a variable.
- The function *SecurityPeriodicPrice* (page 200) is similar to the Excel function *PRICE <https://support.microsoft.com/en-us/office/price-function-3ea9deac-8dfa-436f-a7c8-17ea02c21b0a>*.

Example

The code:

```
_p_spp := SecurityPeriodicPrice(
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2030-01-01",
    ParValue       :  100,
    Redemption     :  100,
    Frequency      :  4,
    CouponRate     :  0.06,
    Yield          :  0.07,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_spp ;
endblock ;
```

Produces:

```
_p_spp := 95.134829 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicRedemption

The function *SecurityPeriodicRedemption* (page 202) returns the repayment at maturity date of a security that pays interest at the end of each coupon period.

```
SecurityPeriodicRedemption(
    SettlementDate, ! (input) scalar string expression
    MaturityDate,   ! (input) scalar string expression
    ParValue,       ! (input) numerical expression
    Price,          ! (input) numerical expression
    Frequency,      ! (input) numerical expression
    CouponRate,     ! (input) numerical expression
    Yield,          ! (input) numerical expression
    [Basis]         ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Price The price of the security at settlement date. *Price* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityPeriodicRedemption* (page 202) returns the amount repaid for the security at the maturity date.

Note: This function can be used in an objective function or constraint and the input parameters *ParValue*, *Price*, *CouponRate*, and *Yield* can be used as a variable.

Example

The code:

```
_p_spr := SecurityPeriodicRedemption(
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2030-01-01",
    ParValue       :  100,
    Price          :  100,
    Frequency      :  4,
    CouponRate     :  0.06,
    Yield          :  0.07,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_spr ;
endblock ;
```

Produces:

```
_p_spr := 107.377754 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicCouponRate

The function `SecurityPeriodicCouponRate` (page 204) returns the coupon rate of a security that pays interest at the end of each coupon period.

```
SecurityPeriodicCouponRate(
    SettlementDate,          ! (input) scalar string expression
    MaturityDate,            ! (input) scalar string expression
    ParValue,                ! (input) numerical expression
    Price,                   ! (input) numerical expression
    Redemption,              ! (input) numerical expression
    Frequency,               ! (input) numerical expression
    Yield,                   ! (input) numerical expression
    [Basis]                  ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Price The price of the security at settlement date. *Price* must be a positive real number.

Redemption The amount repaid for the security at the maturity date. *Redemption* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function `SecurityPeriodicCouponRate` (page 204) returns the interest rate per year of the security as a percentage of the par value.

Note: This function can be used in an objective function or constraint and the input parameters *ParValue*, *Price*, *Redemption*, and *Yield* can be used as a variable.

Example

The code:

```
_p_spcr := SecurityPeriodicCouponRate(
        SettlementDate :  "2024-01-01",
        MaturityDate   :  "2030-01-01",
        ParValue       :  100,
        Price          :  100,
        Redemption     :  100,
        Frequency      :  4,
        Yield          :  0.04,
        Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_spcr ;
endblock ;
```

Produces:

```
_p_spcr := 0.040000 ;
```

References

- Day count basis *methods* (page 122).
  - General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicYieldAll

The procedure *SecurityPeriodicYieldAll* (page 205) returns the yield(s) of a security that pays interest at the end of each coupon period.

```
SecurityPeriodicYieldAll(
    SettlementDate,     ! (input) scalar string expression
    MaturityDate,       ! (input) scalar string expression
    ParValue,           ! (input) numerical expression
    Price,              ! (input) numerical expression
    Redemption,         ! (input) numerical expression
    Frequency,          ! (input) numerical expression
    CouponRate,         ! (input) numerical expression
    Yield,              ! (output) one-dimensional numerical expression
    NumberSolutions,    ! (output) numerical expression
    [Basis,]            ! (optional) numerical expression
    [Mode]              ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Price The price of the security at settlement date. *Price* must be a positive real number.

Redemption The amount repaid for the security at the maturity date. *Redemption* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Yield There is not always a unique solution for yield. Dependent on *Mode* one solution or all the solutions will be given.

NumberSolutions The number of solutions found. If $Mode = 0$ *NumberSolutions* will always be 1.

Basis The day-count basis method to be used. The default is 1.

Mode Indicates whether all the solutions need to be found or just one. $Mode = 0$: the search for solutions stops after one solution is found. $Mode = 1$: the search for solutions continues till all solutions are found.

Note:

- When you want to use this procedure in an objective function or constraint you have to use `SecurityPeriodicYield`.
- The function *SecurityPeriodicYieldAll* (page 205) is similar to the Excel function *YIELD <https://support.microsoft.com/en-us/office/yield-function-f5f5ca43-c4bd-434f-8bd2-ed3c9727a4fe>_*.

Example

The code:

```
_s_sols := ElementRange(1,5);
SecurityPeriodicYieldAll(
    SettlementDate  :  "2024-01-01",
    MaturityDate    :  "2025-01-01",
    ParValue        :  100,
    Price           :  95,
    Redemption      :  100,
    Frequency       :  4,
    CouponRate      :  0.06,
    Yield           :  _p_yield,
    NumberSolutions :  _p_noYield,
    Basis           :  1,
```

(continues on next page)

(continued from previous page)

```
    Mode            :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_noYield, _p_yield ;
endblock ;
```

Produces:

```
_p_noYield := 1 ;


_p_yield := data
{ 1 : 0.113600 } ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicYield

The function *SecurityPeriodicYield* (page 207) returns the yield of a security that pays interest at the end of each coupon period. This function uses the procedure `SecurityPeriodicYieldAll` to determine all possible yields and returns the yield that is within the specified bounds.

```
SecurityPeriodicYield(
    SettlementDate,  ! (input) scalar string expression
    MaturityDate,    ! (input) scalar string expression
    ParValue,        ! (input) numerical expression
    Price,           ! (input) numerical expression
    Redemption,      ! (input) numerical expression
    Frequency,       ! (input) numerical expression
    CouponRate,      ! (input) numerical expression
    [Basis,]         ! (optional) numerical expression
    [LowerBound,]    ! (optional) numerical expression
    [UpperBound,]    ! (optional) numerical expression
    [Error]          ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Price The price of the security at settlement date. *Price* must be a positive real number.

Redemption The amount repaid for the security at the maturity date. *Redemption* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

LowerBound Indicates a minimum for the yield to be accepted by this function. The default is -1.

UpperBound Indicates a maximum for the yield to be accepted by this function. The default is 5.

Error Indicates whether AIMMS should give an error if multiple solutions are found that satisfy the bounds. $Error = 0$: if multiple solutions are found, return the solution with the smallest absolute value. $Error = 1$: if multiple solutions are found, return an error message. The default is 0.

Return Value

The function *SecurityPeriodicYield* (page 207) returns the yield of a security that pays interest at the end of each coupon period.

Note:

- This function can be used in an objective function or constraint and the input parameters *ParValue*, *Price*, *Redemption*, and *CouponRate* can be used as a variable.
- The function *SecurityPeriodicYield* (page 207) is similar to the Excel function *YIELD <https://support.microsoft.com/en-us/office/yield-function-f5f5ca43-c4bd-434f-8bd2-ed3c9727a4fe>_*.

Example

The code:

```
_p_spy := SecurityPeriodicYield(
    SettlementDate :  "2024-01-01",
    MaturityDate   :  "2025-01-01",
    ParValue       :  100,
    Price          :  95,
    Redemption     :  100,
    Frequency      :  4,
    CouponRate     :  0.06,
    Basis          :  1,
    LowerBound     :  -1,
    UpperBound     :  5,
    Error          :  0);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_spy ;
endblock ;
```

Produces:

```
_p_spy := 0.113600 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicAccruedInterest

The function `SecurityPeriodicAccruedInterest` (page 209) returns the accrued interest from the begin of the coupon period until the settlement date for a security that pays interest at the end of each coupon period.

```
SecurityPeriodicAccruedInterest(
    SettlementDate, ! (input) scalar string expression
    MaturityDate,   ! (input) scalar string expression
    ParValue,       ! (input) numerical expression
    Frequency,      ! (input) numerical expression
    CouponRate,     ! (input) numerical expression
    [Basis]         ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function `SecurityPeriodicAccruedInterest` (page 209) returns the interest accrued from the begin of the coupon period until settlement date.

Note: This function can be used in an objective function or constraint and the input parameters *ParValue* and *CouponRate* can be used as a variable.

Example

Security owned for half a year since last coupon date, against 4%, and value 100.

```
_p_spai := SecurityPeriodicAccruedInterest(
    SettlementDate :  "2024-07-01",
    MaturityDate   :  "2025-01-01",
    ParValue       :  100,
    Frequency      :  1,
    CouponRate     :  0.04,
    Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _p_spai ;
endblock ;
```

Results in an accrued interest of 2:

```
_p_spai := 2 ;
```

References

- Day count basis *methods* (page 122).
- General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicDuration

The function *SecurityPeriodicDuration* (page 210) returns the Macauley duration of a security that pays interest at the end of each coupon period. Duration is defined as the weighted average of time it takes to receive a positive cash flow. The present values of the cash flows are used as weights. The duration can be used as a measure of a bond price's response to changes in yield.

```
SecurityPeriodicDuration(
    SettlementDate, ! (input) scalar string expression
    MaturityDate,   ! (input) scalar string expression
    ParValue,       ! (input) numerical expression
    Redemption,     ! (input) numerical expression
    Frequency,      ! (input) numerical expression
    CouponRate,     ! (input) numerical expression
    Yield,          ! (input) numerical expression
    [Basis]         ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Redemption The amount repaid for the security at the maturity date. *Redemption* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function `SecurityPeriodicDuration` (page 210) returns the Macauley duration of a security that pays interest at the end of each coupon period. Duration is defined as the weighted average of the time it takes to receive a positive cash flow.

Equation

The Macauley duration D is computed through the equation

$$D = \frac{\left(N - 1 + \frac{f_{SN}}{f_{PN}}\right) \frac{R}{\left(1 + \frac{r_y}{f}\right)^{N-1+\frac{f_{SN}}{f_{PN}}}} + \sum_{i=1}^{N} \left(i - 1 + \frac{f_{SN}}{f_{PN}}\right) \frac{v_{par} \frac{r_c}{f}}{\left(1 + \frac{r_y}{f}\right)^{i-1+\frac{f_{SN}}{f_{PN}}}}}{\frac{R}{\left(1 + \frac{r_y}{f}\right)^{N-1+\frac{f_{SN}}{f_{PN}}}} + \sum_{i=1}^{N} \frac{v_{par} \frac{r_c}{f}}{\left(1 + \frac{r_y}{f}\right)^{i-1+\frac{f_{SN}}{f_{PN}}}}}$$

where all other variables have the same interpretation as in the general *equations* (page 178) for securities with multiple coupons.

Note:

- This function can be used in an objective function or constraint and the input parameters *ParValue*, *Redemption*, *CouponRate*, and *Yield* can be used as a variable.
- The function `SecurityPeriodicDuration` (page 210) is similar to the Excel function *DURATION <https://support.microsoft.com/en-us/office/duration-function-b254ea57-eadc-4602-a86a-c8e369334038>_*.

Example

The code:

```
_p_spd := SecurityPeriodicDuration(
        SettlementDate :  "2024-01-01",
        MaturityDate   :  "2044-01-01",
        ParValue       :  100,
        Redemption     :  100,
        Frequency      :  1,
        CouponRate     :  0.08,
        Yield          :  0.08,
        Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_spd ;
endblock ;
```

Results in:

```
_p_spd := 10.603599 ;
```

References

- Day count basis *methods* (page 122).
  - General *equations* (page 178) for securities with multiple coupons.

SecurityPeriodicDurationModified

The function *SecurityPeriodicDurationModified* (page 212) returns the modified Macauley duration of a security that pays interest at the end of each coupon period.

```
SecurityPeriodicDurationModified(
    SettlementDate, ! (input) scalar string expression
    MaturityDate,   ! (input) scalar string expression
    ParValue,       ! (input) numerical expression
    Redemption,     ! (input) numerical expression
    Frequency,      ! (input) numerical expression
    CouponRate,     ! (input) numerical expression
    Yield,          ! (input) numerical expression
    [Basis]         ! (optional) numerical expression
    )
```

Arguments

SettlementDate The date of settlement of the security. *SettlementDate* must be in date format.

MaturityDate The date of maturity of the security. *MaturityDate* must also be in date format and must be a date after *SettlementDate*.

ParValue The starting value of the security at issue date. *ParValue* must be a positive real number.

Redemption The amount repaid for the security at the maturity date. *Redemption* must be a positive real number.

Frequency The number of coupon payments in one year. *Frequency* must be 1 (annual), 2 (semi-annual) or 4 (quarterly).

CouponRate The annual interest rate of the security as a percentage of the par value. *CouponRate* must be a nonnegative real number.

Yield The yield of the security. *Yield* must be a nonnegative real number.

Basis The day-count basis method to be used. The default is 1.

Return Value

The function *SecurityPeriodicDurationModified* (page 212) returns the modified Macauley duration of a security that pays interest at the end of each coupon period.

Equation

The modified duration D_{mod} is computed through the equation

$$D_{mod} = \frac{D}{1 + \frac{r_y}{f}}$$

where D is the Macauley duration.

Note:

- This function can be used in an objective function or constraint and the input parameters *ParValue*, *Redemption*, *CouponRate*, and *Yield* can be used as a variable.
- The function *SecurityPeriodicDurationModified* (page 212) is similar to the Excel function MDURATION.

Example

The code:

```
_p_spdm := SecurityPeriodicDurationModified(
        SettlementDate :  "2024-01-01",
        MaturityDate   :  "2044-01-01",
        ParValue       :  100,
        Redemption     :  100,
```

(continues on next page)

(continued from previous page)

```
        Frequency      :  1,
        CouponRate     :  0.08,
        Yield          :  0.08,
        Basis          :  1);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _p_spdm ;
endblock ;
```

Results in:

```
_p_spdm := 9.818147 ;
```

References

- The function *SecurityPeriodicDuration* (page 210).
  - Day count basis *methods* (page 122).
  - General *equations* (page 178) for securities with multiple coupons.

1.7 Distribution and Combinatoric Functions

AIMMS supports several functions to obtain random numbers from discrete or continuous distribution, and additionally some combinatoric functions.

The functions for discrete distributions are:

- *Binomial* (page 215)
- *Geometric* (page 217)
- *HyperGeometric* (page 219)
- *NegativeBinomial* (page 221)
- *Poisson* (page 223)

The functions for continuous distributions are:

- *Beta* (page 225)
- *Exponential* (page 227)
- *ExtremeValue* (page 229)
- *Gamma* (page 231)
- *Logistic* (page 233)
- *LogNormal* (page 235)
- *Normal* (page 237)
- *Pareto* (page 239)
- *Triangular* (page 241)
- *Uniform* (page 243)
- *Weibull* (page 245)

The following functions that operate on distributions are available:

- *DistributionCumulative* (page 247)
- *DistributionDensity* (page 248)
- *DistributionDeviation* (page 248)
- *DistributionInverseCumulative* (page 249)
- *DistributionInverseDensity* (page 250)
- *DistributionKurtosis* (page 250)
- *DistributionMean* (page 251)
- *DistributionSkewness* (page 251)
- *DistributionVariance* (page 252)

The combinatoric functions are:

- *Combination* (page 252)
- *Factorial* (page 253)
- *Permutation* (page 253)

1.7.1 Binomial

The function *Binomial* (page 215) draws a random value from a binomial distribution.

```
Binomial(
        ProbabilityOfSuccess, ! (input) numerical expression
        NumberOfTries         ! (input) integer expression
        )
```

Arguments

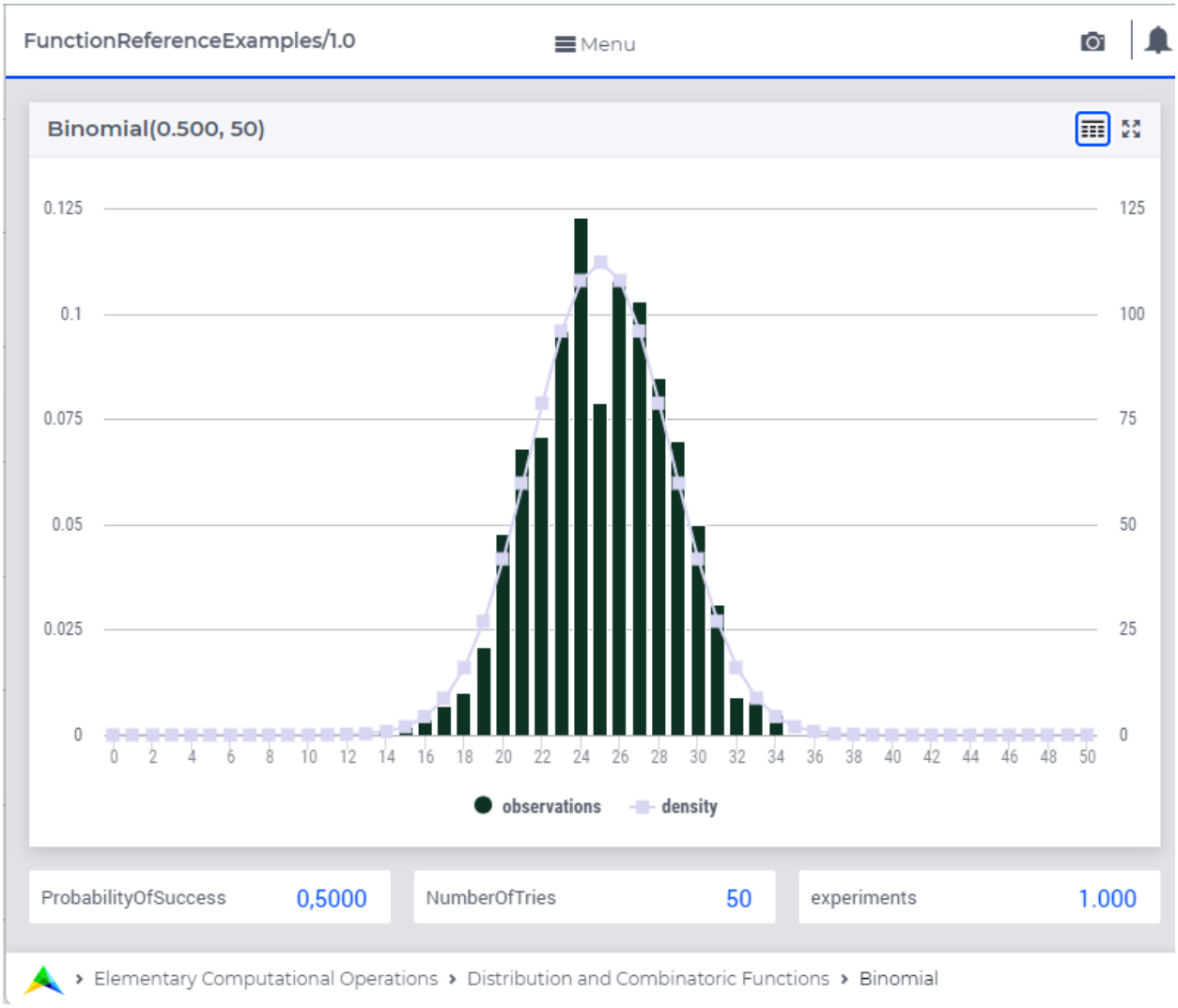
ProbabilityOfSuccess A scalar numerical expression in range $(0, 1)$.

NumberOfTries An integer numerical expression > 0.

Return Value

The function *Binomial* (page 215) returns a random value drawn from a binomial distribution with a probability of success *ProbabilityOfSuccess* and number of tries *NumberOfTries*

Graph



A graph with:

- a histogram for 1000 experiments of drawing from distribution `Binomial(0.5,50)`, and
- the *DistributionDensity* (page 248) for `Binomial(0.5,50)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Binomial( 0.7, 50 );
_p_pointDensity := DistributionDensity( Binomial( 0.7, 50 ), 25 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 37 ;
_p_pointDensity := 0.001436 ;
```

in the listing file.

References

- The *Binomial* (page 215) distribution is discussed in full detail in Discrete Distributions of the Language Reference.
- Wikipedia

1.7.2 Geometric

The function *Geometric* (page 217) draws a random value from a geometric distribution.

It represents the number of failures in a series of drawings whereby the chance of success is `ProbabilityOfSuccess`

```
Geometric(
   ProbabilityOfSuccess    ! (input) numerical expression
         )
```

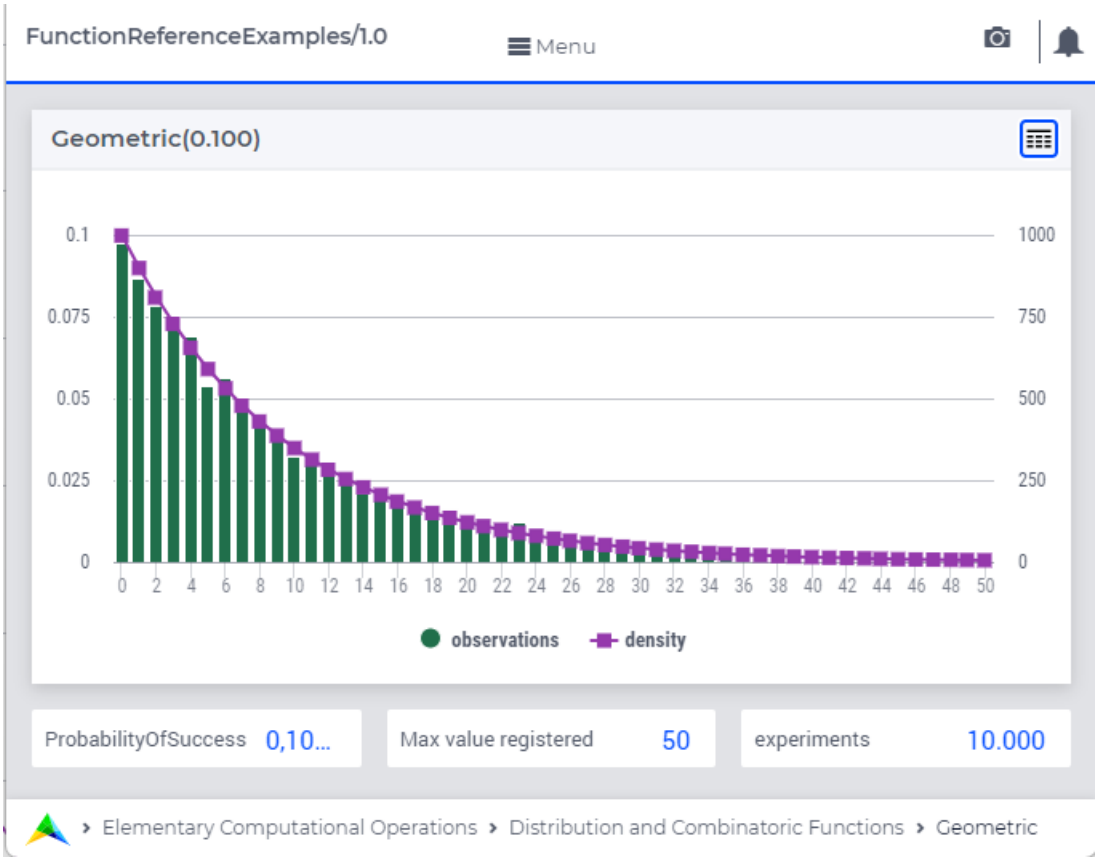
Arguments

ProbabilityOfSuccess A scalar numerical expression in the range $(0, 1)$.

Return Value

The function *Geometric* (page 217) returns a random value drawn from a geometric distribution with a probability of success *ProbabilityOfSuccess*.

Graph



A graph with:

- a histogram for 10000 experiments of drawing from distribution `Geometric(0.1)`, and
- the *DistributionDensity* (page 248) for `Geometric(0.1)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Geometric( 0.1 );
_p_pointDensity := DistributionDensity( Geometric( 0.1 ), 5 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 7 ;
_p_pointDensity := 0.059049 ;
```

in the listing file.

Indicates:

- that for the first drawing: 7 failures before success, when chance of success is 0.1, and
- the probability for result 5 is 0.059049.

References

- The *Geometric* (page 217) distribution is discussed in full detail in Discrete Distributions of the Language Reference.
- Wikipedia

1.7.3 HyperGeometric

The function *HyperGeometric* (page 219) draws a random value from a hypergeometric distribution.

```
HyperGeometric(
    ProbabilityOfSuccess,    ! (input) numerical expression
    NumberOfTries,           ! (input) integer expression
    PopulationSize           ! (input) integer expression
              )
```

Arguments

ProbabilityOfSuccess A scalar numerical expression in the range $(0, 1)$.

NumberOfTries A integer numerical expression in the range $1, \ldots, PopulationSize$.

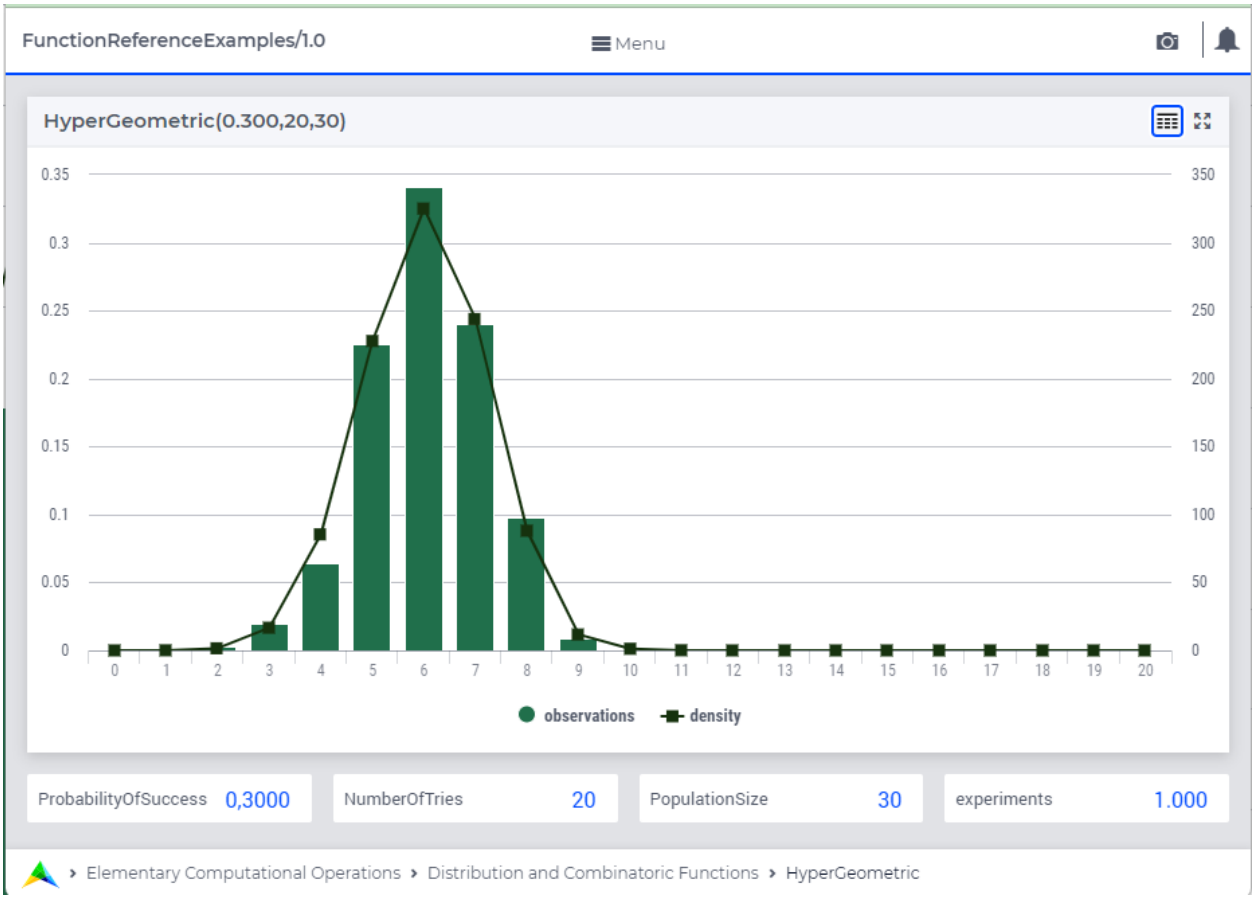
PopulationSize A integer numerical expression > 0.

Return Value

The function *HyperGeometric* (page 219) returns a random value drawn from a hypergeometric distribution with a probability of success *ProbabilityOfSuccess*, number of tries *NumberOfTries* and population size *PopulationSize*.

Note: The probability of success *ProbabilityOfSuccess* must assume one of the values $i/size$, where i is in the range $1, \ldots, PopulationSize - 1$.

Graph



A graph with:

- a histogram for 10000 experiments of drawing from distribution `HyperGeometric(0.3,20,30)`, and
- the *DistributionDensity* (page 248) for `HyperGeometric(0.3,20,30)`

Example

The code:

```
option seed := 1234 ;
_p_draw := HyperGeometric( 0.4, 20, 50 );
_p_pointDensity := DistributionDensity( HyperGeometric( 0.4, 20, 50 ), 5 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 8 ;
_p_pointDensity := 0.051029 ;
```

in the listing file.

Indicates:

- that for the first drawing: 8 success, when sampling 20 out of a population of 50 where each individual has propability 0.4 for success
- the probability for result 5 is 0.059049.

References

- The *HyperGeometric* (page 219) distribution is discussed in full detail in Discrete Distributions of the Language Reference.
- Wikipedia

1.7.4 NegativeBinomial

The function *NegativeBinomial* (page 221) draws a random value from a negative binomial distribution.

```
NegativeBinomial(
    ProbabilityOfSuccess,    ! (input) numerical expression
    NumberOfSuccesses        ! (input) integer expression
    )
```

This value represents the number of failures until `NumberOfSuccesses` were achieved, whereby each trial has a chance of `ProbabilityOfSuccess` to succeed.

Arguments

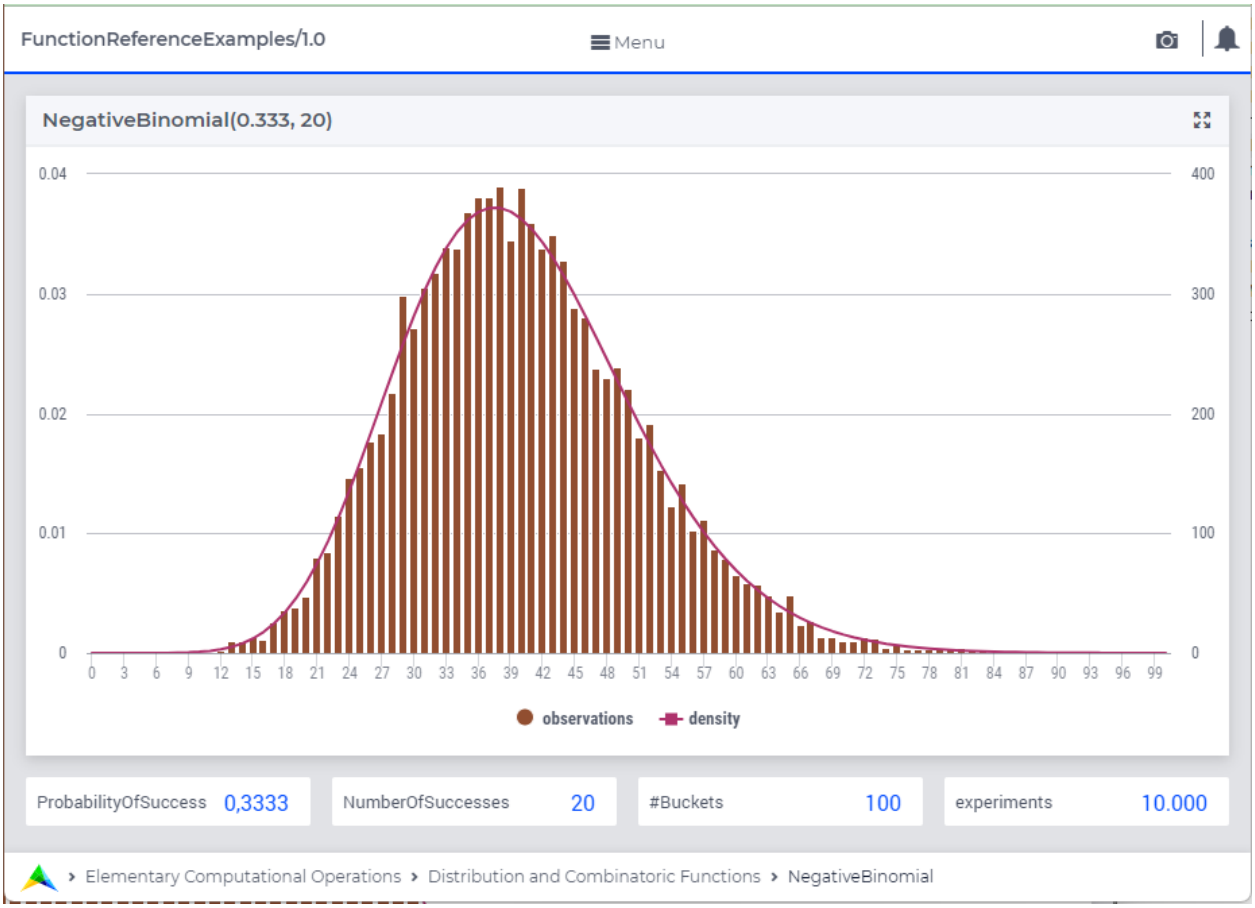
ProbabilityOfSuccess A scalar numerical expression in the range $(0, 1)$.

NumberOfSuccesses An integer numerical expression > 0.

Return Value

The function *NegativeBinomial* (page 221) returns a random value drawn from a negative binomial distribution with probability *ProbabilityOfSuccess* and number of successes *NumberOfSuccesses*.

Graph



A graph with:

- a histogram for 10.000 experiments of drawing from distribution `NegativeBinomial(0.3333,20)`, and
- the *DistributionDensity* (page 248) for `NegativeBinomial(0.3333,20)`

Example

The code:

```
option seed := 1234 ;
_p_draw := NegativeBinomial( 0.7, 50 );
_p_pointDensity := DistributionDensity( NegativeBinomial( 0.7, 50 ), 25 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 31 ;
_p_pointDensity := 0.053424 ;
```

in the listing file.

References

- The *NegativeBinomial* (page 221) distribution is discussed in full detail in Discrete Distributions of the Language Reference.
- Wikipedia

1.7.5 Poisson

The function *Poisson* (page 223) draws a random value from a Poisson distribution.

```
Poisson(
    AverageNumberOfSuccesses      ! (input) numerical expression
    )
```

Arguments

lambda A scalar numerical expression > 0.

Return Value

The function *Poisson* (page 223) returns a random value drawn from a Poisson distribution with average number of occurrences *AverageNumberOfSuccesses*.

Graph

A graph with:

- a histogram for 1000 experiments of drawing from distribution `Poisson(4)`, and
- the *DistributionDensity* (page 248) for `Poisson(4)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Poisson( 10 );
_p_pointDensity := DistributionDensity( Poisson( 10 ), 5 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 12 ;
_p_pointDensity := 0.037833 ;
```

in the listing file.

References

- The *Poisson* (page 223) distribution is discussed in full detail in Discrete Distributions of the Language Reference.
- Wikipedia

1.7.6 Beta

The function *Beta* (page 225) draws a random value from a beta distribution.

```
Beta(
    ShapeAlpha,             ! (input) numerical expression
    ShapeBeta,              ! (input) numerical expression
    Minimum,                ! (optional) numerical expression
    Maximum                 ! (optional) numerical expression
    )
```

Arguments

ShapeAlpha A scalar numerical expression > 0.

ShapeBeta A scalar numerical expression > 0.

Minimum A scalar numerical expression.

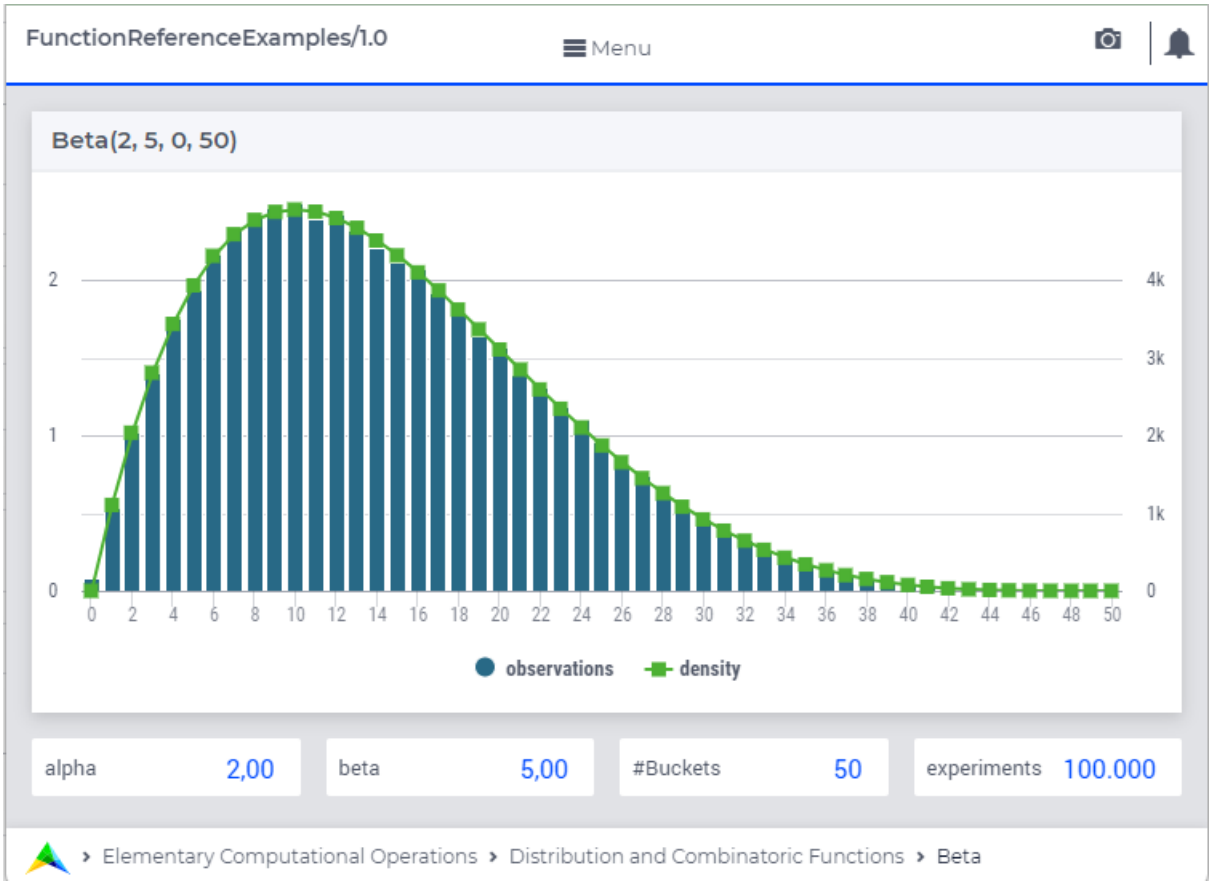
Maximum A scalar numerical expression $>$*min*.

Return Value

The function *Beta* (page 225) returns a random value drawn from a beta distribution with shapes *ShapeAlpha*, *ShapeBeta*, lower bound *Minimum* and upper bound *Maximum*.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. The original function *Beta* (page 225)(*ShapeAlpha*, *ShapeBeta*, *s*) returns a random value drawn from a beta distribution with shapes *ShapeAlpha*, *ShapeBeta* and scale *s*, where $s = Maximum$ and $Minimum = 0$.

Graph



A graph with:

- a histogram for 100.000 experiments of drawing from distribution `beta(2,5,0,50)`, and
- the *DistributionDensity* (page 248) for `beta(2,5,0,50)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Beta( 2, 5 )  ;
_p_pointDensity := DistributionDensity( Beta( 2, 5 ), 0.5 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 0.510666 ;
_p_pointDensity := 0.937500 ;
```

in the listing file.

References

- **The *Beta* (page 225) distribution is discussed in full detail in**

  Continuous Distributions of the Language Reference.

- Wikipedia

1.7.7 Exponential

The function *Exponential* (page 227) draws a random value from an exponential distribution.

```
Exponential(
      lowerbound      ! (optional) numerical expression
      scale           ! (optional) numerical expression
     )
```

Arguments

lowerbound A scalar numerical expression.

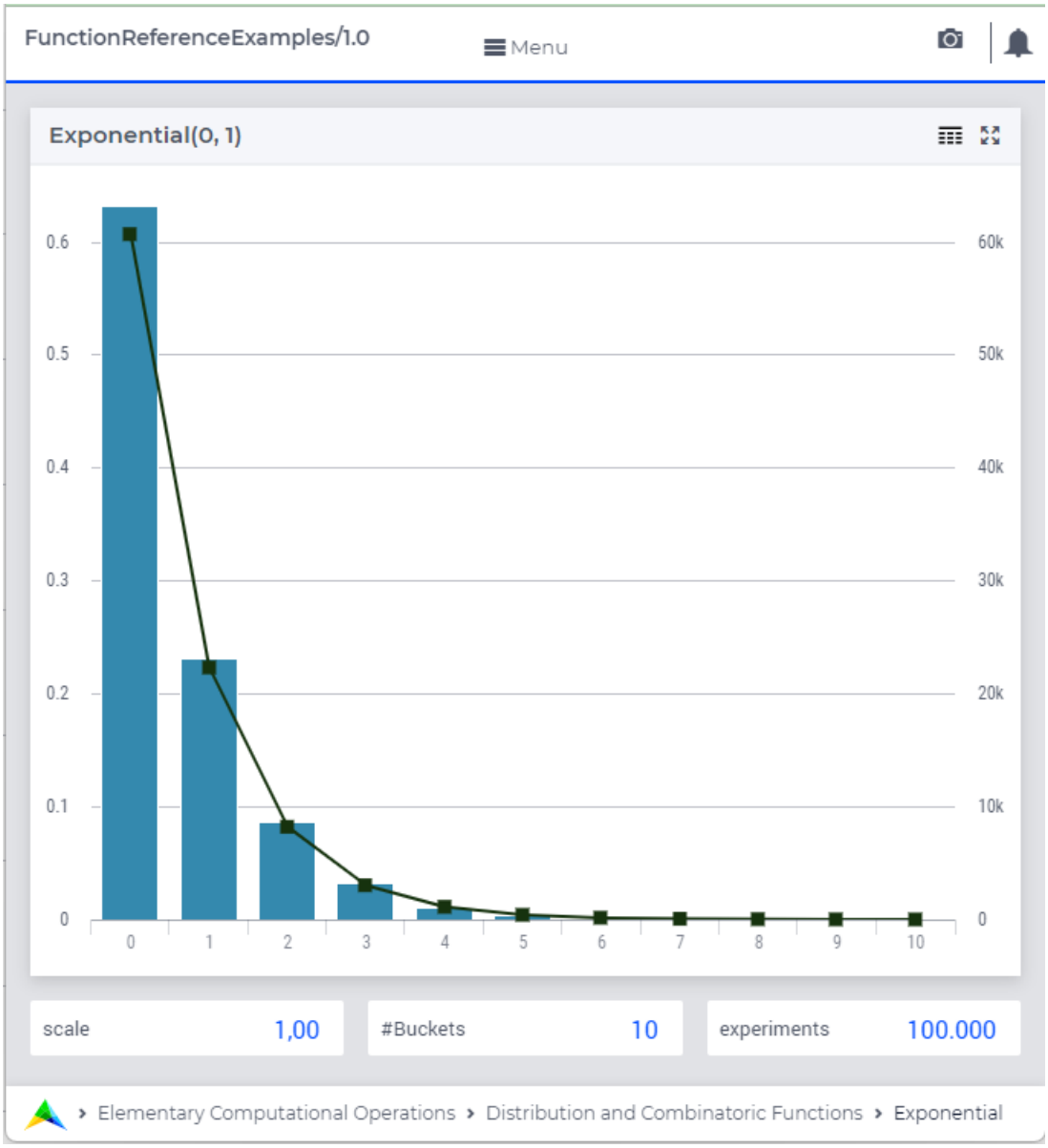
scale A scalar numerical expression > 0.

Return Value

The function *Exponential* (page 227) returns a random value drawn from a exponential distribution with lower bound *lowerbound* and scale *scale*.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. The original function *Exponential* (page 227)(*lambda*) returns a random value drawn from a exponential distribution with rate $lambda = 1/scale$ and lower bound 0.

Graph



A graph with:

- a histogram for 100.000 experiments of drawing from distribution `Exponential(0,1)`, and
- the *DistributionDensity* (page 248) for `Exponential(0,1)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Exponential( 0, 10 )  ;
_p_pointDensity := DistributionDensity( Exponential( 0, 10 ), 3 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 4.948559 ;
_p_pointDensity := 7.408182 ;
```

in the listing file.

References

- The *Exponential* (page 227) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.8 ExtremeValue

The function *ExtremeValue* (page 229) draws a random value from an extreme value distribution.

```
ExtremeValue(
       location,     ! (optional) numerical expression
       scale         ! (optional) numerical expression
       )
```

The `ExtremeValue(location,scale)` distribution provided by AIMMS is a Type I `Gumbel` (μ, β) distribution.

Arguments

location A scalar numerical expression.

scale A scalar numerical expression > 0.

Return Value

The function *ExtremeValue* (page 229) returns a random value drawn from an extreme value distribution with location *location* and scale *scale*.

Graph

A graph with:

- a histogram for 1.000 experiments of drawing from distribution `ExtremeValue(3,4)`, and
- the *DistributionDensity* (page 248) for `ExtremeValue(3,4)`

Example

The code:

```
option seed := 1234 ;
_p_draw := ExtremeValue( 0, 10 )  ;
_p_pointDensity := DistributionDensity( ExtremeValue( 0, 10 ), 3 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := -2.456478 ;
_p_pointDensity := 0.034999 ;
```

in the listing file.

References

- The *ExtremeValue* (page 229) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia generalized extreme value
- Wikipedia gumbel type I

1.7.9 Gamma

The function *Gamma* (page 231) draws a random value from a gamma distribution.

```
Gamma(
     Shape,         ! (input) numerical expression
     Lowerbound,    ! (optional) numerical expression
     Scale          ! (optional) numerical expression
     )
```

Arguments

Shape A scalar numerical expression > 0.

Lowerbound A scalar numerical expression $>= 0$.

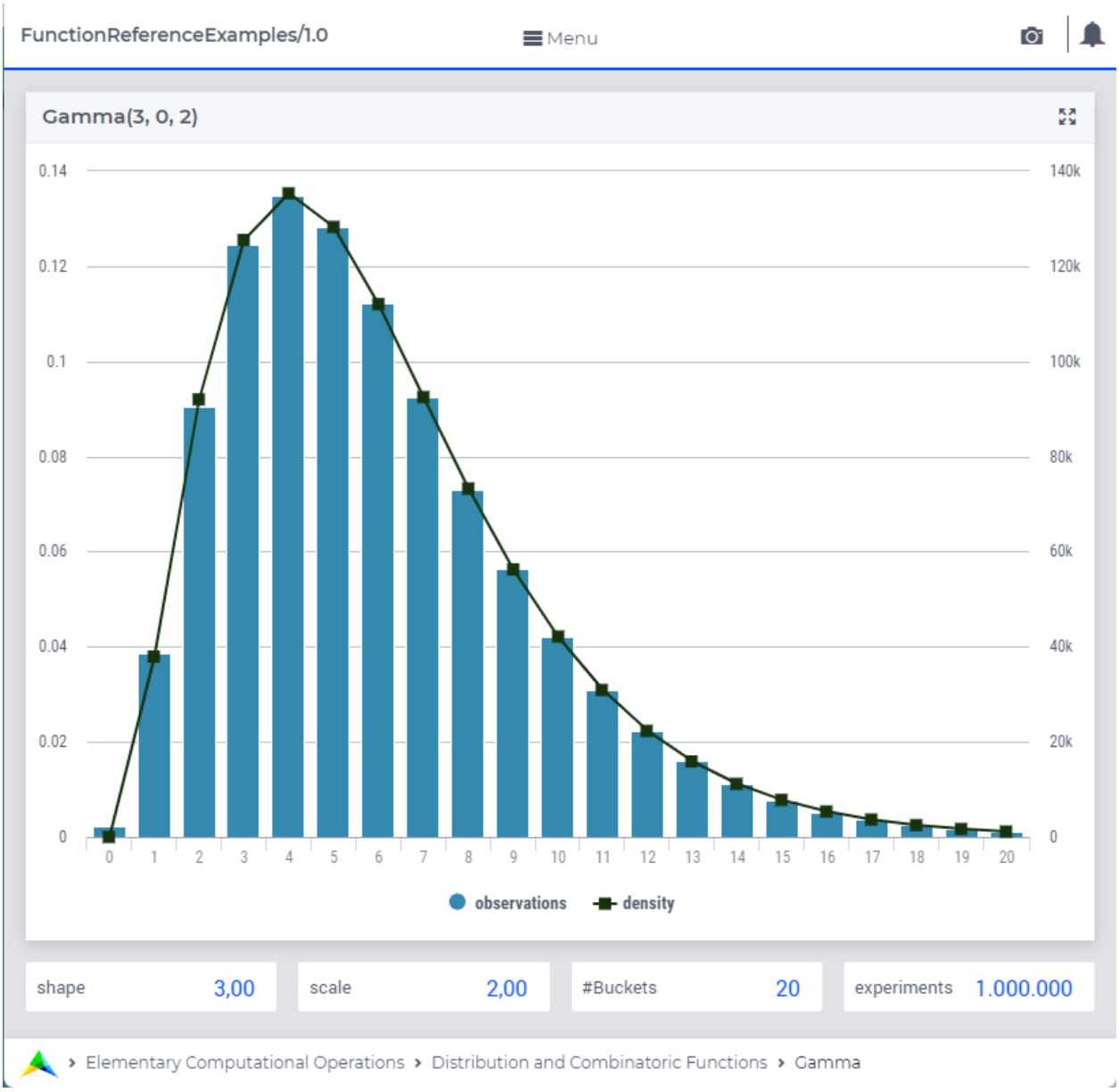
Scale A scalar numerical expression > 0.

Return Value

The function Gamma (page 231) returns a random value drawn from a gamma distribution with shape *Shape*, lower bound *Lowerbound* and scale *Scale*.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. The original function Gamma (page 231)(*alpha*, *Shape*) returns a random value drawn from a gamma distribution with rate $alpha = 1/Scale$, shape *Shape* and lower bound 0.

Graph



A graph with:

- a histogram for 1.000.000 experiments of drawing from distribution `Gamma(3,0,2)`, and
- the *DistributionDensity* (page 248) for `Gamma(3,0,2)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Gamma( 3, 0, 2 )  ;
_p_pointDensity := DistributionDensity( Gamma( 3, 0, 2 ), 9 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 11.206294 ;
_p_pointDensity := 0.056239 ;
```

in the listing file.

References

- The *Gamma* (page 231) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.10 Logistic

The function *Logistic* (page 233) draws a random value from a logistic distribution.

```
Logistic(
        Location,    ! (optional) numerical expression
        Scale        ! (optional) numerical expression
        )
```

Arguments

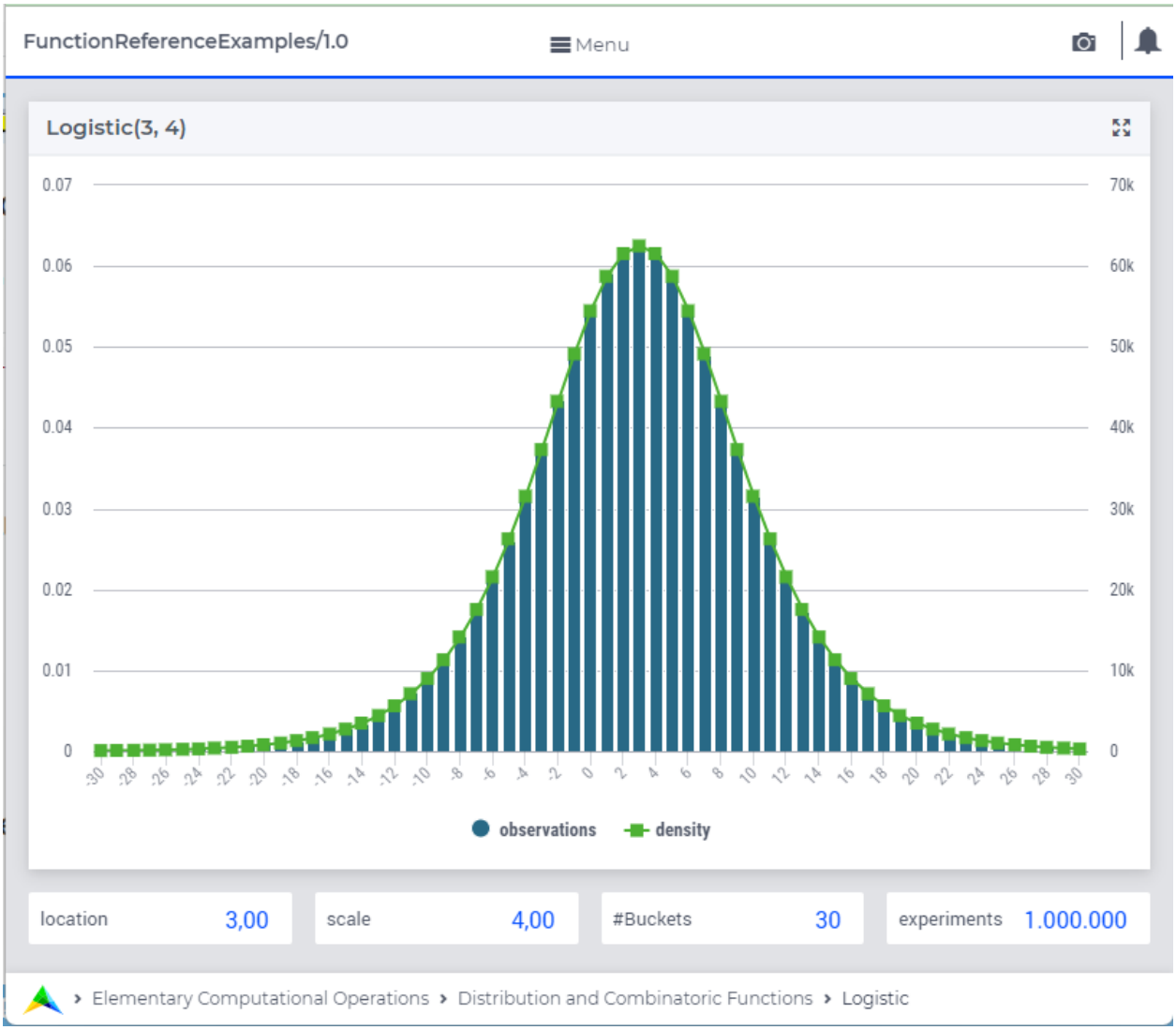
Location A scalar numerical expression.

Scale A scalar numerical expression > 0.

Return Value

The function *Logistic* (page 233) returns a random value drawn from a logistic distribution with mean *Location* and scale *Scale*.

Graph



A graph with:

- a histogram for 1.000.000 experiments of drawing from distribution `Logistic(3,4)`, and
- the *DistributionDensity* (page 248) for `Logistic(3,4)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Logistic( 0, 10 )  ;
_p_pointDensity := DistributionDensity( Logistic( 0, 10 ), 3 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := -21.717284 ;
_p_pointDensity := 0.024446 ;
```

in the listing file.

References

- The *Logistic* (page 233) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.11 LogNormal

The function *LogNormal* (page 235) draws a random value from a lognormal distribution.

```
LogNormal(
    Shape,         ! (input) numerical expression
    Lowerbound,    ! (optional) numerical expression
    Scale          ! (optional) numerical expression
    )
```

Arguments

Shape A scalar numerical expression > 0.

Lowerbound A scalar numerical expression.

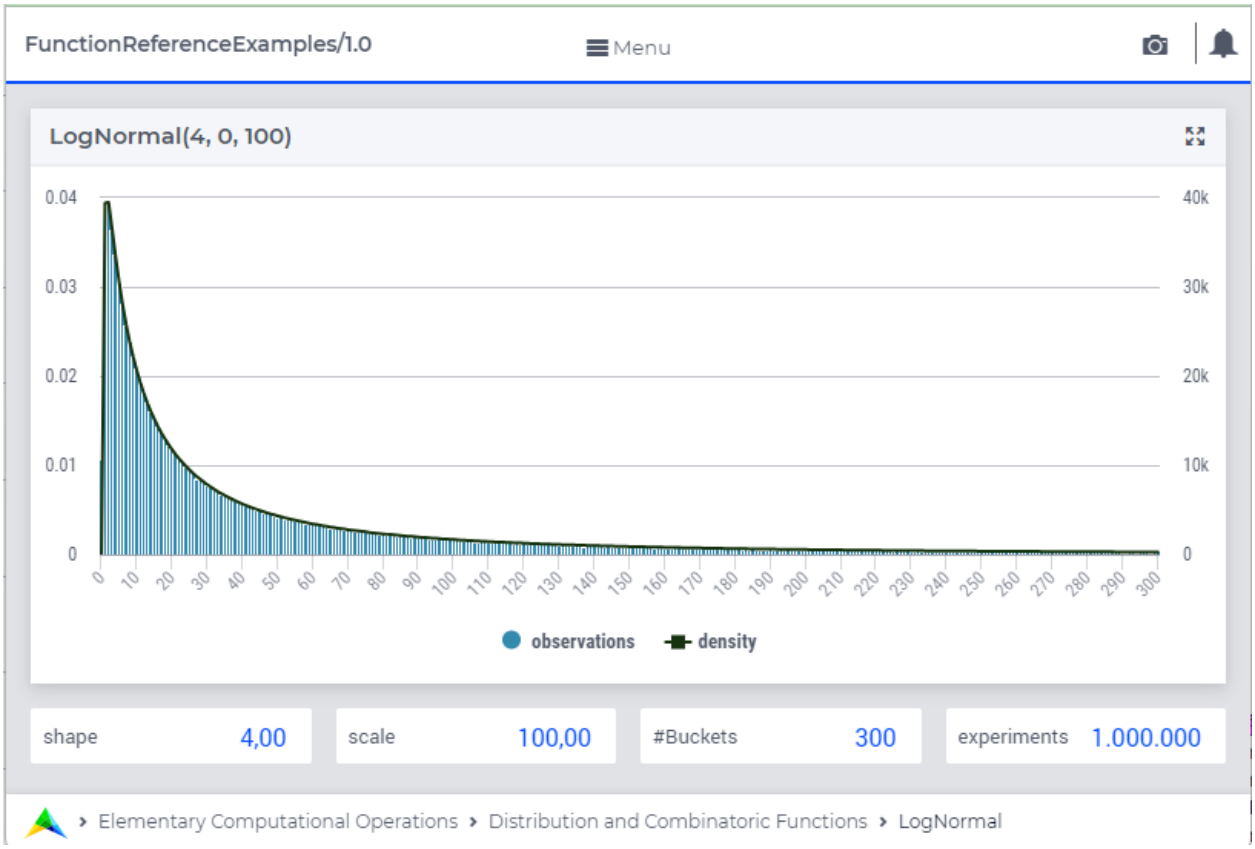
Scale A scalar numerical expression > 0.

Return Value

The function *LogNormal* (page 235) returns a random value drawn from a lognormal distribution with shape *Shape*, lower bound *Lowerbound* and scale *Scale*.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. The original function *LogNormal* (page 235)(*m*, *sd*) returns a random value drawn from a lognormal distribution with mean $m > 0$ and standard deviation $sd > 0$. The same result should now be obtained by setting $Shape = sd/m$, $Lowerbound = 0$ and $Scale = m$.

Graph



A graph with:

- a histogram for 1.000.000 experiments of drawing from distribution `LogNormal(4,0,100)`, and
- the *DistributionDensity* (page 248) for `LogNormal(4,0,100)`

Example

The code:

```
option seed := 1234 ;
_p_draw := LogNormal( 4, 0, 1 )  ;
_p_pointDensity := DistributionDensity( LogNormal( 4, 0, 1 ), 3 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 0.042849 ;
_p_pointDensity := 0.025869 ;
```

in the listing file.

References

- The *LogNormal* (page 235) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.12 Normal

The function *Normal* (page 237) draws a random value from a normal distribution.

```
Normal(
    Mean,       ! (optional) numerical expression
    Deviation   ! (optional) numerical expression
    )
```

Arguments

Mean A scalar numerical expression.

Deviation A scalar numerical expression > 0.

Return Value

The function *Normal* (page 237) returns a random value drawn from a normal distribution with mean *Mean* and standard deviation *Deviation*.

Graph

A graph with:

- a histogram for 1.000 experiments of drawing from distribution `Normal(0,10)`, and
- the *DistributionDensity* (page 248) for `Normal(0,10)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Normal( 0, 10 )  ;
_p_pointDensity := DistributionDensity( Normal( 0, 10 ), 3 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 6.124772 ;
_p_pointDensity := 0.038139 ;
```

in the listing file.

References

- The *Normal* (page 237) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.13 Pareto

The function *Pareto* (page 239) draws a random value from a Pareto distribution.

```
Pareto(
      Shape,           ! (input) numerical expression
      Location,        ! (optional) numerical expression
      Scale            ! (optional) numerical expression
      )
```

Arguments

Shape A scalar numerical expression > 0.

Location A scalar numerical expression.

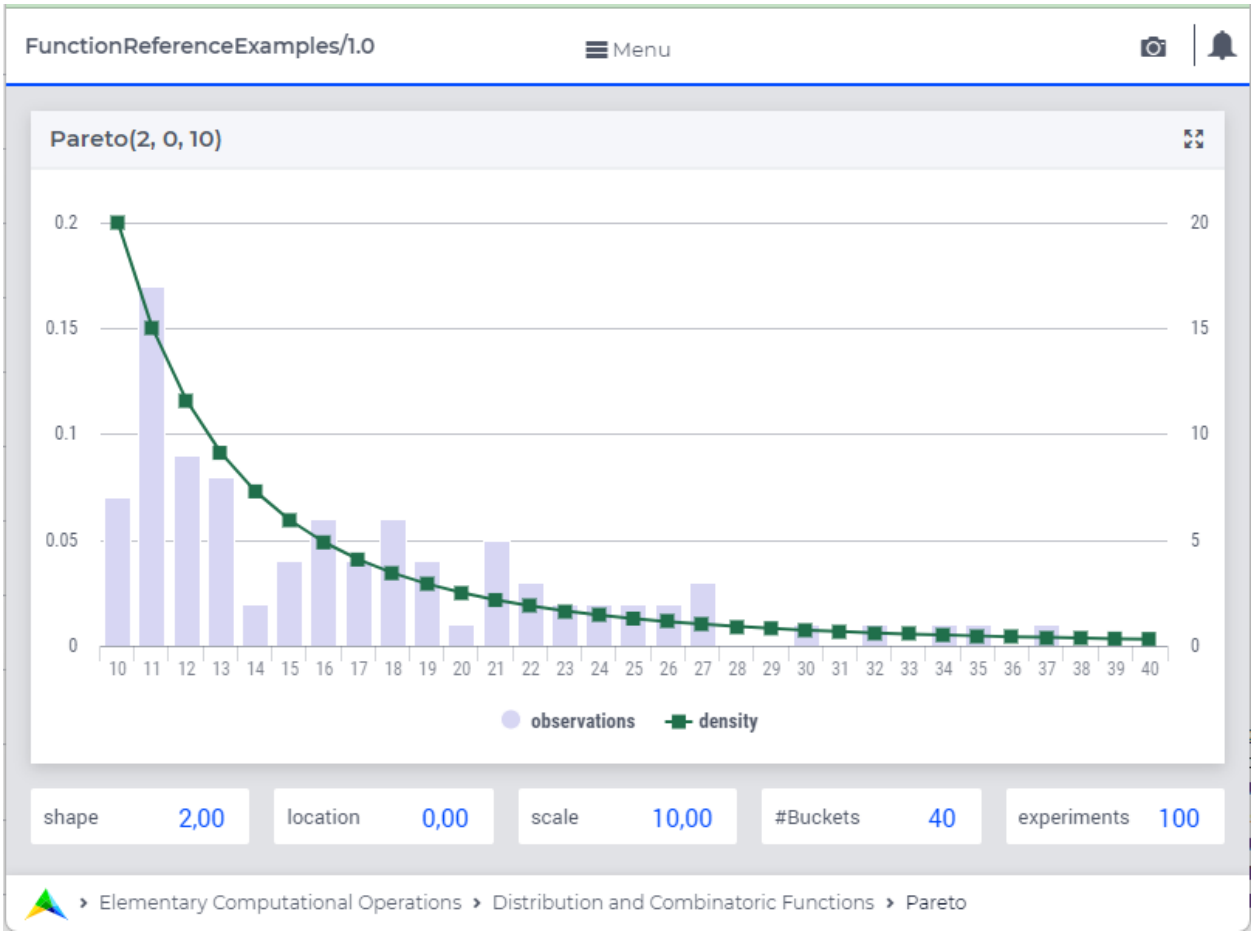
Scale A scalar numerical expression > 0.

Return Value

The function *Pareto* (page 239) returns a random value drawn from a Pareto distribution with shape *Shape*, location *Location* and scale *Scale*.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. The original function *Pareto* (page 239)(*s*, *beta*) returns a random value drawn from a Pareto distribution with shape *beta*, location 0 and scale *s*.

Graph



A graph with:

- a histogram for 100 experiments of drawing from distribution `Pareto(2,0,10)`, and
- the *DistributionDensity* (page 248) for `Pareto(2,0,10)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Pareto( 2, 0, 1 )  ;
_p_pointDensity := DistributionDensity( Pareto( 2, 0, 1 ), 2 );

block where listing_number_precision := 6 ;
        display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 1.262422 ;
_p_pointDensity := 0.250000 ;
```

in the listing file.

References

- The *Pareto* (page 239) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.14 Triangular

The function *Triangular* (page 241) draws a random value from a triangular distribution.

```
Triangular(
     Shape,         ! (input) numerical expression
     Minimum,       ! (optional) numerical expression
     Maximum        ! (optional) numerical expression
     )
```

Arguments

Shape A scalar numerical expression.

Minimum A scalar numerical expression.

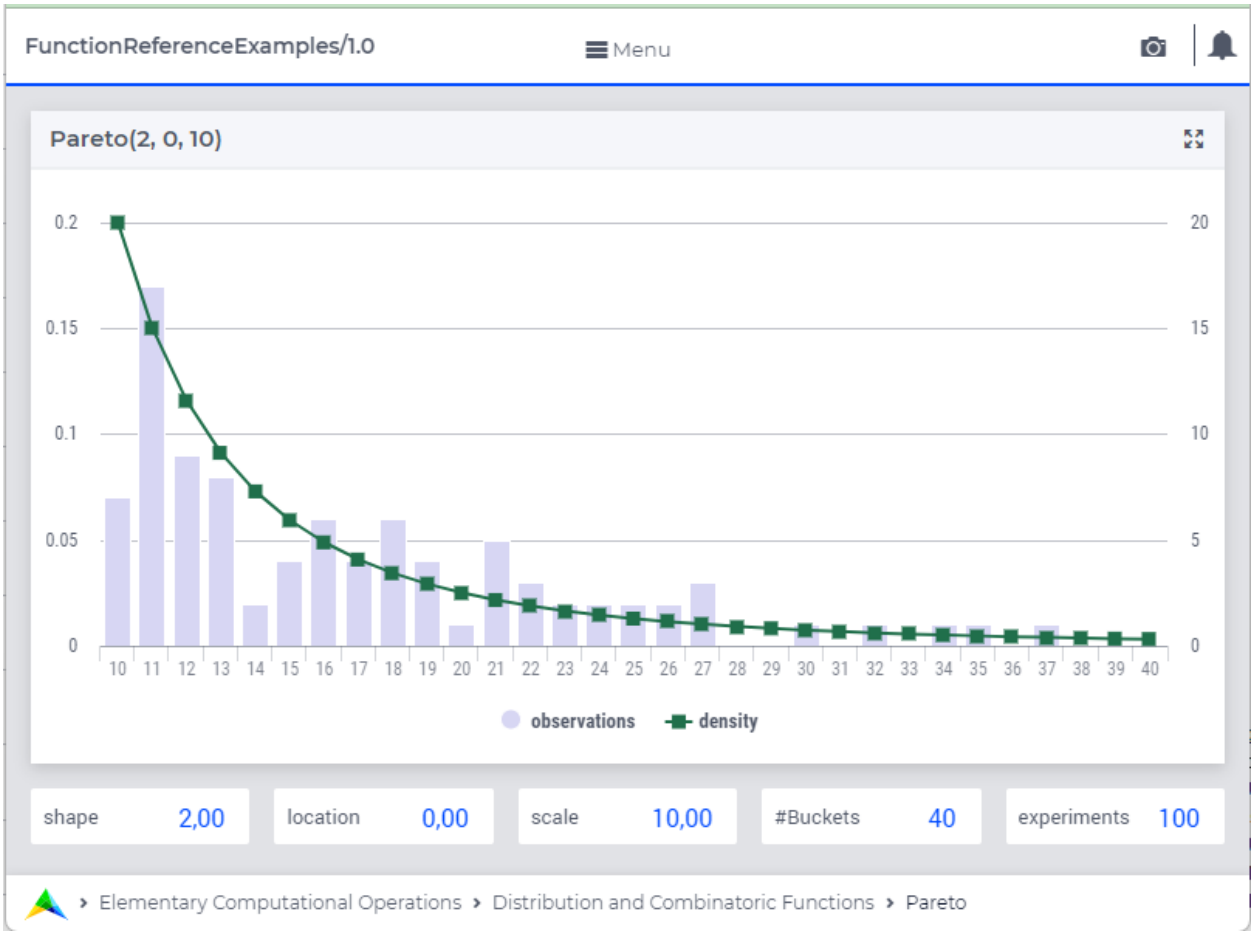
Maximum A scalar numerical expression.

Return Value

The function *Triangular* (page 241) returns a random value drawn from a triangular distribution with shape *Shape*, lower bound *Minimum* and upper bound *Maximum*. The argument *Shape* must satisfy the relation $0 < Shape < 1$.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. The original function *Triangular* (page 241)(a, b, c) returns a random value drawn from a triangular distribution with a lower bound a, likeliest value b and upper bound c. The arguments must satisfy the relation $a < b < c$. The relation between the arguments *Shape* and b is given by $Shape = (b - a)/(c - a)$.

Graph



A graph with:

- a histogram for 100 experiments of drawing from distribution `Triangular(0.300,0,100)`, and
- the *DistributionDensity* (page 248) for `Triangular(0.300,0,100)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Triangular( 0.3, 0, 100 )  ;
_p_pointDensity := DistributionDensity( Triangular( 0.3, 0, 100  ), 20 );

block where listing_number_precision := 6 ;
    display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 40.585179 ;
_p_pointDensity := 0.013333 ;
```

in the listing file.

References

- The *Triangular* (page 241) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.15 Uniform

The function *Uniform* (page 243) draws a random value from a uniform distribution.

```
Uniform(
    Minimum,          ! (optional) numerical expression
    Maximum           ! (optional) numerical expression
    )
```

Arguments

Minimum A scalar numerical expression.

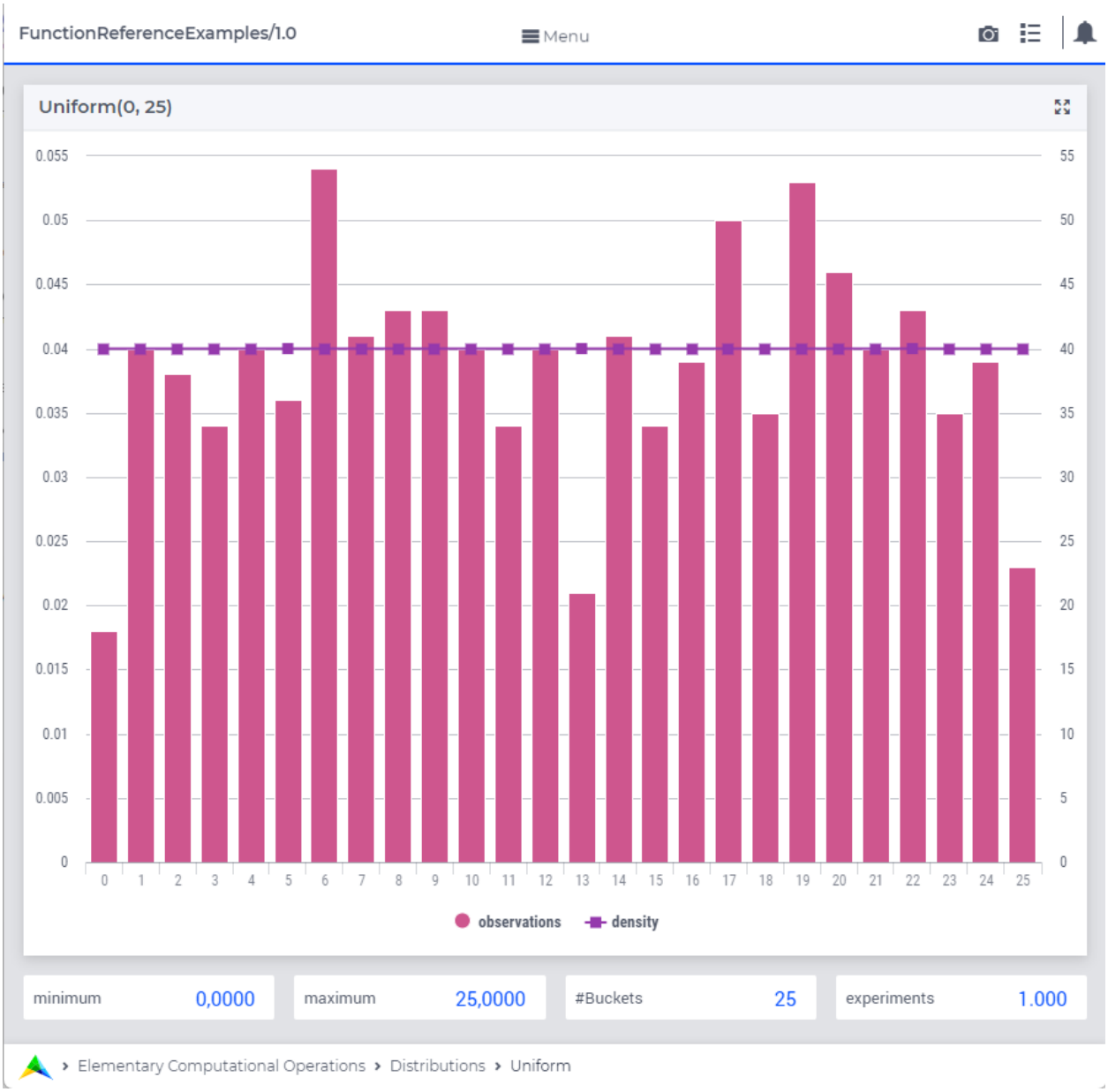
Maximum A scalar numerical expression.

Return Value

The function *Uniform* (page 243) returns a random value drawn from a uniform distribution with lower bound *Minimum* and upper bound *Maximum*.

Note: The arguments must satisfy the relation $Minimum < Maximum$.

Graph



A graph with:

- a histogram for 100 experiments of drawing from distribution `Uniform(0,25)`, and
- the *DistributionDensity* (page 248) for `Uniform(0,25)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Uniform( 0, 100 )  ;
_p_pointDensity := DistributionDensity( Uniform( 0, 100  ), 45 );

block where listing_number_precision := 6 ;
    display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 69.551654 ;
_p_pointDensity := 0.010000 ;
```

in the listing file.

References

- The *Uniform* (page 243) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.16 Weibull

The function *Weibull* (page 245) draws a random value from a Weibull distribution.

```
Weibull(
       Shape,       ! (input) numerical expression
       Lowerbound,  ! (optional) numerical expression
       Scale        ! (optional) numerical expression
       )
```

Arguments

Shape A scalar numerical expression > 0.

Lowerbound A scalar numerical expression.

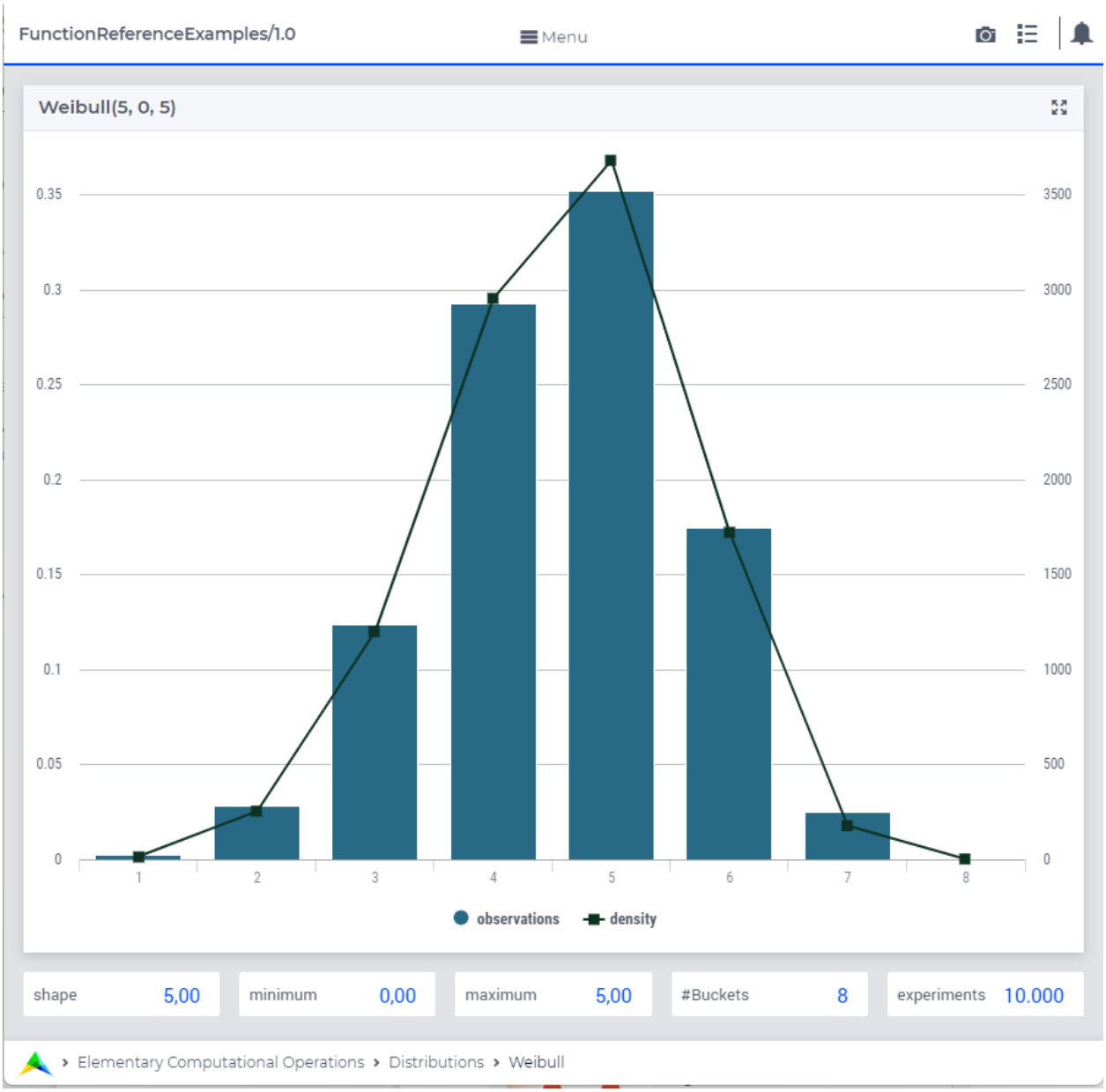
Scale A scalar numerical expression > 0.

Return Value

The function *Weibull* (page 245) returns a random value drawn from a Weibull distribution with shape *Shape* lower bound *Lowerbound*, and scale *Scale*.

Note: The prototype of this function has changed with the introduction of AIMMS 3.4. In order to run models that still use the original prototype, the option `Distribution_compatibility` should be set to `Aimms_3_0`. In the original function *Weibull* (page 245)(*Lowerbound*, *Shape*, *Scale*), the arguments were ordered differently.

Graph



A graph with:

- a histogram for 100 experiments of drawing from distribution `Weibull(5,0,5)`, and
- the *DistributionDensity* (page 248) for `Weibull(5,0,5)`

Example

The code:

```
option seed := 1234 ;
_p_draw := Weibull( 0.3, 0, 100 )  ;
_p_pointDensity := DistributionDensity( Weibull( 0.3, 0, 100  ), 20 );

block where listing_number_precision := 6 ;
    display _p_draw, _p_pointDensity ;
endblock ;
```

will produce

```
_p_draw := 5.483940 ;
_p_pointDensity := 0.004994 ;
```

in the listing file.

References

- The *Weibull* (page 245) distribution is discussed in full detail in Continuous Distributions of the Language Reference.
- Wikipedia

1.7.17 DistributionCumulative

The function *DistributionCumulative* (page 247) computes the cumulative probability value of a given distribution.

```
DistributionCumulative(
           distribution,           ! (input) distribution
           x                       ! (input) numerical expression
           )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

x A scalar numerical expression.

Return Value

The function CumulativeDistribution(*distribution*,x), for $x \in (-\infty, \infty)$ returns the probability $P(X \leq x)$ where the stochastic variable X is distributed according to the given *distribution*.

Note: For continuous distributions AIMMS can compute the derivatives of the cumulative and inverse cumulative distribution functions. As a consequence, you may use these functions in the constraints of a nonlinear model when the second argument is a variable.

See also:

The function *DistributionInverseCumulative* (page 249). The function *DistributionCumulative* (page 247) is discussed in full detail in Discrete Distributions of the Language Reference.

1.7.18 DistributionDensity

The function *DistributionDensity* (page 248) computes the density of a given distribution.

```
DistributionDensity(
          distribution,          ! (input) distribution
          x                      ! (input) numerical expression
          )
```

Arguments

distribution An expression representing any distribution (such as Normal(0,1)).

x A scalar numerical expression.

Return Value

The function *DistributionDensity* (page 248)(*distribution*,x), for $x \in (-\infty, \infty)$ returns the expected density around x of sample points from *distribution*. It is the derivative of DistributionCumulative(distr,x).

See also:

The functions *DistributionCumulative* (page 247), *DistributionInverseDensity* (page 250). The function *DistributionDensity* (page 248) is discussed in full detail in Discrete Distributions of the Language Reference.

1.7.19 DistributionDeviation

The function *DistributionDeviation* (page 248) computes the expected deviation of the given distribution .

```
DistributionDeviation(
            distribution              ! (input) distribution
            )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

Return Value

The function *DistributionDeviation* (page 248)(*distribution*) returns the expected deviation (distance from the mean) of the *distribution*.

See also:

You can find more information about the deviation of a distribution in Discrete Distributions of the Language Reference.

1.7.20 DistributionInverseCumulative

The function *DistributionInverseCumulative* (page 249) computes the inverse cumulative probability value of a given distribution.

```
DistributionInverseCumulative(
            distribution,             ! (input) distribution
            alpha                     ! (input) numerical expression
            )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

alpha A scalar numerical expression within the interval $[0, 1]$.

Return Value

The function *DistributionInverseCumulative* (page 249)(*distribution*,α), for $\alpha \in [0, 1]$ computes the largest $x \in (-\infty, \infty)$ such that the probability $P(X \leq x) \leq \alpha$ where the stochastic variable X is distributed according to the given *distribution*.

Note: For continuous distributions AIMMS can compute the derivatives of the cumulative and inverse cumulative distribution functions. As a consequence, you may use these functions in the constraints of a nonlinear model when the second argument is a variable.

See also:

The function *DistributionCumulative* (page 247). The function *DistributionInverseCumulative* (page 249) is discussed in full detail in Discrete Distributions of the Language Reference.

1.7.21 DistributionInverseDensity

The function *DistributionInverseDensity* (page 250) computes the density of the inverse cumulative function of a given distribution.

```
DistributionInverseDensity(
          distribution,          ! (input) distribution
          alpha                  ! (input) numerical expression
          )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

alpha A scalar numerical expression within the interval $[0, 1]$.

Return Value

The function *DistributionInverseDensity* (page 250)(*distribution*,α), for $\alpha \in [0, 1]$ returns the inverse density from *distribution*. It is the derivative of `DistributionInverseCumulative(distr, alpha)`.

See also:

The function *DistributionDensity* (page 248). The function *DistributionInverseDensity* (page 250) is discussed in full detail in Discrete Distributions of the Language Reference.

1.7.22 DistributionKurtosis

The function *DistributionKurtosis* (page 250) computes the kurtosis of a given distribution.

```
DistributionKurtosis(
          distribution           ! (input) distribution
          )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

Return Value

The function *DistributionKurtosis* (page 250)(*distribution*) returns the kurtosis of the given *distribution*.

See also:

You can find more information about the kurtosis of a distribution in Discrete Distributions of the Language Reference.

1.7.23 DistributionMean

The function *DistributionMean* (page 251) computes the mean of a given distribution.

```
DistributionMean(
          distribution              ! (input) distribution
          )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

Return Value

The function *DistributionMean* (page 251)(*distribution*) returns the mean of the given *distribution*.

See also:

You can find more information about the mean of a distribution in Discrete Distributions of the Language Reference.

1.7.24 DistributionSkewness

The function *DistributionSkewness* (page 251) computes the skewness of a given distribution.

```
DistributionSkewness(
          distribution              ! (input) distribution
          )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

Return Value

The function *DistributionSkewness* (page 251)(*distribution*) returns the skewness of the given *distribution*.

See also:

You can find more information about the skewness of a distribution in Discrete Distributions of the Language Reference.

1.7.25 DistributionVariance

The function *DistributionVariance* (page 252) computes the variance of a given distribution.

```
DistributionVariance(
          distribution              ! (input) distribution
          )
```

Arguments

distribution An expression representing any distribution (such as `Normal(0,1)`).

Return Value

The function *DistributionVariance* (page 252)(*distribution*) returns the variance of the given *distribution*.

See also:

You can find more information about the variance of a distribution in Discrete Distributions of the Language Reference.

1.7.26 Combination

The function *Combination* (page 252) computes the number of combinations of length *m* in *n* items.

```
Combination(
          n,              ! (input) integer expression
          m               ! (input) integer expression
          )
```

Arguments

n An integer numerical expression ≥ 0.

m An integer numerical expression in the range $0, \ldots, n$.

Return Value

The function *Combination* (page 252) returns $\binom{n}{m}$, the number of combinations of length m in a given number of items n.

See also:

Combinatoric functions are discussed in full detail in Combinatoric functions.

1.7.27 Factorial

The function *Factorial* (page 253) returns the factorial of an integer number.

```
Factorial(
          n              ! (input) integer expression
          )
```

Arguments

n An integer numerical expression ≥ 0.

Return Value

The function *Factorial* (page 253) returns the factorial value $n!$.

See also:

Combinatoric functions are discussed in full detail in Combinatoric functions.

1.7.28 Permutation

The function *Permutation* (page 253) computes the number of permutations of length m in n items.

```
Permutation(
           n,            ! (input) integer expression
           m             ! (input) integer expression
           )
```

Arguments

n An integer numerical expression ≥ 0.

m An integer numerical expression in the range $0, \ldots, n$.

Return Value

The function *Permutation* (page 253) returns $m! \cdot \binom{n}{m}$, the number of permutations of length m in a given number of items n.

See also:

Combinatoric functions are discussed in full detail in Combinatoric functions.

1.8 Histogram Functions

AIMMS supports the following functions for creating and managing histograms:

1.8.1 HistogramAddObservation

The procedure *HistogramAddObservation* (page 254) adds a new observation to a histogram that was previously created through the procedure `HistogramCreate`.

```
HistogramAddObservation(
     histogram_id,          ! (input) a scalar parameter
     value                  ! (input) a scalar value
     )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the `HistogramCreate` procedure.

value The value of a new observation that should be added to the histogram.

Return Value

The procedure returns 1 if the new observation is added successfully, or 0 otherwise.

See also:

The procedure *HistogramAddObservations* (page 254), *HistogramCreate* (page 255). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.2 HistogramAddObservations

The procedure *HistogramAddObservations* (page 254) adds a set of observations to a histogram that was previously created through the procedure HistogramCreate.

```
HistogramAddObservations(
    histogram_id,          ! (input) a scalar parameter
    values                 ! (input) a one-dimensional parameter
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate procedure.

values A one-dimensional identifier that contains the values of new observations that should be added to the histogram. The cardinality should be at least 1.

Return Value

The procedure returns 1 if the new observation is added successfully, or 0 otherwise.

See also:

The procedure *HistogramAddObservation* (page 254), *HistogramCreate* (page 255). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.3 HistogramCreate

The function *HistogramCreate* (page 255) sets up a new histogram. The created histogram does not yet contain any observations. These observations must be added later using the function HistogramAddObservation or HistogramAddObservations.

```
HistogramCreate(
    histogram_id,           ! (output) a scalar parameter
    [integer_histogram,]    ! (optional) 0 or 1
    [sample_buffer_size]    ! (optional) a positive integer value
    )
```

Arguments

histogram_id On return, this argument will contain a unique identification number, that is used to refer to the created histogram in other functions.

integer_histogram (optional) A logical indicator that specifies whether the observations will be integer-valued. Default is 0 (not integer).

sample_buffer_size (optional) The sample buffer size used in the histogram. If omitted, a default buffer size of 512 is used.

Return Value

The function returns 1 if the histogram is created successfully, or 0 otherwise.

See also:

The functions *HistogramDelete* (page 256), *HistogramAddObservation* (page 254), *HistogramAddObservations* (page 254). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.4 HistogramDelete

The procedure *HistogramDelete* (page 256) deletes a histogram that was created using the HistogramCreate procedure. After the historgram has been deleted, the histogram id is no longer valid.

```
HistogramDelete(
    histogram_id          ! (input) a scalar parameter
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate procedure. When the procedure returns, this *histogram_id* no longer refers to a valid histogram.

Return Value

The procedure returns 1 if the histogram is deleted successfully, or 0 otherwise.

See also:

The procedure *HistogramCreate* (page 255). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.5 HistogramGetAverage

The function *HistogramGetAverage* (page 256) returns the arithmetic mean of all observations in a histogram.

```
HistogramGetAverage(
    histogram_id          ! (input) a scalar number
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the `HistogramCreate` function.

Return Value

The function returns the arithmetic mean of all observations added to the histogram.

See also:

The functions *HistogramCreate* (page 255), *HistogramGetObservationCount* (page 259), *HistogramGetDeviation* (page 257), *HistogramGetSkewness* (page 260), *HistogramGetKurtosis* (page 259). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.6 HistogramGetBounds

Through the function *HistogramGetBounds* (page 257) you can obtain the lower and upper bounds of frequency interval in a histogram.

```
HistogramGetBounds(
    histogram_id,          ! (input) a scalar number
    left_bound,            ! (output) a one-dimensional parameter
    right_bound            ! (output) a one-dimensional parameter
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the `HistogramCreate` function.

left_bound A one-dimensional identifier that will be filled with the left bound of each interval in the histogram. The cardinality of the domain set should be at least the number of intervals.

right_bound A one-dimensional identifier that will be filled with the right bound of each interval in the histogram. The cardinality of the domain set should be at least the number of intervals.

Return Value

The function returns 1 if the bounds are retrieved successfully, or 0 otherwise.

See also:

The functions *HistogramCreate* (page 255), *HistogramSetDomain* (page 260). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.7 HistogramGetDeviation

The function *HistogramGetDeviation* (page 257) returns the standard deviation of all observations in a histogram.

```
HistogramGetDeviation(
    histogram_id          ! (input) a scalar number
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate function.

Return Value

The function returns the standard deviation of all observations in the histogram.

See also:

The functions *HistogramCreate* (page 255), *HistogramGetObservationCount* (page 259), *HistogramGetAverage* (page 256), *HistogramGetSkewness* (page 260), *HistogramGetKurtosis* (page 259). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.8 HistogramGetFrequencies

Through the procedure *HistogramGetFrequencies* (page 258) you can obtain the observed frequencies for each interval in a histogram.

```
HistogramGetFrequencies(
    histogram_id,         ! (input) a scalar number
    frequencies           ! (output) a one-dimensional parameter
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate procedure.

frequencies A one-dimensional identifier that will be filled with the frequencies of each interval in the histogram. The cardinality of the domain set should be at least the number of intervals.

Return Value

The procedure returns 1 if the frequencies are retrieved successfully, or 0 otherwise.

See also:

The procedures *HistogramCreate* (page 255), *HistogramAddObservation* (page 254), *HistogramAddObservations* (page 254). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.9 HistogramGetKurtosis

The function *HistogramGetKurtosis* (page 259) returns the kurtosis coefficient of all observations in a histogram.

```
HistogramGetKurtosis(
    histogram_id          ! (input) a scalar number
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate function.

Return Value

The function returns the kurtosis coefficient of all observations in the histogram.

See also:

The functions *HistogramCreate* (page 255), *HistogramGetObservationCount* (page 259), *HistogramGetAverage* (page 256), *HistogramGetDeviation* (page 257), *HistogramGetSkewness* (page 260). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.10 HistogramGetObservationCount

The function *HistogramGetObservationCount* (page 259) returns the total number of observations in a histogram.

```
HistogramGetObservationCount(
    histogram_id          ! (input) a scalar number
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate function.

Return Value

The function returns the total number of observations in a histogram.

See also:

The functions *HistogramCreate* (page 255), *HistogramGetAverage* (page 256), *HistogramGetDeviation* (page 257), *HistogramGetSkewness* (page 260), *HistogramGetKurtosis* (page 259). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.11 HistogramGetSkewness

The function *HistogramGetSkewness* (page 260) returns the skewness of all observations in a histogram.

```
HistogramGetSkewness(
    histogram_id          ! (input) a scalar number
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the HistogramCreate function.

Return Value

The function returns the skewness of all observations in the histogram.

See also:

The functions *HistogramCreate* (page 255), *HistogramGetObservationCount* (page 259), *HistogramGetAverage* (page 256), *HistogramGetDeviation* (page 257), *HistogramGetKurtosis* (page 259). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

1.8.12 HistogramSetDomain

With the procedure *HistogramSetDomain* (page 260) you can override the default layout of frequency intervals of a histogram.

```
HistogramSetDomain(
    histogram_id,         ! (input) a scalar number
    intervals,            ! (input) a positive integer number
    [left,]               ! (optional) a scalar expression
    [width,]              ! (optional) a positive scalar number
```

(continues on next page)

(continued from previous page)

```
    [left_tail,]        ! (optional) 0 or 1
    [right_tail]        ! (optional) 0 or 1
    )
```

Arguments

histogram_id A scalar value representing a histogram that was previously created using the `HistogramCreate` procedure.

intervals The number of fixed-width intervals (not including the left_ or right_tail interval).

left (optional) The lower bound of the left-most interval (not including the left-tail interval). If omitted, then the histogram will use the observations to determine this bound.

width (optional) The (fixed) width of each interval. If omitted, then the histogram will use the observations to determine the width.

left_tail (optional) An indicator whether or not a left-tail interval should be created. If this argument is omitted, then the default value of 1 is used (creating a left-tail interval).

right_tail (optional) An indicator whether or not a right-tail interval should be created. If this argument is omitted, then the default value of 1 is used (creating a right-tail interval).

Return Value

The procedure returns 1 if the domain is changed successfully, or 0 otherwise.

See also:

The procedures *HistogramCreate* (page 255), *HistogramGetBounds* (page 257). Histogram support in AIMMS is discussed in full detail in Creating Histograms of the Language Reference.

CHAPTER

TWO

ALGORITHMIC CAPABILITIES

2.1 Constraint Programming Functions

AIMMS supports the following functions for constraint programming:

2.1.1 cp::AllDifferent

This function enforces (a slice of) an indexed variable or expression to be *assigned* all different values, or to *determine* whether (a slice of) an indexed identifier or expression contains all different values.

Mathematical Formulation

The function `cp::AllDifferent(i,x_i)` is equivalent to

$$\forall i, j, i \neq j : x_i \neq x_j$$

Function Prototype

```
cp::AllDifferent(
        valueBinding, ! (input) an index binding
        values        ! (input/output) an expression
)
```

Arguments

valueBinding The index binding for which the `values` argument should have all different values.

values The expression that should have a different value for each element in `valueBinding`. This expression may involve variables, but can only contain integral or element values (i.e. no strings, fractional, or unit values).

Return Value

This function returns 1 if the values in `values` are all distinct, or 0 otherwise. If `valueBinding` results in zero or one element, then this function will also return 1, and may issue a warning on non-binding constraints.

Note: The following two constraints are equivalent, but a constraint programming solver handles the single row instantiated by `Enforcevalues1` much more efficiently than the many instantiated rows resulting from `Enforcevalues2`.

```
Constraint Enforcevalues1 {
    Definition   :  cp::AllDifferent( i, x(i) );
}
```

```
Constraint Enforcevalues2 {
    IndexDomain  :  (i,j) | i < j;
    Definition   :  x(i) <> x(j);
}
```

Example

```
ElementParameter TheElementParameter {
    IndexDomain  : i
    Definition   : {
        data{ 1 : A,
              2 : B,
              3 : C }
    }
}
```

With the above data, `cp::AllDifferent(i, TheElementParameter(i))` returns 1, because all elements are different. However, with the data below, it returns 0 (the element 'A' appears twice).

```
ElementParameter TheElementParameter {
    IndexDomain  : i;
    Definition   : {
        data{ 1 : A,
              2 : B,
              3 : A }
    }
}
```

The following code snippet is extracted from the Sudoku example (in which all rows, columns and blocks should have different values). It illustrates the selection of values; particularly illustrating the use of an index domain condition on the first argument as used in the definition of `DifferentValuesPerBlock`.

```
Constraint DifferentValuesPerRow {
    IndexDomain  :  i;
    Definition   :  cp::AllDifferent( j, x(i,j) );
}
Constraint DifferentValuesPerColumn {
```

(continues on next page)

(continued from previous page)

```
    IndexDomain  :  j;
    Definition   :  cp::AllDifferent( i, x(i,j) );
}
Constraint DifferentValuesPerBlock {
    IndexDomain  :  k;
    Definition   :  cp::AllDifferent( (i,j) | Blck(i,j) = k, x(i,j) );
}
```

See also:

- Constraint Programming in the Language Reference.
- Further information on index binding can be found in Index Binding of the Language Reference.
- The Global Constraint Catalog, which references this function as `alldifferent`.

2.1.2 cp::BinPacking

This function is used to model the assignment of objects in bins: a set of objects, each with its own known 'weight', is to be placed into a set of bins, each with its own known capacity.

Mathematical Formulation

The function `cp::BinPacking(b,c_b,o,a_o,w_o[,u])` returns 1, if, for each bin b, the sum of objects o placed, according to assignment variable a_o, into bin b ($a_o = b$) of weight w_o, is less than or equal to the capacity c_b. In addition, if the argument u is specified, the number of non-empty (i.e. used) bins is set equal to u. `cp::BinPacking(b,c_b,o,a_o,w_o[,u])` is equivalent to

$$\forall b : \sum_{o|a_o=b} w_o \otimes c_b \text{ where } \begin{cases} \otimes \text{ is } = & \text{if } c_b \text{ involves variables} \\ \otimes \text{ is } \leq & \text{if } c_b \text{ does not involve variables} \end{cases}$$

If argument u is present, the following constraint also applies.

$$u = \sum_{b|c_b} 1$$

Function Prototype

```
cp::BinPacking(
        binBinding,       ! (input) an index binding
        binCapacity,      ! (input/output) an expression
        objectBinding,    ! (input) an index binding
        objectAssignment, ! (input/output) an expression
        objectWeight,     ! (input) an expression
        numberOfBinsUsed  ! (optional, input/output) an expression
)
```

Arguments

binBinding The index binding that specifies the available bins.

binCapacity The capacity of the available bins defined over the index binding `binBinding`. This expression may involve variables:

- When the `binCapacity` expression does not involve variables, it is interpreted as an upperbound on the bin capacity.
- When the `binCapacity` expression involves variables, the constraint forces the capacities of the bins to equal this expression.

objectBinding The index binding that specifies the objects that need to be packed.

objectAssignment For each object in `objectBinding`, `objectAssignment` contains a bin in `binBinding` to indicate that the object is assigned to that particular bin. The expression for `objectAssignment` may involve variables.

objectWeight The weight of each object, defined over the binding domain `objectBinding`. This expression cannot involve variables.

numberOfBinsUsed (optional) The number of bins that are used to pack the objects. This argument is an optional expression with a numerical value that may involve variables.

Return Value

The function returns 1 when the placement of objects into bins is such that the capacity of the bins is not exceeded. When the object binding argument `objectBinding` is empty, this function will return 1. In all other cases, the function returns 0.

Example

Let us move 7 benches of size 3, 1, 2, 2, 2, 2, and 3 respectively from one place to the next over several trips with a single truck. The truck we are using has a capacity of 5 (in terms of size, not benches). With the simplest of heuristics, we fill the truck sequentially with these benches until we have no benches left to fill the truck. This heuristic leads to the following schedule:

trip	bench sizes
1	3 1
2	2 2
3	2 2
4	3

With the aid of *cp::BinPacking* (page 265) we can do better. The model is as follows:

```
Set Benches {
    Index        :  bench;
    Definition   :  ElementRange( 1, 7, prefix:"bench-");
}
Parameter BenchSize {
   IndexDomain  :  (bench);
   InitialData  : {
       data { bench-1 : 3, bench-2 : 1, bench-3 : 2, bench-4 : 2,
              bench-5 : 2, bench-6 : 2, bench-7 : 3 }
```

(continues on next page)

(continued from previous page)

```
   }
}
Parameter TruckSize {
    InitialData :  5;
}
Set Trips {
    Index        :  trip;
    Definition   :  ElementRange(1,5,prefix:"trip-");
}
ElementVariable BenchTrip {
    IndexDomain  :  bench;
    Range        :  Trips;
}
Variable NumberOfTripsNeeded {
    Range        :  free;
}
Constraint RespectTruckSize {
    Definition   : {
         cp::BinPacking(trip, TruckSize, bench, BenchTrip(bench),
         BenchSize(bench), NumberOfTripsNeeded)
    }
}
MathematicalProgram TripPlanning {
    Objective    :  NumberOfTripsNeeded;
    Direction    :  minimize;
    Constraints  :  AllConstraints;
    Variables    :  AllVariables;
    Type         :  Automatic;
}
```

Solving this model will provide the following (non-unique) result:

```
NumberOfTripsNeeded := 3 ;

BenchTrip := data { bench-1 : trip-3, bench-2 : trip-1, bench-3 : trip-2,
                    bench-4 : trip-3, bench-5 : trip-1, bench-6 : trip-1,
                    bench-7 : trip-2 } ;
```

Which leads to the following schedule:

trip	bench sizes
1	1 2 2
2	2 3
3	3 2

In the above example, the `binCapacity` argument is a parameter, because `TruckSize` has a fixed value. In such a case, `TruckSize` is an upperbound. In the example below, the truck needs to be rented and we can decide on what size it should be. Therefore, `TruckSize` (the `binCapacity` argument) is a variable. The bounds of that variable are used to limit the `TruckSize`. Note that `TruckSize` is indexed over `trip`, because the `BinPacking` constraint enforces that the fill of the truck is equal to this `TruckSize`. In case `TruckSize` is a scalar, all the trips should be equally loaded, which in practice is not necessary. The example below only displays the new or changed identifiers compared with the example above (the

constraint remains the same, but is displayed for clarity).

```
Parameter MaximumTruckSize {
    InitialData  :  8;
}
Variable TruckSize {
    IndexDomain  :  trip;
    Range        :  {
        {0..MaximumTruckSize}
    }
}
Constraint GetTruckSize {
    Definition   : {
        cp::BinPacking( trip, TruckSize(trip), bench, BenchTrip(bench),
        BenchSize(bench), NumberOfTripsNeeded )
    }
}
```

Solving this model leads to the following (non-unique) result, where the `TruckSize` for the two trips is 7 and 8, so we need to rent a truck of size 8.

```
NumberOfTripsNeeded := 2 ;

BenchTrip := data { bench-1 : trip-2, bench-2 : trip-1, bench-3 : trip-2,
                    bench-4 : trip-1, bench-5 : trip-1, bench-6 : trip-1,
                    bench-7 : trip-2 } ;
```

Which leads to the following schedule:

trip	bench sizes
1	1 2 2 2
2	3 2 3

See also:

- The examples of the function *cp::AllDifferent* (page 263) that illustrate how the index binding and indexed arguments can be used. Further information on index binding can be found in Index Binding
- Constraint Programming on Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `bin_packing`.

2.1.3 cp::Cardinality

The function *cp::Cardinality* (page 268) can be used to restrict the number of occurrences of a particular value in (a slice of) an indexed identifier or expression. This function is typically used in constraints that enforce selected values a limited number of times. The function *cp::Cardinality* (page 268) counts the number of occurrences of a collection of values and either ensures that the number of occurrences is within bounds, or sets this equal to the value of a variable.

Mathematical Formulation

The function `cp::Cardinality(i,x_i,j,c_j,y_j[,u_j])` returns 1 if the number of occurrences where x_i equals c_j is equal to y_j or in the range $\{y_j..u_j\}$for all j. `cp::Cardinality(i,x_i,j,c_j,y_j)` is equivalent to

$$\forall j : \sum_i (x_i = c_j) = y_j$$

and `cp::Cardinality(i,x_i,j,c_j,l_j,u_j)` is equivalent to

$$\forall j : l_j \leq \sum_i (x_i = c_j) \leq u_j$$

Function Prototype

```
cp::Cardinality(
        inspectedBinding,    ! (input) an index binding
        inspectedValues,     ! (input) an expression
        lookupValueBinding,  ! (input) an index binding
        lookupValues,        ! (input) an expression
        numberOfOccurrences, ! (input/output) an expression
        occurrenceLimit)     ! (optional/input) an expression
```

Arguments

inspectedBinding An index binding that specifies and possibly limits the scope of indices. This argument follows the syntax of the index binding argument of iterative operators.

inspectedValues An expression that may involve variables, but can only contain integer or element values (i.e. no strings, fractional or unit values). The result is a vector with an element for each possible value of the indices according to `inspectedBinding`.

lookupValueBinding An index binding that specifies and possibly limits the scope of indices. This argument follows the syntax of the index binding argument of iterative operators.

lookupValues An expression that does not involve variables. The result is a vector with an element for each possible value of the indices according to `lookupValueBinding`.

numberOfOccurrences An expression that may involve variables. The result is a vector with an element for each possible value of the indices according to `lookupValueBinding`.

occurrenceLimit An optional expression that does not involve variables. The result is a vector with an element for each possible value of the indices according to `lookupValueBinding`. In addition, if this argument is specified, the argument `numberOfOccurrences` is not allowed to contain variables either.

Return Value

This function returns 1 if the above condition is met. Also if the index binding argument `lookupValueBinding` is empty, this function will return 1.

Example

In car sequencing the next constraint ensures that the demand `nbCarsPerClass( c )` for each class `c` of type `car(i)` is met. The value of element variable `car` is a class of car.

```
Constraint meetDemand {
    Definition   : {
        cp::Cardinality(
             inspectedBinding    :  i,
             inspectedValues     :  car(i),
             lookupValueBinding  :  c,
             lookupValues        :  c,
             numberOfOccurrences :  nbCarsPerClass( c ),
             occurrenceLimit     :  nbCars)
    }
}
```

See also:

- The functions *cp::Count* (page 272) and *cp::Sequence* (page 278).
- Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `global_cardinality`.

2.1.4 cp::Channel

The function *cp::Channel* (page 270) links two arrays of variables such that they are uniquely matched to each other. For instance, see Fig. 2.1. This function is often used to model different perspectives of the same problem.

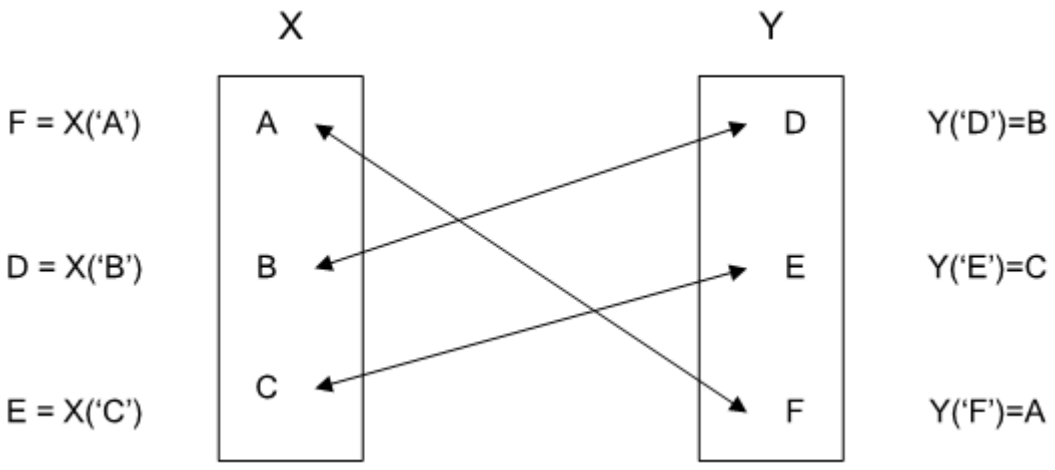


Fig. 2.1: A situation accepted by *cp::Channel* (page 270)

Mathematical Formulation

The function `cp::Channel(i,x_i,j,y_j)` returns 1 if for all i, j: $x_i = j$ implies $y_j = i$ and vice versa. `cp::Channel(i,x_i,j,y_j)` is equivalent to

$$\forall i, j : x_i = j \Leftrightarrow y_j = i$$

Function Prototype

```
cp::Channel(
        mapBinding,         ! (input) an index binding
        map,                ! (input/output) an expression
        inverseMapBinding,  ! (input) an index binding
        inverseMap          ! (input/output) an expression
        )
```

Arguments

mapBinding The index binding corresponding to the domain of the first expression `map`.

map For each element in `mapBinding`, `map` will contain an element in `inverseMapBinding`. This expression may involve variables.

inverseMapBinding The index binding corresponding to the domain of the second expression `inverseMap`.

inverseMap For each element in `inverseMapBinding`, `inverseMap` will contain an element in `mapBinding`. This expression may involve variables.

Return Value

If a unique mapping between the two index bindings is created, this function returns 1. When the index bindings `mapBinding` and `inverseMapBinding` are both empty, this function returns 1 as well. In all other cases, the function returns 0, e.g. when the number of possible values of index binding `mapBinding` is different from that of the index binding `inverseMapBinding`.

Note:

- The *cp::Channel* (page 270) constraint is also referred to in the Constraint Programming literature as `Inverse`.
- The *cp::Channel* (page 270) constraint can be used to implement the `one_factor(i,x(i))` or `symm_AllDifferent(i,x(i))` constraints encountered in the Constraint Programming literature as `cp::Channel(i,X(i),i,X(i))`.

Example

In a sports team scheduling problem, the following constraint

```
Constraint LinkingDuplicateView {
    Definition :  cp::Channel( s, Games(s), g, Slots(g) );
}
```

links the variable `Games(s)` to the variable `Slots(g)`. A game is the identification number of a match between a home and an away team. A slot is the identification number of a week and a match within a week number. For each game, there is a unique slot and for each slot there is a unique game.

See also:

- Constraint Programming on Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `inverse`.

2.1.5 cp::Count

The function *cp::Count* (page 272) can be used to restrict the number of occurrences of a particular value in (a slice of) an indexed identifier or expression. This function is typically used in constraints that enforce a selected value a limited number of times.

Mathematical Formulation

The function `cp::Count(i,x_i,c,\otimes,y)` returns 1 if the number of occurrences of x_i equal to the value c, is related to y according to the relational operator $\otimes$. The function `cp::Count(i,x_i,c,\otimes,y)` is equivalent to

$$\sum_i (x_i = c) \otimes y$$
$$\otimes \in \{\leq, =, \geq, <, >, \neq\}$$

Function Prototype

```
cp::Count(
    inspectedBinding,    ! (input) an index binding
    inspectedValues,     ! (input/output) an expression
    lookupValue,         ! (input) an expression
    relationalOperator,  ! (input) an element
    occurrenceLimit      ! (input/output) an expression
)
```

Arguments

inspectedBinding The index binding that specifies, together with the `inspectedValues` argument, the set of values in which the `lookupValue` should be counted.

inspectedValues The expression indexed over `inspectedBinding` for which the number of occurrences of the value `lookupValue` is counted. This expression may involve variables, but can only contain integer or element values (i.e. no strings, fractional or unit values).

lookupValue The particular value for which the number of occurrences in `inspectedValues` should be counted. This expression cannot involve variables. The data type should match the data type of `inspectedValues`.

relationalOperator The relational operator that indicates how the number of occurrences is limited to the `occurrenceLimit` argument. This can be an expression and should result in an element in the set *AllConstraintProgrammingRowTypes* (page 804). This expression cannot involve variables.

occurrenceLimit The number of occurrences of `lookupValue` is limited to the `occurrenceLimit`. This can be an expression that may involve variables.

Return Value

This function returns 1 if the number of occurences of `lookupValue` does not exceed the `occurenceLimit` argument according to the `relationalOperator`. In all other cases, the function returns 0.

Example

```
ElementParameter TheElementParameter {
    IndexDomain  :  i;
    Definition   :  data{ 1 : A, 2 : B, 3 : A };
}
```

With the above data, the following holds:

```
cp::Count(i, TheElementParameter(i), 'B', '<=', 1) = 1
cp::Count(i, TheElementParameter(i), 'B', '<', 1) = 0
cp::Count(i, TheElementParameter(i), 'A', '=', 2) = 1
```

The following constraint sets the number of stores supplied by a warehouse `w` equal to the variable `warehouseUsage`:

```
Set Warehouses {
    Index        :  w;
}
Set Suppliers {
    Index        :  s;
}
ElementParamter SupplyingWarehouse {
    IndexDomain  :  s;
    Range        :  Warehouses;
}
Variable WarehouseUsage {
    IndexDomain  :  w;
```

(continues on next page)

(continued from previous page)

```
    Range        :  integer;
}
Constraint CountUsedWarehouses {
    IndexDomain  :  w;
    Definition   : {
        cp::count( s, supplyingWarehouse(s), w,
                   '=', warehouseUsage(w) )
    }
}
```

See also:

- The functions *cp::Cardinality* (page 268) and *cp::Sequence* (page 278).
- Constraint Programming on Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `count` or, depending on a particular choice of $\otimes$, as `atleast`, `atmost` or `exactly`.

2.1.6 cp::Lexicographic

The function *cp::Lexicographic* (page 274) ensures that the data of one expression comes lexicographically (i.e. according to the set order) before another expression. This function is often used to reduce symmetry in two variables.

Mathematical Formulation

`cp::Lexicographic(k,x_k,y_k[,e])` is equivalent to

$$\exists i \in \{1..n\} : (\forall j : j < i : x_j = y_j) \wedge \begin{cases} x_i < y_i & \text{if } e = 0 \\ x_i \leq y_i & \text{if } e \neq 0 \end{cases}$$

where n equals `card(range(k))`.

Function Prototype

```
cp::Lexicographic(
    valueBinding, ! (input) an index binding
    firstValues,  ! (input/output) an expression
    secondValues, ! (input/output) an expression
    allowEqual    ! (optional input) an expression
)
```

Arguments

valueBinding The index binding over which the next two arguments are defined.

firstValues The expression that should lexicographically come before `secondValues`. It is defined over index binding `valueBinding` and may involve variables.

secondValues The expression that should lexicographically come after `firstValues`. It is defined over index binding `valueBinding` and may involve variables.

allowEqual When this optional argument is specified and non-zero, the expressions `firstValues` and `secondValues` are allowed to be equal. The `allowEqual` expression may not involve variables. The default of this argument is 0.

Return Value

This function returns 1 if the above condition is met. When the index binding `valueBinding` is empty, this function returns

- 0 if allowEqual is 0
- 1 if allowEqual is not 1.

Note: Please note that the comparison between the two expressions is done, based on the complete specified index binding and not pair-wise for every element in that index domain.

Example

The constraint `x_before_y` ensures that the identifier `x` comes lexicographically before the identifier `y`.

```
Constraint x_before_y {
    Definition   :  cp::Lexicographic( i, x(i), y(i) );
}
```

Suppose

```
x = data { 'a1' : 1, 'a2' : 2, 'a3' : 2 }
y = data { 'a1' : 1, 'a2' : 3, 'a3' : 1 }
```

Then the constraint `x_before_y` is met. Please note that in the case of 'a3', `x = 2` and `y = 1`. Allthough 2 does not come lexicographically before 1, the constraint *is* met. The ordering is based on the *whole* index domain, and not 'pair wise'. Because for 'a2' 2 comes lexicographically before 3, the `x`- and `y`-values for 'a3' are irrelevant here. Higher dimensional variables can also be compared using cp::Lexicographic as is illustrated next. Consider the following declarations:

```
Set S {
    Index        :  i, j;
    InitialData  :  data { a, b, c };
}
Variable X {
    IndexDomain  :  (i,j);
    Range        :  binary;
}
```

(continues on next page)

(continued from previous page)

```
Variable Y {
    IndexDomain  :  (i,j);
    Range        :  binary;
}
Constraint xylex {
    Definition   : {
        cp::Lexicographic(
            (i,j)|ord(i)<=ord(j),
            x(i,j), y(i,j))
    }
}
```

Instantiated constraints are presented in the constraint listing. For the constraint `xylex` this looks as follows:

```
----  xylex

xylex .. [ 1 | 1 | after ]

    cp::Lexicographic({X(a,a), X(a,b), X(a,c), X(b,b), X(b,c), X(c,c)},
                      {Y(a,a), Y(a,b), Y(a,c), Y(b,b), Y(b,c), Y(c,c)},
          allowEqual: 0)

    name    lower level upper
    X(a,a)      0     0     1
    X(a,b)      0     0     1
    X(a,c)      0     0     1
    X(b,b)      0     0     1
    X(b,c)      0     0     1
    X(c,c)      0     0     1
    Y(a,a)      0     1     1
    Y(a,b)      0     0     1
    Y(a,c)      0     0     1
    Y(b,b)      0     0     1
    Y(b,c)      0     0     1
    Y(c,c)      0     0     1
```

Here AIMMS visits all elements of the two dimensional variables `x` and `y`, by varying the indices `i` and `j` in the index binding `(i,j)` and adhering to the index domain condition `ord(i)<=ord(j)`. In the index binding `(i,j)` the index `j` comes after the index `i` and thus the index `j` is varied more.

See also:

- The help text associated with the option `constraint_listing`. This option can be found via the AIMMS menu `settings` > `project options` category `Solvers general` > `Standard reports` > `constraints`.
- Constraint Programming on Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `lex_less` and `lex_lesseq`.

2.1.7 cp::ParallelSchedule

The function `cp::ParallelSchedule(c,j,s_j,d_j,e_j,w_j)` models a resource that can handle multiple jobs j at the same time. The capacity of the resource is c units. The job j starts at period s_j and is active up to but not including period e_j, during d_j periods. Job j requires (a weight of) w_j units of the resource.

Mathematical Formulation

`cp::ParallelSchedule(c,j,s_j,d_j,e_j,w_j)` is equivalent to

$$\forall t : \sum_{j|s_j \leq t < e_j} w_j \leq c$$
$$\forall j : s_j + d_j = e_j.$$

Function Prototype

```
cp::ParallelSchedule(
    resourceCapacity,  ! (input) an expression
    jobBinding,        ! (input) an index binding
    jobBegin,          ! (input/output) an expression
    jobDuration,       ! (input/output) an expression
    jobEnd,            ! (input/output) an expression
    jobWeight          ! (input/output) an expression
)
```

Arguments

resourceCapacity This argument is the capacity that the single resource has available to handle multiple jobs at the same time. It is a integer valued expression and the unit of measurement of this expression should be commensurate to the unit of measurement of `jobWeight`. This expression may not involve variables.

jobBinding The index binding that specifies the jobs that need to be scheduled.

jobBegin An expression that involves variables. When this function is used in a constraint definition it should involve variables. The result is a vector with an element for each possible value of the indices in `jobBinding`. This argument is integer or element valued, i.e. no string, fractional or unit values.

jobDuration An expression that may involve variables. The result of this expression is an integer non-negative value. The result is a vector with an element for each possible value of the indices in `jobBinding`. This argument is integer valued, i.e. no element, string, fractional or unit values, but elements from the set `Integers` are allowed.

jobEnd An expression that involves variables. When this function is used in a constraint definition it should involve variables. This expression has the same data type as `jobBegin`. The result is a vector with an element for each possible value of the indices in `jobBinding`. This argument is integer or element valued, i.e. no string, fractional or unit values.

jobWeight An expression that may involve variables. The result of this expression is an integer non-negative value. The unit of measurement of this expression is commensurate with the unit of measurement of `lowerLimit` and `upperLimit`. The result is a vector with an element for each possible value of the indices in `jobBinding`. This argument is integer valued, i.e. no element, string, fractional or unit values, but elements from the set `Integers` are allowed.

This argument is integer or element valued, i.e. no string, fractional or unit values.

Return Value

This function returns 1 if the jobs can be scheduled within the resource limits. If the index domain argument `jobBinding` is empty, this function also returns 1. Otherwise it returns 0.

Note:

- The arguments of this function involve discrete AIMMS variables and AIMMS parameters, not AIMMS activities.
- This and similar constraints are also known in the Constraint Programming literature as `Cumulative` constraints.

See also:

- The examples at the function *cp::AllDifferent* (page 263) illustrate how the index binding and vector arguments are used.
- Constraint Programming on Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `cumulative`.

2.1.8 cp::Sequence

The function *cp::Sequence* (page 278) is used to limit the number of occurrences of a group of values in each contiguous sequence of a row of variables. It is used to model that some values may occur only a limited number of times in a contiguous subset of the variables.

Mathematical Formulation

The function `cp::Sequence(i,x_i,S,q,l,u[,c])` returns 1 if, for each contiguous sequence of length q, the number of times that x_i is in S is within the range $\{l..u\}$. `cp::Sequence(i,x_i,S,q,l,u,c)` is equivalent to

$$\begin{array}{lllll} \forall i = 1..n-q+1: & l \leq \sum_{j=0}^{q-1}(x_{i+j} \in S) & \leq u & c = 0 \\ \forall i = 1..n: & l \leq \sum_{j=0}^{q-1}(x_{(i+j-1)\%n+1} \in S) & \leq u & c \neq 0 \end{array}$$

Function Prototype

```
cp::Sequence(
    inspectedBinding,  ! (input) an index binding
    inspectedValues,   ! (input/output) an expression
    lookupValues,      ! (input) a set valued expression
    sequenceLength,    ! (input) an expression
    lowerBound,        ! (input) an expression
    upperBound,        ! (input) an expression
    cyclic             ! (optional, input) an expression
)
```

Arguments

inspectedBinding The index binding for which the `inspectedValues` expression should be inspected on occurences of values in the `lookupValues` set.

inspectedValues The expression indexed over `inspectedBinding` for which the number of occurrences of the values in `lookupValues` is limited per subsequence. This expression may involve variables, but can only contain integer or element values (i.e. no strings, fractional or unit values).

lookupValues The set containing the particular values that should occur only a limited number of times in each subsequence. This set valued expression should be a subset of the range of `inspectedValues` and does not involve variables.

sequenceLength The sequence length. An expression that does not involve variables. This argument should be in the range $\{1..\texttt{card}(\texttt{range}(\texttt{inspectedValues}))\}$.

lowerBound The lower bound on the number of occurences. This expression does not involve variables. This argument should be in the range $\{0..\texttt{upperBound}\}$.

upperBound The upper bound on the number of occurences. This expression does not involve variables. This argument should be in the range $\{\texttt{lowerBound}..\texttt{sequenceLength}\}$.

cyclic An optional expression that indicates whether cyclic subsequences should also be inspected. E.g. when you have a set 1,2,3,4,5 then 4,5,1 is a cyclic subsequence of length 3. The `cyclic` expression cannot involve variables and the default of this argument is 0.

Return Value

This function returns 1 if the above condition is met.

Example

In car sequencing the constraint below ensures that no more `cars` of class `c` with option `o` are built in a sequence of length `blockSize(o)` than `maxCarsPerBlock(o)`. Here, the indexed set `classesHavingOption(o)` is, for each option `o`, the classes of car that have that option.

```
Constraint respectCapacity {
    IndexDomain  : (o);
    Definition   : {
        cp::Sequence(
            inspectedBinding :  i,
            inspectedValues  :  car(i),
            lookupValues     :  classesHavingOption(o),
            sequenceLength   :  blockSize(o),
            lowerBound       :  0,
            upperBound       :  maxCarsPerBlock(o) )
    }
}
```

In crew scheduling the constraint below ensures that after a flight an attendant `att` has at least two days off (works at most one day in each sequence of three days). The value 1 is converted to the set {1} by AIMMS.

```
Constraint AssureDaysOff {
    IndexDomain  : (att);
```

(continues on next page)

(continued from previous page)

```
    Definition   : {
        cp::Sequence(
            inspectedBinding :  f,
            inspectedValues  :  CrewOnFlight(att, f),
            lookupValues     :  1,
            sequenceLength   :  3,
            lowerBound       :  0,
            upperBound       :  1,
            cyclic           :  1)
    }
}
```

See also:

- The functions *cp::Count* (page 272) and *cp::Cardinality* (page 268).
- Constraint Programming on Constraint Programming in the Language Reference.
- The Global Constraint Catalog, which references this function as `among_seq`.

2.1.9 cp::SequentialSchedule

The function `cp::SequentialSchedule(j,s_j,d_j,e_j)` models a resource that can handle only one job at a time. A job j is scheduled from start time s_j until, but not including, end time e_j and over a number of periods d_j. This function returns 1 if the jobs are scheduled such that no two jobs overlap.

Mathematical Formulation

`cp::SequentialSchedule(j,s_j,d_j,e_j)` is equivalent to

$$\forall i, j, i \neq j : (s_i + d_i \leq s_j) \vee (s_j + d_j \leq s_i)$$
$$\forall j : s_j + d_j = e_j$$

Function Prototype

```
cp::SequentialSchedule(
        jobBinding,  ! (input) an index binding
        jobBegin,    ! (input) an expression
        jobDuration, ! (input) an expression
        jobEnd       ! (input) an expression
)
```

Arguments

jobBinding An index binding that specifies and possibly limits the scope of indices. This argument follows the syntax of the index binding argument of iterative operators.

jobBegin An expression that involves variables. The result is a vector with an element for each possible value of the indices in `jobBinding`.

jobDuration An expression that may involve variables. The result of this expression is an integer non-negative value. The result is a vector with an element for each possible value of the indices in `jobBinding`.

jobEnd An expression that involves variables. This expression has the same data type as `jobBegin`. The result is a vector with an element for each possible value of the indices in `jobBinding`.

Return Value

This function returns 1 if the jobs can be scheduled such that no two jobs overlap. If the index binding argument `job` is empty, this function will return 1. Otherwise it returns 0.

Note:

- The arguments to this function involve discrete AIMMS variables and AIMMS parameters, not AIMMS activities.
- This and similar constraints are also known in the Constraint Programming literature as `unary` or `disjunctive` constraints.

Example

The following example models the intuitive idea that with an increase in the size of a task also the time window in which that task is to be executed increases.

```
Parameter nrTasks {
    Definition   :  10;
}
Parameter smallestWidth {
    Definition   :  4;
}
Set tasks {
    Index        :  t;
    Definition   :  elementrange( 1, nrTasks, 1, 'task');
}
Parameter release {
    IndexDomain  :  (t);
    Definition   :  Ord(t);
}
Parameter deadline {
    IndexDomain  :  (t);
    Definition   :  2*nrTasks-Ord(t)+smallestWidth;
}
Parameter processingTime {
    IndexDomain  :  (t);
```

(continues on next page)

(continued from previous page)

```
    Definition   :  ceil(0.125*(deadline(t) - release(t)));
}
Variable startTime {
    IndexDomain  :  (t);
    Range        :  {
        {release(t) .. deadline(t)}
    }
}
Variable endTime {
    IndexDomain  :  (t);
    Range        :  {
        {release(t) .. deadline(t)}
    }
}
Constraint UnaryResource {
    Definition   : {
        cp::SequentialSchedule(t, startTime(t),
            processingTime(t), endTime(t))
    }
}
```

This leads to the following results (extracted from the listing file):

name	lower	level	upper
startTime('task01')	1	1	23
startTime('task02')	2	18	22
startTime('task03')	3	15	21
startTime('task04')	4	4	20
startTime('task05')	5	13	19
startTime('task06')	6	6	18
startTime('task07')	7	11	17
startTime('task08')	8	8	16
startTime('task09')	9	9	15
startTime('task10')	10	10	14
endTime('task01')	1	4	23
endTime('task02')	2	21	22
endTime('task03')	3	18	21
endTime('task04')	4	6	20
endTime('task05')	5	15	19
endTime('task06')	6	8	18
endTime('task07')	7	13	17
endTime('task08')	8	9	16
endTime('task09')	9	10	15
endTime('task10')	10	11	14

The following Gantt chart illustrates that the solution satisfies the restricition imposed by *cp::SequentialSchedule* (page 280).

See also:

- The examples at the function *cp::AllDifferent* (page 263) illustrate how the index binding and vector arguments are used.
- Constraint Programming on Constraint Programming in the Language Reference.

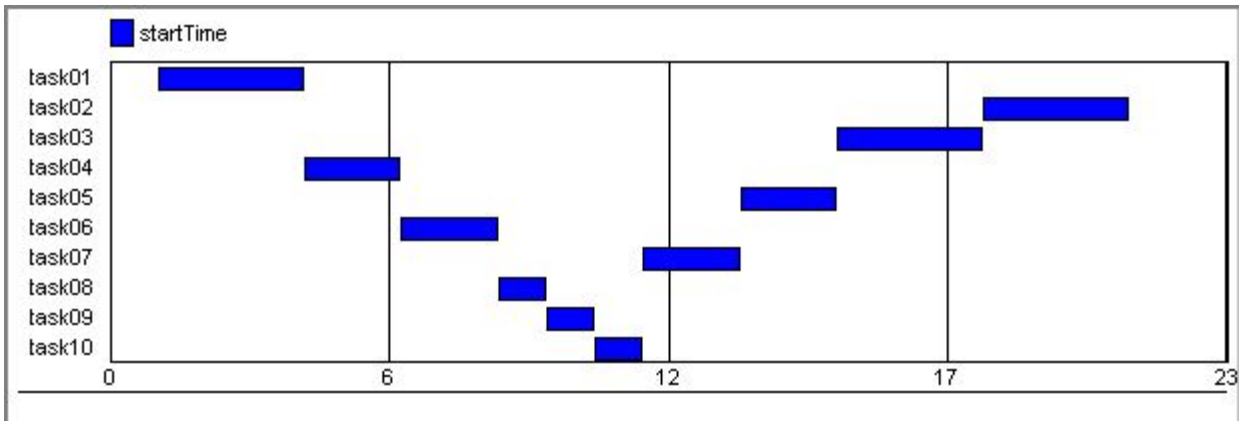


Fig. 2.2: Gantt chart for solution of *cp::SequentialSchedule* (page 280)

- The Global Constraint Catalog, which references this function as `disjunctive`.

2.2 Scheduling Functions

AIMMS supports the following functions for scheduling:

2.2.1 cp::ActivityBegin

The function `cp::ActivityBegin(a,d)` returns the begin of activity a when it is present or default value d when it is absent.

Mathematical Formulation

The function `cp::ActivityBegin(a,d)` is equivalent to

$$\begin{cases} a.\texttt{begin} & \text{if } a.\texttt{present} \\ d & \text{otherwise} \end{cases}$$

This function is typically used in scheduling problems to link activities to other components of the problem.

```
cp::ActivityBegin(
        optionalActivity,    ! (input) an expression
        absentValue          ! (input) an expression
)
```

Arguments

optionalActivity An expression resulting in an activity. This activity may have the property `optional`.

absentValue An expression that results in the value used when activity `optionalActivity` is absent. The result of this expression is an element in the schedule domain of the activity. This expression cannot involve variables.

Return Value

This function returns an element in the schedule domain of the activity and this element is the begin of an activity when that activity is present or a specified default value when it is not.

Example

In the example below, we require that the beginning of the shift represented by element variable `evShift` matches the begin of the optional activity `myAct`.

```
Constraint linkShiftActivity {
    Definition :  cp::ActivityBegin( myAct, first(myCal)) = beginHour(evShift)␣
↪);
}
```

See also:

The functions *cp::Count* (page 272) and *cp::ActivityEnd* (page 284).

2.2.2 cp::ActivityEnd

The function `cp::ActivityEnd(a,d)` returns the end of activity a if it is present or default value d when it is absent.

Mathematical Formulation

The function `cp::ActivityEnd(a,d)` is equivalent to

$$\begin{cases} a.\texttt{end} & \text{if } a.\texttt{present} \\ d & \text{otherwise} \end{cases}$$

This function is typically used in scheduling problems to link activities to other components of the problem.

```
cp::ActivityEnd(
        optionalActivity,    ! (input) an expression
        absentValue          ! (input) an expression
)
```

Arguments

optionalActivity An expression resulting in an activity. This activity may have the property `optional`.

absentValue An expression that results in the value used when activity `optionalActivity` is absent. The result of this expression is an element in the schedule domain of the activity. This expression cannot involve variables.

Return Value

This function returns an element in the schedule domain of the activity and this element is the end of an activity when that activity is present or a specified default value when it is not.

Example

In the example below, we require that the end of the shift represented by element variable `evShift` matches the end of the optional activity `myAct`.

```
Constraint linkShiftActivity {
    Definition :  cp::ActivityEnd( myAct, last(myCal)) = endHour(evShift);
}
```

See also:

The functions *cp::Count* (page 272) and *cp::ActivityBegin* (page 283).

2.2.3 cp::ActivityLength

The function `cp::ActivityLength(a,d)` returns the length of activity a when present and default value d when absent.

Mathematical Formulation

The function `cp::ActivityLength(a,d)` is equivalent to

$$\begin{cases} a.\texttt{length} & \text{if } a.\texttt{present} \\ d & \text{otherwise} \end{cases}$$

This function is typically used in scheduling problems to link activities to other components of the problem.

```
cp::ActivityLength(
        optionalActivity,    ! (input) an expression
        absentValue          ! (input) an expression
)
```

Arguments

optionalActivity An expression resulting in an activity. This activity may have the property `optional`.

absentValue An expression that results in the value used when activity `optionalActivity` is absent. This expression cannot involve variables.

Return Value

This function returns the length of an activity when that activity is present or a specified default value when it is not.

Example

In the example below, we require that the length of an activity is 36, whether or not it is present. When the length of an activity is fixed, if it is present, then this type of constraint might improve the performance of the CP solver.

```
Constraint linkShiftActivity {
    Definition :  cp::ActivityLength( myAct, 36 ) = 36;
}
```

Note that the above constraint is automatically generated when the length attribute of activity `myAct` is specified as `36`.

See also:

The functions *cp::Count* (page 272) and *cp::ActivityBegin* (page 283).

2.2.4 cp::ActivitySize

The function `cp::ActivitySize(a,d)` returns the size of activity a when it is present or default value d when it is absent.

Mathematical Formulation

The function `cp::ActivitySize(a,d)` is equivalent to

$$\begin{cases} a.\texttt{size} & \text{if } a.\texttt{present} \\ d & \text{otherwise} \end{cases}$$

This function is typically used in scheduling problems to link activities to other components of the problem.

```
cp::ActivitySize(
        optionalActivity,    ! (input) an expression
        absentValue          ! (input) an expression
)
```

Arguments

optionalActivity An expression resulting in an activity. This activity may have the property `optional`.

absentValue An expression that results in the value used when activity `optionalActivity` is absent. This expression cannot involve variables.

Return Value

This function returns the size of an activity when that activity is present or a specified default value when it is not.

Example

In the example below, we require that the size of the shift represented by element variable `evShift` matches the size of the optional activity `myAct`.

```
Constraint linkShiftActivity {
    Definition :  cp::ActivitySize( myAct, 3) =, shiftSize(evShift);
}
```

See also:

The functions *cp::Count* (page 272) and *cp::ActivityBegin* (page 283).

2.2.5 cp::Alternative

The function `cp::Alternative(g,i,a_i,n)`, returns

- if activity g is not present, the value 1 if none of the activities a_i are present and 0 otherwise.
- if activity g is present, the value 1 if precisely n activities a_i are present and these present activities match the activity g.

The function `cp::Alternative(g,i,a_i,n)` is equivalent to

$$g.\texttt{Present} = 0 \Leftrightarrow \forall i : a_i.\texttt{Present} = 0$$

and

$$g.\texttt{Present} = 1 \Leftrightarrow \begin{cases} \sum_i a_i.\texttt{Present} = n \\ \forall i : a_i.\texttt{Present} \Rightarrow \begin{cases} g.\texttt{Begin} = a_i.\texttt{Begin} \\ g.\texttt{End} = a_i.\texttt{End} \end{cases} \end{cases}$$

This function is typically used in scheduling problems to denote selected (matching) activities.

```
cp::Alternative(
        globalActivity,    ! (input) an expression
        activityBinding,   ! (input) an activity binding
        subActivity,       ! (input) an expression
        noSelected         ! (optional) an expression
)
```

Arguments

globalActivity An expression resulting in an activity.

activityBinding An index domain that specifies and possibly limits the scope of indices. This argument follows the syntax of the index domain argument of iterative operators.

subActivity An expression resulting in an activity. The result is a vector with an element for each possible value of the indices in index domain `activityBinding`.

noSelected The number of alternatives, the default being 1. This expression may involve variables.

Return Value

This function returns 1 if the above condition is satisfied, or otherwise 0. When the index domain `activityBinding` is empty this function will return an error.

Example

In the example below we require precisely one of the activities `altAct(i)` to match the activity `chosenAct(i)`.

```
Constraint PreciselyOneAlternativeMatches {
    Definition   :  cp::Alternative( chosenAct, i, altAct(i) );
}
```

We could change the above example to allow multiple matches as follows:

```
Variable noMatches {
    Range        :  {
        { 1 .. n }
    }
}
Constraint AtLeastOneAlternativeMatches {
    Definition   :  cp::Alternative( chosenAct, i, altAct(i), noMatches );
}
```

Here, the number of matches is counted in the integer variable `noMatches`.

See also:

The functions *cp::Span* (page 302) and *cp::Synchronize* (page 303).

2.2.6 cp::BeginAtBegin

The function `cp::BeginAtBegin(a,b,d)` returns 1 if one of the activities a and b is absent, or if the begin of activity a plus a nonnegative time period d is equal to the begin of activity b. The function `cp::BeginAtBegin(a,b,d)` is equivalent to

$$\begin{array}{ll} a.\texttt{Present=0} & \vee \\ b.\texttt{Present=0} & \vee \\ a.\texttt{Begin} + d = b.\texttt{Begin} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::BeginAtBegin(
        firstActivity,    ! (input) an expression
        secondActivity,   ! (input) an expression
        delay             ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginBeforeBegin* (page 290) and *cp::BeginBeforeEnd* (page 291), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.7 cp::BeginAtEnd

The function `cp::BeginAtEnd(a,b,d)` returns 1 if one of the activities a and b is absent, or if the begin of activity a plus a nonnegative time period d is equal to the begin of activity b. The function `cp::BeginAtEnd(a,b,d)` is equivalent to

$$\begin{array}{ll} a\texttt{.Present=0} & \vee \\ b\texttt{.Present=0} & \vee \\ a\texttt{.Begin} + d = b\texttt{.End} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::BeginAtEnd(
        firstActivity,    ! (input) an expression
        secondActivity,   ! (input) an expression
        delay             ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginBeforeBegin* (page 290) and *cp::BeginBeforeEnd* (page 291), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.8 cp::BeginBeforeBegin

The function `cp::BeginBeforeBegin(a,b,d)` returns 1 if one of the activities a and b is absent, or if the begin of activity a plus a nonnegative time period d is equal to the begin of activity b. The function `cp::BeginBeforeBegin(a, b,d)` is equivalent to

$$\begin{array}{ll} a\texttt{.Present=0} & \vee \\ b\texttt{.Present=0} & \vee \\ a\texttt{.Begin} + d \leq b\texttt{.Begin} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::BeginBeforeBegin(
        firstActivity,    ! (input) an expression
        secondActivity,   ! (input) an expression
        delay             ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginAtBegin* (page 288) and *cp::BeginBeforeEnd* (page 291), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.9 cp::BeginBeforeEnd

The function `cp::BeginBeforeEnd(a,b,d)` returns 1 if one of the activities a and b is absent, or if the begin of activity a plus a nonnegative time period d is equal to the begin of activity b. The function `cp::BeginBeforeEnd(a, b,d)` is equivalent to

$$\begin{array}{ll} a\texttt{.Present=0} & \vee \\ b\texttt{.Present=0} & \vee \\ a\texttt{.Begin} + d \leq b\texttt{.End} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::BeginBeforeEnd(
        firstActivity,   ! (input) an expression
        secondActivity,  ! (input) an expression
        delay            ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginBeforeBegin* (page 290) and *cp::BeginAtEnd* (page 289), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.10 cp::BeginOfNext

The function *cp::BeginOfNext* (page 291) refers to the begin of the next activity in a sequence of activities. For a resource r, an activity a, timeslots l and d, the function `cp::BeginOfNext(r,a,l,d)` returns

- d if a is absent,
- l if a is present and scheduled as the last activity on r, and
- n`.begin` if a is present and not scheduled as the last activity on r, and n is the next activity of a scheduled on r.

```
cp::BeginOfNext(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        lastValue,            ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

lastValue An optional expression that results in an element in the problem schedule domain. The default value of this expression is the last element in the schedule domain of the sequential resource.

absentValue An optional expression that results in an element in the problem schedule domain. The default value of this expression is the first element in the problem schedule domain.

Return Value

This function returns an element in the problem schedule domain.

See also:

- The functions *cp::BeginOfPrevious* (page 292) and *cp::EndOfNext* (page 296), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.11 cp::BeginOfPrevious

The function *cp::BeginOfPrevious* (page 292) refers to the begin of the previous activity in a sequence of activities. For a resource r, an activity a, timeslots l and d, the function `cp::BeginOfNext(r,a,l,d)` returns

- d if a is absent,
- l if a is present and scheduled as the first activity on r, and
- p`.begin` if a is present and not scheduled as the last activity on r, and p is the previous activity of a scheduled on r.

```
cp::BeginOfPrevious(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
```

(continues on next page)

(continued from previous page)

```
        firstValue,          ! (optional) an expression
        absentValue          ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

firstValue An optional expression that results in an element in the problem schedule domain. The default value of this expression is the first element in the schedule domain of the sequential resource.

absentValue An optional expression that results in an element in the problem schedule domain. The default value of this expression is the first element in the problem schedule domain.

Return Value

This function returns an element in the problem schedule domain.

See also:

- The functions *cp::BeginOfNext* (page 291) and *cp::EndOfPrevious* (page 297), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.12 cp::EndAtBegin

The function `cp::EndAtBegin(a,b,d)` returns 1 if one of the activities a and b is absent, or if the begin of activity a plus a nonnegative time period d is equal to the begin of activity b. The function `cp::EndAtBegin(a,b,d)` is equivalent to

$$\begin{array}{ll} a.\texttt{Present=0} & \vee \\ b.\texttt{Present=0} & \vee \\ a.\texttt{End} + d = b.\texttt{Begin} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::EndAtBegin(
        firstActivity,    ! (input) an expression
        secondActivity,   ! (input) an expression
        delay             ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginBeforeBegin* (page 290) and *cp::BeginBeforeEnd* (page 291), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.13 cp::EndAtEnd

The function `cp::EndAtEnd(a,b,d)` returns 1 if one of the activities a and b is absent, or if the end of activity a plus a nonnegative time period d is equal to the end of activity b. The function `cp::EndAtEnd(a,b,d)` is equivalent to

$$\begin{array}{ll} a\texttt{.Present=0} & \vee \\ b\texttt{.Present=0} & \vee \\ a\texttt{.End} + d = b\texttt{.End} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::EndAtEnd(
        firstActivity,   ! (input) an expression
        secondActivity,  ! (input) an expression
        delay            ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginBeforeBegin* (page 290) and *cp::BeginBeforeEnd* (page 291), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.14 cp::EndBeforeBegin

The function `cp::EndBeforeBegin(a,b,d)` returns 1 if one of the activities a and b is absent, or if the end of activity a plus a nonnegative time period d is less than or equal to the begin of activity b. The function `cp::EndBeforeBegin(a, b,d)` is equivalent to

$$\begin{array}{ll} a\texttt{.Present=0} & \vee \\ b\texttt{.Present=0} & \vee \\ a\texttt{.End} + d \leq b\texttt{.Begin} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::EndBeforeBegin(
        firstActivity,   ! (input) an expression
        secondActivity,  ! (input) an expression
        delay            ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::BeginBeforeBegin* (page 290) and *cp::BeginBeforeEnd* (page 291), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.15 cp::EndBeforeEnd

The function `cp::EndBeforeEnd(a,b,d)` returns 1 if one of the activities a and b is absent, or if the end of activity a plus a nonnegative time period d is less than or equal to the end of activity b. The function `cp::EndBeforeEnd(a, b,d)` is equivalent to

$$\begin{array}{ll} a\texttt{.Present=0} & \vee \\ b\texttt{.Present=0} & \vee \\ a\texttt{.End} + d \leq b\texttt{.End} & \end{array}$$

This function is typically used in scheduling constraints to place a sequencing restriction on activities.

```
cp::EndBeforeEnd(
        firstActivity,   ! (input) an expression
        secondActivity,  ! (input) an expression
        delay            ! (optional) an expression
)
```

Arguments

firstActivity An expression that results in an activity.

secondActivity An expression that results in an activity.

delay An optional expression that results in an integer number of time slots. This expression may involve variables. The default value of this expression is 0.

Return Value

This function returns 1 if the above condition is satisfied, and 0 if it is not.

See also:

- The functions *cp::EndAtEnd* (page 294) and *cp::EndBeforeBegin* (page 295), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.16 cp::EndOfNext

The function *cp::EndOfNext* (page 296) refers to the end of the next activity in a sequence of activities. For a resource r, an activity a, timeslots l and d, the function `cp::EndOfNext(r,a,l,d)` returns

- d if a is absent,
- l if a is present and scheduled as the last activity on r, and
- n`.end` if a is present and not scheduled as the last activity on r, and n is the next activity of a scheduled on r.

```
cp::EndOfNext(
        sequentialResource,  ! (input) an expression
        scheduledActivity,   ! (input) an expression
        lastValue,           ! (optional) an expression
        absentValue          ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

lastValue An optional expression that results in an element in the problem schedule domain. The default value of this expression is the last element in the schedule domain of the sequential resource.

absentValue An optional expression that results in an element in the problem schedule domain. The default value of this expression is the first element in the problem schedule domain.

Return Value

This function returns an element in the problem schedule domain.

See also:

- The functions *cp::BeginOfNext* (page 291) and *cp::EndOfPrevious* (page 297), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.17 cp::EndOfPrevious

The function *cp::EndOfPrevious* (page 297) refers to the end of the previous activity in a sequence of activities. For a resource r, an activity a, timeslots f and d, the function `cp::EndOfPrevious(r,a,f,d)` returns

- d if a is absent,
- f if a is present and scheduled as the first activity on r, and
- p`.end` if a is present and not scheduled as the first activity on r, and p is the previous activity of a scheduled on r.

```
cp::EndOfPrevious(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        firstValue,           ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

firstValue An optional expression that results in an element in the problem schedule domain. The default of this expression is the first element in the schedule domain of the sequential resource.

absentValue An optional expression that results in an element in the problem schedule domain. The default of this expression is the first element in the problem schedule domain.

Return Value

This function returns an element in the problem schedule domain.

See also:

- The functions *cp::BeginOfPrevious* (page 292) and *cp::EndOfNext* (page 296), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.18 cp::GroupOfNext

The function *cp::GroupOfNext* (page 298) refers to the group of the next activity in a sequence of activities. The group of an activity is specified in the `group definition` attribute of the sequential resource to ensure the sequencing. For a resource r, an activity a, groups l and d, the function `cp::GroupOfNext(r,a,l,d)` returns

- d if a is absent,
- l if a is present and scheduled as the last activity on r, and
- $GroupOf(r, n)$ if a is present and not scheduled as the last activity on r, and n is the next activity of a scheduled on r.

```
cp::GroupOfNext(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        lastValue,            ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

lastValue An optional expression that results in a group. The default value of this expression is the last element in the group set of the sequential resource.

absentValue An optional expression that results in a group. The default value of this expression is the last element in the group set of the sequential resource.

Return Value

This function returns a group.

See also:

- The functions *cp::BeginOfNext* (page 291) and *cp::EndOfPrevious* (page 297), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.19 cp::GroupOfPrevious

The function *cp::GroupOfPrevious* (page 298) refers to the group of the previous activity in a sequence of activities. The group of an activity is specified in the `group definition` attribute of the sequential resource to ensure the sequencing. For a resource r, an activity a, groups f and d, the function `cp::GroupOfPrevious(r,a,f,d)` returns

- d if a is absent,
- f if a is present and scheduled as the first activity on r, and
- $GroupOf(r, p)$ if a is present and not scheduled as the first activity on r, and p is the previous activity of a scheduled on r.

```
cp::GroupOfPrevious(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        firstValue,           ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

firstValue An optional expression that results in a group. The default value of this expression is the first element of the group set of the sequential resource.

absentValue An optional expression that results in a group. The default value of this expression is the first element of the group set of the sequential resource.

Return Value

This function returns a group.

See also:

- The functions *cp::BeginOfPrevious* (page 292) and *cp::EndOfNext* (page 296), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.20 cp::LengthOfNext

The function *cp::LengthOfNext* (page 299) refers to the length of the next activity in a sequence of activities. A length is an integer in the range $\{0..card(\texttt{problem schedule domain}) - 1\}$. For a resource r, an activity a, lengths l and d, the function `cp::LengthOfNext(r,a,l,d)` returns

- d if a is absent,
- l if a is present and scheduled as the last activity on r, and
- n`.length` if a is present and not scheduled as the last activity on r, and n is the next activity of a scheduled on r.

```
cp::LengthOfNext(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        lastValue,            ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

lastValue An optional expression that results in a length. The default value of this expression is 0.

absentValue An optional expression that results in a length. The default value of this expression is 0.

Return Value

This function returns a length.

See also:

- The functions *cp::BeginOfNext* (page 291) and *cp::EndOfPrevious* (page 297), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.21 cp::LengthOfPrevious

The function *cp::LengthOfPrevious* (page 300) refers to the length of the previous activity in a sequence of activities. A size is an integer in the range $\{0..card(\texttt{problem schedule domain}) - 1\}$. For a resource r, an activity a, sizes f and d, the function `cp::LengthOfPrevious(r,a,f,d)` returns

- d if a is absent,
- f if a is present and scheduled as the first activity on r, and
- p`.length` if a is present and not scheduled as the first activity on r, and p is the previous activity of a scheduled on r.

```
cp::LengthOfPrevious(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        firstValue,           ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

firstValue An optional expression that results in a length. The default value of this expression is 0.

absentValue An optional expression that results in a length. The default value of this expression is 0.

Return Value

This function returns a length.

See also:

- The functions *cp::BeginOfPrevious* (page 292) and *cp::EndOfNext* (page 296), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.22 cp::SizeOfNext

The function *cp::SizeOfNext* (page 301) refers to the size of the next activity in a sequence of activities. A size is an integer in the range $\{0..card(\texttt{problem schedule domain}) - 1\}$. For a resource r, an activity a, sizes l and d, the function `cp::SizeOfNext(r,a,l,d)` returns

- d if a is absent,
- l if a is present and scheduled as the last activity on r, and
- n`.size` if a is present and not scheduled as the last activity on r, and n is the next activity of a scheduled on r.

```
cp::SizeOfNext(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        lastValue,            ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

lastValue An optional expression that results in a size. The default value of this expression is 0.

absentValue An optional expression that results in a size. The default value of this expression is 0.

Return Value

This function returns a size.

See also:

- The functions *cp::BeginOfNext* (page 291) and *cp::EndOfPrevious* (page 297), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.23 cp::SizeOfPrevious

The function *cp::SizeOfPrevious* (page 302) refers to the size of the previous activity in a sequence of activities. A size is an integer in the range $\{0..card(\texttt{problem schedule domain}) - 1\}$. For a resource r, an activity a, sizes f and d, the function `cp::SizeOfPrevious(r,a,f,d)` returns

- d if a is absent,
- f if a is present and scheduled as the first activity on r, and
- p`.size` if a is present and not scheduled as the first activity on r, and p is the previous activity of a scheduled on r.

```
cp::SizeOfPrevious(
        sequentialResource,   ! (input) an expression
        scheduledActivity,    ! (input) an expression
        firstValue,           ! (optional) an expression
        absentValue           ! (optional) an expression
)
```

Arguments

sequentialResource An expression that results in a sequential resource.

scheduledActivity An expression that results in an activity.

firstValue An optional expression that results in a size. The default of this expression is 0.

absentValue An optional expression that results in a size. The default of this expression is 0.

Return Value

This function returns a size.

See also:

- The functions *cp::BeginOfPrevious* (page 292) and *cp::EndOfNext* (page 296), and
- Constraint Programming on Constraint Programming in the Language Reference.

2.2.24 cp::Span

The function `cp::Span(g,i,a_i)` returns 1 if activity g and activities a_i are all not present, or if the begin of present activity g is equal to the first present activity a_i and the end of activity g is equal to the end of the last present activity a_i. The function `cp::Span(g,i,a_i)` is equivalent to

$$g.\texttt{Present} = 0 \Leftrightarrow \forall i : a_i.\texttt{Present} = 0$$

and

$$g.\texttt{Present} = 1 \Leftrightarrow \begin{cases} \exists i | a_i.\texttt{Present} \\ g.\texttt{Begin} = \min_{i|a_i.\texttt{Present}} a_i.\texttt{Begin} \\ g.\texttt{End} = \max_{i|a_i.\texttt{Present}} a_i.\texttt{End} \end{cases}$$

This function is typically used in scheduling problems to split an activity into sub activities.

```
cp::Span(
        globalActivity,    ! (input) an expression
        activityBinding,   ! (input) an index domain
        subActivity        ! (input) an expression
)
```

Arguments

globalActivity An expression resulting in an activity.

activityBinding An index domain that specifies and possibly limits the scope of indices. This argument follows the syntax of the index domain argument of iterative operators.

subActivity An expression resulting in an activity. The result is a vector with an element for each possible value of the indices in index domain `activityBinding`.

Return Value

This function returns 1 if the above condition is satisfied, 0 otherwise. When the index domain `i` is empty this function will return an error.

See also:

The functions *cp::Alternative* (page 287) and *cp::Synchronize* (page 303).

2.2.25 cp::Synchronize

The function `cp::Synchronize(g,i,a_i)` returns 1 if activity g is not present, or if all present activities a_i match activity g. The function `cp::Synchronize(g,i,a_i)` is equivalent to

$$g.\texttt{Present} \Rightarrow \forall i | a_i.\texttt{Present} : \begin{cases} g.\texttt{Begin} = a_i.\texttt{Begin} \\ g.\texttt{End} = a_i.\texttt{End} \end{cases}$$

This function is typically used in scheduling problems to synchronize activities.

```
cp::Synchronize(
        globalActivity,    ! (input) an expression
        activityBinding,   ! (input) an index domain
        subActivity        ! (input) an expression
)
```

Arguments

globalActivity An expression resulting in an activity.

activityBinding An index domain that specifies and possibly limits the scope of indices. This argument follows the syntax of the index domain argument of iterative operators.

subActivity An expression resulting in an activity. The result is a vector with an element for each possible value of the indices in index domain `activityBinding`.

Return Value

This function returns 1 if the above condition is satisfied, 0 otherwise. When the index domain `activityBinding` is empty this function will return an error.

See also:

The functions *cp::Alternative* (page 287) and *cp::Span* (page 302).

2.3 The GMP Library

Through the GMP library you have direct access to mathematical program instances generated by AIMMS, allowing you to implement advanced algorithms in an efficient manner. The GMP routines can also be used for nonlinear models, unless specified otherwise. All procedures and functions in the GMP library are part of the `GMP` namespace in AIMMS. This namespace is subdivided into the following functional namespaces:

2.3.1 GMP::Benders Procedures and Functions

AIMMS supports the following procedures and functions for implementing an automatic Benders' decomposition algorithm:

GMP::Benders::AddFeasibilityCut

The procedure *GMP::Benders::AddFeasibilityCut* (page 304) generates a feasibility cut for a Benders' master problem using the solution of a Benders' subproblem (or the corresponding feasibility problem). This procedure is typically used in a Benders' decomposition algorithm.

```
GMP::Benders::AddFeasibilityCut(
     GMP1,              ! (input) a generated mathematical program
     GMP2,              ! (input) a generated mathematical program
     solution,          ! (input) a solution
     cutNo,             ! (input) a scalar reference
```

(continues on next page)

(continued from previous page)

```
    [tighten]          ! (optional, default 0) a scalar binary expression
)
```

Arguments

GMP1 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' master problem.

GMP2 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' subproblem (or the corresponding feasibility problem).

solution An integer scalar reference to a solution of *GMP2*.

cutNo An integer scalar reference to a cut number.

tighten A scalar binary value to indicate whether the feasibility cut should be tightened. If the value is 1, tightening is attempted.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP1* should have been created using the function *GMP::Benders::CreateMasterProblem* (page 308).
- The *GMP2* should have been created using the function *GMP::Benders::CreateSubProblem* (page 309) or the function *GMP::Instance::CreateFeasibility* (page 372).
- If the GMP that was created by *GMP::Benders::CreateSubProblem* (page 309) represents the dual of the Benders' subproblem then this GMP should be used as argument *GMP2*. If it represents the primal of the Benders' subproblem then first the feasibility problem should be created which then should be used as argument *GMP2*.
- The *solution* of the Benders' subproblem or feasibility problem (represented by *GMP2*) is used to generate an optimality cut for the Benders' master problem (represented by *GMP1*).
- A feasibility cut $a^T x \geq b$ can be tightened to $1^T x \geq 1$ if x is a vector of binary variables and $a_i \geq b > 0$ for all i.

Example

In the examples below we assume that the Benders' subproblem is infeasible. The way *GMP::Benders::AddFeasibilityCut* (page 304) is called depends on whether the primal or dual of the Benders' subproblem was generated. In the first example we use the dual. In that case an unbounded extreme ray is used to create a feasibility cut. See Benders' Decomposition - Textbook Algorithm of the Language Reference.

```
! Initialization.
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
```

(continues on next page)

(continued from previous page)

```
                                          'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                          useDual : 1, normalizationType : 0 );

NumberOfFeasibilityCuts := 1;

! Switch on solver option for calculating unbounded extreme ray.
GMP::Instance::SetOptionValue( gmpS, 'unbounded ray', 1 );

! First iteration of Benders' decomposition algorithm (simplified).
GMP::Instance::Solve( gmpM );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::Instance::Solve( gmpS );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;
if ( ProgramStatus = 'Unbounded' ) then
    GMP::Benders::AddFeasibilityCut( gmpM, gmpS, 1, NumberOfFeasibilityCuts );
    NumberOfFeasibilityCuts += 1;
endif;
```

In the second example we use the primal of the Benders' subproblem. If that problem turns out to be infeasible then we solve a feasibility problem to get a solution of minimum infeasibility (according to some measurement). The shadow prices of the constraints and the reduced costs of the variables in that solution are used to create a feasibility cut. See Benders' Decomposition - Textbook Algorithm of the Language Reference.

```
! Initialization.
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                          'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                          useDual : 0, normalizationType : 0 );

NumberOfFeasibilityCuts := 1;

! First iteration of Benders' decomposition algorithm (simplified).
GMP::Instance::Solve( gmpM );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::Instance::Solve( gmpS );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;
if ( ProgramStatus = 'Infeasible' ) then
    gmpF := GMP::Instance::CreateFeasibility( gmpS, "FeasProb", useMinMax : 1␣
↪);
```

(continues on next page)

(continued from previous page)

```
    GMP::Instance::Solve( gmpF );

    GMP::Benders::AddFeasibilityCut( gmpM, gmpF, 1, NumberOfFeasibilityCuts );
    NumberOfFeasibilityCuts += 1;
endif;
```

See also:

The routines *GMP::Benders::CreateMasterProblem* (page 308), *GMP::Benders::CreateSubProblem* (page 309), *GMP::Benders::AddOptimalityCut* (page 307), *GMP::Instance::CreateFeasibility* (page 372), *GMP::SolverSession::AddBendersFeasibilityCut* (page 522) and *GMP::SolverSession::AddBendersOptimalityCut* (page 524).

GMP::Benders::AddOptimalityCut

The procedure *GMP::Benders::AddOptimalityCut* (page 307) generates an optimality cut for a Benders' master problem using the (dual) solution of a Benders' subproblem. This procedure is typically used in a Benders' decomposition algorithm.

```
GMP::Benders::AddOptimalityCut(
     GMP1,              ! (input) a generated mathematical program
     GMP2,              ! (input) a generated mathematical program
     solution,          ! (input) a solution
     cutNo              ! (input) a scalar reference
)
```

Arguments

GMP1 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' master problem.

GMP2 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' subproblem.

solution An integer scalar reference to a solution of *GMP2*.

cutNo An integer scalar reference to a cut number.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP1* should have been created using the function *GMP::Benders::CreateMasterProblem* (page 308).
- The *GMP2* should have been created using the function *GMP::Benders::CreateSubProblem* (page 309).
- The *solution* of the Benders' subproblem (represented by *GMP2*) is used to generate an optimality cut for the Benders' master problem (represented by *GMP1*). More precise, the shadow prices of the constraints and the reduced costs of the variables in the Benders' subproblem are used.

Example

In the example below we assume that the Benders' subproblem is feasible. Its program status is stored in the element parameter ProgramStatus with range *AllSolutionStates* (page 807). Note that the subproblem is updated before it is solved.

```
! Initialization.
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                        0, 0 );

NumberOfOptimalityCuts := 1;

! First iteration of Benders' decomposition algorithm (simplified).
GMP::Instance::Solve( gmpM );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::Instance::Solve( gmpS );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;
if ( ProgramStatus = 'Optimal' ) then
    GMP::Benders::AddOptimalityCut( gmpM, gmpS, 1, NumberOfOptimalityCuts );
    NumberOfOptimalityCuts += 1;
endif;
```

See also:

The routines *GMP::Benders::CreateMasterProblem* (page 308), *GMP::Benders::CreateSubProblem* (page 309), *GMP::Benders::AddFeasibilityCut* (page 304), *GMP::SolverSession::AddBendersFeasibilityCut* (page 522) and *GMP::SolverSession::AddBendersOptimalityCut* (page 524).

GMP::Benders::CreateMasterProblem

The function *GMP::Benders::CreateMasterProblem* (page 308) creates a Benders' master problem for a generated mathematical program. This master problem is typically used in a Benders' decomposition algorithm.

```
GMP::Benders::CreateMasterProblem(
     GMP,                ! (input) a generated mathematical program
     Variables,          ! (input) a set of variables
     name,               ! (input) a string expression
     [feasibilityOnly],  ! (optional, default 0) a scalar value
     [addConstraints]    ! (optional, default 0) a scalar value
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Variables A subset of *AllVariables* (page 831).

name A string that holds the name for the Benders' master problem.

feasibilityOnly A scalar binary value to indicate whether this function should (temporary) reformulate the original problem such that the Benders' subproblem becomes a pure feasibility problem.

addConstraints A scalar binary value to indicate whether this function should try to automatically add tightening constraints to the Benders' master problem.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- A call to *GMP::Benders::CreateMasterProblem* (page 308) is typically followed by a call to the function *GMP::Benders::CreateSubProblem* (page 309).
- The *GMP* must have type LP, MIP or RMIP.
- This function cannot be used if the *GMP* is created by the function *GMP::Instance::GenerateStochasticProgram* (page 387).
- The *Variables* argument specifies the variables that become part of the Benders' master problem. All other variables will become part of the Benders' subproblem. The objective variable should be part of the set of master problem variables; if the objective variable is not included in the set *Variables* then this procedure will automatically add it.
- If the *GMP* contains integer variables then they all must be included in the set *Variables*.
- The *feasibilityOnly* argument is discussed in more detail in Subproblem as Pure Feasibility Problem of the Language Reference.
- The *addConstraints* argument is discussed in more detail in Subproblem as Pure Feasibility Problem of the Language Reference.

Example

If the math program has type MIP then often the set of master problem variables equals the set *AllIntegerVariables* (page 823).

```
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );
```

See also:

The routines *GMP::Benders::CreateSubProblem* (page 309), *GMP::Benders::AddFeasibilityCut* (page 304) and *GMP::Benders::AddOptimalityCut* (page 307).

GMP::Benders::CreateSubProblem

The function *GMP::Benders::CreateSubProblem* (page 309) creates a Benders' subproblem for a generated mathematical program. This subproblem is typically used in a Benders' decomposition algorithm.

```
GMP::Benders::CreateSubProblem(
     GMP1,               ! (input) a generated mathematical program
     GMP2,               ! (input) a generated mathematical program
     name,               ! (input) a string expression
     [useDual],          ! (optional, default 0) a scalar value
     [normalizationType] ! (optional, default 0) a scalar value
)
```

Arguments

GMP1 An element in the set *AllGeneratedMathematicalPrograms* (page 833).

GMP2 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' master problem.

name A string that holds the name for the Benders' subproblem.

useDual A scalar binary value to indicate whether this function should create the primal (value 0) or dual (value 1) of the subproblem.

normalizationType A scalar value to indicate which kind of normalization this function should use. Value 0 implies that the standard normalization is used. Value 1 implies that the normalization condition introduced by Fischetti, Salvagnin and Zanette (2010) is used. The normalization condition is added as a constraint to the subproblem.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- The *GMP1* must have type LP, MIP or RMIP.
- The *GMP2* should have been created using the function *GMP::Benders::CreateMasterProblem* (page 308). Note that the call to that function specifies how the variables (and constraints) are divided among the master and subproblem.
- The *useDual* argument is discussed in more detail in Primal Versus Dual Subproblem of the Language Reference.
- The *normalizationType* argument is discussed in more detail in Primal Versus Dual Subproblem of the Language Reference.

Example

If the math program has type MIP then often the set of master problem variables equals the set *AllIntegerVariables* (page 823). All other variables automatically become part of the subproblem.

```
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                        0, 0 );
```

See also:

The routines *GMP::Benders::CreateMasterProblem* (page 308), *GMP::Benders::AddFeasibilityCut* (page 304), *GMP::Benders::AddOptimalityCut* (page 307), *GMP::Benders::UpdateSubProblem* (page 311) and *GMP::Instance::CreateFeasibility* (page 372).

GMP::Benders::UpdateSubProblem

The procedure *GMP::Benders::UpdateSubProblem* (page 311) updates a Benders' subproblem (or the corresponding feasibility problem) using the solution of a Benders' master problem. This procedure is typically used in a Benders' decomposition algorithm.

```
GMP::Benders::UpdateSubProblem(
     GMP1,              ! (input) a generated mathematical program
     GMP2,              ! (input) a generated mathematical program
     solution,          ! (input) a solution
     [round]            ! (optional, default 0) a scalar value
)
```

Arguments

GMP1 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' subproblem.

GMP2 An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' master problem.

solution An integer scalar reference to a solution of *GMP2*.

round A binary scalar indicating whether the level values of the integer variables (if any) should be rounded to the nearest integer value in the solution used to update the subproblem.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP1* should have been created using the function *GMP::Benders::CreateSubProblem* (page 309) or the function *GMP::Instance::CreateFeasibility* (page 372).
- The *GMP2* should have been created using the function *GMP::Benders::CreateMasterProblem* (page 308).
- The *solution* of the Benders' master problem (represented by *GMP2*) is used to update the Benders' subproblem (represented by *GMP1*). That is, the right-hand-side values of the constraints in the subproblem are reevaluated using the level values of the variables in the solution of the Benders' master problem.

Example

Before solving the subproblem it should be updated using a solution of the master problem. In the example below we use the solution at position 1 in the solution repository of the GMP belonging to the master problem.

```
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                        0, 0 );

GMP::Instance::Solve( gmpM );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::Instance::Solve( gmpS );
```

See also:

The functions *GMP::Benders::CreateMasterProblem* (page 308), *GMP::Benders::CreateSubProblem* (page 309) and *GMP::Instance::CreateFeasibility* (page 372).

2.3.2 GMP::Coefficient Procedures and Functions

AIMMS supports the following procedures and functions for modifying the coefficient matrix associated with a generated mathematical program instance:

GMP::Coefficient::Get

The function *GMP::Coefficient::Get* (page 312) retrieves a (linear) coefficient in a generated mathematical program.

```
GMP::Coefficient::Get(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the model or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

column A scalar reference to an existing column in the model or the number of that column in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The value of the specified coefficient in the generated mathematical program.

Note: In case the generated mathematical program is nonlinear, this function will return 0 if the column is part of a nonlinear term in the row. However, if the row is pure quadratic then this function will return the linear coefficient value for a quadratic column.

Example

Consider a GMP containing a constraint `e1` with definition `2*x1 + 3*x2 + x2^3 = 0`. Then `GMP::Coefficient::Get(GMP,e1,x1)` will return 2. Because column `x2` is part of the nonlinear term `x2^3`, `GMP::Coefficient::Get(GMP,e1,x2)` will return 0.

See also:

The routines *GMP::Coefficient::Set* (page 316) and *GMP::QuadraticCoefficient::Get* (page 442).

GMP::Coefficient::GetMinAndMax

The procedure *GMP::Coefficient::GetMinAndMax* (page 313) determines the minimum and maximum value of (linear) coefficients in a generated mathematical program. The domain of this evaluation is indicated by the given row and column sets.

```
GMP::Coefficient::GetMinAndMax(
     GMP,             ! (input)  a generated mathematical program
     rowSet,          ! (input)  a subset of Integers
```

(continues on next page)

(continued from previous page)

```
    colSet,          ! (input)  a subset of Integers
    minCoef,         ! (output) a real-valued parameter
    maxCoef,         ! (output) a real-valued parameter
    [absSense]       ! (optional, default 1) a scalar value
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set Integers, representing a set of row numbers. Each row number should be in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

colSet A subset of the set Integers, representing a set of column numbers. Each column should be in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

minCoef A real-valued parameter indicating the minimum coefficient in the *GMP*.

maxCoef A real-valued parameter indicating the maximum coefficient in the *GMP*.

absSense A binary scalar indicating whether the absolute value of coefficients should be taken into consideration when determining minimum and maximum coefficients. The default is 1, meaning that the absolute value of coefficients are considered.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: This procedure neglects zero coefficients in determining the minimum and maximum values.

See also:

The routines *GMP::Coefficient::Get* (page 312) and `GMP::Coefficient::GetMultiRaw`.

GMP::Coefficient::GetRaw

The procedure *GMP::Coefficient::GetRaw* (page 314) retrieves a collection of (linear) coefficients in a generated mathematical program. The retrieved collection is indicated by the given row and column sets.

```
GMP::Coefficient::GetRaw(
    GMP,             ! (input)  a generated mathematical program
    rowSet,          ! (input)  a subset of Integers
    colSet,          ! (input)  a subset of Integers
    coef             ! (output) a real-valued parameter
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set Integers, representing a set of row numbers. Each row number should be in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

colSet A subset of the set Integers, representing a set of column numbers. Each column should be in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

coef A real-valued parameter over rowSet and colSet indicating the coefficient values for each row and column in rowSet and colSet.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: This procedure is much more efficient than calling the function *GMP::Coefficient::Get* (page 312) to get the coefficients of each row and column in rowSet and colSet individually.

See also:

The routine *GMP::Coefficient::Get* (page 312).

GMP::Coefficient::GetQuadratic

The function *GMP::Coefficient::GetQuadratic* (page 315) retrieves the value of a quadratic product between two columns in a generated mathematical program.

```
GMP::Coefficient::GetQuadratic(
     GMP,             ! (input) a generated mathematical program
     column1,         ! (input) a scalar reference or column number
     column2          ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column1,column2 A scalar reference to an existing column in the model or the number of that column in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The value of the specified quadratic term in the generated mathematical program.

Note:

- If *column1* equals *column2* then AIMMS multiplies the quadratic coefficient by 2 before it is returned by this function.
- This function operates on the objective. To get a quadratic coefficient in a row the function *GMP::QuadraticCoefficient::Get* (page 442) should be used.

See also:

The routines *GMP::Coefficient::SetQuadratic* (page 318) and *GMP::QuadraticCoefficient::Get* (page 442).

GMP::Coefficient::Set

The procedure *GMP::Coefficient::Set* (page 316) sets the value of a (linear) coefficient in a generated mathematical program.

```
GMP::Coefficient::Set(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     column,          ! (input) a scalar reference or column number
     value            ! (input) a scalar numerical value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the model or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

column A scalar reference to an existing column in the model or the number of that column in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value A scalar numerical value indicating the value for the coefficient.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use *GMP::Coefficient::SetMulti* (page 317) or *GMP::Coefficient::SetRaw* (page 319) if many coefficients have to be set because that will be more efficient.
- This procedure cannot be used if the column refers to the objective variable.

- In case the generated mathematical program is nonlinear, this procedure will fail if the column is part of a nonlinear term in the row. However, if the row is pure quadratic, then this procedure can be used to set the linear coefficient value for a quadratic column.
- GMP procedures operate on a generated mathematical program in which all variables are moved to the left-hand-side of each constraint. This can have an influence on the sign of the coeffients as demonstrated in the example below.

Example

Assume that we have the following variable and constraint declarations (in ams format):.

```
Variable y;
Variable z;
Variable x1;
Constraint c1 {
    Definition: x1 - 2*y - 3*z = 0;
}
Variable x2 {
    Definition: 2*y + 3*z;
}
```

To change the coefficient of variable `y` in constraint `c1` to 4 we use:

```
GMP::Coefficient::Set( myGMP, c1, y, 4 );
```

This results in the row `x1 + 4*y - 3*z = 0`.

The definition of variable `x2` is generated as the row `x2 - 2*y - 3*z = 0` by AIMMS. Therefore, using

```
GMP::Coefficient::Set( myGMP, x2_definition, y, -4 );
```

will result in the row `x2 - 4*y - 3*z = 0`.

See also:

The routines *GMP::Coefficient::Get* (page 312), *GMP::Coefficient::SetMulti* (page 317), *GMP::Coefficient::SetRaw* (page 319) and *GMP::QuadraticCoefficient::Set* (page 443).

GMP::Coefficient::SetMulti

The procedure *GMP::Coefficient::SetMulti* (page 317) sets the value of a range of (linear) coefficients for a group of columns and rows, belonging to a variable and constraint, in a generated mathematical program.

```
GMP::Coefficient::SetMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     row,            ! (input) a constraint expression
     column,         ! (input) a variable expression
     value           ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

column A variable that, combined with the *binding* domain, specifies the columns.

value The new coefficient for each combination of row and column, defined over the *binding* domain.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure cannot be used if the objective variable is passed as *column*.
- In case the generated mathematical program is nonlinear, this procedure will fail if one the columns is part of a nonlinear term in one of the rows. However, if the row is pure quadratic, then this procedure can be used to set the linear coefficient value for a quadratic column.
- GMP procedures operate on a generated mathematical program in which all variables are moved to the left-hand-side of each constraint. This can have an influence on the sign of the coeffients as demonstrated in the example of procedure *GMP::Coefficient::Set* (page 316).

Example

To set the coefficients of variable `x(j)` in constraint `c(i)` to `coef(i,j)` we can use:

```
for (i,j) do
    GMP::Column::Set( myGMP, c(i), x(j), coef(i,j) );
endfor;
```

It is more efficient to use:

```
GMP::Coefficient::SetMulti( myGMP, (i,j), c(i), x(j), coef(i,j) );
```

If we only want to set the coefficients of those `x(j)` for which `dom(j)` is unequal to zero, then we use:

```
GMP::Coefficient::SetMulti( myGMP, (i,j) | dom(j), c(i), x(j), coef(i,j) );
```

See also:

The routines *GMP::Coefficient::Get* (page 312), *GMP::Coefficient::Set* (page 316) and *GMP::QuadraticCoefficient::Set* (page 443).

GMP::Coefficient::SetQuadratic

The procedure *GMP::Coefficient::SetQuadratic* (page 318) sets the value of a quadratic product between two columns in a generated mathematical program.

```
GMP::Coefficient::SetQuadratic(
     GMP,             ! (input) a generated mathematical program
     column1,         ! (input) a scalar value or column number
     column2,         ! (input) a scalar value or column number
     value            ! (input) a scalar numerical value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column1,column2 A scalar reference to an existing column in the model or the number of that column in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value A scalar numerical value indicating the value for the quadratic term.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If *column1* equals *column2* then AIMMS multiplies the quadratic coefficient by 0.5 before it is stored (and passed to the solver).
- This procedure operates on the objective. To set a quadratic coefficient in a row the procedure *GMP::QuadraticCoefficient::Set* (page 443) should be used.

See also:

The routines *GMP::Coefficient::GetQuadratic* (page 315) and *GMP::QuadraticCoefficient::Set* (page 443).

GMP::Coefficient::SetRaw

The procedure *GMP::Coefficient::SetRaw* (page 319) sets the value of a range of (linear) coefficients for a group of columns and rows in a generated mathematical program.

```
GMP::Coefficient::SetRaw(
     GMP,             ! (input) a generated mathematical program
     rowSet,          ! (input) a subset of Integers
     colSet,          ! (input) a subset of Integers
     coef,            ! (input) a parameter
     changeZero       ! (input) a binary parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

coef A parameter over (*rowSet*, *colSet*) containing the new coefficient for each (row,column) combination.

changeZero A binary parameter over (*rowSet*, *colSet*) which can be used to mark coefficients with a value of 0 (zero) in *coef* that should be changed by this procedure.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- A coefficient with a value of 0 (zero) will be skipped by this procedure unless *changeZero* is set to 1 for the corresponding (row, column) combination.
- This procedure cannot be used if one of the columns refers to the objective variable.
- In case the generated mathematical program is nonlinear, this procedure will fail if one the columns is part of a nonlinear term in one of the rows. However, if the row is pure quadratic, then this procedure can be used to set the linear coefficient value for a quadratic column.
- GMP procedures operate on a generated mathematical program in which all variables are moved to the left-hand-side of each constraint. This can have an influence on the sign of the coeffients as demonstrated in the example of procedure *GMP::Coefficient::Set* (page 316).

Example

Assume that 'MP' is a mathematical program with the following declaration (in ams format):

```
Variable x {
    IndexDomain: t;
    Range: nonnegative;
}
Variable y {
    IndexDomain: t;
    Range: nonnegative;
}
Constraint c1 {
    IndexDomain: t;
    Definition: - 2 * x(t) + y(t) <= 4;
}
MathematicalProgram MP {
    Objective: obj;
    Direction: minimize;
    Type: LP;
}
```

To use *GMP::Coefficient::SetRaw* (page 319) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set ConstraintSet {
    SubsetOf: AllConstraints;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set RowSet {
    SubsetOf: Integers;
    Index: rr;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
Parameter Coef {
    IndexDomain: (rr,cc);
}
Parameter ChangeZero {
    IndexDomain: (rr,cc);
}
```

To set the coefficients of variables `x(t)` and `y(t)` in constraint `c1(t)` to 1 and 0, respectively, we can use:

```
myGMP := GMP::Instance::Generate( MP );

ConstraintSet := { 'c1' };
RowSet := GMP::Instance::GetRowNumbers( myGMP, ConstraintSet );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

Coef(rr,cc) := 1.0;
ChangeZero(rr,cc) := 0;

GMP::Coefficient::SetRaw( myGMP, RowSet, ColumnSet, Coef, ChangeZero );

VariableSet := { 'y' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

Coef(rr,cc) := 0.0;
ChangeZero(rr,cc) := 1;

GMP::Coefficient::SetRaw( myGMP, RowSet, ColumnSet, Coef, ChangeZero );
```

See also:

The routines *GMP::Coefficient::Get* (page 312), *GMP::Instance::GetColumnNumbers* (page 392) and *GMP::Instance::GetRowNumbers* (page 405).

2.3.3 GMP::Column Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing matrix columns associated with a generated mathematical program instance:

GMP::Column::Add

The procedure *GMP::Column::Add* (page 322) adds a column to a generated mathematical program.

```
GMP::Column::Add(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to a column.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use *GMP::Column::AddMulti* (page 322) if many columns corresponding to some variable have to be added, because that will be more efficient.
- Coefficients for this column can be added to the matrix by using the procedure *GMP::Coefficient::Set* (page 316). After calling *GMP::Column::Add* (page 322) the type and the lower and upper bound of the column are set according to the definition of the corresponding symbolic variable. By using the procedures *GMP::Column::SetType* (page 347), *GMP::Column::SetLowerBound* (page 343) and *GMP::Column::SetUpperBound* (page 350) the column type and the lower and upper bound can be changed.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Coefficient::Set* (page 316), *GMP::Column::AddMulti* (page 322), *GMP::Column::Delete* (page 323), *GMP::Column::SetType* (page 347), *GMP::Column::SetLowerBound* (page 343) and *GMP::Column::SetUpperBound* (page 350).

GMP::Column::AddMulti

The procedure *GMP::Column::AddMulti* (page 322) adds a group of columns, belonging to a variable, to a generated mathematical program.

```
GMP::Column::AddMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     column          ! (input) a variable expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: Coefficients for these columns can be added to the matrix by using the procedure *GMP::Coefficient::Set* (page 316). After calling *GMP::Column::AddMulti* (page 322) the type and the lower and upper bound of the columns are set according to the definition of the corresponding symbolic variable. By using the procedures *GMP::Column::SetType* (page 347), *GMP::Column::SetLowerBound* (page 343) and *GMP::Column::SetUpperBound* (page 350) the column types and the lower and upper bounds can be changed.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Coefficient::Set* (page 316), *GMP::Column::Add* (page 322), *GMP::Column::Delete* (page 323), *GMP::Column::SetType* (page 347), *GMP::Column::SetLowerBound* (page 343) and *GMP::Column::SetUpperBound* (page 350).

GMP::Column::Delete

The procedure *GMP::Column::Delete* (page 323) marks a column in a generated mathematical program as deleted.

```
GMP::Column::Delete(
     GMP,            ! (input) a generated mathematical program
     column          ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use *GMP::Column::DeleteMulti* (page 324) or *GMP::Column::DeleteRaw* (page 325) if many columns have to be deleted, because that will be more efficient.
- The column will not be printed in the constraint listing, nor be visible in the Math Program Inspector and it will be removed from any solver maintained copies.
- Use *GMP::Column::Add* (page 322) to undo this action.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::Add* (page 322), *GMP::Column::DeleteMulti* (page 324) and *GMP::Column::DeleteRaw* (page 325).

GMP::Column::DeleteMulti

The procedure *GMP::Column::DeleteMulti* (page 324) marks a group of columns, belonging to a variable, in a generated mathematical program as deleted.

```
GMP::Column::DeleteMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     column          ! (input) a variable expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The columns will not be printed in the constraint listing, nor be visible in the Math Program Inspector and they will be removed from any solver maintained copies.
- Use *GMP::Column::AddMulti* (page 322) to undo this action.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::AddMulti* (page 322) and *GMP::Column::Delete* (page 323).

GMP::Column::DeleteRaw

The procedure *GMP::Column::DeleteRaw* (page 325) marks a group of columns in a generated mathematical program as deleted.

```
GMP::Column::DeleteRaw(
     GMP,            ! (input) a generated mathematical program
     colSet          ! (input) a subset of Integers
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The columns will not be printed in the constraint listing, nor be visible in the Math Program Inspector and they will be removed from any solver maintained copies.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Column::DeleteRaw* (page 325) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
```

To delete the variable `x(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

GMP::Column::DeleteRaw( myGMP, ColumnSet );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392) and *GMP::Column::Delete* (page 323).

GMP::Column::Freeze

The procedure *GMP::Column::Freeze* (page 326) freezes a column in a generated mathematical program at the given value.

```
GMP::Column::Freeze(
     GMP,            ! (input) a generated mathematical program
     column,         ! (input) a scalar reference or column number
     value           ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value The new value that should be used to freeze the column value.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Use *GMP::Column::FreezeMulti* (page 327) or *GMP::Column::FreezeRaw* (page 328) if many columns have to be frozen, because that will be more efficient.
- The column remains visible in the constraint listing and math program inspector. In addition, it will be retained in solver maintained copies of the generated math program.
- Use *GMP::Column::Unfreeze* (page 354) to undo the freezing.
- During a call to procedure *GMP::Column::Freeze* (page 326) AIMMS stores the upper and lower bound of the column before the procedure was called. This information is used when procedure *GMP::Column::Unfreeze* (page 354) is called thereafter. This information is not copied by the function *GMP::Instance::Copy* (page 364).

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::FreezeMulti* (page 327), *GMP::Column::FreezeRaw* (page 328), *GMP::Column::Unfreeze* (page 354) and *GMP::Instance::Copy* (page 364).

GMP::Column::FreezeMulti

The procedure *GMP::Column::FreezeMulti* (page 327) freezes a group of columns, belonging to a variable, in a generated mathematical program.

```
GMP::Column::FreezeMulti(
    GMP,            ! (input) a generated mathematical program
    binding,        ! (input) an index binding
    column,         ! (input) a variable expression
    value           ! (input) a numerical expression
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

value The new value for each column, defined over the *binding* domain, that should be used to freeze the column value.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- The columns remain visible in the constraint listing and math program inspector. In addition, it will be retained in solver maintained copies of the generated math program.
- Use *GMP::Column::UnfreezeMulti* (page 355) to undo the freezing.
- During a call to procedure *GMP::Column::FreezeMulti* (page 327) AIMMS stores the upper and lower bound of the columns before the procedure was called. This information is used when procedure *GMP::Column::UnfreezeMulti* (page 355) is called thereafter. This information is not copied by the function *GMP::Instance::Copy* (page 364).

Example

To freeze variable `x(i)` to `demand(i)` we can use:

```
for (i) do
    GMP::Column::Freeze( myGMP, x(i), demand(i) );
endfor;
```

It is more efficient to use:

```
GMP::Column::FreezeMulti( myGMP, i, x(i), demand(i) );
```

If we only want to freeze those `x(i)` for which `dom(i)` is unequal to zero, then we use:

```
GMP::Column::FreezeMulti( myGMP, i | dom(i), x(i), demand(i) );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::Freeze* (page 326), *GMP::Column::UnfreezeMulti* (page 355) and *GMP::Instance::Copy* (page 364).

GMP::Column::FreezeRaw

The procedure *GMP::Column::FreezeRaw* (page 328) freezes a group of columns in a generated mathematical program.

```
GMP::Column::FreezeRaw(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     value           ! (input) a parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

value A parameter over *colSet*, defining the value for each column to which it should be frozen.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- The columns remain visible in the constraint listing and math program inspector. In addition, it will be retained in solver maintained copies of the generated math program.
- Use *GMP::Column::UnfreezeRaw* (page 356) to undo the freezing.
- During a call to procedure *GMP::Column::FreezeRaw* (page 328) AIMMS stores the upper and lower bound of the columns before the procedure was called. This information is used when procedure *GMP::Column::UnfreezeRaw* (page 356) is called thereafter. This information is not copied by the function *GMP::Instance::Copy* (page 364).

Example

Assume that 'MP' is a mathematical program. To use *GMP::Column::FreezeRaw* (page 328) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
Parameter FixVal {
    IndexDomain: cc;
}
```

To freeze the variable `x(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

FixVal(cc) := 20.0;

GMP::Column::FreezeRaw( myGMP, ColumnSet, FixVal );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Column::Freeze* (page 326), *GMP::Column::UnfreezeRaw* (page 356) and *GMP::Instance::Copy* (page 364).

GMP::Column::GetLowerBound

The function *GMP::Column::GetLowerBound* (page 330) returns the lower bound of a column in the generated mathematical program.

```
GMP::Column::GetLowerBound(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The lower bound value for the specified column.

Note:

- If the column has a unit then the scaled lower bound is returned (without unit).
- This function can be used to retrieve the lower bound after presolving in case the *GMP* was created by *GMP::Instance::CreatePresolved* (page 375), even if the column was deleted by the AIMMS Presolver.

Example

Assume that 'x1' is a variable in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit      :  kg;
    Conversions   :  ton -> kg : # -> # * 1000;
}
Parameter min_wght {
    Unit          :  ton;
    InitialValue  :  20;
}
Variable x1 {
    Range         :  [min_wght, inf);
```

(continues on next page)

(continued from previous page)

```
    Unit          :  ton;
}
```

If we want to multiply the lower bound by 1.5 and assign it as the new value by using function *GMP::Column::SetLowerBound* (page 343) we can use

```
lb1 := 1.5 * (GMP::Column::GetLowerBound( 'MP', x1 )) [ton];

GMP::Column::SetLowerBound( 'MP', x1, lb1 );
```

if 'lb1' is a parameter with unit [ton], or we can use

```
lb2 := 1.5 * GMP::Column::GetLowerBound( 'MP', x1 );

GMP::Column::SetLowerBound( 'MP', x1, lb2 * GMP::Column::GetScale( 'MP', x1 )␣
↪);
```

if 'lb2' is a parameter without a unit.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::SetLowerBound* (page 343), *GMP::Column::GetUpperBound* (page 335), *GMP::Column::GetScale* (page 332) and *GMP::Instance::CreatePresolved* (page 375).

GMP::Column::GetLowerBoundRaw

The procedure *GMP::Column::GetLowerBoundRaw* (page 331) retrieves a collection of lower bound values corresponding to a given set of columns in the generated mathematical program.

```
GMP::Column::GetLowerBoundRaw(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     lbs             ! (output) a real-valued parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set Integers, representing a set of column numbers. Each column number should be in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

lbs A real-valued parameter over colSet indicating the lower bound values of each column in colSet.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If a column has a unit then the scaled lower bound is retrieved (without unit).
- This procedure is much more efficient than calling the function *GMP::Column::GetLowerBound* (page 330) to get the lower bound of each column in colSet individually.

See also:

The routine *GMP::Column::GetLowerBound* (page 330) and *GMP::Column::GetUpperBoundRaw* (page 336).

GMP::Column::GetName

The function *GMP::Column::GetName* (page 332) returns the name of a column in the matrix of a generated mathematical program.

```
GMP::Column::GetName(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The function returns a string.

See also:

The routines *GMP::Instance::Generate* (page 384) and `GMP::row::GetName`.

GMP::Column::GetScale

The function *GMP::Column::GetScale* (page 332) returns the scaling factor of a column in the generated mathematical program.

```
GMP::Column::GetScale(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The scaling factor for the specified column.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Row::GetScale* (page 465).

GMP::Column::GetStatus

The function *GMP::Column::GetStatus* (page 333) returns the status of a column in a generated mathematical program.

```
GMP::Column::GetStatus(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

An element in the predefined set *AllRowColumnStatuses* (page 805). The set *AllRowColumnStatuses* (page 805) contains the following elements:

- Active,
- Deactivated,
- Deleted,
- NotGenerated,
- PresolveDeleted.

Note:

- This function will return 'PresolveDeleted' only if the generated mathematical program has been created with *GMP::Instance::CreatePresolved* (page 375). Status 'PresolveDeleted' means that the column was generated for the original generated mathematical program but deleted when the presolved mathematical program was created.
- Status 'Deactivated' is not possible for columns.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Instance::CreatePresolved* (page 375).

GMP::Column::GetType

The function *GMP::Column::GetType* (page 334) returns the type of a column in the matrix of a generated mathematical program.

```
GMP::Column::GetType(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

An element in the predefined set *AllColumnTypes* (page 795).

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Column::SetType* (page 347).

GMP::Column::GetUpperBound

The function *GMP::Column::GetUpperBound* (page 335) returns the upper bound of a column in the generated mathematical program.

```
GMP::Column::GetUpperBound(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The upper bound value for the specified column.

Note:

- If the column has a unit then the scaled upper bound is returned (without unit).
- This function can be used to retrieve the upper bound after presolving in case the GMP was created by *GMP::Instance::CreatePresolved* (page 375), even if the column was deleted by the AIMMS Presolver.

Example

Assume that 'x1' is a variable in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit      :  kg;
    Conversions   :  ton -> kg : # -> # * 1000;
}
Parameter max_wght {
    Unit          :  ton;
    InitialValue  :  20;
}
Variable x1 {
```

(continues on next page)

(continued from previous page)

```
    Range          :  [0, max_wght];
    Unit           :  ton;
}
```

If we want to multiply the upper bound by 1.5 and assign it as the new value by using function *GMP::Column::SetUpperBound* (page 350) we can use

```
ub1 := 1.5 * (GMP::Column::GetUpperBound( 'MP', x1 )) [ton];

GMP::Column::SetUpperBound( 'MP', x1, ub1 );
```

if 'ub1' is a parameter with unit [ton], or we can use

```
ub2 := 1.5 * GMP::Column::GetUpperBound( 'MP', x1 );

GMP::Column::SetUpperBound( 'MP', x1, ub2 * GMP::Column::GetScale( 'MP', x1 )␣
↪);
```

if 'ub2' is a parameter without a unit.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::SetUpperBound* (page 350), *GMP::Column::GetLowerBound* (page 330), *GMP::Column::GetScale* (page 332) and *GMP::Instance::CreatePresolved* (page 375).

GMP::Column::GetUpperBoundRaw

The procedure *GMP::Column::GetUpperBoundRaw* (page 336) retrieves a collection of upper bound values corresponding to a given set of columns in the generated mathematical program.

```
GMP::Column::GetUpperBoundRaw(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     ubs             ! (output) a real-valued parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set Integers, representing a set of column numbers. Each column number should be in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

ubs A real-valued parameter over colSet indicating the upper bound values of each column in colSet.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If a column has a unit then the scaled upper bound is retrieved (without unit).
- This procedure is much more efficient than calling the function *GMP::Column::GetUpperBound* (page 335) to get the upper bound of each column in colSet individually.

See also:

The routine *GMP::Column::GetUpperBound* (page 335) and *GMP::Column::GetLowerBoundRaw* (page 331).

GMP::Column::SetAsMultiObjective

The procedure *GMP::Column::SetAsMultiObjective* (page 337) sets a column as one of the multi-objectives of a generated mathematical program, thereby creating a multi-objective optimization problem.

Lexicographic (or hierarchical) multi-objective optimization will optimize for the different objectives in the mathematical program one step at a time, in priority order.

```
GMP::Column::SetAsMultiObjective(
     GMP,            ! (input) a generated mathematical program
     column,         ! (input) a scalar reference or column number
     priority,       ! (input) a numerical expression
     weight,         ! (input) a numerical expression
     [abstol],       ! (input/optional) a numerical expression
     [reltol]        ! (input/optional) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

priority A scalar value specifying the priority of the objective. An objective with the highest priority is considered first.

weight A scalar value specifying the weight of the objective. It defines the weight by which the objective coefficients are multiplied when forming a blended objective, i.e., if multiple objectives have the same priority.

abstol A scalar value specifying the absolute tolerance. In subsequent steps (for objectives with a lower priority) the solution value of *column* may deviate from the optimized value of *column*, as calculated in this step, using this absolute tolerance. The default value is 0.0.

reltol A scalar value specifying the relative tolerance. In subsequent steps (for objectives with a lower priority) the solution value of *column* may deviate from the optimized value of *column*, as calculated in this step, using this relative tolerance. The default value is 0.0.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- The column should be linear and have at exactly one coefficient in the matrix.
- The column should be free, i.e., not have a lower or upper bound.
- If *GMP::Column::SetAsMultiObjective* (page 337) is called twice for the same column then only the information from the second call is used (and the information from the first call is ignored).
- Use the procedure *GMP::Instance::DeleteMultiObjectives* (page 380) to delete all multi-objectives.
- Multi-objective optimization is only supported by CPLEX and Gurobi.
- If you specify n different priorities then CPLEX or Gurobi will optimize the mathematical program n times using different objectives (in priority order).
- During the solution process, you may prefer that optimization problems with different priorities are solved with different option settings. CPLEX and Gurobi make this possible by using parameter files. See the option `Read Parameter File` in the CPLEX Help or the Gurobi Help for more information.
- Multi-objective optimization is not supported for generated mathematical programs created by one of the following functions:
  - GMP::Instance::GenerateRobustCounterpart,
  - GMP::Instance::GenerateStochasticProgram,
  - GMP::Instance::CreatePresolved,
  - GMP::Instance::CreateDual, or
  - GMP::Instance::CreateMasterMIP.
- The meaning of the relaxation of the objective, which is controlled by the *abstol* and *reltol* arguments, depends on whether the multi-objective problem is an LP or MIP. See the Multi-Objective Optimization section in the CPLEX Help or the Gurobi Help for more information.
- The *abstol* and *reltol* arguments are meaningless for the objective(s) with the lowest priority.

Example

In the example below two multi-objectives are specified:

```
myGMP := GMP::Instance::Generate( MP );

GMP::Column::SetAsMultiObjective( myGMP, TotalDist, 2, 1.0, 0, 0.1 );
GMP::Column::SetAsMultiObjective( myGMP, TotalTime, 1, 1.0, 0, 0.0 );

GMP::Instance::Solve( myGMP );
```

We can now switch the priorities of the two objectives by adding:

```
GMP::Column::SetAsMultiObjective( myGMP, TotalDist, 1, 1.0, 0, 0.1 );
GMP::Column::SetAsMultiObjective( myGMP, TotalTime, 2, 1.0, 0, 0.0 );

GMP::Instance::Solve( myGMP );
```

See also:

The procedure *GMP::Instance::DeleteMultiObjectives* (page 380).

GMP::Column::SetAsObjective

The procedure *GMP::Column::SetAsObjective* (page 339) sets a column as the new objective of a generated mathematical program.

```
GMP::Column::SetAsObjective(
     GMP,            ! (input) a generated mathematical program
     column          ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- The column should be linear and have at least one coefficient in the matrix.
- The column should be free, i.e., not have a lower or upper bound.
- After a call to *GMP::Column::SetAsObjective* (page 339) the old objective column will be treated as a normal column.

See also:

The routines *GMP::Column::Add* (page 322) and *GMP::Instance::CreateDual* (page 370).

GMP::Column::SetDecomposition

The procedure *GMP::Column::SetDecomposition* (page 339) can be used to specify a decomposition to be used by a solver. It changes the decomposition value of a single column in the generated mathematical program.

This procedure can be used to specify a decomposition for the Benders algorithm in CPLEX by assigning the columns to the master problem or a subproblem. It can also be used to specify a decompostion for ODH-CPLEX. And it can be used to specify a partition for Gurobi to be used by its partition heuristic.

```
GMP::Column::SetDecomposition(
     GMP,             ! (input) a generated mathematical program
     column,          ! (input) a scalar reference or column number
     value            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value The decomposition value assigned to the column.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Use *GMP::Column::SetDecompositionMulti* (page 341) if the decomposition value of many columns corresponding to some variable have to be set, because that will be more efficient.
- This procedure can be used to specify the decomposition in the Benders algorithm of CPLEX. See the CPLEX option `Benders strategy` for more information.
- For CPLEX, use a value of 0 to assign a column to the master problem, and a value between 1 and N to assign a column to one of the N subproblems (N can be 1 if you only want to use one subproblem). A value of -1 indicates that the column is not assigned to the master problem or a subproblem.
- This procedure can be used to specify model structure or a decomposition used by ODH-CPLEX.
- For ODH-CPLEX, use a value between 1 and N to assign a column to one of the N subproblems. A value of 0 or lower indicates that the column is not assigned to any subproblem.
- This procedure can be used to specify a partition used by the partition heuristic of Gurobi. See the Gurobi option `Partition heuristic` for more information.
- For Gurobi, use a positive value to indicate that the column should be included when the correspondingly numbered sub-MIP is solved, a value of 0 to indicate that the column should be included in every sub-MIP, and a value of -1 to indicate that the column should not be included in any sub-MIP. (Variables that are not included in the sub-MIP are fixed to their values in the current incumbent solution.)
- This procedure is **not** used by the Automatic Benders' Decomposition module in AIMMS.

Example

The first example shows how to specify a decomposition for the Benders algorithm in CPLEX. The integer variable `IntVar` is assigned to the master problem while the continuous variable `ContVar` is assigned to the subproblem.

```
myGMP := GMP::Instance::Generate( MP );

! Switch on CPLEX option for using Benders strategy with decomposition␣
↪specified by user.
GMP::Instance::SetOptionValue( myGMP, 'benders strategy', 1 );

for (i) do
    GMP::Column::SetDecomposition( myGMP, IntVar(i), 0 );
endfor;

for (j) do
    GMP::Column::SetDecomposition( myGMP, ContVar(j), 1 );
endfor;

GMP::Instance::Solve( myGMP );
```

The second example shows how to specify model structure used by ODH-CPLEX. All columns `X(i,j)` and `Y(i,j,k)` with the same `i` are assigned to the same subproblem.

```
myGMP := GMP::Instance::Generate( MP );

for (i,j) do
    GMP::Column::SetDecomposition( myGMP, X(i,j), Ord(i) );
endfor;

for (i,j,k) do
    GMP::Column::SetDecomposition( myGMP, Y(i,j,k), Ord(i) );
endfor;

GMP::Instance::Solve( myGMP );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424) and *GMP::Column::SetDecompositionMulti* (page 341).

GMP::Column::SetDecompositionMulti

The procedure *GMP::Column::SetDecompositionMulti* (page 341) can be used to specify a decomposition to be used by a solver. It changes the decomposition value of a group of columns, belonging to a variable, in the generated mathematical program.

This procedure can be used to specify a decomposition for the Benders algorithm in CPLEX by assigning the columns to the master problem or a subproblem. It can also be used to specify a decompostion for ODH-CPLEX. And it can be used to specify a partition for Gurobi to be used by its partition heuristic.

```
GMP::Column::SetDecompositionMulti(
     GMP,             ! (input) a generated mathematical program
     binding,         ! (input) an index binding
     column,          ! (input) a scalar reference or column number
     value            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

value The new decomposition value for each column, defined over the binding domain *binding*.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- This procedure can be used to specify the decomposition in the Benders algorithm of CPLEX. See the CPLEX option `Benders strategy` for more information.
- For CPLEX, use a value of 0 to assign a column to the master problem, and a value between 1 and N to assign a column to one of the N subproblems (N can be 1 if you only want to use one subproblem). A value of -1 indicates that the column is not assigned to the master problem or a subproblem.
- This procedure can be used to specify model structure or a decomposition used by ODH-CPLEX.
- For ODH-CPLEX, use a value between 1 and N to assign a column to one of the N subproblems. A value of 0 or lower indicates that the column is not assigned to any subproblem.
- This procedure can be used to specify a partition used by the partition heuristic of Gurobi. See the Gurobi option `Partition heuristic` for more information.
- For Gurobi, use a positive value to indicate that the column should be included when the correspondingly numbered sub-MIP is solved, a value of 0 to indicate that the column should be included in every sub-MIP, and a value of -1 to indicate that the column should not be included in any sub-MIP. (Variables that are not included in the sub-MIP are fixed to their values in the current incumbent solution.)
- This procedure is **not** used by the Automatic Benders' Decomposition module in AIMMS.

Example

The first example shows how to specify a decomposition for the Benders algorithm in CPLEX. The integer variable `IntVar` is assigned to the master problem while the continuous variable `ContVar` is assigned to the subproblem.

```
myGMP := GMP::Instance::Generate( MP );

! Switch on CPLEX option for using Benders strategy with decomposition␣
↪specified by user.
GMP::Instance::SetOptionValue( myGMP, 'benders strategy', 1 );

GMP::Column::SetDecompositionMulti( myGMP, i, IntVar(i), 0 );

GMP::Column::SetDecompositionMulti( myGMP, j, ContVar(j), 1 );

GMP::Instance::Solve( myGMP );
```

The second example shows how to specify model structure used by ODH-CPLEX. All columns `X(i,j)` and `Y(i,j,k)` with the same `i` are assigned to the same subproblem.

```
myGMP := GMP::Instance::Generate( MP );

GMP::Column::SetDecompositionMulti( myGMP, (i,j), X(i,j), Ord(i) );

GMP::Column::SetDecompositionMulti( myGMP, (i,j,k), Y(i,j,k), Ord(i) );

GMP::Instance::Solve( myGMP );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424) and *GMP::Column::SetDecomposition* (page 339).

GMP::Column::SetLowerBound

The procedure *GMP::Column::SetLowerBound* (page 343) changes the lower bound of a column in a generated mathematical program.

```
GMP::Column::SetLowerBound(
     GMP,            ! (input) a generated mathematical program
     column,         ! (input) a scalar reference or column number
     value           ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value The new value assigned to the lower bound of the column.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Use *GMP::Column::SetLowerBoundMulti* (page 345) or *GMP::Column::SetLowerBoundRaw* (page 346) if the lower bounds of many columns have to be set, because that will be more efficient.
- If the column has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the column scale, as returned by the function *GMP::Column::GetScale* (page 332).

Example

Assume that 'x1' is a variable in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit      :  kg;
    Conversions   :  ton -> kg : # -> # * 1000;
}
Parameter min_wght {
    Unit          :  ton;
    InitialValue  :  20;
}
Variable x1 {
    Range         :  [min_wght, inf);
    Unit          :  ton;
}
```

Then if we run the following code

```
GMP::Column::SetLowerBound( 'MP', x1, 20 [ton] );
lb1 := GMP::Column::GetLowerBound( 'MP', x1 );
display lb1;

GMP::Column::SetLowerBound( 'MP', x1, 30 );
lb2 := GMP::Column::GetLowerBound( 'MP', x1 );
display lb2;

GMP::Column::SetLowerBound( 'MP', x1, 40 * GMP::Column::GetScale( 'MP', x1 ) );
lb3 := GMP::Column::GetLowerBound( 'MP', x1 );
display lb3;
```

(where 'lb1', 'lb2' and 'lb3' are parameters without a unit) we get the following results:

```
lb1 := 20 ;

lb2 := 0.030 ;

lb3 := 40 ;
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::SetLowerBoundMulti* (page 345), *GMP::Column::SetLowerBoundRaw* (page 346), *GMP::Column::SetUpperBound* (page 350), *GMP::Column::GetLowerBound* (page 330) and *GMP::Column::GetScale* (page 332).

GMP::Column::SetLowerBoundMulti

The procedure *GMP::Column::SetLowerBoundMulti* (page 345) changes the lower bounds of a group of columns, belonging to a variable, in a generated mathematical program.

```
GMP::Column::SetLowerBoundMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     column,         ! (input) a variable expression
     value           ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

value The new lower bound for each column, defined over the *binding* domain.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the variable has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the column scale, as returned by the function *GMP::Column::GetScale* (page 332). See *GMP::Column::SetLowerBound* (page 343) for an example with units.

Example

To set the lower bounds of variable `x(i)` to `lb(i)` we can use:

```
for (i) do
    GMP::Column::SetLowerBound( myGMP, x(i), lb(i) );
endfor;
```

It is more efficient to use:

```
GMP::Column::SetLowerBoundMulti( myGMP, i, x(i), lb(i) );
```

If we only want to set the lower bounds of those `x(i)` for which `dom(i)` is unequal to zero, then we use:

```
GMP::Column::SetLowerBoundMulti( myGMP, i | dom(i), x(i), lb(i) );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::SetLowerBound* (page 343), *GMP::Column::SetUpperBound* (page 350), *GMP::Column::GetLowerBound* (page 330) and *GMP::Column::GetScale* (page 332).

GMP::Column::SetLowerBoundRaw

The procedure *GMP::Column::SetLowerBoundRaw* (page 346) changes the lower bounds of a group of columns in a generated mathematical program.

```
GMP::Column::SetLowerBoundRaw(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     value           ! (input) a parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

value A parameter defining a new lower bound for each column in *colSet*.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the variable has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the column scale, as returned by the function *GMP::Column::GetScale* (page 332). See *GMP::Column::SetLowerBound* (page 343) for an example with units.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Column::SetLowerBoundRaw* (page 346) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
Parameter LB {
    IndexDomain: cc;
}
```

To set the lower bounds of the variable `x(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

LB(cc) := 0.0;

GMP::Column::SetLowerBoundRaw( myGMP, ColumnSet, LB );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Column::SetLowerBound* (page 343), *GMP::Column::SetUpperBound* (page 350), *GMP::Column::GetLowerBound* (page 330) and *GMP::Column::GetScale* (page 332).

GMP::Column::SetType

The procedure *GMP::Column::SetType* (page 347) changes the type of a column in a generated mathematical program.

```
GMP::Column::SetType(
     GMP,            ! (input) a generated mathematical program
     column,         ! (input) a scalar reference or column number
     type            ! (input) an element in AllColumnTypes
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

type An element in *AllColumnTypes* (page 795).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: Use *GMP::Column::SetTypeMulti* (page 348) or *GMP::Column::SetTypeRaw* (page 349) if the types of many columns have to be set, because that will be more efficient.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Column::GetType* (page 334), *GMP::Column::SetTypeMulti* (page 348) and *GMP::Column::SetTypeRaw* (page 349).

GMP::Column::SetTypeMulti

The procedure *GMP::Column::SetTypeMulti* (page 348) changes the types of a group of columns, belonging to a variable, in a generated mathematical program.

```
GMP::Column::SetTypeMulti(
     GMP,             ! (input) a generated mathematical program
     binding,         ! (input) an index binding
     column,          ! (input) a variable expression
     type             ! (input) an element parameter in AllColumnTypes
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

type An element parameter in *AllColumnTypes* (page 795), defined over the *binding* domain.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Column::GetType* (page 334) and *GMP::Column::SetType* (page 347).

GMP::Column::SetTypeRaw

The procedure *GMP::Column::SetTypeRaw* (page 349) changes the types of a group of columns in a generated mathematical program.

```
GMP::Column::SetTypeRaw(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     type            ! (input) an element parameter in AllColumnTypes
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

type An element parameter in *AllColumnTypes* (page 795), defined over *colSet*.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Column::FreezeRaw* (page 328) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
ElementParameter ColType {
    IndexDomain: cc;
```

(continues on next page)

(continued from previous page)

```
    Range: AllColumnTypes;
}
```

To change integer variable `x(i)` into a continuous variable we can use:

```
myGMP := GMP::Instance::Generate( MP );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

ColType(cc) := 'continuous';

GMP::Column::FreezeRaw( myGMP, ColumnSet, ColType );
```

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Column::GetType* (page 334) and *GMP::Column::SetType* (page 347).

GMP::Column::SetUpperBound

The procedure *GMP::Column::SetUpperBound* (page 350) changes the upper bound of a column in the generated mathematical program.

```
GMP::Column::SetUpperBound(
     GMP,            ! (input) a generated mathematical program
     column,         ! (input) a scalar reference or column number
     value           ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value The new value assigned to the upper bound of the column.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Use *GMP::Column::SetUpperBoundMulti* (page 351) or *GMP::Column::SetUpperBoundRaw* (page 352) if the upper bounds of many columns have to be set, because that will be more efficient.
- If the column has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the column scale, as returned by the function *GMP::Column::GetScale* (page 332).

Example

Assume that 'x1' is a variable in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit      :  kg;
    Conversions   :  ton -> kg : # -> # * 1000;
}
Parameter max_wght {
    Unit          :  ton;
    InitialValue  :  20;
}
Variable x1 {
    Range         :  [0, max_wght];
    Unit          :  ton;
}
```

Then if we run the following code

```
GMP::Column::SetUpperBound( 'MP', x1, 20 [ton] );
ub1 := GMP::Column::GetUpperBound( 'MP', x1 );
display ub1;

GMP::Column::SetUpperBound( 'MP', x1, 30 );
ub2 := GMP::Column::GetUpperBound( 'MP', x1 );
display ub2;

GMP::Column::SetUpperBound( 'MP', x1, 40 * GMP::Column::GetScale( 'MP', x1 ) );
ub3 := GMP::Column::GetUpperBound( 'MP', x1 );
display ub3;
```

(where 'ub1', 'ub2' and 'ub3' are parameters without a unit) we get the following results:

```
ub1 := 20 ;

ub2 := 0.030 ;

ub3 := 40 ;
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::SetUpperBoundMulti* (page 351), *GMP::Column::SetUpperBoundRaw* (page 352), *GMP::Column::SetLowerBound* (page 343), *GMP::Column::GetUpperBound* (page 335) and *GMP::Column::GetScale* (page 332).

GMP::Column::SetUpperBoundMulti

The procedure *GMP::Column::SetUpperBoundMulti* (page 351) changes the upper bounds of a group of columns, belonging to a variable, in the generated mathematical program.

```
GMP::Column::SetUpperBoundMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     column,         ! (input) a variable expression
     value           ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

value The new upper bound for each column, defined over the *binding* domain.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the variable has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the column scale, as returned by the function *GMP::Column::GetScale* (page 332). See *GMP::Column::SetUpperBound* (page 350) for an example with units.

Example

To set the upper bounds of variable `x(i)` to `ub(i)` we can use:

```
for (i) do
    GMP::Column::SetUpperBound( myGMP, x(i), ub(i) );
endfor;
```

It is more efficient to use:

```
GMP::Column::SetUpperBoundMulti( myGMP, i, x(i), ub(i) );
```

If we only want to set the upper bounds of those `x(i)` for which `dom(i)` is unequal to zero, then we use:

```
GMP::Column::SetUpperBoundMulti( myGMP, i | dom(i), x(i), ub(i) );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::SetUpperBound* (page 350), *GMP::Column::SetLowerBound* (page 343), *GMP::Column::GetUpperBound* (page 335) and *GMP::Column::GetScale* (page 332).

GMP::Column::SetUpperBoundRaw

The procedure *GMP::Column::SetUpperBoundRaw* (page 352) changes the upper bounds of a group of columns in a generated mathematical program.

```
GMP::Column::SetUpperBoundRaw(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     value           ! (input) a parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

value A parameter defining a new upper bound for each column in *colSet*.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the variable has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the column scale, as returned by the function *GMP::Column::GetScale* (page 332). See *GMP::Column::SetUpperBound* (page 350) for an example with units.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Column::SetUpperBoundRaw* (page 352) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
Parameter UB {
    IndexDomain: cc;
}
```

To set the upper bounds of the variable `x(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

UB(cc) := 1.0;

GMP::Column::SetUpperBoundRaw( myGMP, ColumnSet, UB );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Column::SetUpperBound* (page 350), *GMP::Column::SetLowerBound* (page 343), *GMP::Column::GetUpperBound* (page 335) and *GMP::Column::GetScale* (page 332).

GMP::Column::Unfreeze

The procedure *GMP::Column::Unfreeze* (page 354) removes the frozen status of a column in a generated mathematical program.

```
GMP::Column::Unfreeze(
     GMP,             ! (input) a generated mathematical program
     column           ! (input) a scalar reference or column number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Use *GMP::Column::UnfreezeMulti* (page 355) or *GMP::Column::UnfreezeRaw* (page 356) if many columns have to be unfrozen, because that will be more efficient.
- During a call to procedure *GMP::Column::Freeze* (page 326) AIMMS stores the upper and lower bound of the column before the procedure was called. This information is used when procedure *GMP::Column::Unfreeze* (page 354) is called thereafter. This information is not copied by the function *GMP::Instance::Copy* (page 364). Therefore the call to *GMP::Column::Unfreeze* (page 354) in the following piece of code is useless:

```
GMP::Column::Freeze( gmp1, x1, 20 );
gmp2 := GMP::Instance::Copy( gmp1, "cpy" );
GMP::Column::Unfreeze( gmp2, x1 );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::UnfreezeMulti* (page 355), *GMP::Column::UnfreezeRaw* (page 356), *GMP::Column::Freeze* (page 326) and *GMP::Instance::Copy* (page 364).

GMP::Column::UnfreezeMulti

The procedure *GMP::Column::UnfreezeMulti* (page 355) removes the frozen status of a group of columns, belonging to a variable, in a generated mathematical program.

```
GMP::Column::UnfreezeMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     column          ! (input) a variable expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

column A variable that, combined with the *binding* domain, specifies the columns.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: During a call to procedure *GMP::Column::FreezeMulti* (page 327) AIMMS stores the upper and lower bound of the columns before the procedure was called. This information is used when procedure *GMP::Column::UnfreezeMulti* (page 355) is called thereafter. This information is not copied by the function *GMP::Instance::Copy* (page 364).

Example

To unfreeze variable `x(i)` we can use:

```
for (i) do
    GMP::Column::Unfreeze( myGMP, x(i) );
endfor;
```

It is more efficient to use:

```
GMP::Column::UnfreezeMulti( myGMP, i, x(i) );
```

If we only want to unfreeze those `x(i)` for which `dom(i)` is unequal to zero, then we use:

```
GMP::Column::UnfreezeMulti( myGMP, i | dom(i), x(i) );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::Unfreeze* (page 354), *GMP::Column::FreezeMulti* (page 327) and *GMP::Instance::Copy* (page 364).

GMP::Column::UnfreezeRaw

The procedure *GMP::Column::UnfreezeRaw* (page 356) removes the frozen status of a group of columns in a generated mathematical program.

```
GMP::Column::UnfreezeRaw(
     GMP,            ! (input) a generated mathematical program
     colSet          ! (input) a subset of Integers
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: During a call to procedure *GMP::Column::FreezeRaw* (page 328) AIMMS stores the upper and lower bound of the columns before the procedure was called. This information is used when procedure *GMP::Column::UnfreezeRaw* (page 356) is called thereafter. This information is not copied by the function *GMP::Instance::Copy* (page 364).

Example

Assume that 'MP' is a mathematical program. To use *GMP::Column::UnfreezeRaw* (page 356) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VariableSet {
    SubsetOf: AllVariables;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
```

To unfreeze the variable `x(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

VariableSet := { 'x' };
ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, VariableSet );

GMP::Column::Unfreeze( myGMP, ColumnSet );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Column::Unfreeze* (page 354), *GMP::Column::FreezeRaw* (page 328) and *GMP::Instance::Copy* (page 364).

2.3.4 GMP::Event Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing events:

GMP::Event::Create

The function *GMP::Event::Create* (page 357) creates a new event.

```
GMP::Event::Create(
     Name              ! (input) a string expression
     )
```

Arguments

Name A string expression holding the name of the event.

Return Value

The function returns an element in the set *AllGMPEvents* (page 821).

See also:

The routines *GMP::Event::Delete* (page 357), *GMP::Event::Reset* (page 358) and *GMP::Event::Set* (page 358), and Synchr
Language Reference.

GMP::Event::Delete

The procedure *GMP::Event::Delete* (page 357) deletes an event.

```
GMP::Event::Delete(
     Event             ! (input) an event
     )
```

Arguments

Event An element in the set *AllGMPEvents* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Event::Create* (page 357), *GMP::Event::Reset* (page 358) and *GMP::Event::Set* (page 358), and Synchro
Language Reference.

GMP::Event::Reset

The procedure *GMP::Event::Reset* (page 358) resets an event.

```
GMP::Event::Reset(
     Event              ! (input) an event
     )
```

Arguments

Event An element in the set *AllGMPEvents* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: The state of the event will be reset to the state just after creating the event.

See also:

The routines *GMP::Event::Create* (page 357), *GMP::Event::Delete* (page 357) and *GMP::Event::Reset* (page 358), and Syn
Language Reference.

GMP::Event::Set

The procedure *GMP::Event::Set* (page 358) activates an event.

```
GMP::Event::Set(
     Event              ! (input) an event
     )
```

Arguments

Event An element in the set *AllGMPEvents* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Event::Create* (page 357), *GMP::Event::Delete* (page 357) and *GMP::Event::Reset* (page 358), and Syn
Language Reference.

2.3.5 GMP::Instance Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing generated mathematical program instances:

GMP::Instance::AddIntegerEliminationRows

The procedure *GMP::Instance::AddIntegerEliminationRows* (page 359) adds integer elimination rows to the generated mathematical program which will eliminate an integer solution.

```
GMP::Instance::AddIntegerEliminationRows(
    GMP,          ! (input) a generated mathematical program
    solution,     ! (input) a solution
    refNo         ! (input) a scalar integer value
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

refNo A positive integer scalar value representing a reference number.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Rows and columns added before for *refNo* will be deleted first.
- If the GMP contains only binary variables then only one row will be added; if the GMP contains general integer variables then several rows and columns will be added.

- The exact definitions of the rows and columns that are added are as follows. Let x_i be an integer column whose level value lev_i is between its lower bound lb_i and upper bound ub_i, i.e., $lb_i < lev_i < ub_i$. Then columns $l_i \geq 0$ and $u_i \geq 0$ are added together with a binary column z_i. Also the following three constraints are added:

$$l_i + (lev_i - lb_i)z_i \leq (lev_i - lb_i) \tag{2.1}$$

$$u_i + (lev_i - ub_i)z_i \leq 0 \tag{2.2}$$

$$x_i - u_i + l_i = lev_i \tag{2.3}$$

  Every call to *GMP::Instance::AddIntegerEliminationRows* (page 359) also adds the following constraint:

$$\sum_{i \in S_{lo}} x_i - \sum_{i \in S_{up}} x_i + \sum_{i \in S_{in}} (l_i + u_i) \geq 1 + \sum_{i \in S_{lo}} lev_i - \sum_{i \in S_{up}} lev_i \tag{2.4}$$

  where S_{lo} defines the set of integer columns whose level values equal their lower bounds, S_{up} the set of integer columns whose level values equal their upper bounds, and S_{in} the set of integer columns whose level values are between their bounds.

- By using the suffixes `.ExtendedConstraint` and `.ExtendedVariable` it is possible to refer to the rows and columns respectively that are added by *GMP::Instance::AddIntegerEliminationRows* (page 359):
  - Variables `v.ExtendedVariable('EliminationLowerBound`*k*`',i)`, `v.ExtendedVariable('EliminationUpperBound`*k*`',i)` and `v.ExtendedVariable('Elimination`*k*`',i)` are added for each integer variable `v(i)` with the level value between its bounds. (These variables correspond to l_i, u_i and z_i respectively.)
  - Constraints `v.ExtendedConstraint('EliminationLowerBound`*k*`',i)`, `v.ExtendedConstraint('EliminationUpperBound`*k*`',i)` and `v.ExtendedConstraint('Elimination`*k*`',i)` are added for each integer variable `v(i)` with the level value between its bounds. (These constraints correspond to (2.1), (2.2) and (2.3) respectively.)
  - Constraint `mp.ExtendedConstraint('Elimination`*k*`')`, where `mp` denotes the symbolic mathematical program, is added for every call to *GMP::Instance::AddIntegerEliminationRows* (page 359). (This constraint corresponds to (2.4).)

  Here k denotes the value of the argument *refNo*.

Example

The procedure *GMP::Instance::AddIntegerEliminationRows* (page 359) can be used to find the five best integer solutions for some MIP model:

```
gmp_mip := GMP::Instance::Generate( MIP_Model );

cnt := 1;

while ( cnt <= 5 ) do
    GMP::Instance::Solve( gmp_mip );

    ! Eliminate previous found integer solution.
```

(continues on next page)

(continued from previous page)

```
    GMP::Instance::AddIntegerEliminationRows( gmp_mip, 1, cnt );

    cnt += 1;

    ! Copy solution at position 1 to solution at position cnt
    ! in solution repository.
    GMP::Solution::Copy( gmp_mip, 1, cnt );
endwhile;
```

After executing this code, the five best integer solutions will be stored at positions 2 - 6 in the solution repository, with the best solution at position 2 and the 5th best at position 6.

See also:

The routines *GMP::Instance::DeleteIntegerEliminationRows* (page 379) and *GMP::Solution::IsInteger* (page 498). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Instance::AddLimitBinaryDeviationRow

The procedure *GMP::Instance::AddLimitBinaryDeviationRow* (page 361) adds a row to a GMP that sets a limit on the number of binary columns, from a given set of variables, of which the solution value is allowed to vary.

A scenario in which the procedure could be used is the following. Imagine you have created a production plan based on optimizing some mathematical program and that something unexpected happened that (partly) ruined the plan. You now have to re-optimize the mathematical program, with some changes, but would like the solution of the new optimization to be close to the previous one.

```
GMP::Instance::AddLimitBinaryDeviationRow(
     GMP,          ! (input) a generated mathematical program
     solution,     ! (input) a solution
     variableSet,  ! (input) a reference number
     deviation,    ! (input) a scalar integer value
     [refNo]       ! (optional, default 1) a scalar integer value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

variableSet A subset of *AllVariables* (page 831).

deviation A nonnegative integer scalar value representing the total number of binary variables that are allowed to deviate.

refNo A positive integer scalar value representing a reference number. The default is 1.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure will fail if *variableSet* contains no binary variable.
- Non-binary variables in the *variableSet* will be ignored. It is therefore possible to pass *AllVariables* (page 831) as the *variableSet* (provided that the mathematical program contains a least one binary variable).
- A row added before for *refNo* will be deleted first.
- Every call to *GMP::Instance::AddLimitBinaryDeviationRow* (page 361) adds the following row:

$$\sum_{i \in S_0} x_i - \sum_{i \in S_1} x_i \leq d - |S_1|$$

  where S_0 defines the set of binary columns whose level values equals 0, S_1 the set of binary columns whose level values equals 1 and d the *deviation* value.
- By using the suffix `.ExtendedConstraint` it is possible to refer to the row added by *GMP::Instance::AddLimitBinaryDeviationRow* (page 361). The constraint `mp.ExtendedConstraint('Deviation`*k*`')`, where `mp` denotes the symbolic mathematical program, is added for every call to this procedure. Here k denotes the value of the argument *refNo*.

Example

Assume that 'MP' is a mathematical program containing the binary variables *x(i)* and *y(j)*. Furthermore, we have solved this mathematical program and found a solution but now we want to resolve the mathematical program after we made some changes in the data, resulting in a different generated mathematical program. However, we want to enforce that the solution variables of the binary variables in the second solve does not change much compared to the first solve. This can be achieved using the procedure *GMP::Instance::AddLimitBinaryDeviationRow* (page 361).

To use this procedure we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set VarSet {
    SubsetOf: AllIntegerVariables;
}
```

If we want to enforce that at most 4 of the *x(i)* and *y(j)* variables can get different solution values compared to the first solve then we could use:

```
myGMP := GMP::Instance::Generate(MP);

GMP::Solution::RetrieveFromModel( myGMP, 1 );

VarSet := { 'x', 'y' };
GMP::Instance::AddLimitBinaryDeviationRow( myGMP, 1, varSet, 4, 1 );

GMP::Instance::Solve( myGMP );
```

After executing this code, it could be that all *x(i)* variables get the same solution values as before and that 4 of the *y(j)* variables get different solution values. If we also want to add the restriction that at most 3 of the *y(j)* variables get different solution values then we should use:

```
myGMP := GMP::Instance::Generate(MP);

GMP::Solution::RetrieveFromModel( myGMP, 1 );

VarSet := { 'x', 'y' };
GMP::Instance::AddLimitBinaryDeviationRow( myGMP, 1, varSet, 4, 1 );
VarSet := { 'y' };
GMP::Instance::AddLimitBinaryDeviationRow( myGMP, 1, varSet, 3, 2 );

GMP::Instance::Solve( myGMP );
```

See also:

The routines *GMP::Instance::DeleteIntegerEliminationRows* (page 379). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Instance::CalculateSubGradient

The procedure *GMP::Instance::CalculateSubGradient* (page 363) can be used to solve $By = x$ for a given vector x, where B is the basis matrix of a linear program. This procedure can only be called after the linear program has been solved to optimality.

```
GMP::Instance::CalculateSubGradient(
     GMP,           ! (input) a generated mathematical program
     variableSet,   ! (input) a set of variables
     constraintSet, ! (input) a set of constraints
     [session]      ! (input, optional) a solver session
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833). The mathematical program should have model type LP or RMIP.

variableSet A subset of *AllVariables* (page 831).

constraintSet A subset of *AllConstraints* (page 817).

session An element in the set *AllSolverSessions* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use the `.ExtendedConstraint('RhsChange',*)` suffix of the constraints in *constraintSet* to assign values to the vector *x*.
- The suffix `.ExtendedVariable('RhsChange',*)` of the variables in *variableSet* will be used to store the subgradient *y*.
- The suffixes `.ExtendedConstraint` and `.ExtendedVariable` have no unit and are not scaled.
- This procedure should be called after a normal solve statement or after a successful call to procedure *GMP::Instance::Solve* (page 424).
- This procedure can also be called after a successful call to the procedure *GMP::SolverSession::Execute* (page 530) or the procedure *GMP::SolverSession::AsynchronousExecute* (page 528). In that case the solver session should be passed using the *session* argument.
- A column corresponding to a variable in *variableSet* that is not part of *GMP* will be ignored. A row corresponding to a constraint in *constraintSet* that is not part of *GMP* will also be ignored.
- This procedure is only supported by CPLEX and Gurobi.
- This procedure cannot be used if the *GMP* is created by *GMP::Instance::CreateDual* (page 370).

Example

Assume that 'MP' is a linear mathematical program and `c(i)` is a constraint and `v(j)` is a variable in this mathematical program. The following example shows how to calculate a subgradient after a normal solve statement.

```
solve MP;

! The next statement needs to be called once.
AllGMPExtensions += { 'RhsChange' };

c.ExtendedConstraint('RhsChange',i) := 1.0;

GMP::Instance::CalculateSubGradient('MP',AllVariables,AllConstraints);

display v.ExtendedVariable('RhsChange',j);
```

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::SolverSession::Execute* (page 530) and *GMP::SolverSession::AsynchronousExecute* (page 528). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Instance::Copy

The function *GMP::Instance::Copy* (page 364) creates a copy of a generated mathematical program and an associated new element in the set *AllGeneratedMathematicalPrograms* (page 833).

```
GMP::Instance::Copy(
     GMP,             ! (input) a generated mathematical program
     name             ! (input) a string expression
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

name A string that contains the name for the copy of the generated mathematical program.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- The *name* argument should be different from the name of the original generated mathematical program.
- If an element with name specified by the *name* argument is already present in the set *AllGeneratedMathematicalPrograms* (page 833) then the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved).
- All solutions in the solution repository of the generated mathematical program are also copied.
- The solver selection as specified by *GMP::Instance::SetSolver* (page 423) (if any) will not be copied.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Rename* (page 408) and *GMP::Instance::SetSolver* (page 423).

GMP::Instance::CreateBlockMatrices

The function *GMP::Instance::CreateBlockMatrices* (page 365) generates a set of generated mathematical programs that are independent block representations of the specified generated mathematical program. In other words, the original generated mathematical program will be partitioned into multiple generated mathematical programs.

The mathematical program visualized in Fig. 2.3 contains three blocks (the first row and column represent the objective). This model can be solved by solving each block separately, and combining the solutions.

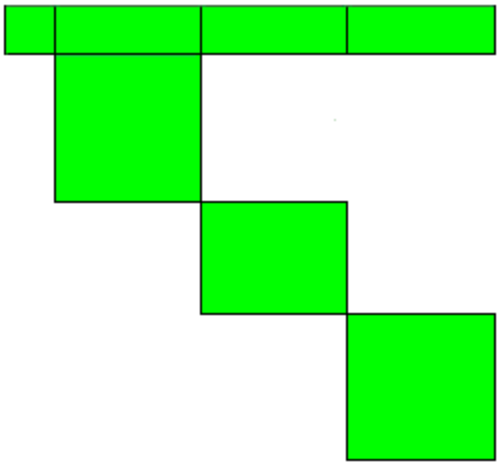


Fig. 2.3: A mathematical program with three blocks

```
GMP::Instance::CreateBlockMatrices(
     GMP,            ! (input) a generated mathematical program
     colSet,         ! (input) a subset of Integers
     blockValue,     ! (input) an integer parameter
     prefix          ! (input) a string expression
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers. This set may be empty.

blockValue An integer parameter over *colSet* indicating the (nonnegative) block value for each column in *colSet*.

prefix A string that holds the prefix used to create the names of each generated mathematical program that is created by this function.

Return Value

A subset of the set *AllGeneratedMathematicalPrograms* (page 833). The subset will be empty in case of an error.

Note:

- The *GMP* must be linear.
- The objective column will be copied to every block GMP. The objective row also, but it will only include the columns assigned to that block GMP.
- The *blockValue* of the objective column will be ignored (because the objective column will be copied to every block GMP).
- A *blockValue* of 0 for a certain column means that AIMMS will automatically assign that column to a block.

- The user can specify the *blockValue* for every column in the *GMP*, in which case this function will create block GMP's based on this partition. If a different non-zero *blockValue* is assigned to two columns, and it turns out that the two corresponding blocks are not independent, then this function will generate an error. If two columns are assigned the same non-zero *blockValue* then these columns will be assigned to the same block, even if this block could be partitioned into two blocks.
- If *colSet* is empty, or if *blockValue* equals 0 for each column, then AIMMS will create block matrices automatically, and as many as possible. If the *GMP* cannot be partitioned in multiple block GMP's then the returned subset of the set *AllGeneratedMathematicalPrograms* (page 833) will only contain one element, namely a copy of the original *GMP*.
- It is possible to assign a positive *blockValue* for a subset of the columns. This function will then automatically assign the other columns to block GMP's.
- The *prefix* argument is used to create the names of the block GMP's, which will consist of the *prefix* value followed by the block number. The block numbers correspond to the *blockValue* assigned to (a selection of) the columns. If extra block GMP's are created (besides those specified through the *blockValue* argument) then their numbering will start at the largest *blockValue* plus one. (And if the *colSet* is empty, or if *blockValue* equals 0 for each column, then the numbering will start at one.)
- If the objective row contains an objective constant then this will be added to the last block GMP only.
- In the unusual situation that the *GMP* contains columns (besides the objective column) that only appear in the objective row, then these columns will be assigned to the last block GMP if their corresponding *blockValue* equals 0.

Example

Assume that 'MP' is a mathematical program with the following declaration (in ams format):

```
Set Periods {
    Index: t;
    Definition: {
        { 'per-1', 'per-2', 'per-3' }
    }
}
Variable x {
    IndexDomain: t;
    Range: nonnegative;
}
Variable y {
    IndexDomain: t;
    Range: nonnegative;
}
Variable obj {
    Definition: sum( t, 7 * x(t) - 2 * y(t) );
}
Constraint c1 {
    IndexDomain: t;
    Definition: - x(t) + 2 * y(t) <= 4;
}
MathematicalProgram MP {
    Objective: obj;
    Direction: minimize;
```

(continues on next page)

(continued from previous page)

```
    Type: LP;
}
```

To use *GMP::Instance::CreateBlockMatrices* (page 365) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set GMPset {
    SubsetOf: AllGeneratedMathematicalPrograms;
    Parameter: CurrentGMP;
}
ElementParameter session {
    Range: AllSolverSessions;
}
Set ColumnSet {
    SubsetOf: Integers;
    Index: cc;
}
Parameter BlockVals {
    IndexDomain: cc;
}
StringParameter ColumnName {
    IndexDomain: cc;
}
Parameter MergeSolution {
    Range: Binary;
}
```

To create block matrices and solve them to create a solution for the original model we can use:

```
myGMP := GMP::Instance::Generate( MP );

ColumnSet := GMP::Instance::GetColumnNumbers( myGMP, AllVariables );

for (cc) do
    ColumnName(cc) := GMP::Column::GetName( myGMP, cc );
endfor;

BlockVals(cc) := 0;

for (cc) do
    if ( FindString( ColumnName(cc), "per-1" ) ) then
        BlockVals(cc) := 1;
    elseif ( FindString( ColumnName(cc), "per-2" ) ) then
        BlockVals(cc) := 2;
    else
        BlockVals(cc) := 3;
    endif;
endfor;
```

(continues on next page)

(continued from previous page)

```
GMPset := GMP::Instance::CreateBlockMatrices( myGMP, ColumnSet, BlockVals,
↪"block-" );

MergeSolution := 0;

while ( Card(GMPset) >= 1 ) do
    CurrentGMP := First(GMPset);

    session := GMP::Instance::CreateSolverSession( CurrentGMP );

    GMP::SolverSession::Execute( session );

    GMP::Solution::RetrieveFromSolverSession( session, 1 );
    if ( Card(GMPset) = 1 ) then
        GMP::Solution::SendToModel( CurrentGMP, 1, merge : MergeSolution,␣
↪evalInline : 1 );
    else
        GMP::Solution::SendToModel( CurrentGMP, 1, merge : MergeSolution,␣
↪evalInline : 0 );
    endif;

    GMP::Instance::Delete( CurrentGMP );   ! Also deletes session

    MergeSolution := 1;
endwhile;

GMP::Instance::Delete( myGMP );
```

The above piece of code creates three block GMP's (with names "block-1", "block-2" and "block-3"). This is also the case if 'ColumnSet' or 'BlockVals' would have been empty. If we assign the block values as follows then only two blocks will be created:

```
BlockVals(cc) := 0;

for (cc) do
    if ( FindString( ColumnName(cc), "per-1" ) ) then
        BlockVals(cc) := 1;
    else
        BlockVals(cc) := 2;
    endif;
endfor;
```

In this case the columns corresponding to the periods "per-2" and "per-3" will be assigned to the same block GMP (with the name "block-2").

The parameter 'MergeSolution' is set to 0 for the first block GMP, otherwise the solution will be merged with an old solution (if one exists).

Note: the first piece of code is optimized for mathematical programs with inline variables because it contains the following code snippet:

```
GMP::Solution::RetrieveFromSolverSession( session, 1 );
if ( Card(GMPset) = 1 ) then
```

(continues on next page)

(continued from previous page)

```
    GMP::Solution::SendToModel( CurrentGMP, 1, merge : MergeSolution,␣
↪evalInline : 1 );
else
    GMP::Solution::SendToModel( CurrentGMP, 1, merge : MergeSolution,␣
↪evalInline : 0 );
endif;
```

If your mathematical program does not contain any inline variables then you can use the following code instead:

```
GMP::Solution::RetrieveFromSolverSession( session, 1 );
GMP::Solution::SendToModel( CurrentGMP, 1, merge : MergeSolution );
```

See also:

The routines *GMP::Instance::CreateSolverSession* (page 378), *GMP::Instance::Generate* (page 384), *GMP::Solution::RetrieveFromSolverSession* (page 502), *GMP::Solution::SendToModel* (page 503) and *GMP::SolverSession::Execute* (page 530).

GMP::Instance::CreateDual

The function *GMP::Instance::CreateDual* (page 370) generates a mathematical program that is the dual representation of the specified generated mathematical program. The generated mathematical program should have model type LP.

```
GMP::Instance::CreateDual(
     GMP,            ! (input) a generated mathematical program
     name            ! (input) a string expression
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

name A string that holds the name for the dual of the generated mathematical program.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- The *name* argument should be different from the name of the original generated mathematical program.
- If an element with name specified by the *name* argument is already present in the set *AllGeneratedMathematicalPrograms* (page 833) the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved).
- The solution of a dual variable can be accessed through the `.ShadowPrice` suffix of the corresponding (primal) constraint.

- Before a generated mathematical program is dualized, AIMMS first transforms it temporary into a standard form using the following rules:
  - Each column x_i that is frozen to 0 is deleted.
  - For each column x_i with upper bound u_i, $u_i \neq 0$ and $u_i < \infty$, an extra row $x_i \leq u_i$ is added.
  - For each column x_i with lower bound l_i, $l_i \neq 0$ and $l_i > -\infty$, an extra row $x_i \geq l_i$ is added.
  - Each ranged row $l_j \leq a^T x \leq u_j$ ($l_j > -\infty$ and $u_j < \infty$) is replaced by two rows $l_j \leq a^T x$ and $a^T x \leq u_j$.
- By using the suffix `.ExtendedConstraint` it is possible to refer to the rows that are added to create the standard form:
  - The constraint `v.ExtendedConstraint('DualUpperBound',i)` is added for a variable `v(i)` with an upper bound unequal to 0 and inf.
  - The constraint `v.ExtendedConstraint('DualLowerBound',i)` is added for a variable `v(i)` with a lower bound unequal to 0 and -inf.
  - The constraints `c.ExtendedConstraint('DualLowerBound',j)` and `c.ExtendedConstraint('DualUpperBound',j)` replace a ranged constraint `c(j)`.

  The solution of these constraints can be accessed through the `.ShadowPrice` suffix, e.g., `v.ExtendedConstraint('DualUpperBound',i).ShadowPrice`.
- The objective variable for the dual mathematical program will become `mp.ExtendedConstraint(DualObjective)` and the objective constraint will be `mp.ExtendedVariable(DualDefinition)`, where `mp` denotes the symbolic mathematical program.

Example

Assume that 'PrimalModel' is a mathematical program with the following declaration (in ams format):

```
Variable x1 {
    Range: [0, 5];
}
Variable x2 {
    Range: nonnegative;
}
Variable obj {
    Definition: - 7 * x1 - 2 * x2;
}
Constraint c1 {
    Definition: - x1 + 2 * x2 <= 4;
}
MathematicalProgram PrimalModel {
    Objective: obj;
    Direction: minimize;
    Type: LP;
}
```

Then *GMP::Instance::CreateDual* (page 370) will create a dual mathematical program with variables

```
name                                                lower  upper
c1                                                   -inf      0
obj_definition                                       -inf    inf
```

(continues on next page)

(continued from previous page)

```
x1.ExtendedConstraint('DualUpperBound')           -inf      0
PrimalModel.ExtendedConstraint('DualObjective')   -inf    inf
```

and constraints

```
x1:
   - c1 + 7 * obj_definition + x1.ExtendedConstraint('DualUpperBound') >= 0 ;

x2:
   + 2 * c1 + 2 * obj_definition >= 0 ;

obj:
   obj_definition = 1 ;

PrimalModel.ExtendedVariable('DualDefinition'):
   - 4 * c1 - 5 * x1.ExtendedConstraint('DualUpperBound')
   + PrimalModel.ExtendedConstraint('DualObjective') = 0 ;
```

See also:

The function *GMP::Instance::Generate* (page 384). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Instance::CreateFeasibility

The function *GMP::Instance::CreateFeasibility* (page 372) creates a mathematical program that is the feasibility problem of a generated mathematical program. Its main purpose is to identify infeasibilities in an infeasible problem. The feasibility problem can be used to minimize the sum of infeasibilities, or to minimize the maximum infeasibility.

This function can be used for both linear and nonlinear problems but not for constraint programming problems.

```
GMP::Instance::CreateFeasibility(
     GMP,             ! (input) a generated mathematical program
     [name],          ! (input, optional) a string expression
     [useMinMax]      ! (input, optional) integer, default 0
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

name A string that contains the name for the feasibility problem.

useMinMax If 0, the sum of infeasibilities will be minimized, else the maximum infeasibility will be minimized.

Mathematical formulation

In this section we show how the feasibility problem is constructed. To simplify the explanation we use a linear problem but the same construction applies to a nonlinear problem.

Consider the following problem where J denotes the set of variables, I_1 the set of $\geq$ inequalities, I_2 the set of $\leq$ inequalities, and I_3 the set of equalities.

$$\begin{aligned}
\max \quad & \sum_{j \in J} a_j x_j \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j \quad \geq b_i \quad i \in I_1 \\
& \sum_{j \in J} a_{ij} x_j \quad \leq b_i \quad i \in I_2 \\
& \sum_{j \in J} a_{ij} x_j \quad = b_i \quad i \in I_3 \\
& x \geq 0
\end{aligned}$$

Then if we minimize the sum of infeasibilities the feasibility problem becomes:

$$\begin{aligned}
\min \quad & \sum_{i \in I_1} z_i^p + \sum_{i \in I_2} z_i^n + \\
& \sum_{i \in I_3} (z_i^p + z_i^n) \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j + z_i^p \quad \geq b_i \quad i \in I_1 \\
& \sum_{j \in J} a_{ij} x_j - z_i^n \quad \leq b_i \quad i \in I_2 \\
& \sum_{j \in J} a_{ij} x_j + z_i^p - z_i^n \quad = b_i \quad i \in I_3 \\
& x, z^p, z^n \geq 0
\end{aligned}$$

If we minimize the maximum infeasibility the feasibility problem becomes:

$$\begin{aligned}
\min \quad & z^m \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j + z^m \quad \geq b_i \quad i \in I_1 \\
& \sum_{j \in J} a_{ij} x_j - z^m \quad \leq b_i \quad i \in I_2 \\
& \sum_{j \in J} a_{ij} x_j + z_i^p - z_i^n \quad = b_i \quad i \in I_3 \\
& z^m - z_i^p - z_i^n \quad \geq 0 \quad i \in I_3 \\
& x, z^p, z^n \geq 0
\end{aligned}$$

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- The *name* argument should be different from the name of the original generated mathematical program.
- If the *name* argument is not specified then AIMMS will name the generated math program as "Feasibility problem of" followed by the name of the *GMP*.
- If an element with name specified by the *name* argument is already present in the set *AllGeneratedMathematicalPrograms* (page 833) the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved).
- By using the suffices `.ExtendedVariable` and `.ExtendedConstraint` it is possible to refer to the columns and rows that are added to create the feasibility problem. In case the sum of infeasibilities is minimized only variables are added:
  - The variable `c.ExtendedVariable('PositiveViolation',i)` is added for a constraint `c(i)` with type $\geq$.
  - The variable `c.ExtendedVariable('NegativeViolation',i)` is added for a constraint `c(i)` with type $\leq$.
  - The variables `c.ExtendedVariable('PositiveViolation',i)` and `c.ExtendedVariable('NegativeViolation',i)` are added for an equality constraint `c(i)`.

  In case the maximum infeasibility is minimized the following variables and constraints are added:
  - The variable `mp.ExtendedVariable('MaximumViolation')` is added for math program `mp`.
  - The variables `c.ExtendedVariable('PositiveViolation',i)` and `c.ExtendedVariable('NegativeViolation',i)` are added for an equality constraint `c(i)`.
  - The constraint `c.ExtendedConstraint('MaximumViolation',i)` is added for an equality constraint `c(i)`.

  In the above mathematical formulation,
  - `c.ExtendedVariable('PositiveViolation',i)` corresponds to z_i^p.
  - `c.ExtendedVariable('NegativeViolation',i)` corresponds to z_i^n.
  - `mp.ExtendedVariable('MaximumViolation')` corresponds to z^m.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Instance::Solve* (page 424).

GMP::Instance::CreateMasterMIP

The function *GMP::Instance::CreateMasterMIP* (page 374) creates a Master MIP copy of the specified generated mathematical program. The copy will remove all nonlinear rows from the GMP.

```
GMP::Instance::CreateMasterMIP(
     GMP,            ! (input) a generated mathematical program
     name            ! (input) a string expression
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

name A string that holds the name for the Master MIP.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- The *name* argument should be different from the name of the original generated mathematical program.
- If an element with name specified by the *name* argument is already present in the set *AllGeneratedMathematicalPrograms* (page 833) the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved).
- The generated mathematical program should have type MINLP (or MIQP or MIQCP). It can also have type NLP in which case the created GMP will have type LP.
- If the objective constraint is nonlinear, *GMP::Instance::CreateMasterMIP* (page 374) adds an extra row and column to the Master MIP. If `mp` denotes the symbolic mathematical program then the extra row will be associated with `mp.ExtendedConstraint(MasterMIPObjective)` and the extra column with `mp.ExtendedVariable(MasterMIPObjective)`. The extra row will be

$$objvar - mp.ExtendedVariable(MasterMIPObjective) = 0$$

where `objvar` denotes the objective variable of the GMP. Column `mp.ExtendedVariable(MasterMIPObjective)` will become the objective column of the Master MIP.

See also:

The function *GMP::Instance::Generate* (page 384). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Instance::CreatePresolved

The function *GMP::Instance::CreatePresolved* (page 375) generates a mathematical program that is the presolved representation of the specified generated mathematical program. The generated mathematical program can be a linear or nonlinear model, and should be generated using the function *GMP::Instance::Generate* (page 384).

```
GMP::Instance::CreatePresolved(
     GMP,            ! (input) a generated mathematical program
     name            ! (input) a string expression
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

name A string that holds the name for the presolved mathematical program.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833), with the name as specified by the *name* argument, if the presolver did not find an infeasibility. Else, the empty element.

Note:

- By using the functions *GMP::Column::GetStatus* (page 333) and *GMP::Row::GetStatus* (page 465) it is possible to check whether a column or row was deleted when the presolved mathematical program was created.
- By using the functions *GMP::Column::GetLowerBound* (page 330) and *GMP::Column::GetUpperBound* (page 335) it is possible to retrieve the lower and upper bound of a column in the presolved mathematical program.
- If the original *GMP* is deleted then the presolved GMP created by *GMP::Instance::CreatePresolved* (page 375) will also be deleted.
- If the option `MINLP Probing` is switched on, then this function will change the mathematical programming type from MINLP (NLP) into MIP (LP) if the presolved model contains no nonlinear constraints.

Example

Assume that 'MP' is a mathematical program and 'gmpMP' and 'gmpPre' are element parameters with range *AllGeneratedMathematicalPrograms* (page 833). To solve the presolved model using GMP functions we can use:

```
gmpMP := GMP::Instance::Generate( MP );
gmpPre := GMP::Instance::CreatePresolved( gmpMP, "PresolvedModel" );

GMP::Instance::Solve( gmpPre ) ;
```

In case the GMP variant of the AOA module is used we can use:

```
gmpMP := GMP::Instance::Generate( MP );
gmpPre := GMP::Instance::CreatePresolved( gmpMP, "PresolvedModel" );

GMPOuterApprox::DoOuterApproximation( gmpPre );
```

Here 'GMPOuterApprox' is the prefix used by the GMP Outer Approximation Module.

See also:

The functions *GMP::Instance::Delete* (page 378), *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::Column::GetStatus* (page 333), *GMP::Row::GetStatus* (page 465), *GMP::Column::GetLowerBound* (page 330) and *GMP::Column::GetUpperBound* (page 335).

GMP::Instance::CreateProgressCategory

The function *GMP::Instance::CreateProgressCategory* (page 377) creates a new GMP progress category for a generated mathematical program. This progress category can be used to display GMP related information in the progress window.

```
GMP::Instance::CreateProgressCategory(
    GMP,              ! (input) a generated mathematical program
    [Name]            ! (input, optional) a string expression
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Name A string that holds the name of the progress category.

Return Value

The function returns an element in the set *AllProgressCategories* (page 834).

Note:

- If no progress category is specified for the generated mathematical program then the GMP progress will be displayed in the general AIMMS progress category for GMP progress. This general AIMMS progress category will be used by all generated mathematical programs for which no progress category is specified. (Progress information for a normal solve is always displayed in the general AIMMS progress category.)
- After calling *GMP::Instance::CreateProgressCategory* (page 377) solver progress will by default be displayed in the solver progress category of the generated mathematical program, and no longer in the general AIMMS progress category for solver progress.
- If the *Name* argument is not specified then the name of the generated mathematical program will be used to name the element in the set *AllProgressCategories* (page 834).
- The information displayed in a GMP progress category is controlled by AIMMS and cannot be modified by the user.
- A progress category created before for the generated mathematical program will be deleted.

See also:

The routines *GMP::ProgressWindow::DeleteCategory* (page 436) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::Instance::CreateSolverSession

The function *GMP::Instance::CreateSolverSession* (page 378) creates a new solver session for a generated mathematical program.

```
GMP::Instance::CreateSolverSession(
    GMP,              ! (input) a generated mathematical program
    [Name],           ! (input, optional) a string expression
    [Solver]          ! (input, optional) a solver
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Name A string that holds the name of the solver session.

Solver An element in the set *AllSolvers* (page 787).

Return Value

The function returns an element in the set *AllSolverSessions* (page 828).

Note:

- The function *GMP::Instance::CreateSolverSession* (page 378) also determines which solver is assigned to the solver session. After the solver session is created it is not possible to change the solver assigned to the solver session! The solver is determined by:
  - the *Solver* argument if it is specified (and not an empty string), else
  - the solver that was assigned to the *GMP* if procedure *GMP::Instance::SetSolver* (page 423) was called before, else
  - the default solver in AIMMS for the GMP its model type.
- If the Name argument is not specified, or if it is the empty string, the names of the symbolic mathematical program, the solver and the host (if any) are used to create a new element in the set *AllGeneratedMathematicalPrograms* (page 833).
- If an element with name specified by the *Name* argument is already present in the set *AllSolverSessions* (page 828) then the corresponding solver session will first be deleted.

See also:

The routines *GMP::Instance::DeleteSolverSession* (page 381), *GMP::Instance::SetSolver* (page 423), *GMP::SolverSession::GetInstance* (page 544) and *GMP::SolverSession::GetSolver* (page 551).

GMP::Instance::Delete

The procedure *GMP::Instance::Delete* (page 378) deletes a generated mathematical program from the set *AllGeneratedMathematicalPrograms* (page 833).

```
GMP::Instance::Delete(
     GMP             ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: All memory associated with the generated mathematical program is also freed.

See also:

The function *GMP::Instance::Generate* (page 384).

GMP::Instance::DeleteIntegerEliminationRows

The procedure *GMP::Instance::DeleteIntegerEliminationRows* (page 379) deletes a particular set of integer elimination rows and columns of a generated mathematical program.

```
GMP::Instance::DeleteIntegerEliminationRows(
     GMP,            ! (input) a generated mathematical program
     refNo           ! (input) a scalar integer value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

refNo A positive integer scalar value representing a reference number.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The procedure *GMP::Instance::AddIntegerEliminationRows* (page 359).

GMP::Instance::DeleteLimitBinaryDeviationRow

The procedure *GMP::Instance::DeleteIntegerEliminationRows* (page 379) deletes a row from a GMP that was added using the procedure *GMP::Instance::AddLimitBinaryDeviationRow* (page 361).

```
GMP::Instance::DeleteLimitBinaryDeviationRow(
     GMP,          ! (input) a generated mathematical program
     [refNo]       ! (optional, default 1) a scalar integer value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

refNo A positive integer scalar value representing a reference number. The default is 1.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The procedure *GMP::Instance::AddLimitBinaryDeviationRow* (page 361).

GMP::Instance::DeleteMultiObjectives

The procedure *GMP::Instance::DeleteMultiObjectives* (page 380) deletes all multi-objectives in a generated mathematical program.

```
GMP::Instance::DeleteMultiObjectives(
     GMP             ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: A column can be specified as a multi-objective by using the procedure *GMP::Column::SetAsMultiObjective* (page 337).

Example

In the example below two multi-objectives are specified after which a multi-objective optimization problem is solved. Next all multi-objectives are deleted by calling *GMP::Instance::CreateDual* (page 370) and the model is solved once again, this time as an ordinary optimization problem with one objective (namely the one specified in the objective attribute of the mathematical programming).

```
myGMP := GMP::Instance::Generate( MP );

GMP::Column::SetAsMultiObjective( myGMP, TotalDist, 2, 1.0, 0, 0.1 );
GMP::Column::SetAsMultiObjective( myGMP, TotalTime, 1, 1.0, 0, 0.0 );

GMP::Instance::Solve( myGMP );

GMP::Instance::DeleteMultiObjectives( myGMP );

GMP::Instance::Solve( myGMP );
```

See also:

The procedure *GMP::Column::SetAsMultiObjective* (page 337).

GMP::Instance::DeleteSolverSession

The procedure *GMP::Instance::DeleteSolverSession* (page 381) deletes the specified solver session.

```
GMP::Instance::DeleteSolverSession(
     solverSession      ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions *GMP::Instance::CreateSolverSession* (page 378) and *GMP::SolverSession::GetInstance* (page 544).

GMP::Instance::FindApproximatelyFeasibleSolution

The procedure *GMP::Instance::FindApproximatelyFeasibleSolution* (page 382) tries to find an approximately feasible solution of a generated mathematical program. It uses the column level values of the first solution as a starting point. The approximately feasible solution is stored in the second solution.

The algorithm used to find the approximately feasible solution is based on the constraint consensus method as developed by John W. Chinneck. The constraint consensus method is an iterative projection algorithm. In each iteration a new point (i.e., a vector of column values) is constructed in such a way that it is likely that it is closer to the feasible region (as defined by the generated mathematical program) then the previous point.

```
GMP::Instance::FindApproximatelyFeasibleSolution(
     GMP,            ! (input) a generated mathematical program
     solution1,      ! (input) a solution
     solution2,      ! (input) a solution
     nrIter,         ! (output) a scalar numerical parameter
     [maxIter],      ! (optional) a scalar value
     [feasTol],      ! (optional) a scalar value
     [moveTol],      ! (optional) a scalar value
     [imprTol],      ! (optional) a scalar value,
     [maxTime],      ! (optional) a scalar value
     [useSum],       ! (optional) a scalar value
     [augIter],      ! (optional) a scalar value
     [useBest]       ! (optional) a scalar value
)
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution1 An integer scalar reference to a solution.

solution2 An integer scalar reference to a solution.

nrIter The number of iterations used by the algorithm.

maxIter The maximal number of iterations that can be used by the algorithm. If its value is 0 (the default) then there is no iteration limit.

feasTol The feasibility distance tolerance. The default is 1e-5.

moveTol The movement tolerance. The default is 1e-5.

imprTol The improvement tolerance. The default is 0.01.

maxTime The maximum time (in seconds) that can be used by the algorithm. If its value is 0 (the default) then there is no time limit.

useSum A scalar binary value to indicate whether the SUM constraint consensus method should be used (value 1) or not (value 0; the default).

augIter An integer scalar reference that specifies the frequency of iterations in which augumentation should be applied. At the default value of 0 no augumentation is applied.

useBest A scalar binary value to indicate whether the best point found (value 1) or the last point found should be returnd (value 0; the default).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The (basic) constraint consensus method is described in: John W. Chinneck, The Constraint Consensus Method for Finding Approximately Feasible Points in Nonlinear Programs, INFORMS Journal on Computing 16(3) (2004), pp. 255-265.
- The SUM constraint consensus method and a constraint consensus method using augumentation are described in: Laurence Smith, John Chinneck, Victor Aitken, Improved constraint consensus methods for seeking feasibility in nonlinear programs, Computational Optimization and Applications 54(3) (2013), pp. 555-578.
- The algorithm terminates if:
  - The iteration limit *maxIter* is exceeded.
  - The time limit *maxTime* is exceeded.
  - The feasibility distance of each row is smaller than the feasibility distance tolerance *feasTol*. The feasibility distance of a row at a point is defined as the row violation normalized by the length of the gradient of the row at that point.
  - The length of the movement vector is smaller than the movement tolerance *moveTol*. The movement vector is the vector along which the point moves from one iteration to another.
  - The relative improvement was smaller than the improvement tolerance *imprTol* for 10 successive iterations. The improvement is defined as the difference between the length of the movement vector of the current iteration and that of the previous iteration.
- The procedure *GMP::Solution::Check* (page 479) can be used to get the sum and number of infeasibilies before and after calling the procedure *GMP::Instance::FindApproximatelyFeasibleSolution* (page 382).

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424) and *GMP::Solution::Check* (page 479).

GMP::Instance::FixColumns

The procedure *GMP::Instance::FixColumns* (page 383) sets the lower and upper bounds of a set of columns in a generated mathematical program (*GMP1*) equal to the level values of the corresponding columns in a solution of a second generated mathematical program (*GMP2*).

```
GMP::Instance::FixColumns(
     GMP1,          ! (input) a generated mathematical program
     GMP2,          ! (input) a generated mathematical program
     solution,      ! (input) a solution
     variableSet,   ! (input) a set of variables
     [round]        ! (optional) a binary scalar value
     )
```

Arguments

GMP1 An element in *AllGeneratedMathematicalPrograms* (page 833).

GMP2 An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution in the solution repository of *GMP2*.

variableSet A subset of *AllVariables* (page 831).

round A binary scalar indicating whether the level values of the integer columns should be rounded to the nearest integer value before fixing the columns. The default is 0 (no rounding).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- A column corresponding to a variable in *variableSet* that is not part of *GMP1* will be ignored. This procedure will fail if a column corresponding to a variable in *variableSet* is not part of *GMP2*.
- If the objective variable is part of the set *variableSet* then it will be ignored, i.e., the objective variable will not be fixed.
- The same generated mathematical program can be used for *GMP1* and *GMP2*.

See also:

The functions *GMP::Instance::CreateSolverSession* (page 378) and *GMP::SolverSession::GetInstance* (page 544).

GMP::Instance::Generate

The function *GMP::Instance::Generate* (page 384) generates a mathematical program instance from a symbolic mathematical program.

```
GMP::Instance::Generate(
     MP,              ! (input) a symbolic mathematical program
     [name]           ! (optional) a string expression
     )
```

Arguments

MP A symbolic mathematical program in the set *AllMathematicalPrograms* (page 824). The mathematical program should have model type LP, MIP, QP, MIQP, QCP, MIQCP, NLP, MINLP, RMIP or RMINLP.

name A string that holds the name for the mathematical program to be generated.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- If the second argument is not specified, or if it is the empty string, the name of the symbolic mathematical program is used to create a new element in the set *AllGeneratedMathematicalPrograms* (page 833).
- If an element with name specified by the *name* argument is already present in the set *AllGeneratedMathematicalPrograms* (page 833) the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved or the second argument is not specified). In that case all existing solver sessions created for the generated mathematical program will be deleted.
- It is possible to generate indexed mathematical program instances. See the example in Indexed Mathematical Program Instances of the Language Reference.
- A callback procedure should be installed using the appropriate GMP procedure, (e.g., *GMP::Instance::SetCallbackIterations* (page 416)) instead of using a suffix of the mathematical program (e.g., suffix `CallbackIterations`).
- If an error occurs during the execution of *GMP::Instance::Generate* (page 384), e.g., if one of the constraints appears to be empty and infeasible, then the program status of the mathematical program will be set to `Infeasible` and the solver status to `PreprocessorError`.

Example

Assume that 'MP' is a symbolic mathematical program which we want to solve twice thereby changing the constraint set:

```
ConstraintSet := { 'C1', 'C2' };
myGMP1 := GMP::Instance::Generate( MP );
GMP::Instance::Solve( myGMP1 );

ConstraintSet := { 'C3', 'C4' };
myGMP2 := GMP::Instance::Generate( MP );
```

After executing these statements, 'myGMP1' and 'myGMP2' will point to the same generated math program, namely the one using the constraints 'C3' and 'C4'. The set *AllGeneratedMathematicalPrograms* (page 833) will contain only one element, namely 'MP'. At this point it makes no difference to use

```
GMP::Instance::Solve( myGMP1 );
```

or

```
GMP::Instance::Solve( myGMP2 );
```

By using the *name* argument we can create two different GMP's from the same symbolic mathematical program:

```
ConstraintSet := { 'C1', 'C2' };
myGMP1 := GMP::Instance::Generate( MP, "FirstGMP" );
GMP::Instance::Solve( myGMP1 );

ConstraintSet := { 'C3', 'C4' };
myGMP2 := GMP::Instance::Generate( MP, "SecondGMP" );
GMP::Instance::Solve( myGMP2 );
```

This time the set *AllGeneratedMathematicalPrograms* (page 833) will contain two elements, namely 'FirstGMP' and 'SecondGMP'.

See also:

The routines *GMP::Instance::Delete* (page 378) and *GMP::Instance::SetCallbackIterations* (page 416).

GMP::Instance::GenerateRobustCounterpart

The function *GMP::Instance::GenerateRobustCounterpart* (page 386) generates the robust counterpart of a (linear) mathematical program.

If the deterministic model is a linear program (LP) then the robust counterpart will be a LP if the uncertainty constraints are linear, or a second-order cone program (SOCP) if some of the uncertainty constraints are ellipsoidal.

If the deterministic model is a mixed-integer program (MIP) then the robust counterpart will be a MIP if the uncertainty constraints are linear, or a mixed-integer second-order cone program (MISOCP) if some of the uncertainty constraints are ellipsoidal.

SOCP and MISOCP problems can be solved by using CPLEX or Gurobi.

```
GMP::Instance::GenerateRobustCounterpart(
     MP,                        ! (input) a symbolic mathematical program
     UncertainParameters,       ! (input) a set of uncertain parameters
     UncertaintyConstraints,    ! (input) a set of uncertainty constraints
     [Name]                     ! (optional) a string expression
)
```

Arguments

MP A symbolic mathematical program in the set *AllMathematicalPrograms* (page 824). The mathematical program should have model type LP or MIP.

UncertainParameters A subset of `AllUncertainParameters`.

UncertaintyConstraints A subset of `AllUncertaintyConstraints`.

Name A string that holds the name for the generated robust counterpart.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- If the *Name* argument is not specified, or if it is the empty string, then the name of the symbolic mathematical program followed by 'robust counterpart' is used to create a new element in the set *AllGeneratedMathematicalPrograms* (page 833).
- If AIMMS detects that the robust counterpart is infeasible during the generation, AIMMS will issue a warning and the robust counterpart will not be generated.
- As part of the generation, AIMMS will check whether the uncertainty set satisfies the Slater condition (controlled by the option `Slater_condition_check`). To do so, AIMMS will solve a linear program (LP) or a second-order cone program (SOCP).
- The created GMP cannot be modified, e.g., it is not allowed to change row or columns in the robust counterpart.

See also:

The procedure *GMP::Instance::Solve* (page 424).

GMP::Instance::GenerateStochasticProgram

The function *GMP::Instance::GenerateStochasticProgram* (page 387) generates the deterministic equivalent of a stochastic mathematical program.

```
GMP::Instance::GenerateStochasticProgram(
     MP,                    ! (input) a symbolic mathematical program
     StochasticParameters,  ! (input) a set of stochastic parameters
     StochasticVariables,   ! (input) a set of stochastic variables
     Scenarios,             ! (input) a set of stochastic scenarios
     ScenarioProbability,   ! (input) a double parameter
     ScenarioTreeMap,       ! (input) an element parameter
     RootScenarioName,      ! (input) a string expression
     [GenerationMode],      ! (optional) a stochatic generation mode
     [Name]                 ! (optional) a string expression
)
```

Arguments

MP A symbolic mathematical program in the set *AllMathematicalPrograms* (page 824). The mathematical program should have model type `LP` or `MIP`.

StochasticParameters A subset of *AllStochasticParameters* (page 829).

StochasticVariables A subset of *AllStochasticVariables* (page 830).

Scenarios A subset of *AllStochasticScenarios* (page 834).

ScenarioProbability A double parameter over `Scenarios` representing the objective probabilities of the scenarios.

ScenarioTreeMap An element parameter that defines the scenario-and-stage to scenario mapping. The range of this parameter should be the set `Scenarios`.

RootScenarioName A string that holds the name of the artificial element that will be added to the set *AllStochasticScenarios* (page 834). This element will be used to store the solution of non-stochastic variables in their respective `.Stochastic` suffixes.

GenerationMode An element in the predefined set *AllStochasticGenerationModes* (page 808). The default is '`SubstituteStochasticVariables`'.

Name A string that holds the name for the generated stochastic mathematical program.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- If the *Name* argument is not specified, or if it is the empty string, then the name of the symbolic mathematical program preceded by 'Stochastic' is used to create a new element in the set *AllGeneratedMathematicalPrograms* (page 833).
- The objective of the symbolic mathematical program must be a defined variable.

See also:

- Stochastic programming is discussed in Stochastic Programming of the Language Reference.
- The procedure *GMP::Instance::Solve* (page 424).

GMP::Instance::GetAttributeValue

The function *GMP::Instance::GetAttributeValue* (page 389) can be used to get the value of certain solver attributes, related to solving a generated mathematical program.

```
GMP::Instance::GetAttributeValue(
     GMP,            ! (input) a generated mathematical program
     attr,           ! (input) a string expression
     [objNo]         ! (input, optional) an integer expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

attr A string that holds the name of the attribute.

objNo A nonnegative scalar value specifying the objective number in a multi-objective optimization problem. The default value is 0.

Return Value

In case of success, the function returns the attribute value. Otherwise it returns `UNDF`.

Note:

- The attribute name is case-insensitive.
- This function is only supported by CPLEX and Gurobi.
- This function cannot be called inside a callback procedure.

Attributes

The table below shows the model attributes supported by CPLEX and Gurobi.

Attribute	CPLEX	Gurobi	Description
ObjCon	✓	✓	Objective constant/offset
NumStart	✓	✓	Number of MIP starts (MIP only)
NumObj	✓	✓	Number of objectives

The table below shows the solution attributes supported by CPLEX and Gurobi.

Attribute	CPLEX	Gurobi	Description
ObjVal	✓	✓	Objective value
ObjBound	✓	✓	Best bound (MIP only)
MIPGap	✓	✓	Relative MIP gap (MIP only)
Runtime	✓	✓	Runtime in seconds
Work	✓	✓	Work or deterministic time
MaxMemUsed		✓	Maximum amount (in GB) of memory used
MemUsed		✓	Current amount (in GB) of memory used
SolCount		✓	Number of solutions (MIP only)
IterCount	✓	✓	Number of simplex iterations
BarIterCount	✓	✓	Number of barrier iterations
NodeCount	✓	✓	Number of branch-and-cut nodes (MIP only)
OpenNodeCount	✓	✓	Number of open branch-and-cut nodes (MIP only)
NodeInt	✓		Node number of the best solution (MIP only)

The table below shows the solution quality attributes supported by CPLEX and Gurobi.

Attribute	CPLEX	Gurobi	Description
MaxVio	✓	✓	Maximum unscaled violation
IntVio	✓	✓	Maximum integrality violation
Kappa	✓	✓	Estimated condition number
KappaExact	✓	✓	Exact condition number
KappaMax	✓		Highest condition number (MIP only)
KappaAttention	✓		Score of numerical difficulties (MIP only)
KappaStable	✓		Percentage of stable simplex bases (MIP only)
KappaSuspicious	✓		Percentage of suspicious simplex bases (MIP only)
KappaUnstable	✓		Percentage of unstable simplex bases (MIP only)
KappaIllposed	✓		Percentage of ill-posed simplex bases (MIP only)

The table below shows the attributes for multi-objective optimization supported by CPLEX and Gurobi. These attribites can be used to retrieve model or solution information for each subproblem solved, as specified by the *objNo* argument.

Attribute	CPLEX	Gurobi	Description
ObjNPriority	✓	✓	Objective priority
ObjNCon		✓	Objective constant/offset
ObjVal/ObjNVal	✓	✓	Objective value
ObjBound	✓		Best bound (MIP only)
MIPGap	✓		Relative MIP gap (MIP only)
Runtime	✓		Runtime in seconds
Work	✓		Work or deterministic time
Status	✓		Solution status
NodeCount	✓		Number of branch-and-cut nodes (MIP only)
OpenNodeCount	✓		Number of open branch-and-cut nodes (MIP only)
IterCount	✓		Number of simplex iterations
BarIterCount	✓		Number of barrier iterations

For Gurobi also other model and solution attributes are supported. For a complete list; see: Model attributes. Attributes with type 'string' are not supported by this function.

Example

In the example below two multi-objectives are specified, each with its own priority. Therefore two subproblems are solved; one for each objective. Below we retrieve the (relative) MIP gap for both subproblems. We also retrieve the total runtime by the solver.

```
myGMP := GMP::Instance::Generate( MP );

GMP::Column::SetAsMultiObjective( myGMP, TotalDist, 2, 1.0 );
GMP::Column::SetAsMultiObjective( myGMP, TotalTime, 1, 1.0 );

GMP::Instance::Solve( myGMP );

gap1 := GMP::Instance::GetAttributeValue( myGMP, "MIPGap", 1 );
gap2 := GMP::Instance::GetAttributeValue( myGMP, "MIPGap", 2 );

runtime := GMP::Instance::GetAttributeValue( myGMP, "Runtime" );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::Column::SetAsMultiObjective* (page 337) and *GMP::SolverSession::GetAttributeValue* (page 537).

GMP::Instance::GetBestBound

The function *GMP::Instance::GetBestBound* (page 391) returns the best known bound for a generated mathematical program.

```
GMP::Instance::GetBestBound(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

In case of success, the function returns the best known bound. Otherwise it returns `UNDF`.

Note:

- This function can be used for GMPs with model type MIP, MIQP or MIQCP.
- This function can also be used for GMPs solved with BARON, regardless of the model type.
- This function can also be used for GMPs with model type QP that are solved with CPLEX, if the CPLEX option *Solution Target* is set to 'Search for global optimum'.
- This function can also be used for GMPs with model type QP, QCP, NLP or MINLP that are solved with GUROBI.

- For multi-objective models, the best bound refers to the (blended) objective with the highest priority.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetMathematicalProgrammingType* (page 395) and *GMP::Instance::GetObjective* (page 401).

GMP::Instance::GetColumnNumbers

The function *GMP::Instance::GetColumnNumbers* (page 392) returns a subset of the column numbers of a generated mathematical program. It returns the column numbers that are generated for a set of variables.

```
GMP::Instance::GetColumnNumbers(
     GMP,                ! (input) a generated mathematical program
     variableSet         ! (input) a set of variables
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

variableSet A subset of the set *AllVariables* (page 831).

Return Value

The function returns a subset of the column numbers as a subset of the set *Integers* (page 812).

Example

Assume we have generated a mathematical program and we want to change the upper bound of the variables `demand(i)` and `supply(j,k)` into 100. This can be done as follows:

```
myGMP := GMP::Instance::Generated( MP );

for (i) do
    GMP::Column::SetUpperBound( myGMP, demand(i), 100 );
endfor;

for (j,k) do
    GMP::Column::SetUpperBound( myGMP, supply(j,k), 100 );
endfor;
```

Using the function *GMP::Instance::GetColumnNumbers* (page 392) this can also be coded as follows. Here `ColNrs` is a subset of *Integers* (page 812) with index `c`, and `Vars` a subset of *AllVariables* (page 831).

```
myGMP := GMP::Instance::Generated( MP );

Vars := { 'demand', 'supply' };
```

(continues on next page)

(continued from previous page)

```
ColNrs := GMP::Instance::GetColumnNumbers( myGMP, Vars );

for (c) do
    GMP::Column::SetUpperBound( myGMP, c, 100 );
endfor;
```

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfColumns* (page 396), *GMP::Instance::GetRowNumbers* (page 405), *GMP::Instance::GetObjectiveColumnNumber* (page 402) and *GMP::Instance::GetObjectiveRowNumber* (page 403).

GMP::Instance::GetDirection

The function *GMP::Instance::GetDirection* (page 393) returns the optimization direction of a generated mathematical program.

```
GMP::Instance::GetDirection(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the optimization direction as an element in *AllMatrixManipulationDirections* (page 802).

See also:

The routines *GMP::Instance::Generate* (page 384) and the procedure *GMP::Instance::SetDirection* (page 419).

GMP::Instance::GetInfeasibleData

Usually, methods for detecting the cause of an infeasibiliy in a mathematical model return information about constraints and variables. The procedure *GMP::Instance::GetInfeasibleData* (page 393) uses a different approach, namely, it can be used to retrieve information about parameters that cause a (linear) mathematical program to become infeasible.

This procedure only considers parameters without a definition, i.e., parameters that can be changed by, e.g., an end user.

This procedure returns information in two ways. First, it will fill the output *message* argument with a text describing the cause of the infeasibility. Second, it will fill the suffix `.SuspicionLevel` for each parameter that contributes to the infeasibility. This suffix may be used to assign identifier annotations to the parameters of interest. Such annotations may be subsequently used in order to highlight the suspicious values of those parameters in the graphical user interface and visually notify the user about potentially incorrect data values.

```
GMP::Instance::GetInfeasibleData(
     GMP,            ! (input) a generated mathematical program
     parSet,         ! (input) a set of parameters
     message,        ! (output) a string parameter
     [method],       ! (optional) a scalar integer value
     [effort],       ! (optional) a scalar integer value
     [textFormat]    ! (optional) a scalar binary value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

parSet A subset of *AllParameters* (page 825).

message A string that describes the cause of the infeasibility.

method An integer scalar value between 0 and 4 that specifies the method used to find the cause of the infeasibility. Default is 0. The methods are described below.

effort An integer scalar value between 0 and 2 that specifies the effort level used by this procedure to find the cause of the infeasibility. Default is 2. A higher value means more effort.

textFormat A scalar binary value to indicate whether the message should use plain text (value 0; the default) or HTML formatting (value 1).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP* must be linear.
- Typically, *AllParameters* (page 825) is passed to the *parSet* argument.
- The procedure *GMP::Instance::GetInfeasibleData* (page 393) will use one of the following methods, as specified by the *method* argument:
  - 0: IIS (default)
  - 1: Solve feasibility problem: minimize sum of infeasibilities
  - 2: Solve feasibility problem: minimize largest infeasibility
  - 3: Solve feasibility problem: minimize sum of infeasibilities after scaling model
  - 4: Solve feasibility problem: minimize largest infeasibility after scaling model

- The calculation of the IIS or solving the feasibility problem can be time consuming if the model is large or complex, and as a consequence, the procedure *GMP::Instance::GetInfeasibleData* (page 393) will then be time consuming.
- The *effort* level influences how much effort is used to find outliers in the (infeasible) data. It does not influence the running time of the *method* used.
- If the text attribute of a parameter is specified then it will be used in the *message*, otherwise the identifier name.
- If the webui::ElementTextIdentifier attribute of a set is specified then it will be used to print the set elements in the *message*.
- The option `Element format` determines the format used for printing elements in the *message*.
- The suffix `.SuspicionLevel` gets a value from the set *AllSuspicionLevels*, or remains empty if the parameter is not part of the infeasible data.
- The procedure *GMP::Instance::GetInfeasibleData* (page 393) is not supported for generated mathematical programs created by one of the following functions:
  - GMP::Instance::GenerateRobustCounterpart,
  - GMP::Instance::GenerateStochasticProgram,
  - GMP::Instance::CreatePresolved,
  - GMP::Instance::CreateDual, or
  - GMP::Instance::CreateMasterMIP.

Example

Assume that 'MP' is a symbolic mathematical program and 'pMessage' a string parameter.

```
solve MP;

if ( MP.ProgramStatus = 'Infeasible'            or
     MP.ProgramStatus = 'InfeasibleOrUnbounded' ) then
    GMP::Instance::GetInfeasibleData( 'MP', AllParameters, pMessage,
                                      method: 4, textFormat: 1 );

    ! Change the font size for the HTML formatted text.
    pMessage := "<span style=\"font-size: 20px\">" + pMessage + "</span>";
endif;
```

See also:

- Section Explainability in the Language Reference.
- Example Infeasible Data.

GMP::Instance::GetMathematicalProgrammingType

The function *GMP::Instance::GetMathematicalProgrammingType* (page 395) returns the model type of a generated mathematical program.

```
GMP::Instance::GetMathematicalProgrammingType(
    GMP             ! (input) a generated mathematical program
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the model type as an element in *AllMathematicalProgrammingTypes* (page 802).

See also:

The function *GMP::Instance::Generate* (page 384) and the procedure *GMP::Instance::SetMathematicalProgrammingType* (page 420).

GMP::Instance::GetMemoryUsed

The function *GMP::Instance::GetMemoryUsed* (page 396) returns for a generated mathematical program the amount of memory used by AIMMS to store it.

```
GMP::Instance::GetMemoryUsed(
    GMP             ! (input) a generated mathematical program
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The amount of megabytes used to store a generated mathematical program.

GMP::Instance::GetNumberOfColumns

The function *GMP::Instance::GetNumberOfColumns* (page 396) returns the number of columns of a generated mathematical program.

```
GMP::Instance::GetNumberOfColumns(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of columns.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfRows* (page 400) and *GMP::Instance::GetNumberOfNonzeros* (page 399).

GMP::Instance::GetNumberOfIndicatorRows

The function *GMP::Instance::GetNumberOfIndicatorRows* (page 397) returns the number of indicator rows of a generated mathematical program.

```
GMP::Instance::GetNumberOfIndicatorRows(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of indicator rows.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Instance::GetNumberOfRows* (page 400).

GMP::Instance::GetNumberOfIntegerColumns

The function *GMP::Instance::GetNumberOfIntegerColumns* (page 397) returns the number of integer columns of a generated mathematical program.

```
GMP::Instance::GetNumberOfIntegerColumns(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of integer columns.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfColumns* (page 396) and *GMP::Instance::GetNumberOfNonlinearColumns* (page 398).

GMP::Instance::GetNumberOfNonlinearColumns

The function *GMP::Instance::GetNumberOfNonlinearColumns* (page 398) returns the number of nonlinear columns of a generated mathematical program.

```
GMP::Instance::GetNumberOfNonlinearColumns(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of nonlinear columns.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfColumns* (page 396) and *GMP::Instance::GetNumberOfIntegerColumns* (page 397).

GMP::Instance::GetNumberOfNonlinearNonzeros

The function *GMP::Instance::GetNumberOfNonlinearNonzeros* (page 398) returns the number of nonlinear nonzero elements in the coefficient matrix of a generated mathematical program.

```
GMP::Instance::GetNumberOfNonlinearNonzeros(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of nonlinear nonzeros.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Instance::GetNumberOfNonzeros* (page 399).

GMP::Instance::GetNumberOfNonlinearRows

The function *GMP::Instance::GetNumberOfNonlinearRows* (page 399) returns the number of nonlinear rows of a generated mathematical program.

```
GMP::Instance::GetNumberOfNonlinearRows(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of nonlinear rows.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Instance::GetNumberOfRows* (page 400).

GMP::Instance::GetNumberOfNonzeros

The function *GMP::Instance::GetNumberOfNonzeros* (page 399) returns the number of nonzero elements in the coefficient matrix of a generated mathematical program.

```
GMP::Instance::GetNumberOfNonzeros(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of nonzeros.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfColumns* (page 396) and *GMP::Instance::GetNumberOfRows* (page 400).

GMP::Instance::GetNumberOfRows

The function *GMP::Instance::GetNumberOfRows* (page 400) returns the number of rows of a generated mathematical program.

```
GMP::Instance::GetNumberOfRows(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of rows.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfColumns* (page 396) and *GMP::Instance::GetNumberOfNonzeros* (page 399).

GMP::Instance::GetNumberOfSOS1Rows

The function *GMP::Instance::GetNumberOfSOS1Rows* (page 400) returns the number of SOS rows of type 1 of a generated mathematical program.

```
GMP::Instance::GetNumberOfSOS1Rows(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of SOS rows of type 1.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfRows* (page 400) and *GMP::Instance::GetNumberOfSOS2Rows* (page 401).

GMP::Instance::GetNumberOfSOS2Rows

The function *GMP::Instance::GetNumberOfSOS2Rows* (page 401) returns the number of SOS rows of type 2 of a generated mathematical program.

```
GMP::Instance::GetNumberOfSOS2Rows(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the number of SOS rows of type 2.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetNumberOfRows* (page 400) and *GMP::Instance::GetNumberOfSOS1Rows* (page 400).

GMP::Instance::GetObjective

The function *GMP::Instance::GetObjective* (page 401) returns the current objective function value of a generated mathematical program.

```
GMP::Instance::GetObjective(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

In case of success, the function returns the current objective function value. Otherwise it returns `UNDF`.

Note: For multi-objective models, the objective value refers to the (blended) objective with the highest priority.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424) and *GMP::Instance::GetBestBound* (page 391).

GMP::Instance::GetObjectiveColumnNumber

The function *GMP::Instance::GetObjectiveColumnNumber* (page 402) returns the column number corresponding to the objective variable of a generated mathematical program.

```
GMP::Instance::GetObjectiveColumnNumber(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the column number as an element of the set *Integers* (page 812). If the generated mathematical program does not contain an objective then -1 is returned.

Note: You should assign the return value of this function to an element parameter with range *Integers* (page 812) if you want to use it as (column) argument to call other GMP routines.

Example

Let ColNo be an element parameter with range *Integers* (page 812).

```
ColNo := GMP::Instance::GetObjectiveColumnNumber( myGMP );

value := GMP::Solution::GetColumnValue( myGMP, 1, ColNo );
```

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Instance::GetObjectiveRowNumber* (page 403) and *GMP::Instance::GetRowNumbers* (page 405).

GMP::Instance::GetObjectiveRowNumber

The function *GMP::Instance::GetObjectiveRowNumber* (page 403) returns the row number corresponding to the constraint or variable definition that defines the objective of a generated mathematical program.

```
GMP::Instance::GetObjectiveRowNumber(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the row number as an element of the set *Integers* (page 812). If the generated mathematical program does not contain an objective then -1 is returned.

Note:

- You should assign the return value of this function to an element parameter with range *Integers* (page 812) if you want to use it as (row) argument to call other GMP routines.
- If the objective variable appears in more than one constraint (or variable definition) then the row number of the first of those constraints will be returned.

Example

Assume that we want to change the coefficients of all integer variables in the objective to 10. This can be done as follows.

```
RowNo := GMP::Instance::GetObjectiveRowNumber( myGMP );

ColNrs := GMP::Instance::GetColumnNumbers( myGMP, AllIntegerVariables );
```

(continues on next page)

(continued from previous page)

```
for (c) do
    GMP::Coefficient::Set( myGMP, RowNo, c, 10 );
endfor;
```

Here RowNo is an element parameter with range *Integers* (page 812) and ColNrs a subset of *Integers* (page 812) with index c.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Instance::GetObjectiveColumnNumber* (page 402) and *GMP::Instance::GetRowNumbers* (page 405).

GMP::Instance::GetOptionValue

The function *GMP::Instance::GetOptionValue* (page 404) returns the value of a solver specific option corresponding to a generated mathematical program as set with the procedure *GMP::Instance::SetOptionValue* (page 421).

This procedure can also be used to retrieve the current option value of certain Solvers General options (see below).

```
GMP::Instance::GetOptionValue(
    GMP,            ! (input) a generated mathematical program
    OptionName      ! (input) a scalar string expression
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

OptionName A string expression holding the name of the option.

Return Value

In case of success, the function returns the current option value. Otherwise it returns UNDF.

Note:

- If the procedure *GMP::Instance::SetOptionValue* (page 421) has not been called then this function will fail and return UNDF (unless the option is a Solvers General option).
- Options for which strings are displayed in the AIMMS **Options** dialog box, are also represented by numerical (integer) values. To obtain the corresponding option keywords, you can use the functions OptionGetString and OptionGetKeywords.
- This procedure can also be used to retrieve the current option value of the following Solvers General options:
  - Cutoff
  - Iteration limit
  - Maximal number of domain errors

- Maximal number of integer solutions
- MIP absolute optimality tolerance
- MIP relative optimality tolerance
- Solver workspace
- Time limit

See also:

The routines *GMP::Instance::SetOptionValue* (page 421), *GMP::SolverSession::GetOptionValue* (page 550), *GMP::SolverSession::SetOptionValue* (page 555), *OptionGetString* (page 742) and *OptionGetKeywords* (page 742).

GMP::Instance::GetRowNumbers

The function *GMP::Instance::GetRowNumbers* (page 405) returns a subset of the row numbers of a generated mathematical program. It returns the row numbers that are generated for a set of constraints.

```
GMP::Instance::GetRowNumbers(
    GMP,                  ! (input) a generated mathematical program
    constraintSet        ! (input) a set of constraints
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

constraintSet A subset of the set *AllConstraints* (page 817).

Return Value

The function returns a subset of the row numbers as a subset of the set *Integers* (page 812).

Example

Assume we have generated a mathematical program and we want to change the right hand side of the constraints `c1(i)` and `c2(j,k)` into 100. This can be done as follows:

```
myGMP := GMP::Instance::Generated( MP );

for (i) do
    GMP::Row::SetUpperBound( myGMP, c1(i), 100 );
endfor;

for (j,k) do
    GMP::Row::SetRightHandSide( myGMP, c2(j,k), 100 );
endfor;
```

Using the function *GMP::Instance::GetRowNumbers* (page 405) this can also be coded as follows. Here RowNrs is a subset of *Integers* (page 812) with index r, and Cons a subset of *AllConstraints* (page 817).

```
myGMP := GMP::Instance::Generated( MP );

Cons := { 'c1', 'c2' };

RowNrs := GMP::Instance::GetRowNumbers( myGMP, Cons );

for (r) do
    GMP::Row::SetRightHandSide( myGMP, r, 100 );
endfor;
```

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Instance::GetNumberOfRows* (page 400), *GMP::Instance::GetObjectiveColumnNumber* (page 402) and *GMP::Instance::GetObjectiveRowNumber* (page 403).

GMP::Instance::GetSolver

The function *GMP::Instance::GetSolver* (page 406) returns for a generated mathematical program the solver that is assigned to it.

```
GMP::Instance::GetSolver(
    GMP             ! (input) a generated mathematical program
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the solver as an element of *AllSolvers* (page 787).

Note: The solver can be assigned by the procedure *GMP::Instance::SetSolver* (page 423), or derived by AIMMS as the default solver for the model class of the generated mathematical program.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Instance::SetSolver* (page 423).

GMP::Instance::GetSymbolicMathematicalProgram

The function *GMP::Instance::GetSymbolicMathematicalProgram* (page 406) returns for a generated mathematical program the originating symbolic mathematical program.

```
GMP::Instance::GetSymbolicMathematicalProgram(
    GMP             ! (input) a generated mathematical program
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The function returns the symbolic mathematical program as an element of *AllMathematicalPrograms* (page 824).

See also:

The function *GMP::Instance::Generate* (page 384).

GMP::Instance::MemoryStatistics

With the procedure *GMP::Instance::MemoryStatistics* (page 407) you can obtain a report containing the statistics collected by AIMMS' memory manager for a single or multiple generated mathemetical programs.

```
GMP::Instance::MemoryStatistics(
    gmpSet,              ! (input) a set of generated mathematical programs
    OutputFileName,      ! (input) scalar string expression
    AppendMode,          ! (optional, default 0) scalar numerical expression
    MarkerText,          ! (optional) scalar string expression
    ShowLeaksOnly,       ! (optional) scalar expression
    ShowTotals,          ! (optional) scalar expression
    ShowSinceLastDump,   ! (optional) scalar expression
    ShowMemPeak,         ! (optional) scalar expression
    ShowSmallBlockUsage, ! (optional) scalar expression
    doAggregate          ! (optional, default 0) scalar expression
    )
```

Arguments

gmpSet A subset of *AllGeneratedMathematicalPrograms* (page 833) with generated mathematical programs whose memory statistics are to be reported.

OutputFileName A string expression holding the name of the file to which the statistics must be written.

AppendMode An 0-1 value indicating whether the file must be overwritten or whether the statistics must be appended to an existing file.

MarkerText A string printed at the top of the memory statistics report.

ShowLeaksOnly A 0-1 value that is only used internally by AIMMS. The value specified doesn't influence the memory statistics report.

ShowTotals A 0-1 value indicating whether the report should include detailed information about the total memory use in AIMMS' own memory management system until the moment of calling *GMP::Instance::MemoryStatistics* (page 407).

ShowSinceLastDump A 0-1 value indicating whether the report should include basic and detailed information about the memory use in AIMMS' own memory management system since the previous call to GMP:Instance::MemoryStatistics.

ShowMemPeak A 0-1 value indicating whether the report should include detailed information about the memory use in AIMMS' own memory management system, when the memory consumption was at its peak level prior to calling *GMP::Instance::MemoryStatistics* (page 407).

ShowSmallBlockUsage A 0-1 value indicating whether the detailed information about the MemoryStatistics memory use in AIMMS' own memory management system is included at all in the memory statistics report. Setting this value to 0 results in a report with only the most basic statistical information about the memory use.

doAggregate A 0-1 value (default 0) indicating whether a single aggregated report is to be presented or multiple individual reports.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The procedure prints a report of the statistics collected by AIMMS' memory manager since the last call to *GMP::Instance::MemoryStatistics* (page 407).
- AIMMS will only collect memory statistics if the option memory_statistics is on.

GMP::Instance::Rename

The procedure *GMP::Instance::Rename* (page 408) can be used to rename a generated mathematical program.

```
GMP::Instance::Rename(
    GMP,            ! (input) a generated mathematical program
    Name            ! (input) a string expression
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Name A string that holds the new name.

Return Value

GMP::Instance::Rename (page 408) has no return value.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Instance::Copy* (page 364).

GMP::Instance::RestoreState

With procedure *GMP::Instance::RestoreState* (page 409) you can restore the state of a generated mathematical program as it was on the moment that you called *GMP::Instance::SaveState* (page 409).

```
GMP::Instance::RestoreState(
     GMP,              ! (input) a generated mathematical program
     state             ! (input) an integer scalar parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

state The value corresponding to a state that you want to restore.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The procedure *GMP::Instance::SaveState* (page 409).

GMP::Instance::SaveState

With the procedure *GMP::Instance::SaveState* (page 409) you can save the current state of a generated mathematical program. Later on, after manipulating the generated mathematical program, you can restore this state by calling *GMP::Instance::RestoreState* (page 409).

```
GMP::Instance::SaveState(
     GMP,              ! (input) a generated mathematical program
     state             ! (output) an integer scalar parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

state On return, contains a positive integer value assigned to the state.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: States are numbered from 1 upwards by AIMMS.

See also:

The procedure *GMP::Instance::RestoreState* (page 409).

GMP::Instance::SetCallbackAddCut

The procedure *GMP::Instance::SetCallbackAddCut* (page 410) installs a callback procedure adding cuts during the solution process of a MIP model.

```
GMP::Instance::SetCallbackAddCut(
     GMP,             ! (input) a generated mathematical program
     callback         ! (input) an AIMMS procedure
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The procedure *GMP::SolverSession::GenerateCut* (page 535) can be used inside a `CallbackAddCut` callback procedure to add cuts during the MIP branch-and-cut process.
- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackAddCut` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.

- A `CallbackAddCut` callback procedure will only be called when solving mixed integer programs with CPLEX, Gurobi or ODH-CPLEX.
- This procedure can also be used for MIQP and MIQCP problems.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415), *GMP::SolverSession::GenerateBinaryEliminationRow* (page 531) and *GMP::SolverSession::GenerateCut* (page 535).

GMP::Instance::SetCallbackAddLazyConstraint

The procedure *GMP::Instance::SetCallbackAddLazyConstraint* (page 411) installs a callback procedure for adding lazy constraints during the solution process of a MIP model.

```
GMP::Instance::SetCallbackAddLazyConstraint(
    GMP,            ! (input) a generated mathematical program
    callback        ! (input) an AIMMS procedure
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The callback procedure is called by the solver in these situations
  - when the solver compares an integer-feasible solution (including an integer-feasible solution provided by a MIP start before any nodes exist) to lazy constraints;
  - when the LP at a node is unbounded, and a lazy constraint might cut off the primal ray.
- The procedure *GMP::SolverSession::GenerateCut* (page 535) can be used inside a `CallbackAddLazyConstraint` callback procedure to add (globally or locally valid) lazy constraints during the MIP branch-and-cut process. Lazy constraints added to the problem are first put into a pool of lazy constraints, so they are not present in the subproblem LP until after the callback is finished.
- If lazy constraints have been added, the subproblem is re-solved and evaluated, and, if the LP solution is still integer feasible and not cut off, the lazy constraint callback is called again.
- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackAddLazyConstraint` callback procedure should have a return value of

  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.
- A `CallbackAddLazyConstraint` callback procedure will only be called when solving mixed integer programs with CPLEX or Gurobi.
- This procedure can also be used for MIQP and MIQCP problems.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415) and *GMP::SolverSession::GenerateCut* (page 535).

GMP::Instance::SetCallbackBranch

The procedure *GMP::Instance::SetCallbackBranch* (page 412) installs a callback procedure to be called after a branch has been selected but before the branch is carried out during the MIP optimization. In the callback routine, the branch selected by the solver can be changed to a user-selected branch.

```
GMP::Instance::SetCallbackBranch(
    GMP,            ! (input) a generated mathematical program
    callback        ! (input) an AIMMS procedure
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This callback is not called when the subproblem is infeasible.
- In the callback procedure at most 2 branches can be specified.
- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackBranch` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.

- The CallbackBranch callback procedure cannot be used to get the column on which the solver will branch.
- A CallbackBranch callback procedure will only be called when solving mixed integer programs with CPLEX.

See also:

The routines *GMP::Solution::RetrieveFromSolverSession* (page 502), *GMP::Solution::SendToModel* (page 503), *GMP::Solution::RetrieveFromModel* (page 501), *GMP::Solution::SendToSolverSession* (page 505), *GMP::SolverSession::GenerateBranchLowerBound* (page 533), *GMP::SolverSession::GenerateBranchUpperBound* (page 534), *GMP::SolverSession::GenerateBranchRow* (page 534), *GMP::SolverSession::GetNumberOfBranchNodes* (page 549), *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414) and *GMP::Instance::SetCallbackIncumbent* (page 415).

GMP::Instance::SetCallbackCandidate

The procedure *GMP::Instance::SetCallbackCandidate* (page 413) installs a callback procedure that is called every time an incumbent solution is found during the solution process of a MIP model. By using the procedure *GMP::SolverSession::RejectIncumbent* (page 554) the incumbent solution can be rejected. If *GMP::SolverSession::RejectIncumbent* (page 554) is not called inside the CallbackCandidate callback procedure then the incumbent solution will be accepted and replace the best incumbent solution found by so far.

```
GMP::Instance::SetCallbackCandidate(
    GMP,            ! (input) a generated mathematical program
    callback        ! (input) an AIMMS procedure
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The CallbackCandidate callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.

  If the return value is 0 (i.e., interrupt the solution process) then the incumbent solution will not be accepted!
- To remove the callback the empty element should be used as the *callback* argument.

- If an **incumbent** callback procedure is installed by using the procedure *GMP::Instance::SetCallbackIncumbent* (page 415), then that callback will be called after the **candidate** callback procedure if the incumbent solution is not rejected inside the **candidate** callback.
- A `CallbackCandidate` callback procedure will only be called when solving mixed integer programs with CPLEX.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415) and *GMP::SolverSession::RejectIncumbent* (page 554).

GMP::Instance::SetCallbackHeuristic

The procedure *GMP::Instance::SetCallbackHeuristic* (page 414) installs a callback procedure that is called during the solution process of a MIP model every time the subproblem has been solved to optimality.

```
GMP::Instance::SetCallbackHeuristic(
    GMP,            ! (input) a generated mathematical program
    callback        ! (input) an AIMMS procedure
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This callback is not called when the subproblem is infeasible or cut off.
- The callback should supply the solver with a heuristically-derived integer solution.
- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackHeuristic` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.
- A `CallbackHeuristic` callback procedure will only be called when solving mixed integer programs with CPLEX, Gurobi or ODH-CPLEX.

See also:

The routines *GMP::Solution::RetrieveFromSolverSession* (page 502), *GMP::Solution::SendToModel* (page 503), *GMP::Solution::RetrieveFromModel* (page 501), *GMP::Solution::SendToSolverSession* (page 505), *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413) and *GMP::Instance::SetCallbackIncumbent* (page 415).

GMP::Instance::SetCallbackIncumbent

The procedure *GMP::Instance::SetCallbackIncumbent* (page 415) installs a callback procedure that is called every time a new incumbent solution is found during the solution process of a MIP model.

```
GMP::Instance::SetCallbackIncumbent(
    GMP,            ! (input) a generated mathematical program
    callback        ! (input) an AIMMS procedure
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackIncumbent` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.
- This procedure is not supported by COPT.
- The functionality of the procedure *GMP::Instance::SetCallbackIncumbent* (page 415) has been changed between AIMMS versions 4.68 and 4.69. In AIMMS version 4.68 and older this procedure was named `GMP::Instance::SetCallbackNewIncumbent`. That procedure has become deprecated. AIMMS version 4.68 and older already contained a procedure that was named `GMP::Instance::SetCallbackIncumbent` but that procedure has been renamed to *GMP::Instance::SetCallbackCandidate* (page 413).

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIterations* (page 416), *GMP::Instance::SetCallbackStatusChange* (page 417) and *GMP::Instance::SetCallbackTime* (page 417).

GMP::Instance::SetCallbackIterations

The procedure *GMP::Instance::SetCallbackIterations* (page 416) installs a callback procedure that is called after a specified number of iterations.

```
GMP::Instance::SetCallbackIterations(
     GMP,            ! (input) a generated mathematical program
     callback,       ! (input) an AIMMS procedure
     [value]         ! (optional) number of iterations
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

value A scalar value indicating after which number of iterations the callback procedure should be called. The default value is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackIterations` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.
- The number of iterations can also be set by the `CallbackIterations` suffix of the symbolic mathematical program, but will be overruled if the *value* is not equal to 0.
- During a MIP solve, the iterations callback will be called irregularly by CPLEX, Gurobi and ODH-CPLEX (especially during the MIP phase).
- The iterations callback will be called less often if CPLEX uses dynamic search as the MIP Search Strategy instead of branch-and-cut.

- This procedure is not supported by COPT.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415), *GMP::Instance::SetCallbackStatusChange* (page 417) and *GMP::Instance::SetCallbackTime* (page 417).

GMP::Instance::SetCallbackStatusChange

The procedure *GMP::Instance::SetCallbackStatusChange* (page 417) installs a callback procedure that is called every time the status changes during the solution process.

```
GMP::Instance::SetCallbackStatusChange(
    GMP,            ! (input) a generated mathematical program
    callback        ! (input) an AIMMS procedure
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackStatusChange` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.
- This procedure is not supported by COPT.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415), *GMP::Instance::SetCallbackIterations* (page 416) and *GMP::Instance::SetCallbackTime* (page 417).

GMP::Instance::SetCallbackTime

The procedure *GMP::Instance::SetCallbackTime* (page 417) installs a callback procedure that is called after a specified number of (elapsed) seconds. By default this callback procedure is called every two seconds.

```
GMP::Instance::SetCallbackTime(
     GMP,            ! (input) a generated mathematical program
     callback        ! (input) an AIMMS procedure
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

callback A reference to a procedure in the set *AllIdentifiers* (page 821).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set *AllSolverSessions* (page 828).
- The `CallbackTime` callback procedure should have a return value of
  - 0, if you want the solution process to stop, or
  - 1, if you want the solution process to continue.
- To remove the callback the empty element should be used as the *callback* argument.
- The `CallbackTime` callback procedure is supported by CPLEX, Gurobi, CBC, COPT, ODH-CPLEX, XA, CP Optimizer, CONOPT, Knitro, SNOPT and IPOPT.
- The number of (elapsed) seconds is determined by the general solvers option `Progress Time Interval`. This option also specifies the interval for updating the Progress Window during a solve. As a consequence, the information passed to this callback procedure will be the same as the information displayed in the Progress Window (except for small differences for the solving time).
- The time callback will be called less often if CPLEX uses dynamic search as the MIP Search Strategy instead of branch-and-cut. In that case the interval between two successive calls might sometimes be larger than the interval as specified by the option `Progress Time Interval`.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415) and *GMP::Instance::SetCallbackStatusChange* (page 417).

GMP::Instance::SetCutoff

The procedure *GMP::Instance::SetCutoff* (page 418) specifies a cutoff value that is used during the solution process of the generated mathematical program.

```
GMP::Instance::SetCutoff(
     GMP,             ! (input) a generated mathematical program
     cutoff           ! (input) scalar numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

cutoff A scalar value.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: This procedure is only used for MIP models.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::Instance::SetIterationLimit* (page 420), `GMP::Instance::GMP::Instance::SetMemoryLimit` and *GMP::Instance::SetTimeLimit* (page 424).

GMP::Instance::SetDirection

The procedure *GMP::Instance::SetDirection* (page 419) changes the direction of a generated mathematical program.

```
GMP::Instance::SetDirection(
     GMP,              ! (input) a generated mathematical program
     direction         ! (input) an optimization direction
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

direction An element expression in the set *AllMatrixManipulationDirections* (page 802).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Instance::GetDirection* (page 393).

GMP::Instance::SetIterationLimit

The procedure *GMP::Instance::SetIterationLimit* (page 420) limits the number of iterations that can be used to solve a generated mathematical program.

```
GMP::Instance::SetIterationLimit(
     GMP,              ! (input) a generated mathematical program
     iterations        ! (input) number of iterations
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

iterations Maximum number of iterations allowed to solve the generated mathematical program.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::Instance::SetCutoff* (page 418), *GMP::Instance::SetMemoryLimit* (page 421) and *GMP::Instance::SetTimeLimit* (page 424).

GMP::Instance::SetMathematicalProgrammingType

The procedure *GMP::Instance::SetMathematicalProgrammingType* (page 420) changes the type of a generated mathematical program from MIP into RMIP (or vice versa), or from MINLP to RMINLP (or vice versa). Also the type can be changed from MIQP or MIQCP to RMINLP, or from MIP or LS to LP, but not vice versa.

```
GMP::Instance::SetMathematicalProgrammingType(
     GMP,                          ! (input) a generated mathematical program
     MathematicalProgrammingType   ! (input) a model type
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

MathematicalProgrammingType One of the elements LP, MIP, RMIP, MINLP or RMINLP (in the set *AllMatrixManipulationProgrammingTypes* (page 803)).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Instance::GetMathematicalProgrammingType* (page 395).

GMP::Instance::SetMemoryLimit

The procedure *GMP::Instance::SetMemoryLimit* (page 421) limits the amount of memory available to solve a generated mathematical program.

```
GMP::Instance::SetMemoryLimit(
     GMP,              ! (input) a generated mathematical program
     memory            ! (input) amount of memory
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

memory Maximum number of megabytes available to solve the generated mathematical program.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Setting the memory limit using this procedure has the same effect as using the Solvers General option `Solver workspace`.
- This procedure has no effect on CPLEX. For CPLEX the option `Tree Memory Limit` should be used instead.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::Instance::SetCutoff* (page 418), *GMP::Instance::SetIterationLimit* (page 420) and *GMP::Instance::SetTimeLimit* (page 424).

GMP::Instance::SetOptionValue

The procedure *GMP::Instance::SetOptionValue* (page 421) sets the value of a solver specific option corresponding to a generated mathematical program.

This procedure can also be used to set certain Solvers General options (see below).

```
GMP::Instance::SetOptionValue(
     GMP,            ! (input) a generated mathematical program
     OptionName,     ! (input) a scalar string expression
     Value           ! (input) a scalar numeric expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

OptionName A string expression holding the name of the option.

Value A scalar numeric expression representing the new value to be assigned to the option.

Return Value

The procedure returns 1 if the option exists and the value can be assigned to the option, or 0 otherwise.

Note:

- All solvers solving the generated mathematical program will use the option value as set by this procedure (provided that the solver contains the option).
- This procedure will be overruled by the procedure *GMP::SolverSession::SetOptionValue* (page 555) in case a solver session is used to solve the generated mathematical program.
- Options for which strings are displayed in the AIMMS **Options** dialog box, are also represented by numerical (integer) values. To obtain the corresponding option keywords, you can use the functions `OptionGetString` and `OptionGetKeywords`.
- This procedure can also be used to set the following Solvers General options:
  - Cutoff
  - Iteration limit
  - Maximal number of domain errors
  - Maximal number of integer solutions
  - MIP absolute optimality tolerance
  - MIP relative optimality tolerance
  - Solver workspace
  - Time limit

See also:

The routines *GMP::Instance::GetOptionValue* (page 404), *GMP::SolverSession::GetOptionValue* (page 550), *GMP::SolverSession::SetOptionValue* (page 555), *OptionGetString* (page 742) and *OptionGetKeywords* (page 742).

GMP::Instance::SetSolver

The procedure *GMP::Instance::SetSolver* (page 423) can be used to select for a generated mathematical program the solver to be called in subsequent calls to *GMP::Instance::Solve* (page 424).

```
GMP::Instance::SetSolver(
     GMP,            ! (input) a generated mathematical program
     solver          ! (input) a solver
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solver An element in the set *AllSolvers* (page 787).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: The solver set in this procedure will also be assigned to any solver session created with the function *GMP::Instance::CreateSolverSession* (page 378) for the *GMP*, unless the *Solver* argument in the procedure *GMP::Instance::CreateSolverSession* (page 378) is specified. Note that the procedure *GMP::Instance::SetSolver* (page 423) cannot be used to change the solver assigned to a solver session after *GMP::Instance::CreateSolverSession* (page 378) has been called.

See also:

The routines *GMP::Instance::CreateSolverSession* (page 378), *GMP::Instance::Generate* (page 384), *GMP::Instance::GetSolver* (page 406) and *GMP::Instance::Solve* (page 424).

GMP::Instance::SetStartingPointSelection

The procedure *GMP::Instance::SetStartingPointSelection* (page 423) specifies a selection of columns for which an initial value is given. This selection is only used for mathematical programs of type COP and CSP.

```
GMP::Instance::SetStartingPointSelection(
     GMP,            ! (input) a generated mathematical program
     colSet          ! (input) a subset of Integers
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392) and *GMP::Instance::Solve* (page 424).

GMP::Instance::SetTimeLimit

The procedure *GMP::Instance::SetTimeLimit* (page 424) limits the elapsed time to solve a generated mathematical program.

```
GMP::Instance::SetTimeLimit(
     GMP,              ! (input) a generated mathematical program
     seconds           ! (input) number of seconds
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

seconds Maximum number of seconds available to solve the generated mathematical program.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::Solve* (page 424), *GMP::Instance::SetCutoff* (page 418), *GMP::Instance::SetIterationLimit* (page 420) and *GMP::Instance::SetMemoryLimit* (page 421).

GMP::Instance::Solve

The procedure *GMP::Instance::Solve* (page 424) starts up a solver session to solve a generated mathematical program. In addition, it copies the initial solution from the model identifiers via solution 1 in the solution repository and stores the final solution via solution 1 back in the model identifiers.

```
GMP::Instance::Solve(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: The procedure *GMP::Instance::Solve* (page 424) automatically creates a solver session with the same name as the generated mathematical program in the set *AllSolverSessions* (page 828).

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::CreateSolverSession* (page 378), *GMP::Solution::RetrieveFromModel* (page 501), *GMP::Solution::SendToSolverSession* (page 505), *GMP::SolverSession::Execute* (page 530), *GMP::Solution::RetrieveFromSolverSession* (page 502) and *GMP::Solution::SendToModel* (page 503).

2.3.6 GMP::Linearization Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing linearizations associated with a generated mathematical program instance:

GMP::Linearization::Add

The procedure *GMP::Linearization::Add* (page 425) adds a linearization row to a generated mathematical program (*GMP1*) with respect to a solution (column level values and row marginals) of a second generated mathematical program (*GMP2*) for each row in a set of nonlinear constraints of that second generated mathematical program.

```
GMP::Linearization::Add(
     GMP1,                  ! (input) a generated mathematical program
     GMP2,                  ! (input) a generated mathematical program
     solution,              ! (input) a solution
     constraintSet,         ! (input) a set of nonlinear constraints
     deviationsPermitted,   ! (input) a binary parameter
     penaltyMultipliers,    ! (input) a double parameter
     linNo,                 ! (input) a linearization number
```

(continues on next page)

(continued from previous page)

```
    [jacTol]              ! (optional) the Jacobian tolerance
    )
```

Arguments

GMP1 An element in *AllGeneratedMathematicalPrograms* (page 833).

GMP2 An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution in the solution repository of *GMP2*.

constraintSet A subset of *AllNonLinearConstraints* (page 824).

deviationsPermitted A binary parameter over *AllNonLinearConstraints* (page 824) indicating whether deviations are permitted for these linearizations. If yes, a new column will also be added to *GMP1* with an objective coefficient equal to the double value given in *penaltyMultiplier* multiplied with the row marginal of the row in *solution*.

penaltyMultipliers A double parameter over *AllNonLinearConstraints* (page 824) representing the penalty multiplier that will be used if *deviationsPermitted* indicates that a deviation is permitted for the linearization.

linNo An integer scalar reference to the rows and columns of the linearization.

jacTol The Jacobian tolerance; if the Jacobian value (in absolute sense) of a column in a nonlinear row is smaller than this value, the column will not be added to the linearization of that row. The default is 1e-5.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure fails if one of the constraints is ranged.
- Rows and columns added before for *linNo* will be deleted first.
- This procedure will fail if *GMP2* contains a column that is not part of *GMP1*. A column that is part of *GMP1* but not of *GMP2* will be ignored, i.e., no coefficient for that column will be added to the linearizations.
- The formula for the linearization of a scalar nonlinear inequality $g(x, y) \leq b_j$ around the point $(x, y) = (x^0, y^0)$ is as follows.

$$g(x^0, y^0) + \bigtriangledown g(x^0, y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} - z_j \leq b_j$$

  where $z_j \geq 0$ is the extra column that is added if a deviation is permitted.
- For a scalar nonlinear equality $g(x, y) = b_j$ the sense of the linearization depends on the shadow price of the equality in the *solution*. The sense will be '$\leq$' if the shadow price is negative and the optimization direction is minimization, or if the shadow price is positive and the optimization direction is maximization. The sense will be '$\geq$' if the shadow price is positive and the optimization direction is minimization, or if the shadow price is negative and the optimization direction is maximization.
- By using the suffixes `.ExtendedConstraint` and `.ExtendedVariable` it is possible to refer to the rows and columns that are added by *GMP::Linearization::Add* (page 425):

- Constraint `c.ExtendedConstraint('Linearization`k`',j)` is added for each nonlinear constraint `c(j)`.
- Variable `c.ExtendedVariable('Linearization`k`',j)` is added for each nonlinear constraint `c(j)` if a deviation is permitted for constraint `c(j)`.

Here k denotes the value of the argument *linNo*.

See also:

The routines *GMP::Linearization::AddSingle* (page 427) and *GMP::Linearization::Delete* (page 429). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Linearization::AddSingle

The procedure *GMP::Linearization::AddSingle* (page 427) adds a single linearization row to a generated mathematical program (*GMP1*) with respect to a solution (column level values and row marginals) of a second generated mathematical program (*GMP2*).

```
GMP::Linearization::AddSingle(
     GMP1,                  ! (input) a generated mathematical program
     GMP2,                  ! (input) a generated mathematical program
     solution,              ! (input) a solution
     row,                   ! (input) a scalar reference or row number
     deviationPermitted,    ! (input) a binary scalar
     penaltyMultiplier,     ! (input) a double scalar
     linNo,                 ! (input) a linearization number
     [jacTol]               ! (optional) the Jacobian tolerance
     )
```

Arguments

GMP1 An element in *AllGeneratedMathematicalPrograms* (page 833).

GMP2 An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution in the solution repository of *GMP2*.

row A scalar reference to an existing nonlinear row in the matrix of *GMP2* or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix of *GMP2*.

deviationPermitted A binary scalar indicating whether a deviation is permitted for this linearization. If yes, a new column will also be added to *GMP1* with an objective coefficient equal to the double value given in *penaltyMultiplier* multiplied with the row marginal of the row in *solution*.

penaltyMultiplier A double value representing the penalty multiplier that will be used if *deviationPermitted* indicates that a deviation is permitted for the linearization.

linNo An integer scalar reference to the rows and columns of the linearization.

jacTol The Jacobian tolerance; if the Jacobian value (in absolute sense) of a column in *row* is smaller than this value, the column will not be added to the linearization. The default is 1e-5.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure fails if the row is ranged.
- Rows and columns added before for *linNo* will be deleted first.
- This procedure will fail if *GMP2* contains a column that is not part of *GMP1*. A column that is part of *GMP1* but not of *GMP2* will be ignored, i.e., no coefficient for that column will be added to the linearizations.
- The formula for the linearization of a scalar nonlinear inequality $g(x, y) \leq b_j$ around the point $(x, y) = (x^0, y^0)$ is as follows:

$$g(x^0, y^0) + \bigtriangledown g(x^0, y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} - z_j \leq b_j$$

  where $z_j \geq 0$ is the extra column that is added if a deviation is permitted.
- For a scalar nonlinear equality $g(x, y) = b_j$ the sense of the linearization depends on the shadow price of the equality in the *solution*. The sense will be '$\leq$' if the shadow price is negative and the optimization direction is minimization, or if the shadow price is positive and the optimization direction is maximization. The sense will be '$\geq$' if the shadow price is positive and the optimization direction is minimization, or if the shadow price is negative and the optimization direction is maximization.
- By using the suffixes `.ExtendedConstraint` and `.ExtendedVariable` it is possible to refer to the row and column that are added by *GMP::Linearization::AddSingle* (page 427):
  - Constraint `c.ExtendedConstraint('Linearization`*k*`',j)` is added for row `c(j)`.
  - Variable `c.ExtendedVariable('Linearization`*k*`',j)` is added for row `c(j)` if a deviation is permitted.

  Here k denotes the value of the argument *linNo*.

Example

Assume that 'prod03' is a mathematical program with the following declaration (in aim format):

```
Variable i1 {
    Range      :  {
        {1..5}
    }
}
Variable i2 {
    Range      :  {
        {1..5}
    }
}
Variable objvar;
Constraint e1 {
    Definition :  - 3*i1 - 2*i2 + objvar = 0;
}
Constraint e2 {
```

(continues on next page)

(continued from previous page)

```
    Definition :  - i1*i2 <= -3.5;
}
MathematicalProgram prod03 {
    Objective  :  objvar;
    Direction  :  minimize;
    Type       :  MINLP;
}
```

Assume that AIMMS has executed the following code in which a mathematical program instance 'gmp1' is generated from 'prod03', its integer variables are relaxed, and it is solved.

```
gmp1 := GMP::Instance::Generate(prod03);
GMP::Instance::SetMathematicalProgrammingType(gmp1,'RMINLP');
GMP::Instance::Solve(gmp1);
```

The optimal solution is i1 $= 1.528$ and i2 $= 2.291$, with Jacobian values -2.291 and -1.528 for i1 and i2 respectively. This solution is stored at position 1 in the solution repository of 'gmp1'. If we have a second generated mathematical program 'gmp2' with the same variables as 'gmp1' then

```
GMP::Linearization::AddSingle(gmp2,gmp1,1,e2,0,0,1);
```

will add a row

```
       e2.ExtendedConstraint('Linearization1'):
          - 2.291 * i1 - 1.528 * i2 <= -7 ;

to 'gmp2'.
```

See also:

The routines *GMP::Linearization::Add* (page 425) and *GMP::Linearization::Delete* (page 429). See Modifying an Extended Math Program Instance of the Language Reference for more details on extended suffixes.

GMP::Linearization::Delete

The procedure *GMP::Linearization::Delete* (page 429) deletes a set of rows and columns corresponding to a linearization in a generated mathematical program.

```
GMP::Linearization::Delete(
     GMP,          ! (input) a generated mathematical program
     linNo         ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Linearization::Add* (page 425) and *GMP::Linearization::AddSingle* (page 427).

GMP::Linearization::GetDeviation

The function *GMP::Linearization::GetDeviation* (page 430) returns the deviation of a linearization of a row in a generated mathematical program.

```
GMP::Linearization::GetDeviation(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo    ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

The function returns the deviation of the row.

See also:

The routines *GMP::Linearization::SetDeviationBound* (page 434) and *GMP::Linearization::GetDeviationBound* (page 430).

GMP::Linearization::GetDeviationBound

The function *GMP::Linearization::GetDeviationBound* (page 430) returns the deviation bound of a linearization of a row in a generated mathematical program. The lower bound of the extra column generated for the linearization is always 0; this function returns the upper bound.

```
GMP::Linearization::GetDeviationBound(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo    ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

The function returns the deviation upperbound of a linearization.

See also:

The routines *GMP::Linearization::SetDeviationBound* (page 434) and *GMP::Linearization::GetDeviation* (page 430).

GMP::Linearization::GetLagrangeMultiplier

The function *GMP::Linearization::GetLagrangeMultiplier* (page 431) returns the Lagrange multiplier used when adding the linearization of a row to a generated mathematical program. (In other words, the marginal value of the row that was used when the linearization was added.)

```
GMP::Linearization::GetLagrangeMultiplier(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo    ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

The function returns the Lagrange multiplier used when adding the linearization of a row.

See also:

The procedure *GMP::Linearization::Add* (page 425).

GMP::Linearization::GetType

The function *GMP::Linearization::GetType* (page 432) returns the row type of a linearization of a row in a generated mathematical program.

```
GMP::Linearization::GetType(
     GMP,    ! (input) a generated mathematical program
     row,    ! (input) a scalar reference or row number
     linNo   ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

An element in the set *AllRowTypes* (page 805).

See also:

The procedure *GMP::Linearization::SetType* (page 434).

GMP::Linearization::GetWeight

The function *GMP::Linearization::GetWeight* (page 432) returns the weight of a linearization of a row in a generated mathematical program. The weight of a linearization is defined as the objective coefficient of the column that was added to the generated mathematical program when the linearization was added and if a deviation was permitted.

```
GMP::Linearization::GetWeight(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo    ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

The function returns the weight of the linearization.

Note:

- This function returns 0 if no extra column was added for the linearization.
- If the objective coefficient of the deviation column (if any) was not changed, the weight equals the penalty multiplier multiplied with the marginal value of the row that was used when the linearization was added with *GMP::Linearization::Add* (page 425) or *GMP::Linearization::AddSingle* (page 427).

See also:

The procedures *GMP::Linearization::Add* (page 425), *GMP::Linearization::AddSingle* (page 427) and *GMP::Linearization::SetWeight* (page 435).

GMP::Linearization::RemoveDeviation

The procedure *GMP::Linearization::RemoveDeviation* (page 433) removes the deviation of a linearization of a row in a generated mathematical program. That is, it deletes the extra column created (if any) when adding the linearization of the row to the generated mathematical program.

```
GMP::Linearization::RemoveDeviation(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo    ! (input) a linearization number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Linearization::GetDeviation* (page 430), *GMP::Linearization::Add* (page 425) and *GMP::Linearization::AddSingle* (page 427).

GMP::Linearization::SetDeviationBound

The procedure *GMP::Linearization::SetDeviationBound* (page 434) sets the deviation bound of a linearization of a row in a generated mathematical program. The lower bound of the extra column generated for the linearization is always 0; this procedure sets the upper bound.

```
GMP::Linearization::SetDeviationBound(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo,   ! (input) a linearization number
     value    ! (input) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

value A scalar value representing the deviaton upper bound of the row.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Linearization::GetDeviationBound* (page 430), *GMP::Linearization::GetDeviation* (page 430) and *GMP::Linearization::RemoveDeviation* (page 433).

GMP::Linearization::SetType

The procedure *GMP::Linearization::SetType* (page 434) sets the row type of linearization of a row in a generated mathematical program.

```
GMP::Linearization::SetType(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo,   ! (input) a linearization number
     rowtype ! (input) a row type
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

rowtype An element (or element parameter or element valued expression) in the predeclared set *AllRowTypes* (page 805).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The function *GMP::Linearization::GetType* (page 432).

GMP::Linearization::SetWeight

The procedure *GMP::Linearization::SetWeight* (page 435) sets the weight of a linearization of a row in a generated mathematical program. The weight of a linearization is defined as the objective coefficient of the column that was added to the generated mathematical program when the linearization was added and if a deviation was permitted.

```
GMP::Linearization::SetWeight(
     GMP,     ! (input) a generated mathematical program
     row,     ! (input) a scalar reference or row number
     linNo,   ! (input) a linearization number
     value    ! (input) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing nonlinear row in the matrix or the number of that row in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

linNo An integer scalar reference to the rows and columns of the linearization.

value A scalar value representing the new weight of the row.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The function *GMP::Linearization::GetWeight* (page 432).

2.3.7 GMP::ProgressWindow Procedures and Functions

AIMMS supports the following procedures and functions for displaying progress information in the Progress Window:

GMP::ProgressWindow::DeleteCategory

The procedure *GMP::ProgressWindow::DeleteCategory* (page 436) deletes a progress category.

```
GMP::ProgressWindow::DeleteCategory(
     Category          ! (input) a progress category
     )
```

Arguments

Category An element in the set *AllProgressCategories* (page 834).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::CreateProgressCategory* (page 377) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::DisplayLine

The procedure *GMP::ProgressWindow::DisplayLine* (page 436) writes one line with progress information in the Progress Window. The *lineNo* argument gives the number of the line in which the information has to be shown. The title contains a string that will be displayed on the left side of the line; the value will be displayed on the right side.

```
GMP::ProgressWindow::DisplayLine(
     lineNo,           ! (input) a line number
     title,            ! (input) a title
     value,            ! (input) a value
     [Category]        ! (optional) a progress category
     )
```

Arguments

lineNo The number of the line in which the information has to be shown. Its value should be a number between 1 and the maximum number of lines available in the Progress Window (currently 6).

title The string that will be displayed on the left side of the line.

value The value that will be displayed on the right side of the line.

Category An element in the set *AllProgressCategories* (page 834).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the *Category* argument is used then the element should be created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).
- To freeze (lock) a line the procedure *GMP::ProgressWindow::FreezeLine* (page 440) should be called. To unfreeze it thereafter the procedure *GMP::ProgressWindow::UnfreezeLine* (page 441) should be called.

See also:

The routines *GMP::ProgressWindow::DisplaySolverStatus* (page 439), *GMP::ProgressWindow::DisplayProgramStatus* (page 437), *GMP::ProgressWindow::DisplaySolver* (page 438), *GMP::ProgressWindow::FreezeLine* (page 440), *GMP::ProgressWindow::UnfreezeLine* (page 441) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::DisplayProgramStatus

The procedure *GMP::ProgressWindow::DisplayProgramStatus* (page 437) writes the program status (or model status) to the Progress Window.

```
GMP::ProgressWindow::DisplayProgramStatus(
     status,           ! (input) a status
     [Category],       ! (optional) a progress category
     [lineNo]          ! (optional) a line number
     )
```

Arguments

status An element in the set *AllSolutionStates* (page 807).

Category An element in the set *AllProgressCategories* (page 834).

lineNo The number of the line in which the program status has to be displayed. The default is 7.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the *Category* argument is used then the element should be created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).
- The program status can also be displayed by using the procedure *GMP::ProgressWindow::DisplayLine* (page 436) with title 'Program Status'.

See also:

The routines *GMP::Solution::GetProgramStatus* (page 494), *GMP::ProgressWindow::DisplayLine* (page 436), *GMP::ProgressWindow::DisplaySolverStatus* (page 439) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::DisplaySolver

The procedure *GMP::ProgressWindow::DisplaySolver* (page 438) writes the solver name to the Progress Window.

```
GMP::ProgressWindow::DisplaySolver(
     name,             ! (input) a solver name
     [Category]        ! (optional) a progress category
     )
```

Arguments

name A scalar string representing the solver name.

Category An element in the set *AllProgressCategories* (page 834).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: If the *Category* argument is used then the element should be created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).

See also:

The routines *GMP::ProgressWindow::DisplaySolverStatus* (page 439), *GMP::ProgressWindow::DisplayProgramStatus* (page 437), *GMP::ProgressWindow::DisplayLine* (page 436) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::DisplaySolverStatus

The procedure *GMP::ProgressWindow::DisplaySolverStatus* (page 439) writes the solver status to the Progress Window.

```
GMP::ProgressWindow::DisplaySolverStatus(
     status,           ! (input) a status
     [Category],       ! (optional) a progress category
     [lineNo]          ! (optional) a line number
     )
```

Arguments

status An element in the set *AllSolutionStates* (page 807).

Category An element in the set *AllProgressCategories* (page 834).

lineNo The number of the line in which the solver status has to be displayed. The default is 8.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the *Category* argument is used then the element should be created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).
- The solver status can also be displayed by using the procedure *GMP::ProgressWindow::DisplayLine* (page 436) with title 'Solver Status'.

See also:

The routines *GMP::Solution::GetSolverStatus* (page 496), *GMP::ProgressWindow::DisplayLine* (page 436), *GMP::ProgressWindow::DisplayProgramStatus* (page 437) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::FreezeLine

The procedure *GMP::ProgressWindow::FreezeLine* (page 440) freezes (or locks) a line in the Progress Window.

```
GMP::ProgressWindow::FreezeLine(
     lineNo,           ! (input) a line number
     [totalFreeze],    ! (optional) a binary
     [Category]        ! (optional) a progress category
     )
```

Arguments

lineNo The number of the line that should be frozen.

totalFreeze If it equals 1 (the default) then the line will never change (untill the procedure *GMP::ProgressWindow::UnfreezeLine* (page 441) is called). If it equals 0 then the line will only change if a `GMP::ProgressWindow` procedure is called for this line.

Category An element in the set *AllProgressCategories* (page 834).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the *Category* argument is used then the element should be created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).
- If the *Category* argument is not specified then this procedure will freeze a line in the general AIMMS progress category for displaying solver progress, or in the solver progress category of the generated mathematical program in case function *GMP::Instance::CreateProgressCategory* (page 377) was called.

See also:

The procedures *GMP::Instance::CreateProgressCategory* (page 377), *GMP::ProgressWindow::DisplayLine* (page 436), *GMP::ProgressWindow::DisplayProgramStatus* (page 437), *GMP::ProgressWindow::DisplaySolverStatus* (page 439), *GMP::ProgressWindow::UnfreezeLine* (page 441) and *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::Transfer

The procedure *GMP::ProgressWindow::Transfer* (page 440) transfers a progress category that was created for a solver session to another solver session. This procedure allows you to share a progress category among several solver sessions.

```
GMP::ProgressWindow::Transfer(
     Category,          ! (input) a progress category
     solverSession      ! (input) a solver session
     )
```

Arguments

Category An element in the set *AllProgressCategories* (page 834).

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *Category* should have been created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).
- The *solverSession* argument specifies the solver session to which the progress category should be transfered.

Example

In the example below we create two GMPs and for each GMP a solver session. Next we create a progress category for the first solver session. After executing the first solver session we transfer the progress category to the second solver session. By transfering the progress category we ensure that both solver sessions use the same area in the progress window.

```
myGMP1 := GMP::Instance::Generated( MP1 );
session1 := GMP::Instance::CreateSolverSession( myGMP1 );

myGMP2 := GMP::Instance::Generated( MP2 );
session2 := GMP::Instance::CreateSolverSession( myGMP2 );

pc := GMP::SolverSession::CreateProgressCategory( session1 );

GMP::SolverSession::Execute( session1 );

GMP::ProgressWindow::Transfer( pc, session2 );

GMP::SolverSession::Execute( session2 );
```

See also:

The procedure *GMP::SolverSession::CreateProgressCategory* (page 529).

GMP::ProgressWindow::UnfreezeLine

The procedure *GMP::ProgressWindow::UnfreezeLine* (page 441) unlocks a frozen line in the Progress Window.

```
GMP::ProgressWindow::UnfreezeLine(
     lineNo,           ! (input) a line number
     [Category]        ! (optional) a progress category
     )
```

Arguments

lineNo The number of the line that should be freed.

Category An element in the set *AllProgressCategories* (page 834).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the *Category* argument is used then the element should be created with the function *GMP::SolverSession::CreateProgressCategory* (page 529).
- If the *Category* argument is not specified then this procedure will unfreeze a line in the general AIMMS progress category for displaying solver progress, or in the solver progress category of the generated mathematical program in case function *GMP::Instance::CreateProgressCategory* (page 377) was called.

See also:

The procedures *GMP::Instance::CreateProgressCategory* (page 377), *GMP::ProgressWindow::DisplayLine* (page 436), *GMP::ProgressWindow::FreezeLine* (page 440) and *GMP::SolverSession::CreateProgressCategory* (page 529).

2.3.8 GMP::QuadraticCoefficient Procedures and Functions

AIMMS supports the following procedures and functions for modifying the quadratic coefficients in the matrix associated with a generated mathematical program instance:

GMP::QuadraticCoefficient::Get

The function *GMP::QuadraticCoefficient::Get* (page 442) retrieves a quadratic coefficient in a quadratic row of a generated mathematical program.

```
GMP::QuadraticCoefficient::Get(
     GMP,              ! (input) a generated mathematical program
     row,              ! (input) a scalar reference
     column1,          ! (input) a scalar reference
     column2           ! (input) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix.

column1 A scalar reference to an existing column in the matrix.

column2 A scalar reference to an existing column in the matrix.

Return Value

The value of the specified quadratic coefficient in the quadratic row.

Note: If *column1* equals *column2* then AIMMS multiplies the quadratic coefficient by 2 before it is returned by this function.

See also:

The routines *GMP::QuadraticCoefficient::Set* (page 443), *GMP::Coefficient::GetQuadratic* (page 315) and *GMP::Coefficient::SetQuadratic* (page 318).

GMP::QuadraticCoefficient::Set

The procedure *GMP::QuadraticCoefficient::Set* (page 443) sets the value for a quadratic coefficient in a quadratic row of a generated mathematical program.

```
GMP::QuadraticCoefficient::Set(
     GMP,              ! (input) a generated mathematical program
     row,              ! (input) a scalar reference
     column1,          ! (input) a scalar reference
     column2,          ! (input) a scalar reference
     value             ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix.

column1 A scalar reference to an existing column in the matrix.

column2 A scalar reference to an existing column in the matrix.

value A scalar numerical value indicating the value for the coefficient.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: If *column1* equals *column2* then AIMMS multiplies the quadratic coefficient by 0.5 before it is stored (and passed to the solver).

See also:

The routines *GMP::QuadraticCoefficient::Get* (page 442), *GMP::Coefficient::GetQuadratic* (page 315) and *GMP::Coefficient::SetQuadratic* (page 318).

2.3.9 GMP::Robust Procedures and Functions

AIMMS supports the following procedures and functions related to robust optimization:

GMP::Robust::EvaluateAdjustableVariables

The procedure *GMP::Robust::EvaluateAdjustableVariables* (page 444) evaluates the values of a set of adjustable variables using the current values of the uncertain parameters inside the model.

```
GMP::Robust::EvaluateAdjustableVariables(
    GMP,              ! (input) a generated mathematical program
    Variables,        ! (input) a set of variables
    [merge]           ! (optional, default 0) a scalar value
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Variables A subset of *AllVariables* (page 831).

merge A scalar binary value to indicate whether the evaluated values for the adjustable variables should be merged with the existing values (value 1) or should replace them (value 0).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP* must have been created using the procedure *GMP::Instance::GenerateRobustCounterpart* (page 386).
- This procedure will ignore variables in the set *Variables* that are not part of the *GMP*. It will also ignore non-adjustable variables.
- The evaluated values will be stored in the `.robust` suffix of the adjustable variables. (Note that no values are stored inside this suffix after the robust counterpart is solved.)
- This procedure will fail if the option `Keep Uncertain Mathematical Program` was not switched on before calling procedure *GMP::Instance::GenerateRobustCounterpart* (page 386).

Example

Assume that `rcGMP` is a robust counterpart GMP with one adjustable variable `Production(i,t)` that depends on the uncertain parameter `Demand(s)`. After solving the robust counterpart you can calculate the values of `Production` for a certain realization of `Demand` as follows:

```
Demand(s) := 5;

GMP::Robust::EvaluateAdjustableVariables( rcGMP, AllVariables );
```

It is also possible to calculate the values of the adjustable variables without using this procedure:

```
Demand(s) := 5;

CalculatedProduction(i,t) := Production.adjustable.Constant(i,t) +
     sum( s, Demand(s) * Production.adjustable.Demand(i,t,s) );
```

Here `CalculatedProduction(i,t)` is a parameter used to store the calculated values of `Production(i,t)`.

See also:

The function *GMP::Instance::GenerateRobustCounterpart* (page 386).

2.3.10 GMP::Row Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing matrix rows associated with a generated mathematical program instance:

GMP::Row::Activate

The procedure *GMP::Row::Activate* (page 445) activates a deactivated row in a generated mathematical program.

```
GMP::Row::Activate(
     GMP,            ! (input) a generated mathematical program
     row             ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: Use *GMP::Row::ActivateMulti* (page 446) or *GMP::Row::ActivateRaw* (page 447) if many rows have to be activated, because that will be more efficient.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::ActivateMulti* (page 446), *GMP::Row::ActivateRaw* (page 447) and *GMP::Row::Deactivate* (page 449).

GMP::Row::ActivateMulti

The procedure *GMP::Row::ActivateMulti* (page 446) activates a group of deactivated rows, belonging to a constraint, in a generated mathematical program.

```
GMP::Row::ActivateMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     row             ! (input) a constraint expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

Return Value

The procedure returns 1 on success, and 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::Activate* (page 445) and *GMP::Row::Deactivate* (page 449).

GMP::Row::ActivateRaw

The procedure *GMP::Row::ActivateRaw* (page 447) activates a group of deactivated rows in a generated mathematical program.

```
GMP::Row::ActivateRaw(
     GMP,            ! (input) a generated mathematical program
     rowSet          ! (input) a subset of Integers
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Row::ActivateRaw* (page 447) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set ConstraintSet {
    SubsetOf: AllConstraints;
}
Set RowSet {
    SubsetOf: Integers;
    Index: rr;
}
```

To activate the constraint `c(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

ConstraintSet := { 'c' };
RowSet := GMP::Instance::GetRowNumbers( myGMP, ConstraintSet );

GMP::Row::ActivateRaw( myGMP, RowSet );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetRowNumbers* (page 405), *GMP::Row::Activate* (page 445) and *GMP::Row::DeactivateRaw* (page 451).

GMP::Row::Add

The procedure *GMP::Row::Add* (page 448) adds an empty row to the matrix of a generated mathematical program.

```
GMP::Row::Add(
     GMP,            ! (input) a generated mathematical program
     row             ! (input) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to a row.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use *GMP::Row::AddMulti* (page 448) if many rows corresponding to some constraint have to be added, because that will be more efficient.
- Coefficients for this row can be added to the matrix by using the procedure *GMP::Coefficient::Set* (page 316).
- The procedure *GMP::Row::Add* (page 448) sets the row type to '<=' and the right-hand-side value to INF. By using the procedures *GMP::Row::SetType* (page 476) and *GMP::Row::SetRightHandSide* (page 472) the row type and the right-hand-side value can be changed.
- Use procedure *GMP::Row::Generate* (page 455) to generate a (non-empty) row according to the definition of its associated symbolic constraint.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Coefficient::Set* (page 316), *GMP::Row::AddMulti* (page 448), *GMP::Row::Delete* (page 452), *GMP::Row::SetType* (page 476), *GMP::Row::SetRightHandSide* (page 472) and *GMP::Row::Generate* (page 455).

GMP::Row::AddMulti

The procedure *GMP::Row::AddMulti* (page 448) adds a group of empty rows, belonging to a constraint, to the matrix of a generated mathematical program.

```
GMP::Row::AddMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     row             ! (input) a constraint expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Coefficients for the rows can be added to the matrix by using the procedure *GMP::Coefficient::SetMulti* (page 317).
- The procedure *GMP::Row::AddMulti* (page 448) sets the row type to '<=' and the right-hand-side values to INF. By using the procedures *GMP::Row::SetTypeMulti* (page 477) and *GMP::Row::SetRightHandSideMulti* (page 474) the row type and the right-hand-side value can be changed.
- Use procedure *GMP::Row::GenerateMulti* (page 457) to generate (non-empty) rows according to the definition of the associated symbolic constraint.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Coefficient::SetMulti* (page 317), *GMP::Row::Add* (page 448), *GMP::Row::DeleteMulti* (page 453), *GMP::Row::SetTypeMulti* (page 477), *GMP::Row::SetRightHandSideMulti* (page 474) and *GMP::Row::GenerateMulti* (page 457).

GMP::Row::Deactivate

The procedure *GMP::Row::Deactivate* (page 449) deactivates a row in a generated mathematical program. A deactivated row will not be passed to a solver session.

```
GMP::Row::Deactivate(
     GMP,            ! (input) a generated mathematical program
     row             ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: Use *GMP::Row::DeactivateMulti* (page 450) or *GMP::Row::DeactivateRaw* (page 451) if many rows have to be deactivated, because that will be more efficient.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::Activate* (page 445), *GMP::Row::DeactivateMulti* (page 450) and *GMP::Row::DeactivateRaw* (page 451).

GMP::Row::DeactivateMulti

The procedure *GMP::Row::DeactivateMulti* (page 450) deactivates a group of rows, belonging to a constraint, in a generated mathematical program.

```
GMP::Row::DeactivateMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     row             ! (input) a constraint expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

Return Value

The procedure returns 1 on success, and 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::Activate* (page 445) and *GMP::Row::Deactivate* (page 449).

GMP::Row::DeactivateRaw

The procedure *GMP::Row::DeactivateRaw* (page 451) deactivates a group of rows in a generated mathematical program.

```
GMP::Row::DeactivateRaw(
     GMP,            ! (input) a generated mathematical program
     rowSet          ! (input) a subset of Integers
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Row::DeactivateRaw* (page 451) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set ConstraintSet {
    SubsetOf: AllConstraints;
}
Set RowSet {
```

(continues on next page)

(continued from previous page)

```
    SubsetOf: Integers;
    Index: rr;
}
```

To deactivate the constraint `c(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

ConstraintSet := { 'c' };
RowSet := GMP::Instance::GetRowNumbers( myGMP, ConstraintSet );

GMP::Row::DeactivateRaw( myGMP, RowSet );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetRowNumbers* (page 405), *GMP::Row::ActivateRaw* (page 447) and *GMP::Row::Deactivate* (page 449).

GMP::Row::Delete

The procedure *GMP::Row::Delete* (page 452) marks a row in a generated mathematical program as deleted.

```
GMP::Row::Delete(
     GMP,            ! (input) a generated mathematical program
     row             ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use *GMP::Row::DeleteMulti* (page 453) or *GMP::Row::DeleteMulti* (page 453) if many rows have to be deleted, because that will be more efficient.
- A deleted row remains present in the generated mathematical program but its content will not be copied to a solver session.
- The row will not be printed in the constraint listing, nor be visible in the Math Program Inspector and it will be removed from any solver maintained copies.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::Add* (page 448), *GMP::Row::DeleteMulti* (page 453) and *GMP::Row::DeleteRaw* (page 454).

GMP::Row::DeleteIndicatorCondition

The procedure *GMP::Row::DeleteIndicatorCondition* (page 453) deletes an indicator column and condition from a row in a generated mathematical program.

```
GMP::Row::DeleteIndicatorCondition(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: This procedure transforms an indicator row into a normal row.

See also:

The routines *GMP::Row::GetIndicatorColumn* (page 459), *GMP::Row::GetIndicatorCondition* (page 459) and *GMP::Row::SetIndicatorCondition* (page 467).

GMP::Row::DeleteMulti

The procedure *GMP::Row::DeleteMulti* (page 453) marks a group of rows in a generated mathematical program, belonging to a constraint, as deleted.

```
GMP::Row::DeleteMulti(
     GMP,             ! (input) a generated mathematical program
     binding,         ! (input) an index binding
     row              ! (input) a constraint expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Deleted rows remain present in the generated mathematical program but their content will not be copied to a solver session.
- The rows will not be printed in the constraint listing, nor be visible in the Math Program Inspector and they will be removed from any solver maintained copies.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::AddMulti* (page 448) and *GMP::Row::Delete* (page 452).

GMP::Row::DeleteRaw

The procedure *GMP::Row::DeleteRaw* (page 454) marks a group of rows in a generated mathematical program as deleted.

```
GMP::Row::DeleteRaw(
     GMP,            ! (input) a generated mathematical program
     rowSet          ! (input) a subset of Integers
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Deleted rows remain present in the generated mathematical program but their content will not be copied to a solver session.
- The rows will not be printed in the constraint listing, nor be visible in the Math Program Inspector and they will be removed from any solver maintained copies.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Row::DeleteRaw* (page 454) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set ConstraintSet {
    SubsetOf: AllConstraints;
}
Set RowSet {
    SubsetOf: Integers;
    Index: rr;
}
```

To delete the constraint `c(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

ConstraintSet := { 'c' };
RowSet := GMP::Instance::GetRowNumbers( myGMP, ConstraintSet );

GMP::Row::DeleteRaw( myGMP, RowSet );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetRowNumbers* (page 405), *GMP::Row::Add* (page 448) and *GMP::Row::Delete* (page 452).

GMP::Row::Generate

The procedure *GMP::Row::Generate* (page 455) generates a row and adds it to the matrix of a generated mathematical program. The row is generated according to the definition of its associated symbolic constraint, or to the definition of its associated symbolic variable in case the row refers to the definition of a variable.

```
GMP::Row::Generate(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference
     [autoAddColumn] ! (optional) a binary scalar
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to a row.

autoAddColumn A binary scalar indicating whether this procedure should automatically add columns that are not in the *GMP*. The default is 0 meaning that no columns are added.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Use *GMP::Row::GenerateMulti* (page 457) if many rows corresponding to some constraint have to be generated, because that will be more efficient.
- Before generating the row all existing matrix coefficients for this row are removed.
- The row type and the right-hand-side value (and, if the row type is `'ranged'`, the left-hand-side value) are set according to the constraint definition.
- This procedure cannot be used if the row contains the objective variable, and the row was added or generated before using a different coefficient for the objective variable.
- If the value of *autoAddColumn* equals 0, then this procedure will generate an error if it encounters a column that is not in the *GMP*. You then have to add that column before calling this procedure by using the procedure *GMP::Column::Add* (page 322).
- Setting the value of *autoAddColumn* to 1 should only be done if you know exactly which columns are automatically added by this procedure. Otherwise you might end up with a model in which some columns only appear in this row, possibly making this row redundant.
- This procedure will never add columns that were deleted before with the procedure *GMP::Column::Delete* (page 323).

Example

To generate the row corresponding to constraint `c(i)` for element `'1'`, we can use:

```
GMP::Row::Generate( myGMP, c('1') );
```

If the row refers to the definition of a variable then we have to place '_definition' behind the name of the variable. For example, if `v(j)` is a variable with a definition and we want to generate a row according to its definition for element `'2'` then we have to use:

```
GMP::Row::Generate( myGMP, v_definition('2') );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::Add* (page 322), *GMP::Column::Delete* (page 323), *GMP::Row::Add* (page 448), *GMP::Row::Delete* (page 452) and *GMP::Row::GenerateMulti* (page 457).

GMP::Row::GenerateMulti

The procedure *GMP::Row::GenerateMulti* (page 457) generates a group of rows, belonging to a constraint, and adds them to the matrix of a generated mathematical program. The rows are generated according to the definition of the associated symbolic constraint, or to the definition of the associated symbolic variable in case the rows refer to the definition of a variable.

```
GMP::Row::GenerateMulti(
     GMP,             ! (input) a generated mathematical program
     binding,         ! (input) an index binding
     row,             ! (input) a constraint expression
     [autoAddColumn] ! (optional) a binary scalar
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

autoAddColumn A binary scalar indicating whether this procedure should automatically add columns that are not in the *GMP*. The default is 0 meaning that no columns are added.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- Before generating the rows all existing matrix coefficients for this group of rows are removed.
- The row types and the right-hand-side values (and, if the row type is `'ranged'`, the left-hand-side values) are set according to the constraint definition.
- This procedure cannot be used if one of the rows contains the objective variable, and the row was added or generated before using a different coefficient for the objective variable.
- If the value of *autoAddColumn* equals 0, then this procedure will generate an error if it encounters a column that is not in the *GMP*. You then have to add that column before calling this procedure by using the procedure *GMP::Column::Add* (page 322).
- Setting the value of *autoAddColumn* to 1 should only be done if you know exactly which columns are automatically added by this procedure. Otherwise you might end up with a model in which some columns only appear in one of these rows, possibly making this row redundant.
- This procedure will never add columns that were deleted before with the procedure *GMP::Column::Delete* (page 323).

Example

To generate the rows corresponding to constraint `c(i)` we can use:

```
GMP::Row::GenerateMulti( myGMP, i, c(i) );
```

If the row refers to the definition of a variable then we have to place '_definition' behind the name of the variable. For example, if `v(j)` is a variable with a definition and we want to generate all corresponding rows according to its definition then we have to use:

```
GMP::Row::GenerateMulti( myGMP, j, v_definition(j) );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Column::Add* (page 322), *GMP::Column::Delete* (page 323), *GMP::Row::Add* (page 448), *GMP::Row::Delete* (page 452) and *GMP::Row::Generate* (page 455).

GMP::Row::GetConvex

The function *GMP::Row::GetConvex* (page 458) returns 1 for a row in a generated mathematical program if it has been marked as being convex; otherwise it returns 0.

```
GMP::Row::GetConvex(
     GMP,            ! (input) a generated mathematical program
     row             ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns 1 if the row is convex, and 0 otherwise.

Note: AIMMS cannot detect whether a row is convex or not. A row is marked as being convex if the procedure *GMP::Row::SetConvex* (page 467) has been called before or if the Convex suffix has been set to 1 for the corresponding constraint.

See also:

The procedure *GMP::Row::SetConvex* (page 467). The Convex suffix is explained in full detail in Constraint Suffices for Global Optimization of the Language Reference.

GMP::Row::GetIndicatorColumn

The function *GMP::Row::GetIndicatorColumn* (page 459) returns, for a row in a generated mathematical program, the column number of the indicator column.

```
GMP::Row::GetIndicatorColumn(
    GMP,            ! (input) a generated mathematical program
    row             ! (input) a scalar reference or row number
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns the column number if the indicator column exists, and -1 otherwise.

See also:

The routines *GMP::Row::DeleteIndicatorCondition* (page 453), *GMP::Row::GetIndicatorCondition* (page 459) and *GMP::Row::SetIndicatorCondition* (page 467).

GMP::Row::GetIndicatorCondition

The function *GMP::Row::GetIndicatorCondition* (page 459) returns the indicator condition of a row in a generated mathematical program.

```
GMP::Row::GetIndicatorCondition(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix.

Return Value

The function returns the indicator condition.

Note: This function fails if the row has no indicator column.

See also:

The routines *GMP::Row::DeleteIndicatorCondition* (page 453), *GMP::Row::GetIndicatorColumn* (page 459) and *GMP::Row::SetIndicatorCondition* (page 467).

GMP::Row::GetLeftHandSide

The function *GMP::Row::GetLeftHandSide* (page 460) returns the left-hand-side value of a row as present in the generated mathematical program. This function is typically used for ranged constraints.

Note that this function does not return the (evaluated) level value of a row; you should use the function *GMP::Solution::GetRowValue* (page 495) instead.

```
GMP::Row::GetLeftHandSide(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns the left-hand-side value of the specified row.

Note: If the row has a unit then the scaled left-hand-side value is returned (without unit).

Example

Assume that 'c1' is a constraint in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit      :  kg;
    Conversions   :  ton -> kg : # -> # * 1000;
}
Parameter wght_lower {
    Unit          :  ton;
    InitialValue  :  20;
}
Parameter wght_upper {
    Unit          :  ton;
    InitialValue  :  60;
}
Constraint c1 {
    Unit          :  ton;
    Definition    :  wght_lower <= -x1 + 2 * x2 <= wght_upper;
}
```

If we want to multiply the left-hand-side value by 1.5 and assign it as the new value by using function *GMP::Row::SetLeftHandSide* (page 468) we can use

```
lhs1 := 1.5 * (GMP::Row::GetLeftHandSide( 'MP', c1 )) [ton];

GMP::Row::SetLeftHandSide( 'MP', c1, lhs1 );
```

if 'lhs1' is a parameter with unit [ton], or we can use

```
lhs2 := 1.5 * GMP::Row::GetLeftHandSide( 'MP', c1 );

GMP::Row::SetLeftHandSide( 'MP', c1, lhs2 * GMP::Row::GetScale( 'MP', c1 ) );
```

if 'lhs2' is a parameter without a unit.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::SetLeftHandSide*

(page 468), *GMP::Row::GetRightHandSide* (page 463), *GMP::Row::GetScale* (page 465) and *GMP::Solution::GetRowValue* (page 495).

GMP::Row::GetName

The function *GMP::Row::GetName* (page 462) returns the name of a row in the matrix of a generated mathematical program.

```
GMP::Row::GetName(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns a string.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Column::GetName* (page 332).

GMP::Row::GetRelaxationOnly

The function *GMP::Row::GetRelaxationOnly* (page 462) returns 1 for a row in a generated mathematical program if it has been marked as being a relaxation-only row; otherwise it returns 0.

```
GMP::Row::GetRelaxationOnly(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns 1 if the row is a relaxation-only row, and 0 otherwise.

Note: A row is marked as being a relaxation-only row if the procedure *GMP::Row::SetRelaxationOnly* (page 472) has been called before or if the `RelaxationOnly` suffix has been set to 1 for the corresponding constraint.

See also:

The procedure *GMP::Row::SetRelaxationOnly* (page 472). The `RelaxationOnly` suffix is explained in full detail in Constraint Suffices for Global Optimization of the Language Reference.

GMP::Row::GetRightHandSide

The function *GMP::Row::GetRightHandSide* (page 463) returns the right-hand-side value of a row as present in the generated mathematical program.

```
GMP::Row::GetRightHandSide(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns the right-hand-side value of the specified row.

Note: If the row has a unit then the scaled right-hand-side value is returned (without unit).

Example

Assume that 'c1' is a constraint in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit      :  kg;
    Conversions   :  ton -> kg : # -> # * 1000;
}
Parameter wght {
    Unit          :  ton;
    InitialValue  :  20;
```

(continues on next page)

(continued from previous page)

```
}
Constraint c1 {
    Unit          :  ton;
    Definition    :  -x1 + 2 * x2 <= wght;
}
```

If we want to multiply the right-hand-side value by 1.5 and assign it as the new value by using function *GMP::Row::SetRightHandSide* (page 472) we can use

```
rhs1 := 1.5 * (GMP::Row::GetRightHandSide( 'MP', c1 )) [ton];

GMP::Row::SetRightHandSide( 'MP', c1, rhs1 );
```

if 'rhs1' is a parameter with unit [ton], or we can use

```
rhs2 := 1.5 * GMP::Row::GetRightHandSide( 'MP', c1 );

GMP::Row::SetRightHandSide( 'MP', c1, rhs2 * GMP::Row::GetScale( 'MP', c1 ) );
```

if 'rhs2' is a parameter without a unit.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::SetRightHandSide* (page 472), *GMP::Row::GetLeftHandSide* (page 460) and *GMP::Row::GetScale* (page 465).

GMP::Row::GetRightHandSideRaw

The procedure *GMP::Row::GetRightHandSideRaw* (page 464) retrieves a collection of right-hand-side values corresponding to a given set of row numbers in the generated mathematical program.

```
GMP::Row::GetRightHandSideRaw(
     GMP,            ! (input)  a generated mathematical program
     rowSet,         ! (input)  a subset of Integers
     rhs             ! (output) a real-valued parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set Integers, representing a set of row numbers. Each row number should be in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

rhs A real-valued parameter over rowSet indicating the right hand side values of each row in rowSet.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If a row has a unit then the scaled right-hand-side value is retrieved (without unit).
- This procedure is much more efficient than calling the function *GMP::Row::GetRightHandSide* (page 463) to get the right hand side of each row in rowSet individually.

See also:

The routine *GMP::Row::GetRightHandSide* (page 463).

GMP::Row::GetScale

The function *GMP::Row::GetScale* (page 465) returns the scaling factor of a row in the generated mathematical program.

```
GMP::Row::GetScale(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The scaling factor for the specified row.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Column::GetScale* (page 332).

GMP::Row::GetStatus

The function *GMP::Row::GetStatus* (page 465) returns the status of a row in the matrix of a generated mathematical program.

```
GMP::Row::GetStatus(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

An element in the predefined set *AllRowColumnStatuses* (page 805). The set *AllRowColumnStatuses* (page 805) contains the following elements:

- Active,
- Deactivated,
- Deleted,
- NotGenerated,
- PresolveDeleted.

Note: This function will return '`PresolveDeleted`' only if the generated mathematical program has been created with *GMP::Instance::CreatePresolved* (page 375). Status '`PresolveDeleted`' means that the row was generated for the original generated mathematical program but deleted when the presolved mathematical program was created.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Instance::CreatePresolved* (page 375).

GMP::Row::GetType

The function *GMP::Row::GetType* (page 466) returns the type of a row in the matrix of a generated mathematical program.

```
GMP::Row::GetType(
     GMP,             ! (input) a generated mathematical program
     row              ! (input) a scalar reference or row number
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

Return Value

The function returns an element in the predefined set *AllRowTypes* (page 805).

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Row::SetType* (page 476).

GMP::Row::SetConvex

The procedure *GMP::Row::SetConvex* (page 467) can be used to indicate that a row in a generated mathematical program is convex. Some solvers (like BARON) can make use of this information.

```
GMP::Row::SetConvex(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     value            ! (input) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

value A scalar reference to a 0-1 value.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: AIMMS cannot detect whether a row is convex or not. A row is marked as being convex after this procedure is called with the *value* argument equal to 1 or if the `Convex` suffix has been set to 1 for the corresponding constraint.

See also:

The function *GMP::Row::GetConvex* (page 458). The `Convex` suffix is explained in full detail in Constraint Suffices for Global Optimization of the Language Reference.

GMP::Row::SetIndicatorCondition

The procedure *GMP::Row::SetIndicatorCondition* (page 467) assigns an indicator column and condition to a row in a generated mathematical program.

```
GMP::Row::SetIndicatorCondition(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     column,          ! (input) a scalar reference or column number
     value            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value A binary value that will be used as indicator condition.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Assigning an indicator column and condition to a row means that the row must (only) be satisfied if the level value of the indicator column equals the indicator condition.
- This procedure fails if the row is nonlinear or if the column is not binary.

See also:

The routines *GMP::Row::DeleteIndicatorCondition* (page 453), *GMP::Row::GetIndicatorColumn* (page 459) and *GMP::Row::GetIndicatorCondition* (page 459).

GMP::Row::SetLeftHandSide

The procedure *GMP::Row::SetLeftHandSide* (page 468) changes the left-hand-side of a row in a generated mathematical program.

```
GMP::Row::SetLeftHandSide(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     value            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

value The new value that should be assigned to the left-hand-side of the row.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the row has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the row scale, as returned by the function *GMP::Row::GetScale* (page 465).

Example

Assume that 'c1' is a constraint in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit     :  kg;
    Conversions  :  ton -> kg : # -> # * 1000;
}
Constraint c1 {
    Unit         :  ton;
    Definition   :  -x1 + 2 * x2 <= wght;
}
```

Then if we run the following code

```
GMP::Row::SetLeftHandSide( 'MP', c1, 20 [ton] );
lhs1 := GMP::Row::GetLeftHandSide( 'MP', c1 );
display lhs1;

GMP::Row::SetLeftHandSide( 'MP', c1, 30 );
lhs2 := GMP::Row::GetLeftHandSide( 'MP', c1 );
display lhs2;

GMP::Row::SetLeftHandSide( 'MP', c1, 40 * GMP::Row::GetScale( 'MP', c1 ) );
lhs3 := GMP::Row::GetLeftHandSide( 'MP', c1 );
display lhs3;
```

(where 'lhs1', 'lhs2' and 'lhs3' are parameters without a unit) we get the following results:

```
lhs1 := 20 ;

lhs2 := 0.030 ;

lhs3 := 40 ;
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::SetRightHandSide* (page 472), *GMP::Row::GetLeftHandSide* (page 460) and *GMP::Row::GetScale* (page 465).

GMP::Row::SetPoolType

The procedure *GMP::Row::SetPoolType* (page 470) can be used to indicate that a row in a generated mathematical program should become part of a pool of lazy constraints or a pool of (user) cuts. The solvers CPLEX, Gurobi and ODH-CPLEX can make use of this information.

```
GMP::Row::SetPoolType(
     GMP,            ! (input) a generated mathematical program
     row,            ! (input) a scalar reference or row number
     value,          ! (input) a scalar reference
     [mode]          ! (optional) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

value A scalar reference to a value. The value 1 specifies that the row should be added to the **lazy constraint pool** and 2 specifies that the row should be added to the **cut pool**. The value 0 indicates that the row will be removed from either pools (and treated as a normal row).

mode A scalar reference to a value representing the lazy constraint mode. The value should be a number between 0 and 3. The default is 0. The meaning of these values is explained below.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- The lazy constraint pool is supported by CPLEX, Gurobi and ODH-CPLEX while the cut pool is supported by CPLEX and ODH-CPLEX.
- Use *GMP::Row::SetPoolTypeMulti* (page 471) if the pool type of many rows corresponding to some constraint have to be set, because that will be more efficient.
- The *mode* is only used if the row should be added to the lazy constraint pool (i.e., if *value* equals 1) and if Gurobi is used. The *mode* should be a value between 0 and 3, and these values have the following meaning:
  - 0: The mode is specified by the Gurobi option `Lazy constraint mode`.
  - 1: The lazy constraint can be used to cut off a feasible solution, but it won't necessarily be pulled in if another lazy constraint also cuts off the solution.
  - 2: Lazy constraints that are violated by a feasible solution will be pulled into the model.
  - 3: Lazy constraints that cut off the relaxation solution at the root node are also pulled into the model.

See also:

The procedure *GMP::Row::SetPoolTypeMulti* (page 471). The lazy constraint pool and the cut pool are explained in full detail in Indicator Constraints, Lazy Constraints and Cut Pools of the Language Reference.

GMP::Row::SetPoolTypeMulti

The procedure *GMP::Row::SetPoolTypeMulti* (page 471) can be used to indicate that a group of rows, belonging to a constraint in a generated mathematical program, should become part of a pool of lazy constraints or a pool of (user) cuts. The solvers CPLEX, Gurobi and ODH-CPLEX can make use of this information.

```
GMP::Row::SetPoolTypeMulti(
     GMP,            ! (input) a generated mathematical program
     binding,        ! (input) an index binding
     row,            ! (input) a constraint expression
     value,          ! (input) a numerical expression
     mode            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

value The pool type for each row, defined over the *binding* domain. A value of 1 specifies that the row should be added to the **lazy constraint pool** and 2 specifies that the row should be added to the **cut pool**. The value 0 indicates that the row will be removed from either pools (and treated as a normal row).

mode The lazy constraint mode for each row, defined over the *binding* domain. Its value should be a number between 0 and 3. The meaning of these values is explained below.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- The lazy constraint pool is supported by CPLEX, Gurobi and ODH-CPLEX while the cut pool is supported by CPLEX and ODH-CPLEX.
- The *mode* is only used if the row should be added to the lazy constraint pool (i.e., if *value* equals 1) and if Gurobi is used. The *mode* should be a value between 0 and 3, and these values have the following meaning:
  - 0: The mode is specified by the Gurobi option `Lazy constraint mode`.
  - 1: The lazy constraint can be used to cut off a feasible solution, but it won't necessarily be pulled in if another lazy constraint also cuts off the solution.
  - 2: Lazy constraints that are violated by a feasible solution will be pulled into the model.

– 3: Lazy constraints that cut off the relaxation solution at the root node are also pulled into the model.

See also:

The procedure *GMP::Row::SetPoolType* (page 470). The lazy constraint pool and the cut pool are explained in full detail in Indicator Constraints, Lazy Constraints and Cut Pools of the Language Reference.

GMP::Row::SetRelaxationOnly

The procedure *GMP::Row::SetRelaxationOnly* (page 472) can be used to indicate that a row in a generated mathematical is a relaxation-only row. Some solvers (like BARON) can make use of this information.

```
GMP::Row::SetRelaxationOnly(
     GMP,            ! (input) a generated mathematical program
     row,            ! (input) a scalar reference or row number
     value           ! (input) a scalar reference
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

value A scalar reference to a 0-1 value.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: A row is marked as being a relaxation-only row after this procedure is called with the *value* argument equal to 1 or if the `RelaxationOnly` suffix has been set to 1 for the corresponding constraint.

See also:

The function *GMP::Row::GetRelaxationOnly* (page 462). The `RelaxationOnly` suffix is explained in full detail in Constraint Suffices for Global Optimization of the Language Reference.

GMP::Row::SetRightHandSide

The procedure *GMP::Row::SetRightHandSide* (page 472) changes the right-hand-side of a row in a generated mathematical program.

```
GMP::Row::SetRightHandSide(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     value            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

value The new value that should be assigned to the right-hand-side of the row.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note:

- Use *GMP::Row::SetRightHandSideMulti* (page 474) or *GMP::Row::SetRightHandSideRaw* (page 475) if the right-hand-side values of many rows have to be set, because that will be more efficient.
- If the row has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the row scale, as returned by the function *GMP::Row::GetScale* (page 465).

Example

Assume that 'c1' is a constraint in mathematical program 'MP' with a unit as defined by:

```
Quantity SI_Mass {
    BaseUnit     :  kg;
    Conversions  :  ton -> kg : # -> # * 1000;
}
Constraint c1 {
    Unit         :  ton;
    Definition   :  -x1 + 2 * x2 <= wght;
}
```

Then if we run the following code

```
GMP::Row::SetRightHandSide( 'MP', c1, 20 [ton] );
rhs1 := GMP::Row::GetRightHandSide( 'MP', c1 );
display rhs1;
```

(continues on next page)

(continued from previous page)

```
GMP::Row::SetRightHandSide( 'MP', c1, 30 );
rhs2 := GMP::Row::GetRightHandSide( 'MP', c1 );
display rhs2;

GMP::Row::SetRightHandSide( 'MP', c1, 40 * GMP::Row::GetScale( 'MP', c1 ) );
rhs3 := GMP::Row::GetRightHandSide( 'MP', c1 );
display rhs3;
```

(where 'rhs1', 'rhs2' and 'rhs3' are parameters without a unit) we get the following results:

```
rhs1 := 20 ;

rhs2 := 0.030 ;

rhs3 := 40 ;
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::SetRightHandSideMulti* (page 474), *GMP::Row::SetRightHandSideRaw* (page 475), *GMP::Row::SetLeftHandSide* (page 468), *GMP::Row::GetRightHandSide* (page 463) and *GMP::Row::GetScale* (page 465).

GMP::Row::SetRightHandSideMulti

The procedure *GMP::Row::SetRightHandSideMulti* (page 474) changes the right-hand-sides of a group of rows, belonging to a constraint, in a generated mathematical program.

```
GMP::Row::SetRightHandSideMulti(
     GMP,             ! (input) a generated mathematical program
     binding,         ! (input) an index binding
     row,             ! (input) a constraint expression
     value            ! (input) a numerical expression
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

value The new right-hand-side for each row, defined over the *binding* domain.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the constraint has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the row scale, as returned by the function *GMP::Row::GetScale* (page 465). See *GMP::Row::SetRightHandSide* (page 472) for an example with units.

Example

To set the right-hand-side values of constraint `c(i)` to `rhs(i)` we can use:

```
for (i) do
    GMP::Row::SetRightHandSide( myGMP, c(i), rhs(i) );
endfor;
```

It is more efficient to use:

```
GMP::Row::SetRightHandSideMulti( myGMP, i, c(i), rhs(i) );
```

If we only want to set the right-hand-side values of those `c(i)` for which `dom(i)` is unequal to zero, then we use:

```
GMP::Row::SetRightHandSideMulti( myGMP, i | dom(i), c(i), rhs(i) );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::SetRightHandSide* (page 472), *GMP::Row::SetLeftHandSide* (page 468), *GMP::Row::GetRightHandSide* (page 463) and *GMP::Row::GetScale* (page 465).

GMP::Row::SetRightHandSideRaw

The procedure *GMP::Row::SetRightHandSideRaw* (page 475) changes the right-hand-sides of a group of rows in a generated mathematical program.

```
GMP::Row::SetRightHandSideRaw(
     GMP,            ! (input) a generated mathematical program
     rowSet,         ! (input) a subset of Integers
     value           ! (input) a parameter
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

value A parameter defining a new right-hand-side for each row in *rowSet*.

Return Value

The procedure returns 1 on success, and 0 otherwise.

Note: If the constraint has a unit then *value* should have the same unit. If *value* has no unit then you should multiply it by the row scale, as returned by the function *GMP::Row::GetScale* (page 465). See *GMP::Row::SetRightHandSide* (page 472) for an example with units.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Row::SetRightHandSideRaw* (page 475) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set ConstraintSet {
    SubsetOf: AllConstraints;
}
Set RowSet {
    SubsetOf: Integers;
    Index: rr;
}
Parameter RHS {
    IndexDomain: rr;
}
```

To change the right-hand-side values of the constraint `c(i)` we can use:

```
myGMP := GMP::Instance::Generate( MP );

ConstraintSet := { 'c' };
RowSet := GMP::Instance::GetRowNumbers( myGMP, ConstraintSet );

RHS(rr) := 5.0;

GMP::Row::SetRightHandSideRaw( myGMP, RowSet, RHS );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::SetRightHandSide* (page 472), *GMP::Row::SetLeftHandSide* (page 468), *GMP::Row::GetRightHandSide* (page 463) and *GMP::Row::GetScale* (page 465).

GMP::Row::SetType

The procedure *GMP::Row::SetType* (page 476) changes the type of a row in a generated mathematical program.

```
GMP::Row::SetType(
     GMP,             ! (input) a generated mathematical program
     row,             ! (input) a scalar reference or row number
     type             ! (input) an element in AllRowTypes
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

type An element in *AllRowTypes* (page 805).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: Use *GMP::Row::SetTypeMulti* (page 477) or *GMP::Row::SetTypeRaw* (page 478) if the types of many rows have to be changed, because that will be more efficient.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::GetType* (page 466), *GMP::Row::SetTypeMulti* (page 477) and *GMP::Row::SetTypeRaw* (page 478).

GMP::Row::SetTypeMulti

The procedure *GMP::Row::SetTypeMulti* (page 477) changes the types of a group of rows, belonging to a constraint, in a generated mathematical program.

```
GMP::Row::SetTypeMulti(
     GMP,             ! (input) a generated mathematical program
     binding,         ! (input) an index binding
     row,             ! (input) a constraint expression
     type             ! (input) an element parameter in AllRowTypes
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

binding An index binding that specifies and possibly limits the scope of indices.

row A constraint that, combined with the *binding* domain, specifies the rows.

type An element parameter in *AllRowTypes* (page 805), defined over the *binding* domain.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::GetType* (page 466) and *GMP::Row::SetType* (page 476).

GMP::Row::SetTypeRaw

The procedure *GMP::Row::SetTypeRaw* (page 478) changes the types of a group of rows in a generated mathematical program.

```
GMP::Row::SetTypeRaw(
     GMP,            ! (input) a generated mathematical program
     rowSet,         ! (input) a subset of Integers
     type            ! (input) an element parameter in AllRowTypes
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

type An element parameter in *AllRowTypes* (page 805), defined over the *binding* domain.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Example

Assume that 'MP' is a mathematical program. To use *GMP::Row::SetTypeRaw* (page 478) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
Set ConstraintSet {
    SubsetOf: AllConstraints;
}
Set RowSet {
    SubsetOf: Integers;
    Index: rr;
}
ElementParameter RowType {
    IndexDomain: rr;
    Range: AllRowTypes;
}
```

To change the constraint `c(i)` in a less-than-or-equal-to constraint we can use:

```
myGMP := GMP::Instance::Generate( MP );

ConstraintSet := { 'c' };
RowSet := GMP::Instance::GetRowNumbers( myGMP, ConstraintSet );

RowType(rr) := '<=';

GMP::Row::SetTypeRaw( myGMP, RowSet, RowType );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::GetType* (page 466) and *GMP::Row::SetType* (page 476).

2.3.11 GMP::Solution Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing solutions in the solution repository associated with a generated mathematical program instance:

GMP::Solution::Check

The procedure *GMP::Solution::Check* (page 479) checks the validity of a solution for a generated mathematical program.

```
GMP::Solution::Check(
    GMP,            ! (input) a generated mathematical program
    solution,       ! (input) a solution
    numInfeas,      ! (output) number of infeasibilities
    sumInfeas,      ! (output) sum of infeasibilities
```

(continues on next page)

(continued from previous page)

```
    maxInfeas,      ! (output) maximum infeasibility
    [skipObj],      ! (optional, default 0) a scalar value
    [feasTol],      ! (optional, default -1) a scalar value
    [checkColumns]  ! (optional, default 0) a scalar value
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

numInfeas Number of infeasibilities for the solution.

sumInfeas Sum of all infeasibilities for the solution.

maxInfeas Maximum infeasibility for the solution.

skipObj A scalar binary value to indicate whether constraints containing the objective variable should be skipped (value 1) or not (value 0).

feasTol Feasibility tolerance in checking the constraints violations. If this argument is negative, the value of the option `Constraint Listing Feasibility Tolerance` is used.

checkColumns A scalar binary value to indicate whether to check for variable bound violations and integrality violations (value 1) or not (value 0).

The procedure returns 1 on success, or 0 otherwise.

Note: The optional argument feasTol determines the feasibility tolerance used by this procedure. If a constraint violation is smaller than this tolerance then it will be ignored. If this argument is not passed, or if it is set to a negative value, the option `Constraint Listing Feasibility Tolerance` is used as the feasibility tolerance.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::RetrieveFromModel* (page 501) and *GMP::Solution::RetrieveFromSolverSession* (page 502).

GMP::Solution::ConstraintListing

The procedure *GMP::Solution::ConstraintListing* (page 480) outputs a detailed description of a generated mathematical program to file. It uses the solution to provide feasibility, left hand side and derivative information.

```
GMP::Solution::ConstraintListing(
    GMP,            ! (input) a generated mathematical program
    solution,       ! (input) a solution
    Filename,       ! (input) a string
    [AppendMode]    ! (input/optional) integer, default 0
    )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer that is a reference to a solution.

Filename The name of the file to which the output is written.

AppendMode If non-zero, the output will be appended to the file, instead of overwritten.

This function allows one to inspect a generated mathematical program after it is generated, modified, or solved.

Usage example

Given the following declarations:

```
MathematicalProgram sched;
ElementParameter cp_gmp {
    Range        :  AllGeneratedMathematicalPrograms;
}
Parameter vars_in_cl {
    Range        :  binary;
    InitialData  :  0;
    Comment      : {
        "When 1 the variables and bounds are printed
       in the constraint listing"
     }
```

The use of the function *GMP::Solution::ConstraintListing* (page 480) is illustrated in the following code fragment.

```
cp_gmp := gmp::Instance::Generate( sched );
if cp_gmp then
    GMP::Solution::RetrieveFromModel( cp_gmp, 1 );
    block where constraint_listing_variable_values := vars_in_cl ;
        GMP::Solution::ConstraintListing( cp_gmp, 1, "sched.constraintlisting" );
    endblock;
endif ;
```

The following remarks apply to this code fragment:

- Directly after generation, the generated mathematical program referenced by `cp_gmp` does not contain a solution. The current values in the model can be used to obtain such a solution using *GMP::Solution::RetrieveFromModel* (page 501).
- The actual call to *GMP::Solution::ConstraintListing* (page 480) is placed in a block statement, to permit the programmatic control of output steering options. The available output steering options are in the option category `Solvers General - Standard Reports - Constraints`.

Output

The description that is output by the function *GMP::Solution::ConstraintListing* (page 480) is split into a header, a body, and a footer.

The header of a constraint listing

The brief header contains the solve number (the suffix .number) of the mathematical program and the name of the generated mathematical program. Whenever this suffix is less than or equal to twenty, it is written as a word. When the generated mathematical program is a scheduling problem, containing activities as documented in Activity, the problem schedule domain is also printed, as illustrated in the following example:

```
This is the first constraint listing of mySched.

The schedule domain of mySched is the calendar "TimeLine" containing 61 elements
                                      in the range { '2011-03-31' .. '2011-05-30' }.
```

This is a constraint listing whereby the scheduling problem mySched is solved once. In addition, the problem schedule domain is detailed.

The body of a constraint listing

The body of the constraint listing contains all details in the rows of the generated mathematical program. The information detailed depends both on option settings and the type of row. Lets begin with a linear row.

An LP row

From AIMMS example Transportation model:

```
----  MeetDemand  The amount transported to customer c should meet its demand

MeetDemand(Alkmaar) .. [ 1 | 2 | Optimal ]

    + 1 * Transport(Eindhoven ,Alkmaar) + 1 * Transport(Haarlem   ,Alkmaar)
    + 1 * Transport(Heerenveen,Alkmaar) + 1 * Transport(Middelburg,Alkmaar)
    + 1 * Transport(Zutphen   ,Alkmaar) >= 793 ; (lhs=793, scale=0.001)

    name                              lower    level    upper    scale
    Transport(Eindhoven,Alkmaar)          0        0      inf    0.001
    Transport(Haarlem,Alkmaar)            0      793      inf    0.001
    Transport(Heerenveen,Alkmaar)         0        0      inf    0.001
    Transport(Middelburg,Alkmaar)         0        0      inf    0.001
    Transport(Zutphen,Alkmaar)            0        0      inf    0.001
```

For each group of constraints, the name of that constraint and its text are printed. Next comes each row of that group, whereby the number of rows per symbolic constraint can be limited by the option Number_of_Rows_per_Constraint_in_Listing. A row starts with its name and then, within square brackets, the solve number, the row number, and the solution status of the solution. For that row, it is followed by its contents, whereby all terms containing variables are moved to the left and all terms without variables to the right and summed to mimic the LP form $Ax \leq b$. Between parentheses the lhs is computed by filling in the values of the variables. In this version of the model the base unit for weight is ton, but the constraint uses the unit kg which is 0.001 * ton.

AIMMS computes the LP matrix with respect to the base units and subsequently scales to the units of the variables and constraints. Thus we have a scaling factor of 0.001 for both the constraint and the variables. The coefficients presented are the coefficients after this scaling and as such passed to the solver. The last part of this example shows the variable values, their bounds, and, when relevant, the scaling factor. This last part is obtained by setting the option constraint listing variable values to on.

An NLP row

Consider the arbitrary objective definition

```
Variable o {
    Range      :  free;
    Definition :  x^3 - y^4 + x / y;
}
```

Filling in the definition attribute of variable o will let AIMMS construct the constraint o_definition with the same index domain, empty here, and unit, empty here. This constraint is presented as follows in the constraint listing.

```
----  o_definition

o_definition .. [ 0 | 2 | not solved yet ]

    + [-4] * x + [5] * y + 1 * o = 0 ; (lhs=-1) ****

    name  lower level upper
    x         1     1     4
    y         1     1     5
    o      -inf     0   inf

Hessian:
                                x                     y
                    --------------------  --------------------
                x                    -6                     1
                y                     1                    10
```

This example is similar to the example of the linear row, but with some extras. First, the coefficients -4 and 5 are denoted between brackets to indicate that they are not fixed coefficients, but first order derivative values taken at the level values of the variables. We say that the variables x and y appear non-linear in the constraint o_definition. The coefficient 1 before the variable o is also a first order derivative, but the value of this coefficient does not depend on the values of the variables and is therefore not denoted between brackets. We say that the variable o appears linearly in the constraint o_definition. Next, to indicate that the constraint is infeasible, it is postfixed by ****. Finally, the Hessian containing the second order derivative values is presented, by switching the option constraint_listing_Hessian to on. The Hessian is only presented for those variables that appear non-linear in the constraint presented. A typical question concerns the accuracy of these first and second order derivative values. These derivative values are exact when the non-linear expressions in the constraint only reference differentiable AIMMS intrinsic functions. The first order derivative values are approximated using differencing, when there is a non-linear expression in the constraint referencing an external function. The second order derivative values are not available when a non-linear expression references an external function.

A COP row

Consider the artificial constraint:

```
Constraint element_constraint {
    Definition   :  P(eV) = 7;
}
```

This constraint will lead to the following in the constraint listing.

```
----  element_constraint

element_constraint .. [ 0 | 2 | not solved yet ]

    [1,4,7,10,13,..., 28 (size=10)][eV]
    = 7 ****

    name     lower    level    upper
    eV       'a01'    'a01'    'a10'
```

The main difference between this example and the previous examples is that the presentation is an instantiated symbolic form of the constraints as the presentation of the first and second order derivatives is meaningless in the context of constraint programming.

The footer of a constraint listing

The footer of the constraint listing contains statistics regarding the size of the problem to give an impression of the relative difficulty of the instance presented to other instances with the same structure. It should be noted, that the structure of an instance may have more influence on the difficulty to a solver than sheer size. The structure of an instance depends on how it is modeled.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: A SOLVE statement may produce this constraint listing, depending on the option `constraint_listing`, in the listing file.

See also:

- The Mathematical Program Inspector is an interactive alternative to constraint listings and has additional facilities such as searching for an irreducible infeasibility set for linear program.
- The routine *GMP::Instance::Generate* (page 384).

GMP::Solution::ConstructMean

The procedure *GMP::Solution::ConstructMean* (page 484) constructs the weighted average of two solutions of a generated mathematical program by using the column level values in both solutions. The first solution is replaced by the resulting mean solution.

```
GMP::Solution::ConstructMean(
     GMP,            ! (input) a generated mathematical program
     solution1,      ! (input) a solution
     solution2,      ! (input) a solution
     weight          ! (input) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution1 An integer scalar reference to a solution.

solution2 An integer scalar reference to a solution.

weight The weight used for *solution1*.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: The *weight* argument defines the weight used for *solution1*; for *solution2* a weight of 1 is used. The constructed mean solution is divided by (*weight*+1), and placed in *solution1*.

GMP::Solution::Copy

The procedure *GMP::Solution::Copy* (page 485) copies one solution to another solution in the solution repository of a generated mathematical program.

```
GMP::Solution::Copy(
     GMP,            ! (input) a generated mathematical program
     fromSolution,   ! (input) a solution
     toSolution      ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

fromSolution An integer scalar reference to a solution.

toSolution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Solution::Move* (page 500).

GMP::Solution::Count

The function *GMP::Solution::Count* (page 486) returns the number of non-empty solutions in the solution repository of a generated mathematical program.

```
GMP::Solution::Count(
     GMP              ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The number of non-empty solutions stored in the solution repository.

Note: In order to make the solution repository flexible, it may contain both feasible and infeasible solutions; any solution algorithm, or hybrid combinations thereof, may add or remove solutions.

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Solution::GetSolutionsSet* (page 496).

GMP::Solution::Delete

The procedure *GMP::Solution::Delete* (page 486) deletes a solution from the solution repository of a generated mathematical program.

```
GMP::Solution::Delete(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Solution::DeleteAll* (page 487).

GMP::Solution::DeleteAll

The procedure *GMP::Solution::DeleteAll* (page 487) empties the solution repository of a generated mathematical program.

```
GMP::Solution::DeleteAll(
     GMP             ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Solution::Delete* (page 486).

GMP::Solution::GetBestBound

The function *GMP::Solution::GetBestBound* (page 487) returns the the best known bound on a solution.

```
GMP::Solution::GetBestBound(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

In case of success, the best known bound. Otherwise it returns `UNDF`.

Note:

- This function can be used for GMPs with model type MIP, MIQP or MIQCP.
- This function can also be used for GMPs solved with BARON, regardless of the model type.
- This function can also be used for GMPs with model type QP that are solved with CPLEX, if the CPLEX option *Solution Target* is set to 'Search for global optimum'.
- This function can also be used for GMPs with model type QP or QCP that are solved with GUROBI, if the GUROBI option *Nonconvex Strategy* is set to 'Translate'.
- For multi-objective models, the best bound refers to the (blended) objective with the highest priority.

See also:

The procedure *GMP::Solution::GetObjective* (page 492).

GMP::Solution::GetColumnValue

The function *GMP::Solution::GetColumnValue* (page 488) returns the level value or reduced cost of a column in a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetColumnValue(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     column,         ! (input) a scalar reference or column number
     [valueType]     ! (input/optional) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

valueType A scalar value specifying the value type. If 0 (the default) then the level value will be returned. If 1, the reduced cost. If 3, the basic state.

Return Value

The level value or reduced cost of the column.

Note:

- To get the reduced cost of a column the option `Always Store Marginals` should be switched on or the **ReducedCost** property of the corresponding variable should be set.
- To get the basic state of a column the option `Always Store Basics` should be switched on or the **Basic** property of the corresponding variable should be set.
- This function returns 0 as basic state if the column is **nonbasic**; it returns 1 if the column is **basic** and 2 if the column is **superbasic** (nonlinear models only).
- If the column has a unit then the scaled value is returned (without unit). You can get the scale factor by using the function *GMP::Column::GetScale* (page 332).

See also:

The routines *GMP::Column::GetScale* (page 332), *GMP::Instance::Generate* (page 384), *GMP::Solution::GetRowValue* (page 495) and *GMP::Solution::SetColumnValue* (page 506).

GMP::Solution::GetDistance

The function *GMP::Solution::GetDistance* (page 489) calculates the Euclidean distance between the vectors of column level values in a first and second solution of a generated mathematical program.

```
GMP::Solution::GetDistance(
     GMP,            ! (input) a generated mathematical program
     solution1,      ! (input) a solution
     solution2       ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution1 An integer scalar reference to a solution.

solution2 An integer scalar reference to a solution.

Return Value

In case of success, the Euclidean distance between both solutions. Otherwise it returns UNDF.

Note: The level value of the objective column (if any) is not used.

GMP::Solution::GetFirstOrderDerivative

The function *GMP::Solution::GetFirstOrderDerivative* (page 490) returns the first order derivative for a column in a row in a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetFirstOrderDerivative(
    GMP,            ! (input) a generated mathematical program
    solution,       ! (input) a solution
    row,            ! (input) a scalar reference or row number
    column          ! (input) a scalar reference or column number
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

Return Value

The first order derivative of the column in the row.

Note: If this function is called for multiple rows and columns, then AIMMS will calculate the first order derivatives more efficiently if this function is called row wise instead of column wise. That is, it is better to call this function for all columns in a certain row before calling it for the next row.

See also:

The routines *GMP::Instance::Generate* (page 384).

GMP::Solution::GetIterationsUsed

The function *GMP::Solution::GetIterationsUsed* (page 490) returns the number of iterations used to create a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetIterationsUsed(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The number of iterations used to create a solution.

See also:

The procedures *GMP::Instance::SetIterationLimit* (page 420) and *GMP::Solution::SetIterationCount* (page 507).

GMP::Solution::GetMemoryUsed

The function *GMP::Solution::GetMemoryUsed* (page 491) returns the amount of (peak) memory used to create a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetMemoryUsed(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The amount of megabytes used to create a solution.

Note: For CPLEX and Gurobi, AIMMS calculates the memory in use based on the virtual memory used by the process. This approach is not reliable for asynchronous solver sessions.

See also:

The procedure *GMP::Instance::SetMemoryLimit* (page 421).

GMP::Solution::GetNodesUsed

The function *GMP::Solution::GetNodesUsed* (page 492) returns the number of nodes used to create a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetNodesUsed(
    GMP,            ! (input) a generated mathematical program
    solution        ! (input) a solution
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The number of nodes used to create a solution.

Note:

- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can be used inside a **candidate**, **cut** or **heuristic** callback.

See also:

The routines *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackCandidate* (page 413) and *GMP::Instance::SetCallbackHeuristic* (page 414).

GMP::Solution::GetObjective

The function *GMP::Solution::GetObjective* (page 492) retrieves the objective function value of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetObjective(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The objective function value of the solution.

Note:

- The objective function value is only available if the solution has been retrieved from the solver, or if the function `GMP::Solution::SetObjective` has been called before.
- For multi-objective models, the objective value refers to the (blended) objective with the highest priority.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::GetProgramStatus* (page 494), *GMP::Solution::GetSolverStatus* (page 496) and *GMP::Solution::SetObjective* (page 509).

GMP::Solution::GetPenalizedObjective

The function *GMP::Solution::GetPenalizedObjective* (page 493) calculates the penalized objective for a generated mathematical program by using the level values of the columns in a first solution and the shadow prices in a second solution as the penalty multipliers for the rows. To avoid a very large value, the penalized objective value is divided by the square of the number of rows.

```
GMP::Solution::GetPenalizedObjective(
     GMP,            ! (input) a generated mathematical program
     solution1,      ! (input) a solution
     solution2,      ! (input) a solution
     [skipObj]       ! (optional, default 0) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution1 An integer scalar reference to a solution.

solution2 An integer scalar reference to a solution.

skipObj A scalar binary value to indicate whether the objective defining constraint should be skipped (value 1) or not (value 0).

Return Value

In case of success, the penalized objective function value of the *GMP* associated with both solutions. Otherwise it returns -1e80 for a maximization problem, and 1e80 for a minimization problem (or a feasibility problem).

Note: Assume that x denotes the level values of the columns in *solution1* and w the shadow prices of the rows in *solution2*. Then the penalized objective function $P(x, w)$ is defined as

$$P(x, w) = \frac{f(x) + dirval * \sum_{i=1}^{m} \left(w_i * viol(g_i(x))\right)}{m^2},$$

where $f(x)$ denotes the objective function value, m is the number of rows and the function $viol(g_i(x))$ equals the absolute amount by which the ith row is violated at the point x. Here $dirval$ is 1 in case of minization and -1 in case of maximization.

See also:

The procedure *GMP::Solution::UpdatePenaltyWeights* (page 512).

GMP::Solution::GetProgramStatus

The function *GMP::Solution::GetProgramStatus* (page 494) retrieves the program status of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetProgramStatus(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

An element in the set *AllSolutionStates* (page 807).

Note: The program status is only available if the solution has been retrieved from the solver, or if the procedure *GMP::Solution::SetProgramStatus* (page 510) has been called before.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::GetSolverStatus* (page 496), *GMP::Solution::GetObjective* (page 492) and *GMP::Solution::SetProgramStatus* (page 510).

GMP::Solution::GetRowValue

The function *GMP::Solution::GetRowValue* (page 495) returns the level value or shadow price of a row in a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetRowValue(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     row,            ! (input) a scalar reference or row number
     [valueType]     ! (input/optional) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

valueType A scalar value specifying the value type. If 0 (the default) then the level value as calculated by the solver (or algorithm) will be returned. If 1, the shadow price. If 2, the level value after evaluating the row using the column values in the solution. If 3, the basic state.

Return Value

The level value or shadow price of the row.

Note:

- To get the level value of a row, if *valueType* is set to 0, the option `Always Store Constraint Levels` should be switched on or the **Level** property of the corresponding constraint should be set.
- To get the shadow price of a row the option `Always Store Marginals` should be switched on or the **Shadow-Price** property of the corresponding constraint should be set.
- To get the basic state of a row the option `Always Store Basics` should be switched on or the **Basic** property of the corresponding constraint should be set.

- This function returns 0 as basic state if the row is **nonbasic**; it returns 1 if the row is **basic** and 2 if the row is **superbasic** (nonlinear models only).
- If the row has a unit then the scaled value is returned (without unit). You can get the scale factor by using the function *GMP::Row::GetScale* (page 465).

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Row::GetScale* (page 465), *GMP::Solution::GetColumnValue* (page 488) and *GMP::Solution::SetRowValue* (page 510).

GMP::Solution::GetSolutionsSet

The function *GMP::Solution::GetSolutionsSet* (page 496) returns the set of non-empty solutions in the solution repository of a generated mathematical program.

```
GMP::Solution::GetSolutionsSet(
     GMP            ! (input) a generated mathematical program
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

Return Value

A subset of *Integers* (page 812).

See also:

The functions *GMP::Instance::Generate* (page 384) and *GMP::Solution::Count* (page 486) and the section on Managing the solution repository Managing the Solution Repository of the Language Reference.

GMP::Solution::GetSolverStatus

The function *GMP::Solution::GetSolverStatus* (page 496) retrieves the solver status of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetSolverStatus(
     GMP,           ! (input) a generated mathematical program
     solution       ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

An element in the set *AllSolutionStates* (page 807).

Note: The solver status is only available if the solution has been retrieved from the solver, or if the procedure *GMP::Solution::SetSolverStatus* (page 512) has been called before.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::GetProgramStatus* (page 494) and *GMP::Solution::GetObjective* (page 492) and *GMP::Solution::SetSolverStatus* (page 512).

GMP::Solution::GetTimeUsed

The function *GMP::Solution::GetTimeUsed* (page 497) returns the elapsed time (in 1/100th seconds) used to create a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::GetTimeUsed(
     GMP,             ! (input) a generated mathematical program
     solution         ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The number of 1/100th seconds used to create a solution.

See also:

The procedure *GMP::Instance::SetTimeLimit* (page 424).

GMP::Solution::IsDualDegenerated

The function *GMP::Solution::IsDualDegenerated* (page 497) checks whether the solution for a generated mathematical program, with model type LP, RMIP or QP, is dual degenerated.

```
GMP::Solution::IsDualDegenerated(
     GMP,            ! (input) a generated mathematical program
     solution        ! (input) a solution
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The function returns 1 if the solution is dual degenerated, and 0 otherwise.

Note:

- A solution is dual degenerated if a non-basic variable has a zero marginal, or if a non-equality constraint is non-basic and has a zero marginal. In that case the primal solution is not unique.
- This function will always return 0 if the barrier algorithm (without crossover) of CPLEX was used to solve the problem because the barrier algorithm (without crossover) of CPLEX does not provide a basic solution.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::IsPrimalDegenerated* (page 499) and *GMP::Solution::RetrieveFromSolverSession* (page 502).

GMP::Solution::IsInteger

The function *GMP::Solution::IsInteger* (page 498) checks whether the solution for a generated mathematical program is an integer solution.

```
GMP::Solution::IsInteger(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     [tolerance]     ! (optional) a tolerance
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

tolerance A numerical value. The default is 0.

Return Value

The function returns 1 if the solution is integer, and 0 otherwise.

Note: If the mathematical program contains Special Ordered Sets (SOS) then this function also checks whether the solution satisfies them. If one of the SOS sets is violated then this function returns 0.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::RetrieveFromModel* (page 501) and *GMP::Solution::RetrieveFromSolverSession* (page 502).

GMP::Solution::IsPrimalDegenerated

The function *GMP::Solution::IsPrimalDegenerated* (page 499) checks whether the solution for a generated mathematical program, with model type LP, RMIP or QP, is primal degenerated.

```
GMP::Solution::IsPrimalDegenerated(
     GMP,             ! (input) a generated mathematical program
     solution         ! (input) a solution
     )
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The function returns 1 if the solution is primal degenerated, and 0 otherwise.

Note:

- A solution is primal degenerated if a basic variable is at a bound, or if a non-equality constraint is basic and at a bound. In that case the dual solution is not unique.
- This function will always return 0 if the barrier algorithm (without crossover) of CPLEX was used to solve the problem because the barrier algorithm (without crossover) of CPLEX does not provide a basic solution.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::IsDualDegenerated* (page 497) and *GMP::Solution::RetrieveFromSolverSession* (page 502).

GMP::Solution::Move

The procedure *GMP::Solution::Move* (page 500) moves one solution to another solution in the solution repository of a generated mathematical program.

```
GMP::Solution::Move(
     GMP,            ! (input) a generated mathematical program
     fromSolution,   ! (input) a solution
     toSolution      ! (input) a solution
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

fromSolution An integer scalar reference to a solution.

toSolution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: After calling this procedure, the solution at position *fromSolution* in the solution repository will be empty. This is not the case if you use the procedure *GMP::Solution::Copy* (page 485).

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Solution::Copy* (page 485).

GMP::Solution::RandomlyGenerate

The procedure *GMP::Solution::RandomlyGenerate* (page 500) generates random level values in a solution for all columns in a generated mathematical program. Each level value is sampled from the uniform distribution by using the lower and upper bound of the column as parameters.

```
GMP::Solution::RandomlyGenerate(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     [maxVarBound],  ! (optional) a scalar value
     [startPoint],   ! (optional) a solution
```

(continues on next page)

(continued from previous page)

```
    [perturbation]  ! (optional) a scalar value
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

maxVarBound The maximal variable bound. If a column has no upper bound then the sampled level value will be smaller than the maximal variable bound, and if a column has no lower bound then the sampled level value will be greater than minus the maximal variable bound. The default is 1000.

startPoint An integer scalar reference to a solution representing a starting point. If specified then the sampled level value of a column will be around its level value in the starting point. By default no starting point is used.

perturbation Used in combination with argument *startPoint*. A value between 0 and 1 that represents the (relative) perturbation around the starting pount. The default is 0.1.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure should be called after calling the function *GMP::Instance::CreatePresolved* (page 375) if it is used in combination with that function. Otherwise the sampled level values might be outside the range of the columns in the presolved model.
- If argument *startPoint* is specified then for each column the sampled value will be in the range

  where x denotes the level value of the column, lb and ub its lower and upper bound respectively, and p the *perturbation* value.
- *startPoint* cannot be equal to *solution*.

See also:

The function *GMP::Instance::CreatePresolved* (page 375).

GMP::Solution::RetrieveFromModel

The procedure *GMP::Solution::RetrieveFromModel* (page 501) stores the solution from the model identifiers into the solution repository of a generated mathematical program.

```
GMP::Solution::RetrieveFromModel(
    GMP,            ! (input) a generated mathematical program
    solution        ! (input) a solution
    )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: A solution vector in the solution repository only contains solution data for the generated columns and rows of the GMP. Hence, no solution data is stored in the solution repository for columns and rows that were not generated.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::SendToModel* (page 503), *GMP::Solution::RetrieveFromSolverSession* (page 502) and *GMP::Solution::SendToSolverSession* (page 505).

GMP::Solution::RetrieveFromSolverSession

The procedure *GMP::Solution::RetrieveFromSolverSession* (page 502) stores the solution from a solver session into the solution repository of a generated mathematical program.

```
GMP::Solution::RetrieveFromSolverSession(
     solverSession,  ! (input) a solver session
     solution        ! (input) a solution
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

solution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- For a solver session belonging to a GMP with type MIP, this procedure retrieves the best integer solution found by so far (i.e., the incumbent), except when this procedure is called inside a **branch**, **cut**, **heuristic** or **lazy constraint** callback. In that case this procedure retrieves the LP solution of the current node (**branch**, **cut**, **heuristic**) or an integer feasible solution (**lazy constraint**).
- The function *GMP::SolverSession::GetNodeObjective* (page 547) can be used to get the objective value corresponding to the solution retrieved with this procedure inside a **branch**, **candidate**, **cut**, **heuristic** or **lazy constraint** callback.

- By using the procedure *GMP::SolverSession::RejectIncumbent* (page 554) the incumbent solution can be rejected inside a **candidate** callback.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Solution::SendToSolverSession* (page 505), *GMP::Solution::RetrieveFromModel* (page 501), *GMP::Solution::SendToModel* (page 503), *GMP::SolverSession::GetNodeObjective* (page 547) and *GMP::SolverSession::RejectIncumbent* (page 554).

GMP::Solution::SendToModel

The procedure *GMP::Solution::SendToModel* (page 503) initializes the model identifiers with the values in the solution from the solution repository of a generated mathematical program.

```
GMP::Solution::SendToModel(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     [merge],        ! (optional, default 0) a scalar value
     [evalInline]    ! (optional, default 1) a scalar binary value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

merge A scalar value to indicate whether the values of the variables and constraints in the mathematical program should be replaced by (value 0) or merged with (value 1 or 2) the solution.

evalInline A scalar binary value to indicate whether the level values of inline variables (if any) in the mathematical program should be evaluated (value 1) or not (value 0).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- A solution vector in the solution repository only contains solution data for the generated columns and rows of the GMP. Hence, no solution data is stored in the solution repository for columns and rows that were not generated.
- By default the values of the variables in the Variables set of the mathematical program will be emptied for all index tuples before sending the solution values to the variables. If the argument *merge* is set to 1 or 2 then only values of columns (i.e., individual variables) present in the *GMP* will be replaced. (The same holds for the constraints.)

- The difference between values 1 and 2 for argument *merge* only affects the objective variable. With value 1 the level value of the objective variable will be **increased** by the level value for the corresponding column in the *solution*. With value 2 the level value of the objective variable will be **replaced** by the level value for the corresponding column in the *solution*.
- With the argument *merge* set to 1, if the level value of the objective variable equals NA, or if the level value for the corresponding column in the *solution* equals NA, then the level value of the objective variable will remain or be set to NA.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::RetrieveFromModel* (page 501), *GMP::Solution::RetrieveFromSolverSession* (page 502), *GMP::Solution::SendToModelSelection* (page 504) and *GMP::Solution::SendToSolverSession* (page 505).

GMP::Solution::SendToModelSelection

The procedure *GMP::Solution::SendToModelSelection* (page 504) initializes a part of the model identifiers with the values in the solution from the solution repository of a generated mathematical program.

```
GMP::Solution::SendToModelSelection(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     Identifiers,    ! (input) a set expression
     Suffices,       ! (input) a set expression
     [merge],        ! (optional, default 0) a scalar value
     [evalInline]    ! (optional, default 1) a scalar binary value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Identifiers A subset of the predefined set *AllVariablesConstraints* (page 832), containing the set of all variables and constraints for which the values have to be changed into those of *solution*.

Suffices A subset of the predefined set *AllSuffixNames* (page 809), containing the set of suffixes for which the values of *Identifiers* have to be changed into those of *solution*.

merge A scalar value to indicate whether the values of the variables and constraints in the math program should be replaced by (value 0) or merged with (value 1 or 2) the solution.

evalInline A scalar binary value to indicate whether the level values of inline variables (if any) in the mathematical program should be evaluated (value 1) or not (value 0).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the subset *Identifiers* contains a variable or constraint that is not part of the generated mathematical program, then that variable or constaint will be ingnored and its data will not change.
- If the subset *Suffices* contains a suffix other than 'Level', 'Basic', 'ReducedCost', 'ShadowPrice', 'SmallestCoefficient', 'NominalCoefficient', 'LargestCoefficient', 'SmallestValue', 'LargestValue', 'SmallestRightHandSide', 'NominalRightHandSide', 'LargestRightHandSide', 'SmallestShadowPrice' and 'LargestShadowPrice', then that suffix will be ingnored and its data will not change.
- A solution vector in the solution repository only contains solution data for the generated columns and rows of the GMP. Hence, no solution data is stored in the solution repository for columns and rows that were not generated.
- By default the values of the variables in the Variables set of the mathematical program will be emptied for all index tuples before sending the solution values to the variables. If the argument *merge* is set to 1 or 2 then only values of columns (i.e., individual variables) present in the *GMP* will be replaced. (The same holds for the constraints.) Note that setting *merge* to 1 or 2 has no impact on scalar variables (and constraints).
- The difference between values 1 and 2 for argument *merge* only affects the objective variable. With value 1 the level value of the objective variable will be **increased** by the level value for the corresponding column in the *solution*. With value 2 the level value of the objective variable will be **replaced** by the level value for the corresponding column in the *solution*.
- With the argument *merge* set to 1, if the level value of the objective variable equals NA, or if the level value for the corresponding column in the *solution* equals NA, then the level value of the objective variable will remain or be set to NA.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::RetrieveFromModel* (page 501), *GMP::Solution::RetrieveFromSolverSession* (page 502), *GMP::Solution::SendToSolverSession* (page 505) and *GMP::Solution::SendToModel* (page 503)

GMP::Solution::SendToSolverSession

The procedure *GMP::Solution::SendToSolverSession* (page 505) initializes a solver session with the values in the solution from the solution repository of a generated mathematical program.

```
GMP::Solution::SendToSolverSession(
     solverSession,  ! (input) a solver session
     solution        ! (input) a solution
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

solution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::RetrieveFromSolverSession* (page 502), *GMP::Solution::RetrieveFromModel* (page 501) and *GMP::Solution::SendToModel* (page 503).

GMP::Solution::SetColumnValue

The procedure *GMP::Solution::SetColumnValue* (page 506) sets the level value, reduced cost, hint value or hint priority of a column in a solution in the solution repository of a generated mathematical program.

Hint values and hint priorities can be used as follows: If you know that a variable is likely to take a particular value in high quality solutions of a MIP model, you can provide that value as a hint. You can also (optionally) provide a hint priority which resembles your level of confidence in a hint.

```
GMP::Solution::SetColumnValue(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     column,         ! (input) a scalar reference or column number
     value,          ! (input) a scalar value
     [valueType]     ! (input/optional) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

column A scalar reference to an existing column in the matrix or an element in the set *Integers* (page 812) in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

value The value to be assigned to the column.

valueType A scalar value specifying the value type. If 0 (the default) then the level value will be set. If 1, the reduced cost. If 2, the hint value, and if 3 the hint priority.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the column has a unit then the scaled value should be passed. You can get the scale factor by using the function *GMP::Column::GetScale* (page 332).
- Hint values and priorities are only supported by Gurobi.

Example

Assume we have a GMP for which we have two solutions in the solution repository at positions 1 and 2. Our goal is to add up the level values of each column in the solutions, and place the result in the solution at position 3 in the solution repository. This can be done in a generic way using the function *GMP::Instance::GetColumnNumbers* (page 392) as follows. Here `ColumnNrs` is a subset of *Integers* (page 812) with index `c`.

```
! Get the column numbers of all variables in myGMP.
ColumnNrs := GMP::Instance::GetColumnNumbers( myGMP, AllVariables );

for ( c ) do
    ! Get level value of column c in solution 1.
    val1 := GMP::Solution::GetColumnValue( myGMP, 1, c );
    ! Get level value of column c in solution 2.
    val2 := GMP::Solution::GetColumnValue( myGMP, 2, c );

    ! Assign the sum to column c in solution 3.
    GMP::Solution::SetColumnValue( myGMP, 3, c, val1 + val2 );
endfor;

! Send solution 3 to the (symbolic) model identifiers.
GMP::Solution::SendToModel( myGMP, 3 );
```

In the next example, we use the current level values of the variable `JobSchedule` as variable hints:

```
myGMP := GMP::Instance::Generate( FlowShopModel );

for (j,s) do
    GMP::Solution::SetColumnValue( myGMP, 1, JobSchedule(j,s),
                                   JobSchedule(j,s).level, 2 );
    GMP::Solution::SetColumnValue( myGMP, 1, JobSchedule(j,s), 10, 3 );
endfor;

GMP::Instance::Solve( myGMP );
```

In this example the hint priority for `JobSchedule` is set to 10.

See also:

The routines *GMP::Column::GetScale* (page 332), *GMP::Instance::Generate* (page 384), *GMP::Instance::GetColumnNumbers* (page 392), *GMP::Solution::GetColumnValue* (page 488), *GMP::Solution::SendToModel* (page 503) and *GMP::Solution::SetRowValue* (page 510).

GMP::Solution::SetIterationCount

The procedure *GMP::Solution::SetIterationCount* (page 507) sets the iteration count of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::SetIterationCount(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     iterationCount  ! (input) iteration count
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

iterationCount An integer scalar.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::SolverSession::GetIterationsUsed* (page 545) and *GMP::Solution::SetProgramStatus* (page 510).

GMP::Solution::SetMIPStartFlag

The procedure *GMP::Solution::SetMIPStartFlag* (page 508) can be used to mark a solution in the solution repository of a generated mathematical program such that it should be used as a MIP start during the a MIP solve (or a MIQP or MIQCP solve).

```
GMP::Solution::SetMIPStartFlag(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     flag,           ! (input) a scalar value
     [effortLevel]   ! (optional, default 0) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

flag A scalar binary value to indicate whether the solution should be marked (value 1) or unmarked (value 0) as MIP start.

effortLevel A scalar value to specify the level of effort that the solver should apply to the solution when using it as MIP start solution. The default value of 0 indicates that the solver should decide; the other values are explained below.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure is only supported by CPLEX, Gurobi and COPT.
- The argument *effortLevel* is only used by CPLEX.
- The levels of effort and their effect as specified by argument *effortLevel* are:
  - Level 0: The solver decides.
  - Level 1: The solver checks feasibility of the corresponding MIP start.
  - Level 2: The solver solves the fixed LP problem specified by the MIP start.
  - Level 3: The solver solves a subMIP.
  - Level 4: The solver attempts to repair the MIP start if it is infeasible.
  - Level 5: A complete solution is injected without the solver performing the usual checks. If the solution defined by the MIP start is infeasible, behavior is undefined.

See also:

The routines *GMP::Instance::Generate* (page 384).

GMP::Solution::SetObjective

The procedure *GMP::Solution::SetObjective* (page 509) sets the objective function value of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::SetObjective(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     value           ! (input) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

value A scalar value to be assigned.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions *GMP::Instance::Generate* (page 384), *GMP::Solution::GetObjective* (page 492) and *GMP::Solution::SendToModel* (page 503).

GMP::Solution::SetProgramStatus

The procedure *GMP::Solution::SetProgramStatus* (page 510) sets the program status of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::SetProgramStatus(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     status          ! (input) a status
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

status An element in the set *AllSolutionStates* (page 807).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::GetProgramStatus* (page 494) and *GMP::Solution::SetSolverStatus* (page 512).

GMP::Solution::SetRowValue

The procedure *GMP::Solution::SetRowValue* (page 510) sets the level value or shadow price of a row in a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::SetRowValue(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     row,            ! (input) a scalar reference or row number
     value,          ! (input) a scalar value
     [valueType]     ! (input/optional) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

row A scalar reference to an existing row in the matrix or an element in the set *Integers* (page 812) in the range $\{0..m-1\}$ where m is the number of rows in the matrix.

value The value to be assigned to the row.

valueType A scalar value specifying the value type. If 0 (the default) then the level value will be set. If 1, the shadow price.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: If the row has a unit then the scaled value should be passed. You can get the scale factor by using the function *GMP::Row::GetScale* (page 465).

Example

Assume we have a GMP for which we want to multiply all shadow prices in a solution by some value, say 10. This can be done in a generic way using the function *GMP::Instance::GetRowNumbers* (page 405) as follows. Here `RowNrs` is a subset of *Integers* (page 812) with index `r`.

```
! Get the row numbers of all constraints in myGMP.
RowNrs := GMP::Instance::GetRowNumbers( myGMP, AllConstraints );

for ( r ) do
    ! Get shadow price of row r in solution 1.
    val := GMP::Solution::GetRowValue( myGMP, 1, r, valueType : 1 );

    ! Assign new value for shadow price to row r in solution 1.
    GMP::Solution::SetRowValue( myGMP, 1, r, 10 * val, valueType : 1 );
endfor;
```

(continues on next page)

(continued from previous page)

```
! Send solution to the (symbolic) model identifiers.
GMP::Solution::SendToModel( myGMP, 1 );
```

Note: the shadow prices will only be stored in the data structures of the constraints if the `ShadowPrice` property of the variables is set, or if the option `Always_Store_Marginals` is set.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::GetRowNumbers* (page 405), *GMP::Row::GetScale* (page 465), *GMP::Solution::GetRowValue* (page 495), *GMP::Solution::SendToModel* (page 503) and *GMP::Solution::SetColumnValue* (page 506).

GMP::Solution::SetSolverStatus

The procedure *GMP::Solution::SetSolverStatus* (page 512) sets the solver status of a solution in the solution repository of a generated mathematical program.

```
GMP::Solution::SetSolverStatus(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     status          ! (input) a status
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

status An element in the set *AllSolutionStates* (page 807).

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Solution::GetSolverStatus* (page 496) and *GMP::Solution::SetProgramStatus* (page 510).

GMP::Solution::UpdatePenaltyWeights

The procedure *GMP::Solution::UpdatePenaltyWeights* (page 512) updates the penalty weights which are stored as shadow prices in a first solution of a generated mathematical program. The shadow price of a row in this solution is compared with the shadow price of the same row in the second solution, and replaced by the maximum of both shadow prices.

```
GMP::Solution::UpdatePenaltyWeights(
     GMP,            ! (input) a generated mathematical program
     solution1,      ! (input) a solution
     solution2,      ! (input) a solution
     [minValue]      ! (optional) a scalar value
     )
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution1 An integer scalar reference to a solution.

solution2 An integer scalar reference to a solution.

minValue The minimum value for each shadow price. The default is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: If for a certain row both the shadow prices in *solution1* and *solution2* are smaller than *minValue*, the new value assigned to the shadow price in *solution1* will be *minValue*.

See also:

The function *GMP::Solution::GetPenalizedObjective* (page 493).

2.3.12 GMP::Solver Procedures and Functions

AIMMS supports the following procedures and functions for retrieving solver related information, and managing solver environments:

GMP::Solver::FreeEnvironment

The procedure *GMP::Solver::FreeEnvironment* (page 513) can be used to free a solver environment. By using the procedure *GMP::Solver::InitializeEnvironment* (page 516) you can initialize a solver environment; by using this procedure you can free it again.

Normally AIMMS initializes solver environments at startup and frees them when it is closed. The procodures *GMP::Solver::InitializeEnvironment* (page 516) and *GMP::Solver::FreeEnvironment* (page 513) can be used to initialize and free a solver environment multiple times inside one AIMMS sesstion. Both procedures are typically used for solvers running on a remote server or a cloud system.

```
GMP::Solver::FreeEnvironment(
     solver      ! (input) a solver
     )
```

Arguments

solver An element in the set *AllSolvers* (page 787).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure can be used in combination with a normal solve statement.
- This procedure is only supported by Gurobi.
- This procedure cannot be called inside a solver callback procedure.
- This procedure cannot be called if one of the solver sessions is asynchronous executing.

Example

Assume that 'MIPSolver' is an element parameter with range *AllSolvers* (page 787) and 'myGMP' is an element parameter with range *AllGeneratedMathematicalPrograms* (page 833).

```
MIPSolver := 'Gurobi 12.0';

! First solve using normal solve statement.

GMP::Solver::InitializeEnvironment( MIPSolver );

solve MP1;

GMP::Solver::FreeEnvironment( MIPSolver );

! Second solve using GMP solve.

GMP::Solver::InitializeEnvironment( MIPSolver );

mgGMP := GMP::Instance::Generate( MP2 );
GMP::Instance::Solve( myGMP );

GMP::Solver::FreeEnvironment( MIPSolver );
```

See also:

The procedure *GMP::Solver::InitializeEnvironment* (page 516).

GMP::Solver::GetAsynchronousSessionsLimit

The function *GMP::Solver::GetAsynchronousSessionsLimit* (page 514) returns the maximum number of asynchronous solver sessions that can run simultaneous for a certain solver. This number depends on the AIMMS license.

```
GMP::Solver::GetAsynchronousSessionsLimit(
    solver,      ! (input) a solver
    [cores],     ! (input, optional) a binary scalar value
    [GMP]        ! (input, optional) a generated mathematical program
    )
```

Arguments

solver An element in the set *AllSolvers* (page 787).

cores A binary scalar indicating whether the this function should take into account the number of cores on the machine. The default is 0 (cores are not used).

GMP An element in *AllGeneratedMathematicalPrograms* (page 833). By default this argument is empty.

Return Value

The maximal number of asynchronous solver sessions that can run simultaneous using *solver*, or any other version of the same solver.

If the *cores* argument equals 1 then this function returns the number of cores on the machine if that number is smaller than the maximal number of asynchronous solver sessions.

If the *GMP* argument is used then this function will return 0 if the specified generated mathematical program cannot be used for asynchronous executing (e.g., if it contains a constraint with a nonlinear expression referencing an external function).

Note:

- The function returns 0 if the solver cannot be found or is not licensed. It also returns 0 if the solver cannot be used to do an asynchronous solve (e.g., BARON, CBC, ODH-CPLEX).
- The function returns 1 if the solver is not thread-safe (e.g., IPOPT, SNOPT).
- The number of asynchronous solver sessions running in parallel using a certain solver should not exceed the limit on asynchronous solver sessions, as returned by this function. To count the number of asynchronous solver sessions currently running with a solver, AIMMS checks all solver versions available. For example, if one asynchronous solver session is running with CPLEX 22.1 and another simultaneous with CPLEX 20.1 then solver CPLEX is running two asynchronous solver sessions. The value returned by this function limits all solver versions together (even though the argument passed to the function refers to a particular solver version).

Example

Assume that 'MaxSes' is a parameter then the following statement returns the maximal number of asynchronous solver sessions for CPLEX:

```
MaxSes := GMP::Solver::GetAsynchronousSessionsLimit( 'CPLEX 22.1' );
```

The value MaxSes is the limit on asynchronous solver sessions that can run at the same time with CPLEX 22.1 plus CPLEX 20.1 plus CPLEX 12.10, etc.

See also:

The routine *GMP::SolverSession::AsynchronousExecute* (page 528).

GMP::Solver::InitializeEnvironment

The procedure *GMP::Solver::InitializeEnvironment* (page 516) can be used to initialize a solver environment. By using the procedure *GMP::Solver::FreeEnvironment* (page 513) you can free a solver environment; by using this procedure you can initialize it again.

Normally AIMMS initializes solver environments at startup and frees them when it is closed. The procedures *GMP::Solver::InitializeEnvironment* (page 516) and *GMP::Solver::FreeEnvironment* (page 513) can be used to initialize and free a solver environment multiple times inside one AIMMS sesstion. Both procedures are typically used for solvers running on a remote server or a cloud system.

Several environment parameters can be set using the optional arguments. Instead you can also use one of the following procedures to set these or other parameters:

- *GMP::Solver::SetEnvironmentDoubleParameter* (page 518)
- *GMP::Solver::SetEnvironmentIntegerParameter* (page 519)
- *GMP::Solver::SetEnvironmentStringParameter* (page 521)

```
GMP::Solver::InitializeEnvironment(
    solver,             ! (input) a solver
    [computeserver],    ! (input, optional) a string expression
    [password],         ! (input, optional) a string expression
    [priority],         ! (input, optional) integer, default 0
    [timeout],          ! (input, optional) integer, default -1
    [logfile]           ! (input, optional) a string expression
    )
```

Arguments

solver An element in the set *AllSolvers* (page 787).

computeserver A string containing a comma-separated list of compute servers. You can refer to compute server machines using their names or their IP addresses.

password The password for gaining access to the specified compute servers. Do not specify this argument if no password is required.

priority The priority of the job. Priorities must be between -100 and 100, with a default value of 0. Higher priority jobs are chosen from the server job queue before lower priority jobs.

timeout Job queue timeout (in seconds). If the job does not reach the front of the queue before the specified timeout, the call will exit with an error. Use the default value of -1 to indicate that the call should never timeout.

logfile The name of the log file for this environment. If this argument is not specified then no log file will be created for this environment.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the solver environment is already initialized when this procedure is called, the solver environment will be freed first.
- This procedure can be used in combination with a normal solve statement.
- This procedure is only supported by Gurobi.
- If the *computeserver* argument is not specified then the compute server must be specified via a Gurobi client license key file, or using the procedure *GMP::Solver::SetEnvironmentStringParameter* (page 521).
- The *computeserver* argument can refer to a server using its name or its IP address. If you are using a non-default port, the server name should be followed by the port number (e.g., myserver1:61000).
- The *computeserver* argument may contain a comma-separated list of servers to increase robustness. If the first server in the list does not respond then the second will be tried, etc.
- The optional arguments *password*, *priority*, *timeout* and *logfile* are only used if the optional argument *computeserver* is specified.
- A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.
- This procedure cannot be called inside a solver callback procedure.
- This procedure cannot be called if one of the solver sessions is asynchronous executing.

Example

Assume that 'MIPSolver' is an element parameter with range *AllSolvers* (page 787) and 'myGMP' is an element parameter with range *AllGeneratedMathematicalPrograms* (page 833).

```
MIPSolver := 'Gurobi 12.0';

! First solve using normal solve statement.

GMP::Solver::InitializeEnvironment( MIPSolver, computeserver: "myserver1:61000
↪", priority: 10 );

solve MP1;

GMP::Solver::FreeEnvironment( MIPSolver );

! Second solve using GMP solve.

GMP::Solver::SetEnvironmentStringParameter( MIPSolver, "ComputeServer",
↪"myserver1:61000" );
GMP::Solver::SetEnvironmentIntegerParameter( MIPSolver, "CSPriority", 10 );

GMP::Solver::InitializeEnvironment( MIPSolver );

mgGMP := GMP::Instance::Generate( MP2 );
GMP::Instance::Solve( myGMP );

GMP::Solver::FreeEnvironment( MIPSolver );
```

See also:

The procedures *GMP::Solver::FreeEnvironment* (page 513), *GMP::Solver::SetEnvironmentDoubleParameter* (page 518), *GMP::Solver::SetEnvironmentIntegerParameter* (page 519) and *GMP::Solver::SetEnvironmentStringParameter* (page 521).

GMP::Solver::SetEnvironmentDoubleParameter

The procedure *GMP::Solver::SetEnvironmentDoubleParameter* (page 518) can be used to set a double-valued parameter to be used for starting a solver environment. This procedure is typically used for solvers running on a remote server or a cloud system.

```
GMP::Solver::SetEnvironmentDoubleParameter(
     solver,            ! (input) a solver
     parameter,         ! (input) a string expression
     value              ! (input) a numerical expression
     )
```

Arguments

solver An element in the set *AllSolvers* (page 787).

parameter The name of the parameter.

value The desired value of the parameter.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure is only supported by Gurobi. Typically it is only used in combination with a Gurobi link-only license.
- This procedure cannot be called inside a solver callback procedure.

Parameters

This procedure and the procedures *GMP::Solver::SetEnvironmentIntegerParameter* (page 519) and *GMP::Solver::SetEnvironmentStringParameter* (page 521) can be used to set Gurobi Configuration Parameters. Typically these procedures are used to set Gurobi parameters for Cloud, Compute Server, Cluster Manager or Token Server. Note that normally these parameters are set in the Gurobi license file.

Example

```
MIPSolver := 'Gurobi 12.0';

GMP::Solver::SetEnvironmentStringParameter( MIPSolver, "ComputeServer",
↪"myserver1:61000" );
GMP::Solver::SetEnvironmentDoubleParameter( MIPSolver, "CSQueueTimeout", 60 );
GMP::Solver::SetEnvironmentDoubleParameter( MIPSolver, "MemLimit", 64 );

GMP::Solver::InitializeEnvironment( MIPSolver );

solve MP1;

GMP::Solver::FreeEnvironment( MIPSolver );
```

See also:

The procedures *GMP::Solver::InitializeEnvironment* (page 516), *GMP::Solver::SetEnvironmentIntegerParameter* (page 519) and *GMP::Solver::SetEnvironmentStringParameter* (page 521).

GMP::Solver::SetEnvironmentIntegerParameter

The procedure *GMP::Solver::SetEnvironmentIntegerParameter* (page 519) can be used to set a integer-valued parameter to be used for starting a solver environment. This procedure is typically used for solvers running on a remote server or a cloud system.

```
GMP::Solver::SetEnvironmentIntegerParameter(
     solver,            ! (input) a solver
     parameter,         ! (input) a string expression
     value              ! (input) a numerical expression
     )
```

Arguments

solver An element in the set *AllSolvers* (page 787).

parameter The name of the parameter.

value The desired value of the parameter.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure is only supported by Gurobi. Typically it is only used in combination with a Gurobi link-only license.
- This procedure cannot be called inside a solver callback procedure.

Parameters

This procedure and the procedures *GMP::Solver::SetEnvironmentDoubleParameter* (page 518) and *GMP::Solver::SetEnvironmentStringParameter* (page 521) can be used to set Gurobi Configuration Parameters. Typically these procedures are used to set Gurobi parameters for Cloud, Compute Server, Cluster Manager or Token Server. Note that normally these parameters are set in the Gurobi license file.

Example

```
MIPSolver := 'Gurobi 12.0';

GMP::Solver::SetEnvironmentStringParameter( MIPSolver, "ComputeServer",
↪"myserver1:61000" );
GMP::Solver::SetEnvironmentIntegerParameter( MIPSolver, "ServerTimeout", 30 );

GMP::Solver::InitializeEnvironment( MIPSolver );
```

(continues on next page)

(continued from previous page)

```
solve MP1;

GMP::Solver::FreeEnvironment( MIPSolver );
```

See also:

The procedures *GMP::Solver::InitializeEnvironment* (page 516), *GMP::Solver::SetEnvironmentDoubleParameter* (page 518) and *GMP::Solver::SetEnvironmentStringParameter* (page 521).

GMP::Solver::SetEnvironmentStringParameter

The procedure *GMP::Solver::SetEnvironmentStringParameter* (page 521) can be used to set a string-valued parameter to be used for starting a solver environment. This procedure is typically used for solvers running on a remote server or a cloud system.

```
GMP::Solver::SetEnvironmentStringParameter(
     solver,            ! (input) a solver
     parameter,         ! (input) a string expression
     value              ! (input) a string expression
     )
```

Arguments

solver An element in the set *AllSolvers* (page 787).

parameter The name of the parameter.

value The desired value of the parameter.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure is only supported by Gurobi. Typically it is only used in combination with a Gurobi link-only license.
- This procedure cannot be called inside a solver callback procedure.

Parameters

This procedure and the procedures *GMP::Solver::SetEnvironmentDoubleParameter* (page 518) and *GMP::Solver::SetEnvironmentIntegerParameter* (page 519) can be used to set Gurobi Configuration Parameters. Typically these procedures are used to set Gurobi parameters for Cloud, Compute Server, Cluster Manager or Token Server. Note that normally these parameters are set in the Gurobi license file.

Example

```
MIPSolver := 'Gurobi 12.0';

GMP::Solver::SetEnvironmentStringParameter( MIPSolver, "ComputeServer",
↪"myserver1:61000" );

GMP::Solver::InitializeEnvironment( MIPSolver );

solve MP1;

GMP::Solver::FreeEnvironment( MIPSolver );
```

See also:

The procedures *GMP::Solver::InitializeEnvironment* (page 516), *GMP::Solver::SetEnvironmentDoubleParameter* (page 518) and *GMP::Solver::SetEnvironmentIntegerParameter* (page 519).

2.3.13 GMP::SolverSession Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing solver sessions associated with a generated mathematical program instance:

GMP::SolverSession::AddBendersFeasibilityCut

The procedure *GMP::SolverSession::AddBendersFeasibilityCut* (page 522) generates a feasibility cut for a Benders' master problem using the solution of a Benders' subproblem (or the corresponding feasibility problem). The Benders' master problem must be a MIP problem.

The cut is typically added as a lazy constraint in a callback during the MIP branch-and-cut search. This procedure is typically used in a Benders' decomposition algorithm in which a single master MIP problem is solved.

```
GMP::SolverSession::AddBendersFeasibilityCut(
     solverSession,     ! (input) a solver session
     GMP,               ! (input) a generated mathematical program
     solution,          ! (input) a solution
     [local],           ! (optional, default 0) a scalar binary expression
     [purgeable],       ! (optional, default 0) a scalar binary expression
     [tighten]          ! (optional, default 0) a scalar binary expression
)
```

Arguments

solverSession An element in the set `AllSolverSessions` (page 828) representing a solver session for the Benders' master problem.

GMP An element in the set `AllGeneratedMathematicalPrograms` (page 833) representing a Benders' subproblem.

solution An integer scalar reference to a solution of *GMP2*.

local A scalar binary value to indicate whether the cut is valid for the local problem (i.e. the problem corresponding to the current node in the solution process and all its descendant nodes) only (value 1) or for the global problem (value 0).

purgeable A scalar binary value to indicate whether the solver is allowed to purge the cut if it deems it ineffective. If the value is 1, then it is allowed.

tighten A scalar binary value to indicate whether the feasibility cut should be tightened. If the value is 1, tightening is attempted.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The generated mathematical program corresponding to the *solverSession* should have been created using the function `GMP::Benders::CreateMasterProblem` (page 308).
- The *GMP* should have been created using the function `GMP::Benders::CreateSubProblem` (page 309) or the function `GMP::Instance::CreateFeasibility` (page 372).
- If the function `GMP::Benders::CreateSubProblem` (page 309) was used to create a GMP representing the dual of the Benders' subproblem then this GMP should be used as argument *GMP2*. If it represents the primal of the Benders' subproblem then first the feasibility problem should be created which then should be used as argument *GMP2*.
- The *solution* of the *GMP* is used to generate an optimality cut for the Benders' master problem (represented by *solverSession*).
- See Benders' Decomposition - Textbook Algorithm of the Language Reference for more information about the Benders' decomposition algorithm in which a single master MIP problem is solved.
- A feasibility cut $a^T x \geq b$ can be tightened to $1^T x \geq 1$ if x is a vector of binary variables and $a_i \geq b > 0$ for all i.

Example

The way `GMP::Benders::AddFeasibilityCut` (page 304) is called depends on whether the primal or dual of the Benders' subproblem was generated. In the example below we use the dual. In that case an unbounded extreme ray is used to create a feasibility cut. In this example we solve only one Benders' master problem (which is a MIP). During the solve, whenever the solver finds an integer (incumbent) solution we want to run a callback for lazy constraints. Therefore we install a callback for it.

```
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                        useDual : 1, normalizationType : 0 );

GMP::Instance::SetCallbackAddLazyConstraint( gmpM, 'LazyCallback' );

! Switch on solver option for calculating unbounded extreme ray.
GMP::Instance::SetOptionValue( gmpS, 'unbounded ray', 1 );

GMP::Instance::Solve( gmpM );
```

The callback procedure `LazyCallback` has one argument, namely `ThisSession` which is an element parameter with range *AllSolverSessions* (page 828). Inside the callback procedure we solve the Benders' subproblem. We assume that the Benders' subproblem is always unbounded. The program status of the subproblem is stored in the element parameter `ProgramStatus` with range *AllSolutionStates* (page 807). Note that the subproblem is updated before it is solved.

```
! Get MIP incumbent solution.
GMP::Solution::RetrieveFromSolverSession( ThisSession, 1 );
GMP::Solution::SendToModel( gmpM, 1 );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::Instance::Solve( gmpS );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;
if ( ProgramStatus = 'Unbounded' ) then
    GMP::SolverSession::AddBendersFeasibilityCut( ThisSession, gmpF, 1 );
endif;
```

In this example we skipped the check for optimality of the Benders' decomposition algorithm.

See also:

The routines *GMP::Benders::CreateMasterProblem* (page 308), *GMP::Benders::CreateSubProblem* (page 309), *GMP::Benders::AddFeasibilityCut* (page 304), *GMP::Benders::AddOptimalityCut* (page 307), *GMP::Instance::CreateFeasibility* (page 372) and *GMP::SolverSession::AddBendersOptimalityCut* (page 524).

GMP::SolverSession::AddBendersOptimalityCut

The procedure *GMP::SolverSession::AddBendersOptimalityCut* (page 524) generates an optimality cut for a Benders' master problem using the (dual) solution of a Benders' subproblem. The Benders' master problem must be a MIP problem. The cut is typically added as a lazy constraint in a callback during the MIP branch-and-cut search. This procedure is typically used in a Benders' decomposition algorithm in which a single master MIP problem is solved.

```
GMP::SolverSession::AddBendersOptimalityCut(
     solverSession,     ! (input) a solver session
     GMP,               ! (input) a generated mathematical program
     solution,          ! (input) a solution
     [local],           ! (optional, default 0) a scalar binary expression
     [purgeable]        ! (optional, default 0) a scalar binary expression
)
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828) representing a solver session for the Benders' master problem.

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833) representing a Benders' subproblem.

solution An integer scalar reference to a solution of *GMP2*.

local A scalar binary value to indicate whether the cut is valid for the local problem (i.e. the problem corresponding to the current node in the solution process and all its descendant nodes) only (value 1) or for the global problem (value 0).

purgeable A scalar binary value to indicate whether the solver is allowed to purge the cut if it deems it ineffective. If the value is 1, then it is allowed.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The generated mathematical program corresponding to the *solverSession* should have been created using the function *GMP::Benders::CreateMasterProblem* (page 308).
- The *GMP* should have been created using the function *GMP::Benders::CreateSubProblem* (page 309).
- The *solution* of the Benders' subproblem (represented by *GMP*) is used to generate an optimality cut for the Benders' master problem (represented by *solverSession*). More precise, the shadow prices of the constraints and the reduced costs of the variables in the Benders' subproblem are used.
- See Benders' Decomposition - Textbook Algorithm of the Language Reference for more information about the Benders' decomposition algorithm in which a single master MIP problem is solved.

Example

In the example below we solve only one Benders' master problem (which is a MIP). During the solve, whenever the solver finds an integer (incumbent) solution we want to run a callback for lazy constraints. Therefore we install a callback for it.

```
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );

gmpS := GMP::Benders::CreateSubProblem( myGMP, masterGMP, 'BendersSubProblem',
                                        0, 0 );

GMP::Instance::SetCallbackAddLazyConstraint( gmpM, 'LazyCallback' );

GMP::Instance::Solve( gmpM );
```

The callback procedure `LazyCallback` has one argument, namely `ThisSession` which is an element parameter with range *AllSolverSessions* (page 828). Inside the callback procedure we solve the Benders' subproblem. We assume that the Benders' subproblem is always feasible. The program status of the subproblem is stored in the element parameter `ProgramStatus` with range *AllSolutionStates* (page 807). Note that the subproblem is updated before it is solved.

```
! Get MIP incumbent solution.
GMP::Solution::RetrieveFromSolverSession( ThisSession, 1 );
GMP::Solution::SendToModel( gmpM, 1 );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::Instance::Solve( gmpS );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;
if ( ProgramStatus = 'Optimal' ) then
    GMP::SolverSession::AddBendersOptimalityCut( ThisSession, gmpF, 1 );
endif;
```

In this example we skipped the check for optimality of the Benders' decomposition algorithm.

See also:

The routines *GMP::Benders::CreateMasterProblem* (page 308), *GMP::Benders::CreateSubProblem* (page 309), *GMP::Benders::AddFeasibilityCut* (page 304), *GMP::Benders::AddOptimalityCut* (page 307) and *GMP::SolverSession::AddBendersFeasibilityCut* (page 522).

GMP::SolverSession::AddLinearization

The procedure *GMP::SolverSession::AddLinearization* (page 526) adds a linearization row to a solver session with respect to a solution (column level values and row marginals) of a generated mathematical program for each row in a set of nonlinear constraints of that generated mathematical program.

```
GMP::SolverSession::AddLinearization(
     solverSession,     ! (input) a solver session
     GMP,               ! (input) a generated mathematical program
     solution,          ! (input) a solution
     constraintSet,     ! (input) a set of nonlinear constraints
     [jacTol],          ! (optional) the Jacobian tolerance
     [local],           ! (optional, default 0) a scalar binary expression
     [purgeable]        ! (optional, default 0) a scalar binary expression
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution in the solution repository of *GMP*.

constraintSet A subset of *AllNonLinearConstraints* (page 824).

jacTol The Jacobian tolerance; if the Jacobian value (in absolute sense) of a column in a nonlinear row is smaller than this value, the column will not be added to the linearization of that row. The default is 1e-5.

local A scalar binary value to indicate whether the linearization is valid for the local problem (i.e. the problem corresponding to the current node in the solution process and all its descendant nodes) only (value 1) or for the global problem (value 0).

purgeable A scalar binary value to indicate whether the solver is allowed to purge the cut if it deems it ineffective. If the value is 1, then it is allowed.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure fails if one of the constraints is ranged.
- This procedure can only be called from within a `CallbackAddCut` or `CallbackAddLazyConstraint` callback procedure.
- A `CallbackAddCut` callback procedure will only be called when solving mixed integer programs with CPLEX or Gurobi. In case of Gurobi the cuts are always local even if argument *local* has value 0.
- A `CallbackAddLazyConstraint` callback procedure will only be called when solving mixed integer programs with CPLEX or Gurobi.
- Argument *purgeable* can only be used with CPLEX. If the cut is local then the cut will not be purgeable even if argument *purgeable* has value 1.

- This procedure will fail if *GMP* contains a column that is not part of the generated mathematical program corresponding to *solverSession*. A column that is part of *GMP* but not of the generated mathematical program corresponding to *solverSession* will be ignored, i.e., no coefficient for that column will be added to the linearizations.
- The formula for the linearization of a scalar nonlinear inequality $g(x, y) \leq b_j$ around the point $(x, y) = (x^0, y^0)$ is as follows.

$$g(x^0, y^0) + \bigtriangledown g(x^0, y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} \leq b_j$$

See also:

The routines *GMP::Linearization::Add* (page 425), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411) and *GMP::SolverSession::GenerateCut* (page 535).

GMP::SolverSession::AsynchronousExecute

The procedure *GMP::SolverSession::AsynchronousExecute* (page 528) invokes the solution algorithm to asynchronous solve a generated mathematical program by using a solver session.

```
GMP::SolverSession::AsynchronousExecute(
    solverSession   ! (input) a solver session
    )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure will not copy the initial solution into the solver, or copy the final solution back into solution repository or model identifiers. When you use this function you always have to explicitly call functions from the `GMP::Solution` namespace to accomplish these tasks.
- The following solvers are thread-safe and can be used for solving multiple mathematical programs in parallel using the same solver: CPLEX, Gurobi, XA, CONOPT, and Knitro.
- The following solvers are not thread-safe but the AIMMS-solver interface is thread safe and therefore they can be used in parallel with another solver: IPOPT and SNOPT. For example, SNOPT 7.1 can be used in parallel with IPOPT but it cannot be used in parallel with SNOPT 7.1.
- The procedure *GMP::SolverSession::AsynchronousExecute* (page 528) cannot be used by the following solvers: BARON, CBC, ODH-CPLEX, and PATH.
- Calling *GMP::SolverSession::AsynchronousExecute* (page 528) inside a callback procedure is not allowed.

- The procedure *GMP::SolverSession::AsynchronousExecute* (page 528) cannot be used if an external function is used in a constraint.
- The procedures *GMP::SolverSession::WaitForCompletion* (page 556) and *GMP::SolverSession::WaitForSingleCompletion* (page 557) can be used to let AIMMS wait until one or more asynchronous executing solver sessions are finished.
- Normal solve statements will be ignored during an asynchronous execution of a solver session.
- Sensitivity ranges will not be calculated during an asynchronous solve.
- This procedure does not create a listing file but you can use the procedure *GMP::Solution::ConstraintListing* (page 480) for that.

See also:

The routines *GMP::Instance::Copy* (page 364), *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::ExecutionStatus* (page 531) *GMP::SolverSession::Interrupt* (page 553), *GMP::SolverSession::WaitForCompletion* (page 556), *GMP::SolverSession::WaitForSingleCompletion* (page 557), *GMP::Solution::ConstraintListing* (page 480) and *GMP::Solver::GetAsynchronousSessionsLimit* (page 514).

GMP::SolverSession::CreateProgressCategory

The function *GMP::SolverSession::CreateProgressCategory* (page 529) creates a new progress category for a solver session. This progress category can be used to display solver (session) related information in the Progress Window.

There are three levels of progress categories for solver information. By default all solver progress will be displayed in the general AIMMS progress category for solver progress. If a progress category was created for the GMP with procedure *GMP::Instance::CreateProgressCategory* (page 377), then all solver progress related to that GMP will by default be displayed in the solver progress category of the GMP. For displaying solver session progress in a separated category the function *GMP::SolverSession::CreateProgressCategory* (page 529) can be used.

```
GMP::SolverSession::CreateProgressCategory(
     solverSession,     ! (input) a solver session
     [Name],            ! (optional) a string expression
     [Size]             ! (optional) an integer expession
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Name A string that holds the name of the progress category.

Size The number of lines in the progress category. The default is 0 meaning that the size of the progress window will be automatically adjusted to the number of progress lines used by the solver.

Return Value

The function returns an element in the set *AllProgressCategories* (page 834).

Note:

- If the *Name* argument is not specified then the name of the solver session will be used to name the element in the set *AllProgressCategories* (page 834).
- The information displayed in the solver session progress window can be controlled by using the procedures *GMP::ProgressWindow::DisplayLine* (page 436) and *GMP::ProgressWindow::FreezeLine* (page 440).
- A progress category created before for the solver session will be deleted.
- The procedure *GMP::ProgressWindow::Transfer* (page 440) can be used to share a progress category among several solver sessions.

See also:

The routines `GMP::ProgressWindow::CreateProgressCategory`, *GMP::ProgressWindow::DeleteCategory* (page 436), *GMP::ProgressWindow::DisplayLine* (page 436), *GMP::ProgressWindow::FreezeLine* (page 440), *GMP::ProgressWindow::UnfreezeLine* (page 441) and *GMP::ProgressWindow::Transfer* (page 440).

GMP::SolverSession::Execute

The procedure *GMP::SolverSession::Execute* (page 530) invokes the solution algorithm to solve the mathematical program for which it had been generated.

```
GMP::SolverSession::Execute(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure will not copy the initial solution into the solver, or copy the final solution back into solution repository or model identifiers. When you use this function you always have to explicitly call functions from the `GMP::Solution` namespace to accomplish these tasks.
- This procedure does not create a listing file but you can use the procedure *GMP::Solution::ConstraintListing* (page 480) for that.

See also:

The routines *GMP::Instance::CreateSolverSession* (page 378), *GMP::Instance::Solve* (page 424), *GMP::SolverSession::AsynchronousExecute* (page 528) and *GMP::Solution::ConstraintListing* (page 480).

GMP::SolverSession::ExecutionStatus

The function *GMP::SolverSession::ExecutionStatus* (page 531) returns the execution status of a solver session.

```
GMP::SolverSession::ExecutionStatus(
     solverSession     ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

An element in the set *AllExecutionStatuses* (page 798). This set contains the following elements:

- NotStarted,
- Pending,
- Running,
- Interrupted,
- Finished.

See also:

The routines *GMP::SolverSession::AsynchronousExecute* (page 528), *GMP::SolverSession::Interrupt* (page 553), *GMP::SolverSession::WaitForCompletion* (page 556) and *GMP::SolverSession::WaitForSingleCompletion* (page 557).

GMP::SolverSession::GenerateBinaryEliminationRow

The procedure *GMP::SolverSession::GenerateBinaryEliminationRow* (page 531) adds a binary row to a solver session which will eliminate a binary solution.

```
GMP::SolverSession::GenerateBinaryEliminationRow(
     solverSession,    ! (input) a solver session
     solution,         ! (input) a solution
     branch            ! (input) a scalar value
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

solution An integer scalar reference to a solution.

branch An integer scalar reference to a branch. Value should be either 1 or 2.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure will fail if the GMP corresponding to the solver session does not have model type MIP.
- This procedure can only be called from within a `CallbackBranch`, `CallbackAddCut` or `CallbackAddLazyConstraint` callback procedure.
- The *branch* argument will be ignored if this procedure is called from within a `CallbackAddCut` or `CallbackAddLazyConstraint` callback procedure.
- Every call to *GMP::SolverSession::GenerateBinaryEliminationRow* (page 531) adds the following row:

$$\sum_{i \in S_0} x_i - \sum_{i \in S_1} x_i \geq 1 - |S_1|$$

where S_0 defines the set of binary columns whose level values equals 0 and S_1 the set of binary columns whose level values equals 1.

Example

The procedure *GMP::SolverSession::GenerateBinaryEliminationRow* (page 531) can be used to enforce a MIP solver to branch a node that would have been fathomed otherwise. We can achieve this by installing a branching callback using procedure *GMP::Instance::SetCallbackBranch* (page 412) and adding the following code to the callback procedure:

```
! Get LP solution at the current node.

GMP::Solution::RetrieveFromSolverSession(ThisSession,1);

! Get the number of nodes that the MIP solver wants to create from the
! current branch.

NrBranches := GMP::SolverSession::GetNumberOfBranchNodes(ThisSession);

if ( NrBranches = 0 ) then

    ! The LP solution at the current node appears to be integer feasible.
    ! We enforce the MIP solver to branch the current node by creating a
    ! branch containing one constraint that cuts off this LP solution.

    GMP::SolverSession::GenerateBinaryEliminationRow(ThisSession,1,1);
```

(continues on next page)

(continued from previous page)

```
endif;

return 1;
```

Here 'ThisSession' is an input argument of the callback procedure and a scalar element parameter into the set *AllSolverSessions* (page 828).

See also:

The routines *GMP::Instance::AddIntegerEliminationRows* (page 359), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411) and *GMP::SolverSession::GetNumberOfBranchNodes* (page 549).

GMP::SolverSession::GenerateBranchLowerBound

The procedure *GMP::SolverSession::GenerateBranchLowerBound* (page 533) specifies the lower bound change of a column in a branch to be taken from the current node during MIP branch-and-cut.

```
GMP::SolverSession::GenerateBranchLowerBound(
     solverSession,     ! (input) a solver session
     column,            ! (input) a scalar reference
     bound,             ! (input) a numerical expression
     branch             ! (input) a branch number
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

column A scalar reference to an existing column in the model.

bound The value assigned to the lower bound change of the column in the branch.

branch An integer scalar reference to the branch number. It should be equal to 1 or 2.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- A branch can be specified by adding multiple bound changes and rows (with *GMP::SolverSession::GenerateBranchRow* (page 534)) to the node problem.
- This procedure can only be called from within a `CallbackBranch` callback procedure.
- A `CallbackBranch` callback procedure will only be called when solving mixed integer programs with CPLEX.

See also:

The procedures *GMP::Instance::SetCallbackBranch* (page 412), *GMP::SolverSession::GenerateBranchUpperBound* (page 534) and *GMP::SolverSession::GenerateBranchRow* (page 534).

GMP::SolverSession::GenerateBranchRow

The procedure *GMP::SolverSession::GenerateBranchRow* (page 534) adds a row to a branch to be taken from the current node during MIP branch-and-cut.

```
GMP::SolverSession::GenerateBranchRow(
     solverSession,     ! (input) a solver session
     row,               ! (input) a scalar reference
     branch             ! (input) a branch number
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

row A scalar reference to an existing row in the model.

branch An integer scalar reference to the branch number. It should be equal to 1 or 2.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- A branch can be specified by adding multiple rows and bound changes (with *GMP::SolverSession::GenerateBranchLowerBound* (page 533) and *GMP::SolverSession::GenerateBranchUpperBound* (page 534)) to the node problem.
- This procedure can only be called from within a `CallbackBranch` callback procedure.
- A `CallbackBranch` callback procedure will only be called when solving mixed integer programs with CPLEX.

See also:

The procedures *GMP::Instance::SetCallbackBranch* (page 412), *GMP::SolverSession::GenerateBranchLowerBound* (page 533) and *GMP::SolverSession::GenerateBranchUpperBound* (page 534).

GMP::SolverSession::GenerateBranchUpperBound

The procedure *GMP::SolverSession::GenerateBranchUpperBound* (page 534) specifies the upper bound change of a column in a branch to be taken from the current node during MIP branch-and-cut.

```
GMP::SolverSession::GenerateBranchUpperBound(
     solverSession,     ! (input) a solver session
     column,            ! (input) a scalar reference
     bound,             ! (input) a numerical expression
     branch             ! (input) a branch number
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

column A scalar reference to an existing column in the model.

bound The value assigned to the upper bound change of the column in the branch.

branch An integer scalar reference to the branch number. It should be equal to 1 or 2.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- A branch can be specified by adding multiple bound changes and rows (with *GMP::SolverSession::GenerateBranchRow* (page 534)) to the node problem.
- This procedure can only be called from within a `CallbackBranch` callback procedure.
- A `CallbackBranch` callback procedure will only be called when solving mixed integer programs with CPLEX.

See also:

The procedures *GMP::Instance::SetCallbackBranch* (page 412), *GMP::SolverSession::GenerateBranchLowerBound* (page 533) and *GMP::SolverSession::GenerateBranchRow* (page 534).

GMP::SolverSession::GenerateCut

The procedure *GMP::SolverSession::GenerateCut* (page 535) adds a cut to the LP subproblem of the current node during MIP branch-and-cut. It can also be used to add a lazy constraint inside a callback for adding lazy constraints.

```
GMP::SolverSession::GenerateCut(
     solverSession,     ! (input) a solver session
     row,               ! (input) a scalar reference
     [local],           ! (optional, default 0) a scalar binary expression
     [purgeable]        ! (optional, default 0) a scalar binary expression
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

row A scalar reference to a row in the model.

local A scalar binary value to indicate whether the cut is valid for the local problem (i.e. the problem corresponding to the current node in the solution process and all its descendant nodes) only (value 1) or for the global problem (value 0).

purgeable A scalar binary value to indicate whether the solver is allowed to purge the cut if it deems it ineffective. If the value is 1, then it is allowed.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure can only be called from within a `CallbackAddCut` or `CallbackAddlazyConstraint` callback procedure.
- A `CallbackAddCut` callback procedure will only be called when solving mixed integer programs with CPLEX, Gurobi or ODH-CPLEX. In case of Gurobi the cuts are always local even if argument *local* has value 0.
- A `CallbackAddLazyConstraint` callback procedure will only be called when solving mixed integer programs with CPLEX or Gurobi.
- Argument *purgeable* can only be used with CPLEX. If the cut is local then the cut will not be purgeable even if argument *purgeable* has value 1.
- This procedure can also be used for MIQP and MIQCP problems.

Example

We have a math program, with variables `x(v1)` and `y(v1,v2)`,for which we want add certain cuts, namely triangle cut and triangle clique constraints, during the solve. For that we use a callback procedure that is called whenever the MIP solver finds a new fractional solution (typically after solving a subproblem in the branch-and-bound algorithm). The cut callback procedure can be implemented as follows.

```
! Get fractional solution from solver and pass it to the AIMMS identifiers.

myGMP := GMP::SolverSession::GetInstance( solvSess );

GMP::Solution::RetrieveFromSolverSession( solvSess, 1 );
GMP::Solution::SendToModel( myGMP, 1 );

! Find violated triangle cut and triangle clique constraints and pass them
! as cuts to the MIP solver.

for ( (v1,v2,v3) | v1 < v2 and v2 < v3 ) do
    ! Triangle Cut.
```

(continues on next page)

(continued from previous page)

```
    if ( x(v1) + x(v2) + x(v3) - y(v1,v2) - y(v1,v3) - y(v2,v3) > 1 ) then
        GMP::SolverSession::GenerateCut( solvSess, Triangle_Cut(v1,v2,v3) );
    endif;

    ! Triangle Clique.

    if ( y(v1,v2) + y(v1,v3) - x(v1) - y(v2,v3) > 0 ) then
        GMP::SolverSession::GenerateCut( solvSess, Triangle_Clique(v1,v2,v3) );
    endif;
endfor;

return 1;
```

Here 'solvSess' is an input argument of the callback procedure and a scalar element parameter into the set *AllSolverSessions* (page 828). And 'myGMP' is a scalar element parameter into the set *AllGeneratedMathematicalPrograms* (page 833), defined as a local parameter of the callback procedure.

See also:

The procedures *GMP::Instance::SetCallbackAddCut* (page 410) and *GMP::Instance::SetCallbackAddLazyConstraint* (page 411). See Managing Generated Mathematical Program Instances of the Language Reference for more details on how to install a callback procedure to add cuts.

GMP::SolverSession::GetAttributeValue

The function *GMP::Instance::GetAttributeValue* (page 389) can be used to get the value of certain solver attributes, related to an executed solver session.

```
GMP::SolverSession::GetAttributeValue(
     solverSession,  ! (input) a solver session
     attr,           ! (input) a string expression
     [objNo]         ! (input, optional) an integer expression
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

attr A string that holds the name of the attribute.

objNo A nonnegative scalar value specifying the objective number in a multi-objective optimization problem. The default value is 0.

Return Value

In case of success, the function returns the attribute value. Otherwise it returns `UNDF`.

Note:

- The attribute name is case-insensitive.
- This function is only supported by CPLEX and Gurobi.
- This function can be called inside an incumbent, lazy, cuts or heuristic callback procedure, but only for Gurobi.

Attributes

The table below shows the model attributes supported by CPLEX and Gurobi.

Attribute	CPLEX	Gurobi	Description
ObjCon	✓	✓	Objective constant/offset
NumStart	✓	✓	Number of MIP starts (MIP only)
NumObj	✓	✓	Number of objectives

The table below shows the solution attributes supported by CPLEX and Gurobi.

Attribute	CPLEX	Gurobi	Description
ObjVal	✓	✓	Objective value
ObjBound	✓	✓	Best bound (MIP only)
MIPGap	✓	✓	Relative MIP gap (MIP only)
Runtime	✓	✓	Runtime in seconds
Work	✓	✓	Work or deterministic time
MaxMemUsed		✓	Maximum amount (in GB) of memory used
MemUsed		✓	Current amount (in GB) of memory used
SolCount		✓	Number of solutions (MIP only)
IterCount	✓	✓	Number of simplex iterations
BarIterCount	✓	✓	Number of barrier iterations
NodeCount	✓	✓	Number of branch-and-cut nodes (MIP only)
OpenNodeCount	✓	✓	Number of open branch-and-cut nodes (MIP only)
NodeInt	✓		Node number of the best solution (MIP only)

The table below shows the solution quality attributes supported by CPLEX and Gurobi.

Attribute	CPLEX	Gurobi	Description
MaxVio	✓	✓	Maximum unscaled violation
IntVio	✓	✓	Maximum integrality violation
Kappa	✓	✓	Estimated condition number
KappaExact	✓	✓	Exact condition number
KappaMax	✓		Highest condition number (MIP only)
KappaAttention	✓		Score of numerical difficulties (MIP only)
KappaStable	✓		Percentage of stable simplex bases (MIP only)
KappaSuspicious	✓		Percentage of suspicious simplex bases (MIP only)
KappaUnstable	✓		Percentage of unstable simplex bases (MIP only)
KappaIllposed	✓		Percentage of ill-posed simplex bases (MIP only)

The table below shows the attributes for multi-objective optimization supported by CPLEX and Gurobi. These attribites can be used to retrieve model or solution information for each subproblem solved, as specified by the *objNo* argument.

Attribute	CPLEX	Gurobi	Description
ObjNPriority	✓	✓	Objective priority
ObjNCon		✓	Objective constant/offset
ObjVal/ObjNVal	✓	✓	Objective value
ObjBound	✓		Best bound (MIP only)
MIPGap	✓		Relative MIP gap (MIP only)
Runtime	✓		Runtime in seconds
Work	✓		Work or deterministic time
Status	✓		Solution status
NodeCount	✓		Number of branch-and-cut nodes (MIP only)
OpenNodeCount	✓		Number of open branch-and-cut nodes (MIP only)
IterCount	✓		Number of simplex iterations
BarIterCount	✓		Number of barrier iterations

For Gurobi also other model and solution attributes are supported. For a complete list; see: Model attributes. Attributes with type 'string' are not supported by this function.

The table below shows the attributes that can be called from inside an incumbent, lazy, cuts or heuristic callback procedure, but only for Gurobi.

Attribute	Description
ObjBest	Current best objective
ObjBound	Current best objective bound
NodeCount	Current explored node count
SolCount	Current number of solutions found
Phase	Current phase in the MIP solution
ObjVal	Objective value for new solution

Example

In the example below two multi-objectives are specified, each with its own priority. Therefore two subproblems are solved; one for each objective. Below we retrieve the (relative) MIP gap for both subproblems. We also retrieve the total runtime by the solver.

```
myGMP := GMP::Instance::Generate( MP );

GMP::Column::SetAsMultiObjective( myGMP, TotalDist, 2, 1.0 );
GMP::Column::SetAsMultiObjective( myGMP, TotalTime, 1, 1.0 );

solSes := GMP::Instance::CreateSolverSession( myGMP 0;

GMP::SolverSession::Execute( solSes );

GMP::Solution::RetrieveFromSolverSession( solSes, 1 );
GMP::Solution::SendToModel( myGMP, 1 );

gap1 := GMP::SolverSession::GetAttributeValue( solSes, "MIPGap", 1 );
gap2 := GMP::SolverSession::GetAttributeValue( solSes, "MIPGap", 2 );

runtime := GMP::SolverSession::GetAttributeValue( solSes, "Runtime" );
```

See also:

The routines *GMP::Instance::CreateSolverSession* (page 378), *GMP::Instance::Generate* (page 384), *GMP::SolverSession::Execute* (page 530), *GMP::Column::SetAsMultiObjective* (page 337) and *GMP::Instance::GetAttributeValue* (page 389).

GMP::SolverSession::GetBestBound

The function *GMP::SolverSession::GetBestBound* (page 540) returns the best known bound for a solver session.

```
GMP::SolverSession::GetBestBound(
     solverSession     ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

In case of success, the best known bound. Otherwise it returns UNDF.

Note:

- This function can be used for GMPs with model type MIP, MIQP or MIQCP.
- This function can also be used for GMPs solved with BARON, regardless of the model type.
- This function can also be used for GMPs with model type QP that are solved with CPLEX, if the CPLEX option *Solution Target* is set to 'Search for global optimum'.
- This function can also be used for GMPs with model type QP, QCP, NLP or MINLP that are solved with GUROBI.
- For multi-objective models, the best bound refers to the (blended) objective with the highest priority.

See also:

The routines *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::GetObjective* (page 549), *GMP::SolverSession::GetIterationsUsed* (page 545), *GMP::SolverSession::GetMemoryUsed* (page 545) and *GMP::SolverSession::GetTimeUsed* (page 552).

GMP::SolverSession::GetCallbackInterruptStatus

The function *GMP::SolverSession::GetCallbackInterruptStatus* (page 541) returns the type of the last callback function that had been called during a specific solver session.

```
GMP::SolverSession::GetCallbackInterruptStatus(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

An element in the set *AllSolverInterrupts* (page 807).

Note: When the solver session has not yet been executed, the empty element will be returned.

See also:

The procedures *GMP::SolverSession::Execute* (page 530), *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch*

(page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackIncumbent* (page 415), *GMP::Instance::SetCallbackIterations* (page 416), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackStatusChange* (page 417) and *GMP::Instance::SetCallbackTime* (page 417).

GMP::SolverSession::GetCandidateObjective

The function *GMP::SolverSession::GetCandidateObjective* (page 542) returns the objective value of a candidate solution during MIP optimization from within a candidate or lazy constraint callback.

```
GMP::SolverSession::GetCandidateObjective(
     solverSession     ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

In case of success, the objective value at the current node. Otherwise it returns `UNDF`.

Note:

- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can only be used inside a **candidate** or **lazy constraint** callback.
- The procedure *GMP::Solution::RetrieveFromSolverSession* (page 502) can be used to retrieve a candidate solution inside a **candidate** or **lazy constraint** callback.
- This function is only supported by CPLEX and Gurobi. Please note that the **candidate** callback is not supported by Gurobi.

See also:

The routines *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackCandidate* (page 413) and *GMP::Solution::RetrieveFromSolverSession* (page 502).

GMP::SolverSession::GetIIS

The procedure *GMP::SolverSession::GetIIS* (page 542) returns an irreducible infeasible set (IIS) for an infeasible math program, by returning the numbers of the rows and columns that are part of the IIS.

```
GMP::SolverSession::GetIIS(
     solverSession,     ! (input) a solver session
     rowSet,            ! (output) a subset of Integers
     colSet             ! (output) a subset of Integers
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

rowSet A subset of the set *Integers* (page 812), representing a set of row numbers.

colSet A subset of the set *Integers* (page 812), representing a set of column numbers.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure is only supported by BARON, CPLEX and Gurobi. In case of BARON, the BARON option `Compute IIS` should be set to a non-default value.
- Calculating the IIS procedure can be time consuming, especially if the model contains binary or integer variables.

Example

To use *GMP::SolverSession::GetIIS* (page 542) we declare the following identifiers (in ams format):

```
ElementParameter myGMP {
    Range: AllGeneratedMathematicalPrograms;
}
ElementParameter session {
    Range: AllSolverSessions;
}
ElementParameter ProgramStatus {
    Range: AllSolutionStates;
}
Set RowSet {
    SubsetOf: Integers;
    Index: rr;
}
Set ColSet {
    SubsetOf: Integers;
    Index: cc;
```

(continues on next page)

(continued from previous page)

```
}
StringParameter name;
```

To retrieve the IIS, and print the rows and columns that are part of the IIS, we can use:

```
myGMP := GMP::Instance::Generate( MP );
session := GMP::Instance::CreateSolverSession( myGMP );

GMP::SolverSession::Execute( session );

ProgramStatus := GMP::SolverSession::GetProgramStatus( session );

if ( ProgramStatus = 'Infeasible' or ProgramStatus = 'InfeasibleOrUnbounded' )␣
↪then
    GMP::SolverSession::GetIIS( session, RowSet, ColSet );

    for ( rr ) do
        name := GMP::Row::GetName( myGMP, rr );
        display name;
    endfor;

    for ( cc ) do
        name := GMP::Column::GetName( myGMP, cc );
        display name;
    endfor;
else
    GMP::Solution::RetrieveFromSolverSession( session, 1 );
    GMP::Solution::SendToModel( myGMP, 1 );
endif;

GMP::Instance::Delete( myGMP );
```

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Instance::CreateSolverSession* (page 378), *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::GetProgramStatus* (page 551), *GMP::Column::GetName* (page 332) and *GMP::Row::GetName* (page 462).

GMP::SolverSession::GetInstance

The function *GMP::SolverSession::GetInstance* (page 544) returns the generated mathematical program that was used to create a solver session.

```
GMP::SolverSession::GetInstance(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

An element in the set *AllGeneratedMathematicalPrograms* (page 833).

See also:

The routines *GMP::Instance::Generate* (page 384) and *GMP::Instance::CreateSolverSession* (page 378).

GMP::SolverSession::GetIterationsUsed

The function *GMP::SolverSession::GetIterationsUsed* (page 545) returns the number of iterations used by a solver session.

```
GMP::SolverSession::GetIterationsUsed(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The number of iterations used by the solver session.

See also:

The routines *GMP::SolverSession::Execute* (page 530), *GMP::Instance::SetIterationLimit* (page 420), *GMP::SolverSession::GetMemoryUsed* (page 545) and *GMP::SolverSession::GetTimeUsed* (page 552).

GMP::SolverSession::GetMemoryUsed

The function *GMP::SolverSession::GetMemoryUsed* (page 545) returns the amount of memory used by the solver session.

During a solve this function returns the current amount of memory used by the solver. After the solve, this function returns the peak memory used by the solver.

```
GMP::SolverSession::GetMemoryUsed(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The amount of megabytes used to execute a solver session.

Note:

- This function should be called inside a callback procedure to retrieve the current amount of memory used by the solver during a solve.
- During a solve, the memory used by the solver can fluctuate.
- For CPLEX and Gurobi, AIMMS calculates the memory in use based on the virtual memory used by the process. This approach is not reliable for asynchronous solver sessions.

See also:

The routines *GMP::Instance::SetCallbackIterations* (page 416), *GMP::Instance::SetCallbackTime* (page 417), *GMP::Instance::SetMemoryLimit* (page 421), *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::GetIterationsUsed* (page 545) and *GMP::SolverSession::GetTimeUsed* (page 552).

GMP::SolverSession::GetNodeNumber

The function *GMP::SolverSession::GetNodeNumber* (page 546) returns the number of the current node during MIP optimization from within a node callback.

```
GMP::SolverSession::GetNodeNumber(
    solverSession   ! (input) a solver session
    )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The number of the node for which the callback is called. It returns -1 if this function is not called inside a solver callback, or if it is not supported by the solver.

Note:

- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can only be used inside a **branch**, **candidate**, **cut** or **heuristic** callback.
- This function is only supported by CPLEX.
- The root node in a branch-and-bound tree gets number 0.

See also:

The routines *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414) and *GMP::SolverSession::GetNodesUsed* (page 548).

GMP::SolverSession::GetNodeObjective

The function *GMP::SolverSession::GetNodeObjective* (page 547) returns the objective value for the subproblem at the current node during MIP optimization from within a node callback.

```
GMP::SolverSession::GetNodeObjective(
     solverSession     ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

In case of success, the objective value at the current node. Otherwise it returns `UNDF`.

Note:

- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can only be used inside a **branch**, **cut** or **heuristic** callback.
- The procedure *GMP::Solution::RetrieveFromSolverSession* (page 502) can be used to retrieve the node solution inside a **branch**, **cut** or **heuristic** callback.
- This function is only supported by CPLEX, however it is not supported if the CPLEX option `Use generic callbacks` is switched on.

See also:

The routines *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Solution::RetrieveFromSolverSession* (page 502) and *GMP::SolverSession::GetNodeNumber* (page 546).

GMP::SolverSession::GetNodesLeft

The function *GMP::SolverSession::GetNodesLeft* (page 547) returns the number of unexplored nodes left in the branch-and-bound tree for a solver session.

```
GMP::SolverSession::GetNodesLeft(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The number of unexplored nodes left in the branch-and-bound tree.

Note:

- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can be used inside a **branch**, **candidate**, **cut** or **heuristic** callback.
- This function is only supported by CPLEX and Gurobi.

See also:

The routines *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::SolverSession::GetNodeNumber* (page 546) and *GMP::SolverSession::GetNodesUsed* (page 548).

GMP::SolverSession::GetNodesUsed

The function *GMP::SolverSession::GetNodesUsed* (page 548) returns the number of nodes that are processed by a solver session.

```
GMP::SolverSession::GetNodesUsed(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The number of nodes that are processed by the solver session.

Note:

- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can be used inside a **branch**, **candidate**, **cut** or **heuristic** callback.

See also:

The routines *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::SolverSession::GetNodeNumber* (page 546) and *GMP::SolverSession::GetNodesLeft* (page 547).

GMP::SolverSession::GetNumberOfBranchNodes

The function *GMP::SolverSession::GetNumberOfBranchNodes* (page 549) returns the number of nodes that the solver will create from the current branch.

```
GMP::SolverSession::GetNumberOfBranchNodes(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The number of nodes that the solver will create from the current branch.

Note:

- If the value returned equals 0, the node will be fathomed unless user-specified branches are made. That is, no child nodes are created and the node itself is discarded.
- This function has only meaning for solver sessions belonging to a GMP with type MIP, MIQP or MIQCP.
- This function can be used inside a **branch** callback.

See also:

The routines *GMP::Instance::SetCallbackBranch* (page 412).

GMP::SolverSession::GetObjective

The function *GMP::SolverSession::GetObjective* (page 549) returns the objective function value associated with a solver session.

```
GMP::SolverSession::GetObjective(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The objective function value associated with a solver session.

Note: For multi-objective models, the objective value refers to the (blended) objective with the highest priority.

See also:

The routines *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::GetBestBound* (page 540), *GMP::SolverSession::GetIterationsUsed* (page 545), *GMP::SolverSession::GetMemoryUsed* (page 545), *GMP::SolverSession::GetTimeUsed* (page 552) and *GMP::SolverSession::SetObjective* (page 554).

GMP::SolverSession::GetOptionValue

The function *GMP::SolverSession::GetOptionValue* (page 550) returns the value of a solver specific option for a solver session.

```
GMP::SolverSession::GetOptionValue(
     solverSession,    ! (input) a solver session
     OptionName        ! (input) a scalar string expression
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

OptionName A string expression holding the name of the option.

Return Value

In case of success, the function returns the current option value. Otherwise it returns UNDF.

Note: Options for which strings are displayed in the AIMMS **Options** dialog box, are also represented by numerical (integer) values. To obtain the corresponding option keywords, you can use the functions OptionGetString and OptionGetKeywords.

See also:

The routines *GMP::Instance::GetOptionValue* (page 404), *GMP::Instance::SetOptionValue* (page 421), *GMP::SolverSession::SetOptionValue* (page 555), *OptionGetString* (page 742) and *OptionGetKeywords* (page 742).

GMP::SolverSession::GetProgramStatus

The function *GMP::SolverSession::GetProgramStatus* (page 551) returns the program status of the last execution of a solver session.

```
GMP::SolverSession::GetProgramStatus(
     solverSession    ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

An element in the set *AllSolutionStates* (page 807).

See also:

The routines *GMP::SolverSession::Execute* (page 530) and *GMP::SolverSession::GetSolverStatus* (page 552).

GMP::SolverSession::GetSolver

The function *GMP::SolverSession::GetSolver* (page 551) returns the solver belonging to a solver session.

```
GMP::SolverSession::GetSolver(
     solverSession     ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The solver belonging to a solver session as an element of *AllSolvers* (page 787).

Note: Which solver is assigned to the solver session is determined by the routines *GMP::Instance::CreateSolverSession* (page 378) and *GMP::Instance::SetSolver* (page 423). Note that if the *Solver* argument of *GMP::Instance::CreateSolverSession* (page 378) is used then it overrules *GMP::Instance::SetSolver* (page 423).

See also:

The routines *GMP::Instance::CreateSolverSession* (page 378) and *GMP::Instance::SetSolver* (page 423).

GMP::SolverSession::GetSolverStatus

The function *GMP::SolverSession::GetSolverStatus* (page 552) returns the solver status of the last execution of a solver session.

```
GMP::SolverSession::GetSolverStatus(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

An element in the set *AllSolutionStates* (page 807).

See also:

The routines *GMP::SolverSession::Execute* (page 530) and *GMP::SolverSession::GetProgramStatus* (page 551).

GMP::SolverSession::GetTimeUsed

The function *GMP::SolverSession::GetTimeUsed* (page 552) returns the elapsed time (in 1/100th seconds) needed to execute a solver session.

```
GMP::SolverSession::GetTimeUsed(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The number of 1/100th seconds used to execute a solver session.

See also:

The routines *GMP::Instance::SetTimeLimit* (page 424), *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::GetIterationsUsed* (page 545) and *GMP::SolverSession::GetMemoryUsed* (page 545).

GMP::SolverSession::Interrupt

The procedure *GMP::SolverSession::Interrupt* (page 553) interrupts a solver session that is (asynchronous) executing.

```
GMP::SolverSession::Interrupt(
     solverSession,    ! (input) a solver session
     [timeout]         ! (optional) timeout interval
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

timeout A scalar value indicating the time-out interval (in seconds). The default value is 600.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This interrupt procedure will wait until the solver session is successfully interrupted or the time-out interval elapses.

- This procedure can also be called for a solver session that is not asynchronous executing. In that case the *timeout* argument will be ignored.

See also:

The routines *GMP::SolverSession::AsynchronousExecute* (page 528), *GMP::SolverSession::ExecutionStatus* (page 531), *GMP::SolverSession::Interrupt* (page 553), *GMP::SolverSession::WaitForCompletion* (page 556) and *GMP::SolverSession::WaitForSingleCompletion* (page 557).

GMP::SolverSession::RejectIncumbent

The procedure *GMP::SolverSession::RejectIncumbent* (page 554) rejects the integer solution found by a solver session during the solution process of a MIP model.

```
GMP::SolverSession::RejectIncumbent(
     solverSession   ! (input) a solver session
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure can only be called from within a `CallbackCandidate` callback procedure.
- A `CallbackCandidate` callback procedure will only be called when solving mixed integer programs with CPLEX.

See also:

The procedure *GMP::Instance::SetCallbackCandidate* (page 413). See Managing Generated Mathematical Program Instances of the Language Reference for more details on how to install a candidate callback procedure.

GMP::SolverSession::SetObjective

The procedure *GMP::SolverSession::SetObjective* (page 554) sets the objective value for the solution belonging to a solver session.

```
GMP::SolverSession::SetObjective(
     solverSession,    ! (input) a solver session
     Value             ! (input) a scalar numeric expression
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

Value A scalar numeric expression representing the new value to be assigned as the objective value.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The routine *GMP::SolverSession::Execute* (page 530) and *GMP::SolverSession::GetObjective* (page 549).

GMP::SolverSession::SetOptionValue

The procedure *GMP::SolverSession::SetOptionValue* (page 555) sets the value of a solver specific option for a solver session. To a solver session corresponds to one unique solver, and the option will only be set for that solver.

```
GMP::SolverSession::SetOptionValue(
     solverSession,     ! (input) a solver session
     OptionName,        ! (input) a scalar string expression
     Value              ! (input) a scalar numeric expression
     )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

OptionName A string expression holding the name of the option.

Value A scalar numeric expression representing the new value to be assigned to the option.

Return Value

The procedure returns 1 if the option exists and the value can be assigned to the option, or 0 otherwise.

Note:

- The option value of a solver specific option can also be set in other ways. The value of an option belonging to a solver session is determined by:
  - the procedure *GMP::SolverSession::SetOptionValue* (page 555) if it is called for the solver session, else
  - the procedure *GMP::Instance::SetOptionValue* (page 421) if it is called for the generated mathematical program corresponding to the solver session, else
  - the value used in the `OPTION` statement if that statement is used (see also The OPTION and PROPERTY Statements of the Language Reference), else
  - the option value in the option tree.

- Options for which strings are displayed in the AIMMS **Options** dialog box, are also represented by numerical (integer) values. To obtain the corresponding option keywords, you can use the functions `OptionGetString` and `OptionGetKeywords`.

See also:

The routines *GMP::Instance::GetOptionValue* (page 404), *GMP::Instance::SetOptionValue* (page 421), *GMP::SolverSession::GetOptionValue* (page 550), *OptionGetString* (page 742) and *OptionGetKeywords* (page 742).

GMP::SolverSession::Transfer

The procedure *GMP::SolverSession::Transfer* (page 556) can be used to transfer a solver session from its current GMP to another similar GMP. Both GMPs should be created from the same symbolic math program.

Currently this procedure is only supported for stochastic Benders decomposition.

```
GMP::SolverSession::Transfer(
    solverSession,  ! (input) a solver session
    GMP             ! (input) a generated mathematical program
    )
```

Arguments

solverSession An element in the set *AllSolverSessions* (page 828).

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: If each GMP has its own solver session then more memory is required which might not be available for large models or if many GMPs are used. To save memory this procedure can be used since it allows similar GMPs to share one solver session. After transfering a solver session to a *GMP*, only the differences between the old and new GMP will be passed as updates to the solver.

See also:

The routines *GMP::Instance::CreateSolverSession* (page 378), *GMP::Instance::GenerateStochasticProgram* (page 387) and *GMP::Stochastic::BendersFindReference* (page 560).

GMP::SolverSession::WaitForCompletion

The procedure *GMP::SolverSession::WaitForCompletion* (page 556) has a set of objects as its input. The set of objects may contain solver sessions that are asynchronous executing and events. This procedure lets AIMMS wait until all the solver sessions have completed their asynchronous execution and all the events get activated.

```
GMP::SolverSession::WaitForCompletion(
     solSesSet           ! (input) a set of objects
     )
```

Arguments

solSesSet A subset of *AllSolverSessionCompletionObjects* (page 828).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: This procedure ignores solver sessions that are not asynchronous executing but using the procedure *GMP::SolverSession::Execute* (page 530).

See also:

The routines *GMP::Event::Create* (page 357), *GMP::Event::Set* (page 358), *GMP::SolverSession::AsynchronousExecute* (page 528), *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::ExecutionStatus* (page 531), *GMP::SolverSession::Interrupt* (page 553) and *GMP::SolverSession::WaitForSingleCompletion* (page 557).

GMP::SolverSession::WaitForSingleCompletion

The routine *GMP::SolverSession::WaitForSingleCompletion* (page 557) has a set of objects as its input. The set of objects may contain solver sessions that are asynchronous executing and events. This routine lets AIMMS waits until one of the solver sessions has completed its asynchronous execution or one of the events gets activated, and it returns the completed object.

```
GMP::SolverSession::WaitForSingleCompletion(
     Objects           ! (input) a set of objects
     )
```

Arguments

Objects A subset of *AllSolverSessionCompletionObjects* (page 828).

Return Value

An element in the set *AllSolverSessionCompletionObjects* (page 828).

Note:

- This routine ignores solver sessions that are not asynchronous executing but using the procedure *GMP::SolverSession::Execute* (page 530).
- This routine will return immediately if one of the objects is a solver session that has execution status 'Finished'.

See also:

The routines *GMP::Event::Create* (page 357), *GMP::Event::Set* (page 358), *GMP::SolverSession::AsynchronousExecute* (page 528), *GMP::SolverSession::Execute* (page 530), *GMP::SolverSession::ExecutionStatus* (page 531), *GMP::SolverSession::Interrupt* (page 553) and *GMP::SolverSession::WaitForCompletion* (page 556).

2.3.14 GMP::Stochastic Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing generated stochastic mathematical program instances:

GMP::Stochastic::AddBendersFeasibilityCut

The procedure *GMP::Stochastic::AddBendersFeasibilityCut* (page 558) adds a Benders feasibility cut to the parent of a Benders feasibility problem. (The parent of a Benders feasibility problem is the parent of the corresponding Benders problem.) It uses the dual information from a solution of the Benders feasibility problem.

```
GMP::Stochastic::AddBendersFeasibilityCut(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     cutNo           ! (input) a scalar reference
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

cutNo An integer scalar reference to a cut number.

Return Value

The procedure returns 1 on success, or 0 otherwise..

Note:

- The *GMP* should have been created by the function `GMP::Stochastic::CreateBendersFeasibilitySubproblem`.
- By using the suffix `.SubproblemFeasibilityCuts` of the associated symbolic mathematical program it is possible to refer to the row that is added by *GMP::Stochastic::AddBendersFeasibilityCut* (page 558). Let `gmpBen` be a Benders problem corresponding to the symbolic mathematical program `mp`. Then the row `mp.SubproblemFeasibilityCuts(gmpBen,lbl)` is added to the *GMP*, where `lbl` is an element in the set *AllGMPExtensions* (page 798) created by this procedure using *cutNo*.

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::AddBendersOptimalityCut* (page 559), `GMP::Stochastic::CreateBendersFeasibilitySubproblem` and *GMP::Stochastic::BendersFindReference* (page 560).

GMP::Stochastic::AddBendersOptimalityCut

The procedure *GMP::Stochastic::AddBendersOptimalityCut* (page 559) adds a Benders optimality cut to the parent of a Benders problem by using the dual information from a solution of the Benders problem.

```
GMP::Stochastic::AddBendersOptimalityCut(
     GMP,            ! (input) a generated mathematical program
     solution,       ! (input) a solution
     cutNo           ! (input) a scalar reference
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

cutNo An integer scalar reference to a cut number.

Return Value

The procedure returns 1 on success, or 0 otherwise..

Note:

- The *GMP* should have been created by the function *GMP::Stochastic::BendersFindReference* (page 560).
- By using the suffix `.SubproblemOptimalityCuts` of the associated symbolic mathematical program it is possible to refer to the row that is added by *GMP::Stochastic::AddBendersOptimalityCut* (page 559). Let `gmpBen` be a Benders problem corresponding to the symbolic mathematical program `mp`. Then the row

`mp.SubproblemOptimalityCuts(gmpBen,lbl)` is added to the *GMP*, where `lbl` is an element in the set *AllGMPExtensions* (page 798) created by this procedure using *cutNo*.

- The first time this procedure is called for a Benders problem a new column `mp.SubproblemObjectiveBound(gmpBen)` is added to the parent of the Benders problem. For this column a coefficient equal to the relative weight of the Benders problem will be added to the objective of the parent. For this column a coefficient of 1 is added to the optimality cut.

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::AddBendersFeasibilityCut* (page 558), *GMP::Stochastic::BendersFindReference* (page 560), *GMP::Stochastic::GetObjectiveBound* (page 562) and *GMP::Stochastic::GetRelativeWeight* (page 563).

GMP::Stochastic::BendersFindFeasibilityReference

The function *GMP::Stochastic::BendersFindFeasibilityReference* (page 560) returns the reference to the (feasibility) generated math program belonging to a node in the scenario tree. This generated math program represents the Benders feasibility problem for a stage and for some representive scenario in the scenario tree of a stochastic mathematical program.

```
GMP::Stochastic::BendersFindFeasibilityReference(
     GMP,               ! (input) a generated mathematical program
     stage,             ! (input) a scalar reference
     scenario           ! (input) a scenario
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

stage An integer scalar reference to a stage.

scenario An element in the set *AllStochasticScenarios* (page 834).

Return Value

An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Note:

- The function *GMP::Stochastic::CreateBendersRootproblem* (page 561) creates all Benders feasibility problems for all nodes in the scenario tree, and must be called before calling *GMP::Stochastic::BendersFindReference* (page 560).
- The *GMP* should correspond to a root node, i.e., be created by using the function *GMP::Stochastic::CreateBendersRootproblem* (page 561).

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::BendersFindReference* (page 560) and *GMP::Stochastic::CreateBendersRootproblem* (page 561).

GMP::Stochastic::BendersFindReference

The function *GMP::Stochastic::BendersFindReference* (page 560) returns the reference to the generated math program belonging to a node in the scenario tree. This generated math program represents the Benders problem for a stage and for some representive scenario in the scenario tree of a stochastic mathematical program.

```
GMP::Stochastic::BendersFindReference(
     GMP,                ! (input) a generated mathematical program
     stage,              ! (input) a scalar reference
     scenario            ! (input) a scenario
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

stage An integer scalar reference to a stage.

scenario An element in the set *AllStochasticScenarios* (page 834).

Return Value

An element in the set *AllGeneratedMathematicalPrograms* (page 833).

Note:

- The function *GMP::Stochastic::CreateBendersRootproblem* (page 561) creates all Benders problems for all nodes in the scenario tree, and must be called before calling *GMP::Stochastic::BendersFindReference* (page 560).
- The *GMP* should correspond to a root node, i.e., be created by using the function *GMP::Stochastic::CreateBendersRootproblem* (page 561).

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::BendersFindFeasibilityRefer* (page 560) and *GMP::Stochastic::CreateBendersRootproblem* (page 561).

GMP::Stochastic::CreateBendersRootproblem

The function *GMP::Stochastic::CreateBendersRootproblem* (page 561) generates a mathematical program that represents the Benders problem at the unique node at stage 1 in the scenario tree of a stochastic mathematical program, and it also creates all Benders problems for all other nodes.

This function collects all columns and rows that correspond to the unique (representive) scenario at stage 1 in the scenario tree.

```
GMP::Stochastic::CreateBendersRootproblem(
     GMP,                ! (input) a generated mathematical program
     [name]              ! (optional) a string expression
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

name A string that holds the name for the Benders problem created for *GMP* at stage 1.

Return Value

A new element in the set *AllGeneratedMathematicalPrograms* (page 833) with the name as specified by the *name* argument.

Note:

- The *GMP* should have been created by the function *GMP::Instance::GenerateStochasticProgram* (page 387).
- The generated math program belonging to the node of a Benders subproblem can be obtained by using the function *GMP::Stochastic::BendersFindReference* (page 560).
- If the *name* argument is not specified, or if it is the empty string, then the name of the *GMP*, stage 1 and the unique representive scenario at stage 1 are used to create a new element in the set *AllGeneratedMathematicalPrograms* (page 833).

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::BendersFindReference* (page 560) and *GMP::Stochastic::UpdateBendersSubproblem* (page 565). See Basic Concepts of the Language Reference for more details on scenario tree, scenarios and stages.

GMP::Stochastic::GetObjectiveBound

The function *GMP::Stochastic::GetObjectiveBound* (page 562) returns the level value of the column `mp.SubproblemObjectiveBound` in a solution of a Benders problem, where `mp` denotes the corresponding symbolic mathematical program.

```
GMP::Stochastic::GetObjectiveBound(
     GMP,           ! (input) a generated mathematical program
     solution       ! (input) a solution
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

In case of success, the level value. Otherwise it returns `UNDF`.

Note:

- The *GMP* should have been created by the function *GMP::Stochastic::BendersFindReference* (page 560).
- Initially, the column `mp.SubproblemObjectiveBound` is not part of the Benders problem but it will be added if the procedure *GMP::Stochastic::AddBendersOptimalityCut* (page 559) is called.

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::AddBendersOptimalityCut* (page 559) and *GMP::Stochastic::BendersFindReference* (page 560).

GMP::Stochastic::GetRelativeWeight

The function *GMP::Stochastic::GetRelativeWeight* (page 563) returns the relative weight of a scenario at some stage in the scenario tree belonging to a stochastic mathematical program. The weight is relative to the sum of the weights of all scenarios that have the same parent at that stage.

```
GMP::Stochastic::GetRelativeWeight(
     GMP,            ! (input) a generated mathematical program
     stage,          ! (input) a scalar reference
     scenario        ! (input) a scenario
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

stage An integer scalar reference to a stage.

scenario An element in the set *AllStochasticScenarios* (page 834).

Return Value

In case of success, the relative weight. Otherwise it returns UNDF.

Note: The *GMP* should have been created by the function *GMP::Instance::GenerateStochasticProgram* (page 387).

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387) and *GMP::Stochastic::GetRepresentativeScenario* (page 564). See Basic Concepts of the Language Reference for more details on scenario tree, scenarios and stages.

GMP::Stochastic::GetRepresentativeScenario

The function *GMP::Stochastic::GetRepresentativeScenario* (page 564) returns the representive scenario of a scenario at some stage in the scenario tree belonging to a stochastic mathematical program.

```
GMP::Stochastic::GetRepresentativeScenario(
     GMP,            ! (input) a generated mathematical program
     stage,          ! (input) a scalar reference
     scenario        ! (input) a scenario
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

stage An integer scalar reference to a stage.

scenario An element in the set *AllStochasticScenarios* (page 834).

Return Value

An element in the set *AllStochasticScenarios* (page 834).

Note: The *GMP* should have been created by the function *GMP::Instance::GenerateStochasticProgram* (page 387).

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387) and *GMP::Stochastic::GetRelativeWeight* (page 563). See Basic Concepts of the Language Reference for more details on scenario tree, scenarios and stages.

GMP::Stochastic::MergeSolution

The procedure *GMP::Stochastic::MergeSolution* (page 564) merges a solution of a Benders problem into a solution of the stochastic mathematical program belonging to the Benders problem. Only the level values of the columns are merged. The objective level value is updated by using the objective definition and the level values in the solution.

```
GMP::Stochastic::MergeSolution(
     GMP,            ! (input) a generated mathematical program
     solution1,      ! (input) a solution
     solution2,      ! (input) a solution
     [updObj]        ! (optional) a binary scalar value
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution1 An integer scalar reference to a solution of *GMP*.

solution2 An integer scalar reference to a solution of the stochastic mathematical program that belongs to *GMP*.

updObj A binary scalar indicating whether the (stochastic) objective value should be updated. Its default value is 1 which means that the objective is updated.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP* should have been created by the function *GMP::Stochastic::CreateBendersRootproblem* (page 561) or by the function *GMP::Stochastic::BendersFindReference* (page 560).
- It is most efficient to only update the objective value during the last call to *GMP::Stochastic::MergeSolution* (page 564), i.e., set *updObj* to 1 for the last call and to 0 for all preceding calls.

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::CreateBendersRootproblem* (page 561) and *GMP::Stochastic::BendersFindReference* (page 560).

GMP::Stochastic::UpdateBendersSubproblem

The procedure *GMP::Stochastic::UpdateBendersSubproblem* (page 565) updates the right hand side values of a Benders problem by using a solution of the parent Benders problem.

```
GMP::Stochastic::UpdateBendersSubproblem(
     GMP,             ! (input) a generated mathematical program
     solution         ! (input) a solution
)
```

Arguments

GMP An element in the set *AllGeneratedMathematicalPrograms* (page 833).

solution An integer scalar reference to a solution.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The *GMP* should have been created by the function *GMP::Stochastic::CreateBendersRootproblem* (page 561) or obtained by the function *GMP::Stochastic::BendersFindReference* (page 560).
- This procedure does not use the *solution* if the *GMP* belongs to the Benders problem at (the unique node at) stage 1, i.e., if it was created by the function *GMP::Stochastic::CreateBendersRootproblem* (page 561).

See also:

The routines *GMP::Instance::GenerateStochasticProgram* (page 387), *GMP::Stochastic::BendersFindReference* (page 560) and *GMP::Stochastic::CreateBendersRootproblem* (page 561).

2.3.15 GMP::Tuning Procedures and Functions

AIMMS supports the following procedures and functions for tuning models:

GMP::Tuning::SolveSingleMPS

The procedure *GMP::Tuning::SolveSingleMPS* (page 566) solves a MPS, LP or SAV file.

```
GMP::Tuning::SolveSingleMPS(
     FileName,              ! (input) scalar string expression
     Solver,                ! (input) scalar element parameter
     SolverStatus,          ! (output) scalar element parameter
     ProgramStatus,         ! (output) scalar element parameter
     Objective,             ! (output) scalar numerical parameter
     Iterations,            ! (output) scalar numerical parameter
     Nodes,                 ! (output) scalar numerical parameter
```

(continues on next page)

(continued from previous page)

```
    SolutionTime,          ! (output) scalar numerical parameter
    [SolutionFile]         ! (optional) a scalar numerical expression
)
```

Arguments

FileName The name of the file, with file format '.mps', '.lp' or '.sav', to be solved.

Solver An element in the set *AllSolvers* (page 787).

SolverStatus The solver status as an element in the set *AllSolutionStates* (page 807).

ProgramStatus The program status as an element in the set *AllSolutionStates* (page 807).

Objective The objective value returned by the solver.

Iterations The number of iterations used by the solver to solve the model.

Nodes The number of nodes used by the solver to solve the model.

SolutionTime The solution time (in seconds) used by the solver to solve the model.

SolutionFile A 0-1 value indicating whether a solution file should be created. If 1, then the solution file will be named '*FileName*.sol'. The default is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The solver will use the option settings as specified in the AIMMS project.
- This procedure is supported by the solvers CPLEX, Gurobi, CBC, ODH-CPLEX and XA. XA does not support the LP format. Only CPLEX supports the SAV format.

Example

To solve model 'mod1.mps' using CPLEX 12.10 execute:

```
GMP::Tuning::SolveSingleMPS( 'mod1.mps', 'CPLEX 12.10', SolStat, ProStat, obj,␣
↪iter,
                             nodes, soltime );
```

See also:

The routine *GMP::Tuning::TuneMultipleMPS* (page 567).

GMP::Tuning::TuneMultipleMPS

The procedure *GMP::Tuning::TuneMultipleMPS* (page 567) tunes the solver options for a set of problems represented by MPS, LP or SAV files.

```
GMP::Tuning::TuneMultipleMPS(
     DirectoryName,         ! (input) scalar string expression
     Solver,                ! (input) scalar element parameter
     FixedOptions,          ! (input) set expression
     [ApplyTunedSettings], ! (optional) scalar numerical expression
     [OptionFileName]       ! (optional) scalar string expression
)
```

Arguments

DirectoryName The name of the directory containing the problems to be tuned. All problems with file format '.mps', '.lp' or '.sav' inside the directory will be used.

Solver An element in the set *AllSolvers* (page 787).

FixedOptions A subset of the predefined set *AllOptions* (page 786), containing the set of all solver options that should *not* be tuned by the solver. For fixed options the current AIMMS project settings are used.

ApplyTunedSettings A 0-1 value indicating whether the tuned option settings should be used inside the project immediately. The default is 0.

OptionFileName The name of the options file to which the tuned options will be written. If this argument is not specified then no options file will be created.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- All solver options not in the set *FixedOptions* will be subject to tuning even if such an option is set to a non-default value inside the AIMMS project.
- Mixed problem sets are not supported, i.e., you cannot mix LP problems with MIP problems.
- The tuned options will be written to the listing file.
- The options file (if any) can be imported into the AIMMS project using the options dialog box.
- This procedure is only supported by CPLEX and Gurobi.
- Only CPLEX supports the SAV format.

Example

Assume we have a set 'FixedOptions' defined as:

```
Set FixedOptions {
    SubsetOf   :  AllOptions;
    Definition :  data { 'CPLEX 12.10::mip_search_strategy' };
}
```

Using CPLEX 12.10 we tune all '.mps', '.lp' and '.sav' problems inside the directory 'Set1' by executing:

```
GMP::Tuning::TuneMultipleMPS( "Set1", 'CPLEX 12.10', FixedOptions );
```

Note that the opion 'mip search strategy' is fixed and will not be tuned.

See also:

The routines *GMP::Tuning::SolveSingleMPS* (page 566) and *GMP::Tuning::TuneSingleGMP* (page 569).

GMP::Tuning::TuneSingleGMP

The procedure *GMP::Tuning::TuneSingleGMP* (page 569) tunes the solver options for a generated mathematical program.

```
GMP::Tuning::TuneSingleGMP(
     GMP,                  ! (input) generated mathematical program
     FixedOptions,         ! (input) set expression
     [ApplyTunedSettings], ! (optional) scalar numerical expression
     [OptionFileName]      ! (optional) scalar string expression
)
```

Arguments

GMP An element in *AllGeneratedMathematicalPrograms* (page 833).

FixedOptions A subset of the predefined set *AllOptions* (page 786), containing the set of all solver options that should *not* be tuned by the solver. For fixed options the current AIMMS project settings are used.

ApplyTunedSettings A 0-1 value indicating whether the tuned option settings should be used inside the project immediately. The default is 0.

OptionFileName The name of the options file to which the tuned options will be written. If this argument is not specified then no options file will be created.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- All solver options not in the set *FixedOptions* will be subject to tuning even if such an option is set to a non-default value inside the AIMMS project.
- This procedure does not return a solution for the GMP and therefore the model identifiers are not changed.
- The tuned options will be written to the listing file.
- The options file (if any) can be imported into the AIMMS project using the options dialog box.
- This procedure is only supported by CPLEX and Gurobi.

Example

Assume that 'MP' is a mathematical program and 'gmpMP' an element parameter with range 'AllGeneratedMathematicalPrograms'. Furthermore, we have a set 'FixedOptions' defined as:

```
Set FixedOptions {
    SubsetOf   :  AllOptions;
    Definition :  data { 'CPLEX 12.10::mip_search_strategy' };
}
```

To tune 'MP' we have to run:

```
gmpMP := GMP::Instance::Generate( MP );

GMP::Tuning::TuneSingleGMP( gmpMP, FixedOptions );
```

Here the opion 'mip search strategy' is fixed and will not be tuned (assuming we are using solver CPLEX 12.10).

See also:

The routines *GMP::Instance::Generate* (page 384), *GMP::Tuning::SolveSingleMPS* (page 566) and *GMP::Tuning::TuneMultipleMPS* (page 567).

2.4 Miscellaneous Functions

The following function can be used to add cuts during the solution process of a mixed integer program, or retrieve the current variable values during a solve statement:

2.4.1 GenerateCut

The procedure *GenerateCut* (page 570) adds a cut during the solution process of a mixed integer program.

```
GenerateCut(
     Arow,         ! (input) a scalar value
     [local]       ! (optional, default 1) a scalar binary expression
     )
```

Arguments

Arow A scalar reference to an existing row name in the model.

local A scalar binary value to indicate whether the cut is valid for the local problem (i.e. the problem corresponding to the current node in the solution process and all its descendant nodes) only (value 1) or for the global problem (value 0).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This procedure can only be called from within a `CallbackAddCut` callback procedure.
- A `CallbackAddCut` callback procedure will only be called when solving mixed integer programs with CPLEX, GUROBI or ODH-CPLEX.

See also:

See Suffices and Callbacks of the Language Reference for more details on how to install a callback procedure to add cuts.

2.4.2 RetrieveCurrentVariableValues

With the procedure *RetrieveCurrentVariableValues* (page 571) you can obtain the variable values for a given collection of variables during a running solution process. This procedure can only be called from within the context of a solver callback procedure.

```
RetrieveCurrentVariableValues(
     Variables     ! (input) a subset of AllVariables
     )
```

Arguments

Variables A subset of `AllVariables` (page 831), holding all the variables for which you want to retrieve the current values.

See also:

Solver callback procedures are discussed in full detail in Suffices and Callbacks of the Language Reference

CHAPTER

THREE

MODEL HANDLING

3.1 Model Query Functions

AIMMS supports the following functions to query the structure of the identifiers in the model:

3.1.1 AttributeToString

The function *AttributeToString* (page 573) converts a specified attribute for a given identifier to a string.

```
AttributeToString(
     IdentifierName,       ! (input) scalar element parameter
     AttributeName         ! (input) scalar element parameter
     )
```

Arguments

IdentifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which an attribute should be converted to a string.

AttributeName An element expression in the predefined set *AllAttributeNames* (page 793) specifying the attribute that should be converted to string format.

Return Value

This function returns a string representation of the attribute on success or the empty string otherwise and the predeclared identifier *CurrentErrorMessage* (page 835) contains an appropriate error message.

Note: In order to protect the intellectual property of the model developer, the string `Encrypted` is returned and the predeclared identifier *CurrentErrorMessage* (page 835) contains an appropriate error message, when the identifier is in an encrypted section of the model. There is one exception; if the procedure making the call `AttributeToString(id, attr)` is in the same component as the identifier `id`, the attribute `attr` is still returned as string. Here component is the main model or one of the libraries.

Example

Given the declaration:

```
Parameter p {
    Comment: "Hello""}
```

inside the nested module `chapterModel::sectionModelQuery::funcAttributeToString`, the code that queries the comment attribute of identifier `p` is:

```
1 _ep_p := StringToElement(AllIdentifiers,
2     "chapterModel::sectionModelQuery::funcAttributeToString::p",
3     create: 0);
4 _sp_cmt := AttributeToString(_ep_p, 'comment');
5 block where single_column_display := 1, listing_number_precision := 6 ;
6     display { _ep_p, _sp_cmt };
7 endblock ;
```

Remark:

- Lines 1-3: Note that the name of the identifier contains all prefixes of the modules it is in.

The results in the listing file:

```
_ep_p   := 'chapterModel::sectionModelQuery::funcAttributeToString::p' ;
_sp_cmt :=                                                  "Hello\n" ;
```

References

- *StringToElement* (page 66),
- *me::GetAttribute* (page 610),
- *AttributeContainsString* (page 575), and
- *AttributeLength* (page 574).

3.1.2 AttributeLength

The function *AttributeLength* (page 574) returns the length of a string representation of a specified attribute for a given identifier.

```
AttributeLength(
     IdentifierName,   ! (input) scalar element parameter
     AttributeName     ! (input) scalar element parameter
     )
```

Arguments

IdentifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which an attribute length has to be returned.

AttributeName An element expression in the predefined set *AllAttributeNames* (page 793) specifying the attribute.

Return Value

This function returns the length of a string representation of the attribute on success or zero otherwise and the predeclared identifier *CurrentErrorMessage* (page 835) contains an appropriate error message.

Example

Given the declaration:

```
Parameter p {
    Comment: "Hello""}
```

inside the nested module `chapterModel::sectionModelQuery::funcAttributeLength`, the code that queries the comment attribute of identifier `p` is:

```
1 _ep_p := StringToElement(AllIdentifiers,
2     "chapterModel::sectionModelQuery::funcAttributeLength::p",
3     create: 0);
4 _p_cmt := AttributeLength(_ep_p, 'comment');
5 block where single_column_display := 1, listing_number_precision := 6 ;
6     display { _ep_p, _p_cmt };
7 endblock ;
```

Remark:

- Lines 1-3: Note that the name of the identifier contains all prefixes of the modules it is in.

The results in the listing file:

```
_ep_p  := 'chapterModel::sectionModelQuery::funcAttributeLength::p' ;
_p_cmt :=                                                         5 ;
```

References

- *AttributeToString* (page 573),
- *AttributeContainsString* (page 575).

3.1.3 AttributeContainsString

The function *AttributeContainsString* (page 575) returns 1 if the string representation of a specified attribute for a given identifier contains the specified text (case sensitive).

```
AttributeContainsString(
     IdentifierName,        ! (input) scalar element parameter
     AttributeName          ! (input) scalar element parameter
     Key                    ! (input) scalar string expression
     )
```

Arguments

IdentifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier.

AttributeName An element expression in the predefined set *AllAttributeNames* (page 793) specifying the attribute.

Key A string specifying the text to search for.

Return Value

1 if the attribute text contains the Key string (always case sensitive), 0 otherwise. In case of failure, the return value is 0 and the predeclared identifier *CurrentErrorMessage* (page 835) contains an appropriate error message.

Example

Given the declaration:

```
Parameter p {
    Comment: "Hello""}
```

inside the nested module `chapterModel::sectionModelQuery::funcAttributeLength`, the code that queries the comment attribute of identifier `p` is:

```
1    _ep_p := StringToElement(AllIdentifiers,
2            "chapterModel::sectionModelQuery::funcAttributeLength::p",
3            create: 0);
4    _bp_loPresent := AttributeContainsString(
5            IdentifierName :  _ep_p,
6            AttributeName  :  'comment',
7            Key            :  "lo") ;
8    block where single_column_display := 1, listing_number_precision := 6 ;
9            display { _ep_p, _bp_loPresent };
10   endblock ;
```

Remark:

- Lines 1-3: Note that the name of the identifier contains all prefixes of the modules it is in.

The results in the listing file:

```
_ep_p         := 'chapterModel::sectionModelQuery::funcAttributeLength::p' ;
_bp_loPresent :=                                                        1 ;
```

References

- *AttributeToString* (page 573),
  - *AttributeLength* (page 574),
- *me::GetAttribute* (page 610).

3.1.4 CallerAttribute

The function *CallerAttribute* (page 577) returns the attribute of a node that is on the current execution stack.

```
CallerAttribute(
               Depth      ! (optional) scalar element parameter
               )
```

Arguments

Depth An numeric optional expression with default 1. The value should be in the range {1 . . . *CallerNumberOfLocations* (page 580) }. The value 1, refers to the caller of the currently running procedure.

Return Value

This function returns an element in *AllAttributeNames* (page 793).

Example

The function *CallerAttribute* (page 577) is usually part of utility code, as follows:

```
1 Procedure pr_fragmentCallerAttribute {
2     Body: {
3         _ep_callerAttribute := CallerAttribute(1);
4         display _ep_callerAttribute ;
5     }
6     ElementParameter _ep_callerAttribute {
7         Range: AllAttributeNames;
8     }
9 }
```

Which can then be used as follows:

```
Procedure pr_testCallerAttribute {
    Body: {
        pr_fragmentCallerAttribute();
    }
}
```

A call to pr_testCallerAttribute produces in the listing file:

```
_ep_callerAttribute := 'body' ;
```

For a more extended illustration of how the function *CallerAttribute* (page 577) can be part of utility code, please check the example at *CallerNumberOfLocations* (page 580).

References

- *errh::Attribute* (page 728),
- *CallerLine* (page 578),
- *CallerNode* (page 579), and
- *CallerNumberOfLocations* (page 580).

3.1.5 CallerLine

The function *CallerLine* (page 578) returns the line of a node that is on the current execution stack.

```
CallerLine(
     Depth       ! (optional) scalar element parameter
     )
```

Arguments

Depth An numeric optional expression with default 1. The value should be in the range {1 . . . *CallerNumberOfLocations* (page 580) }. The value 1, refers to the caller of the currently running procedure.

Return Value

This function returns a line number.

Example

The function *CallerLine* (page 578) is usually part of utility code, as follows:

```
Procedure pr_fragmentCallerLine {
    Body: {
        _p_callerLine := CallerLine(1);
        display _p_callerLine ;
    }
    Parameter _p_callerLine;
}
```

Which can then be used as follows:

```
Procedure pr_testCallerLine {
    Body: {
        ! Hello

        pr_fragmentCallerLine();
    }
}
```

A call to `pr_testCallerLine` produces in the listing file:

```
_p_callerLine := 3 ;
```

For a more extended illustration of how the function *CallerLine* (page 578) can be part of utility code, please check the example at *CallerNumberOfLocations* (page 580).

References

- *CallerAttribute* (page 577),
- *errh::Line* (page 735),
- *CallerNode* (page 579), and
- *CallerNumberOfLocations* (page 580).

3.1.6 CallerNode

The function *CallerNode* (page 579) returns the node that is on the current execution stack.

```
CallerNode(
     Depth      ! (optional) scalar element parameter
     )
```

Arguments

Depth An numeric optional expression with default 1. The value should be in the range {1 . . . *CallerNumberOfLocations* (page 580) }. The value 1, refers to the caller of the currently running procedure.

Return Value

This function returns an element in *AllSymbols* (page 787).

Example

The function *CallerNode* (page 579) is usually part of utility code, as follows:

```
Procedure pr_fragmentCallerNode {
    Body: {
        _ep_callerNode := CallerNode(1);
        display _ep_callerNode ;
    }
    ElementParameter _ep_callerNode {
        Range: AllSymbols;
    }
}
```

Which can then be used as follows:

```
Procedure pr_testCallerNode {
    Body: {
        ! Hello

        pr_fragmentCallerNode();
    }
    aimmsunit::TestSuite: modelQuery;
}
```

A call to `pr_testCallerNode` produces in the listing file:

```
_ep_callerNode := 'chapterModel::sectionModelQuery::funcCallerNode::pr_testCallerNode' ;
```

For a more extended illustration of how the function *CallerNode* (page 579) can be part of utility code, please check the example at *CallerNumberOfLocations* (page 580).

References

- *CallerAttribute* (page 577),
- *CallerLine* (page 578),
- *errh::Node* (page 739), and
- *CallerNumberOfLocations* (page 580).

3.1.7 CallerNumberOfLocations

The function *CallerNumberOfLocations* (page 580) returns the number of nodes on the current execution stack, not counting the current internal procedure or function.

```
CallerNumberOfLocations( )
```

Example

The following code provides the skeleton of a simple stack dump.

```
Parameter noLocs ;
Parameter aDepth ;
Parameter aLine ;
ElementParameter aNode {
    range : AllIdentifiers ;
}
ElementParameter anAttr {
    range : AllAttributeNames ;
}
File outf {
    Name: "a41t001.put";
}
Procedure reportStack {
    Body: {
        noLocs := callerNumberOfLocations();
        aDepth := 1 ;
        put outf, "Current execution stack: ", / ;
        put "depth":5, " ", "node":20, " ", "attribute":12, " ", "line":4, / ;
        put "-"*5,     " ", "-"*20,    " ", "-"*12,          " ", "-"*4,    / ;
        while aDepth <= noLocs do
            aLine := callerLine( aDepth );
            aNode := callerNode( aDepth );
            anAttr := callerAttribute( aDepth );
            put aDepth:5:0, " ", aNode:20, " ", anAttr:12, " ", aLine:4:0, " ", / ;
            aDepth += 1 ;
        endwhile ;
        putclose ;
    }
}
```

An instance of its output might be:

```
Current execution stack:
depth node                attribute    line
----- ------------------- ------------ ----
    1 work1               body            4
    2 MainExecution       body            1
```

References

- *CallerAttribute* (page 577),
- *CallerLine* (page 578),
- *CallerNode* (page 579), and
- *errh::NumberOfLocations* (page 739).

3.1.8 ConstraintVariables

The function *ConstraintVariables* (page 582) returns all the symbolic variables that are referred in a certain collection of constraints, including the variables that are referred in the definitions of these variables.

```
ConstraintVariables(
     Contraints      ! (input) a subset of AllConstraints
     )
```

Arguments

Contraints The set of constraints for which you want to retrieve the referred variables.

Note: This function operates on the compiled definition of constraints; it will skip inline variables.

Example

```
Model Main_cv {
    Variable x {
        Range: free;
    }
    Variable y {
        Range: free;
    }
    Variable z {
        Range: free;
        Property: Inline;
        Definition: x + y;
    }
    Constraint c {
        Definition: z > 0;
    }
    Set S {
        SubsetOf: AllConstraints;
        Index: i;
        Definition: data { c };
    }
    Set T {
        SubsetOf: AllVariables;
```

(continues on next page)

(continued from previous page)

```
        Index: j;
    }
    Set U {
        SubsetOf: AllVariables;
        Index: k;
    }
    Set setje {
        Index: ii;
        Definition: data { a, b };
    }
    Parameter P {
        IndexDomain: ii;
        Definition: data { a : 3,  b : 4 };
    }
    ElementParameter colPar {
        IndexDomain: ii;
        Range: AllColors;
        Definition: data { a : red,  b : yellow };
    }
    Procedure MainInitialization;
    Procedure MainExecution {
        Body: {
            T := ConstraintVariables( S );
            U := ReferencedIdentifiers( S, AllAttributeNames, recursive: 1 );
            display T, U ;
        }
    }
    Procedure MainTermination {
        Body: {
            return 1 ;
        }
    }
}
```

Running `MainExecution` will create the following listing file:

```
T := data { x, y } ;

U := data { x, y, z } ;
```

Because `z` is an inline variable.

Return Value

The function returns a subset of the set *AllVariables* (page 831), containing the variables found.

See also:

The function *VariableConstraints* (page 598) and *ReferencedIdentifiers* (page 595).

3.1.9 DeclaredSubset

The function *DeclaredSubset* (page 583) returns 1 if both subsetName and superName refer to a one-dimensional set and `subsetName` is directly or indirectly declared to be a subset of `supersetName`.

```
DeclaredSubset(
     subsetName,         ! (input) scalar element parameter
     supersetName        ! (input) scalar element parameter
     )
```

Arguments

subsetName An element expression in the predefined set *AllIdentifiers* (page 821).

supersetName An element expression in the predefined set *AllIdentifiers* (page 821).

Return Value

This function returns 1 iff `subsetName` is directly or indirectly a subset of `supersetName`. If `subsetName` or `supersetName` does not refer to a one-dimensional set, this function will return 0 without any warning or error message.

Example

With the following declarations:

```
Set MasterSet {
    Index        :  ms;
}
Set DomainSet {
    SubsetOf     :  MasterSet;
    Index        :  ds;
}
Set ActiveSet {
    SubsetOf     :  DomainSet;
    Index        :  as;
}
File outf {
    Name         :  "outf.put";
}
```

The following statements:

```
put outf ;
put "ActiveSet(=DomainSet =", DeclaredSubset('ActiveSet', 'DomainSet'):0:0,/;
put "ActiveSet(=MasterSet =", DeclaredSubset('ActiveSet', 'MasterSet'):0:0,/;
put "MasterSet(=ActiveSet =", DeclaredSubset('MasterSet', 'ActiveSet'):0:0,/;
put "MasterSet(=outf      =", DeclaredSubset('MasterSet', 'outf'     ):0:0,/;
putclose ;
```

Return the following output.

```
ActiveSet(=DomainSet =1 ! ActiveSet is directly a subset of DomainSet
ActiveSet(=MasterSet =1 ! ActiveSet is indirectly a subset of MasterSet
MasterSet(=ActiveSet =0 ! But the reverse is not true.
MasterSet(=outf      =0 ! outf isn't even a set.
```

See also:

The function *IndexRange* (page 593).

3.1.10 DomainIndex

The function *DomainIndex* (page 585) returns the `indexPosition`-th index of `identifierName` as an element in *AllIdentifiers* (page 821).

```
DomainIndex(
     identifierName,       ! (input) scalar element parameter
     indexPosition
     )        ! (input) scalar integer parameter
```

Arguments

identifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which an index should be obtained.

indexPosition An expression in the range $\{1..dim\}$ where dim is the dimension of `identifierName`.

Return Value

This function returns an element in the set *AllIdentifiers* (page 821) representing the `indexPosition` index of `identifierName`. If `identifierName` is not an indexed parameter, variable or constraint, or if `indexPosition` is outside the range $\{1..dim\}$, the empty element is returned without further warning.

Example

The following code uses the function *DomainIndex* (page 585) to obtain the indices of the index domain of a parameter:

```
put outf ;
for ( IndexParameters | IdentifierDimension( IndexParameters ) > 0 ) do
   put IndexParameters:0, "(" ;
   while loopcount <= IdentifierDimension( IndexParameters ) do
       put DomainIndex( IndexParameters, loopcount ):0 ;
       if loopCount < IdentifierDimension( IndexParameters ) then put "," ;␣
↪endif ;
   endwhile ;
   put ")", /  ;
endfor ;
putclose ;
```

A fragment of the output of this code might look as follows:

```
LowFP(f,p)
UppFP(f,p)
Supply(c)
Demand(f)
```

See also:

The functions *IdentifierDimension* (page 587), *DeclaredSubset* (page 583) and *IndexRange* (page 593).

3.1.11 GetAnnotationValues

The pocedure *GetAnnotationValues* (page 586) collects for all identifiers the annotation value for annotation AnnotationKey.

```
GetAnnotationValues(
     AnnotationKey,  ! (input)   string expression
             Values          ! (output)  string parameter indexed over AllIdentifiers
)
```

Arguments

AnnotationKey A string expression that results in a valid annotation key

Values A string parameter that is indexed over *AllIdentifiers* (page 821)

Return Value

This procedure returns 1 upon success.

Example

Given the declaration:

```
StringParameter _sp_annot {
        IndexDomain: IndexIdentifiers;
}
```

The code:

```
GetAnnotationValues(
        AnnotationKey :  "aimmsunit::TestSuite",
        Values        :  _sp_annot);
display _sp_annot('chapterModel::sectionModelQuery::funcGetAnnotationValues::pr_
↪testGetAnnotationValues');
```

Produces in the listing file:

```
_sp_annot('chapterModel::sectionModelQuery::funcGetAnnotationValues::pr_
↪testGetAnnotationValues') := "modelQuery" ;
```

See also:

The functions *DeclaredSubset* (page 583) and *DomainIndex* (page 585).

3.1.12 IdentifierAttributes

The function *IdentifierAttributes* (page 587) determines which attributes a specified identifier has.

```
IdentifierAttributes(
     IdentifierName       ! (input) scalar element parameter
     )
```

Arguments

IdentifierName An element expression specifying the identifier for which the attributes should be determined.

Return Value

This function returns a subset of *AllAttributeNames* (page 793) containing all the attributes for the specified identifier.

Example

The code:

```
1    _ep_mp := StringToElement(AllIdentifiers,
2            "chapterModel::sectionModelQuery::funcIdentifierAttributes::mp_loc",
3            create: 0);
4    _s_someAttributes := IdentifierAttributes( _ep_mp );
5    block where single_column_display := 1, listing_number_precision := 6 ;
6            display _s_someAttributes  ;
7    endblock ;
```

produces in the listing file:

```
_s_someAttributes := data
{ comment             ,
  constraints         ,
  convention          ,
  direction           ,
  objective           ,
  text                ,
  type                ,
  variables           ,
  'violation penalty' } ;
```

3.1.13 IdentifierDimension

The function *IdentifierDimension* (page 587) returns the data dimension of `identifierName`.

```
IdentifierDimension(
     identifierName)       ! (input) scalar element parameter
```

Arguments

identifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which the dimension should be obtained.

Return Value

This function returns a non-negative integer. If `identifierName` is not an identifier, an error message is issued. If `identifierName` is not an indexed parameter, variable or constraint, a 0 is returned without further warning.

Note: This function replaces the deprecated suffix *.dim* (page 863).

Example

Given the declaration of a two dimensional parameter `p_d(i_a,i_b)`, the code:

```
1 _ep_p := StringToElement(AllIdentifiers,
2     "chapterModel::sectionModelQuery::funcIdentifierDimension::p_d",
3     create: 0);
4 _p_dim := IdentifierDimension( _ep_p );
5 block where single_column_display := 1, listing_number_precision := 6 ;
6     display _p_dim ;
7 endblock ;
```

produces in the listing file:

```
_p_dim := 2 ;
```

References

- The functions *DomainIndex* (page 585) and *IndexRange* (page 593).
- Working with the Set AllIdentifiers of the Language Reference.
- The common example in Listing 3.1.

3.1.14 IdentifierElementRange

The function *IdentifierElementRange* (page 588) returns the range as a set.

```
IdentifierElementRange(
     identifierName)       ! (input) scalar element parameter
```

Arguments

identifierName An element expression in the predefined set *AllSymbols* (page 787) specifying the identifier for which the range should be obtained.

Return Value

This function returns the set, as an element in *AllSymbols* (page 787), that is the range of `identifierName` if it is element valued. If `identifierName` is not an identifier, an error message is issued. If `identifierName` is not element valued, the empty element is returned without further warning.

Example

Given the declaration:

```
Set s_a;
ElementParameter ep_d {
        Range: s_a;
}
```

the code:

```
1    _ep_p := StringToElement(AllIdentifiers,
2            "chapterModel::sectionModelQuery::funcIdentifierElementRange::ep_d",
3            create: 0);
4    _ep_s := IdentifierElementRange( _ep_p );
5    block where single_column_display := 1, listing_number_precision := 6 ;
6            display _ep_s ;
7    endblock ;
```

produces in the listing file:

```
_ep_s := 'chapterModel::sectionModelQuery::funcIdentifierElementRange::s_a' ;
```

References

- The functions *DomainIndex* (page 585), *IdentifierDimension* (page 587), and *IndexRange* (page 593).
- Working with the Set AllIdentifiers of the Language Reference.
- The common example in Listing 3.1.

3.1.15 IdentifierText

The function *IdentifierText* (page 590) returns the string representation of the text attribute of `identifierName` or, if the text is not specified, the name of the identifier.

```
IdentifierText(
     identifierName)       ! (input) scalar element parameter
```

Arguments

identifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which the text should be obtained.

Return Value

This function returns a string containing the text attribute of `identifierName`. If `identifierName` is not an identifier, an error message is issued. When the text is not specified, the name of the identifier is returned.

Note: This function replaces the deprecated suffix *.txt* (page 863).

Example

Given the declaration:

```
Parameter p_d {
    Text: "a scalar";
}
```

the code:

```
_ep_p := StringToElement(AllIdentifiers,
    "chapterModel::sectionModelQuery::funcIdentifierText::p_d",
    create: 0);
_sp_txt := IdentifierText( _ep_p );
block where single_column_display := 1, listing_number_precision := 6 ;
    display _sp_txt ;
endblock ;
```

produces in the listing file:

```
_sp_txt := "a scalar" ;
```

References

- The functions *IdentifierText* (page 590).
- Working with the Set AllIdentifiers of the Language Reference.
- The common example in Listing 3.1.

3.1.16 IdentifierType

The function *IdentifierType* (page 591) returns the type of `identifierName` as an element in *AllIdentifierTypes* (page 800).

```
IdentifierType(
     identifierName)       ! (input) scalar element parameter
```

Arguments

identifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which the type should be obtained.

Return Value

This function returns a type as an element in *AllIdentifierTypes* (page 800). If `identifierName` is not an identifier, an error message is issued.

Note: This function replaces the suffix *.type* (page 864); this suffix is deprecated.

Example

Given the declaration:

```
Parameter p_d  ;
```

the code:

```
1 _ep_p := StringToElement(AllIdentifiers,
2     "chapterModel::sectionModelQuery::funcIdentifierType::p_d",
3     create: 0);
4 _ep_type := IdentifierType( _ep_p );
5 block where single_column_display := 1, listing_number_precision := 6 ;
6     display _ep_type ;
7 endblock ;
```

produces in the listing file:

```
_ep_type := 'parameter' ;
```

References

- The functions *IdentifierDimension* (page 587) and *IdentifierUnit* (page 592).
- Working with the Set AllIdentifiers of the Language Reference.
- The common example in Listing 3.1.

3.1.17 IdentifierUnit

The function *IdentifierUnit* (page 592) returns the unit of `identifierName` as it is declared.

```
IdentifierUnit(
     identifierName)       ! (input) scalar element parameter
```

Arguments

identifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which the unit should be obtained.

Return Value

This function returns a unit. If `identifierName` is not an identifier, an error message is issued. If `identifierName` is not a parameter, variable or constraint, the unit `[]` is returned without further warning.

Note: This function complements the suffix *.unit* (page 865); when the unit of an identifier is a unit parameter, this function will return that unit parameter, whilst the suffix `unit` will return the value of that unit parameter.

Example

Given the declaration:

```
Parameter p_d {
        Unit: m;
}
```

the code:

```
1    _ep_p := StringToElement(AllIdentifiers,
2             "chapterModel::sectionModelQuery::funcIdentifierUnit::p_d",
3             create: 0);
4    _up_p := IdentifierUnit( _ep_p );
5    block where single_column_display := 1, listing_number_precision := 6 ;
6             display _up_p ;
7    endblock ;
```

produces in the listing file:

```
_up_p := [m] ;
```

References

- The functions *IdentifierDimension* (page 587) and *IdentifierType* (page 591).
- Working with the Set AllIdentifiers of the Language Reference.
- The common example in Listing 3.1.

3.1.18 IndexRange

The function *IndexRange* (page 593) returns the range of an index as an element in *AllIdentifiers* (page 821).

```
IndexRange(
     indexName       ! (input) scalar element parameter
     )
```

Arguments

indexName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the index for which the range should be returned.

Return Value

This function returns the range of index `indexName` as an element in *AllIdentifiers* (page 821). If `indexName` is not an index or if it does not have a range the empty element is returned.

Example

With the declarations

```
Set MasterSet {
    Index      :  a;
}
Index b {
    Range      :  MasterSet;
}
Index c;
```

The output of the statements

```
put "IndexRange( 'a' ) = \"", IndexRange( 'a' ):10, "\"", / ;
put "IndexRange( 'b' ) = \"", IndexRange( 'b' ):10, "\"", / ;
put "IndexRange( 'c' ) = \"", IndexRange( 'c' ):10, "\"", / ;
```

is:

```
IndexRange( 'a' ) = "MasterSet "
IndexRange( 'b' ) = "MasterSet "
IndexRange( 'c' ) = "          "
```

See also:

The functions *DeclaredSubset* (page 583) and *DomainIndex* (page 585).

3.1.19 IdentifierShowAttributes

The function *IdentifierShowAttributes* (page 594) allows you to programmatically open the attribute window of a specific identifier in your model. The function only works in a developer system, in an end-user system the function raises an error message.

```
IdentifierShowAttributes(
     identifier       ! (input) element in AllIdentifiers
     )
```

Arguments

identifier The identifier for which you want to open the attribute window.

See also:

The function *IdentifierShowTreeLocation* (page 594).

3.1.20 IdentifierShowTreeLocation

The function *IdentifierShowTreeLocation* (page 594) allows you to programmatically show the position of a specific identifier in the Model Explorer tree. If the Model Explorer is not currently opened, it will open automatically. The function only works in a developer system, in an end-user system the function raises an error message.

```
IdentifierShowTreeLocation(
     identifier       ! (input) element in AllIdentifiers
     )
```

Arguments

identifier The identifier for which you want to show the location in the Model Explorer.

See also:

The function *IdentifierShowAttributes* (page 594).

3.1.21 IsRuntimeIdentifier

The function *IsRuntimeIdentifier* (page 594) returns 1 when the argument `identifierName` is created at runtime.

```
IsRuntimeIdentifier(
     identifierName)       ! (input) scalar element parameter
```

Arguments

identifierName An element expression in the predefined set *AllIdentifiers* (page 821) specifying the identifier for which it should be determined whether or not it is created at runtime.

Return Value

This function returns 0 or 1. If `identifierName` is not an identifier, an error message is issued.

Note: In order to determine whether or not the value of string parameter `myStr` is an identifier, you can use `StringToElement(AllIdentifiers, myStr)` or `myStr in AllIdentifiers`.

Example

The code:

```
_bp_isRuntime := IsRuntimeIdentifier('webui_runtime::AllImplicitPublicIdentifiers');
display _bp_isRuntime ;
```

produces in the listing file:

```
_bp_isRuntime := 1 ;
```

References

- Runtime Libraries and the Model Edit Functions
- Retrieve Value of Dynamic Identifier
- Repeat Data with Model Query and Model Edit
- The functions *StringToElement* (page 66), *DeclaredSubset* (page 583) and *IndexRange* (page 593).
- Working with the Set AllIdentifiers of the Language Reference.
- The common example in Listing 3.1.

3.1.22 ReferencedIdentifiers

The function *ReferencedIdentifiers* (page 595) determines which identifiers are used in the specified attributes of a subset of *AllIdentifiers* (page 821).

```
ReferencedIdentifiers(
     searchIdentSet       ! (input) subset of AllIdentifiers
     searchAttrSet        ! (input) subset of AllAttributeNames
     recursive            ! (optional) numerical expression
     )
```

Arguments

searchIdentSet The set of identifiers to search in for referenced identifiers. This is a subsetof *AllIdentifiers* (page 821).

searchAttrSet The set of attributes to search in for referenced identifiers. This is a subset of *AllAttributeNames* (page 793).

recursive Optional argument, default 0, if 1 this function will also search in the referenced identifiers for identifier references.

Return Value

This function returns a subset of *AllIdentifiers* (page 821) containing all the identifiers that are referenced in the attributes in *searchAttrSet* in one of the identifiers in *searchIdentSet*.

Example

Given the declarations:

```
Parameter p1;
Parameter p2;
Parameter pd {
    Definition: p1 + P2;
}
```

The following code, checks the definition of `pd` to determine that the identifiers `p1` and `p2` are referenced.

```
1  _ep_pd := StringToElement(AllIdentifiers,
2      "chapterModel::sectionModelQuery::funcReferencedIdentifiers::pd",
3      create:0);
4  _s_ids += _ep_pd ;
5  _s_refIds := ReferencedIdentifiers(
6      searchIdentSet :  _s_ids,
7      searchAttrSet  :  AllAttributeNames,
8      recursive      :  1);
9  block where single_column_display := 1, listing_number_precision := 6 ;
10     display _s_refIds ;
11 endblock ;
```

produces in the listing file:

```
_s_refIds := data
{ 'chapterModel::sectionModelQuery::funcReferencedIdentifiers::p1',
  'chapterModel::sectionModelQuery::funcReferencedIdentifiers::p2' } ;
```

References

- *ConstraintVariables* (page 582), and
- *VariableConstraints* (page 598)
- ReferencedIdentifiers is essential in https://how-to.aimms.com/Articles/582/582-reduce-client-server-exchange.html
- Working with the Set AllIdentifiers

3.1.23 SectionIdentifiers

The function *SectionIdentifiers* (page 597) determines which identifiers are declared within a specific section in the model tree.

```
SectionIdentifiers(
     SectionName       ! (input) scalar element parameter
     )
```

Arguments

SectionName An element expression in the set *AllSections* (page 827) specifying the section for which the identifiers should be listed.

Return Value

This function returns a subset of *AllIdentifiers* (page 821) containing all the identifiers that are declared within the specified section, excluding the section itself and its prefix (if the section is a module or library). When SectionName is the empty element, the empty set is returned.

Example

Given the declarations:

```
Parameter p1;
Parameter p2;
Parameter pd {
    Definition: p1 + p2;
}
```

The following code collects all identifiers referenced in a section, includining the procedure.

```
_ep_sec := StringToElement(AllIdentifiers,
    "chapterModel::sectionModelQuery::Function_SectionIdentifiers",
    create:0);
_s_refIds := SectionIdentifiers(_ep_sec);
block where single_column_display := 1, listing_number_precision := 6 ;
    display _s_refIds ;
endblock ;
```

produces in the listing file:

```
_s_refIds := data
{ 'chapterModel::sectionModelQuery::funcSectionIdentifiers::pr_testSectionIdentifiers',
  'chapterModel::sectionModelQuery::funcSectionIdentifiers::p1'                       ,
  'chapterModel::sectionModelQuery::funcSectionIdentifiers::p2'                       ,
  'chapterModel::sectionModelQuery::funcSectionIdentifiers::pd'                       } ;
```

References

- Working with the Set AllIdentifiers

3.1.24 VariableConstraints

The function *VariableConstraints* (page 598) returns all the symbolic constraints that refer to one or more variables in a given set of variables.

```
VariableConstraints(
     Variables  ! (input) a subset of AllVariables
     )
```

Arguments

Variables The set of variables for which you want to retrieve the constraints that refer to them. This is a subset of *AllVariables* (page 831).

Note: This function operates on the compiled definition of constraints; it will skip inline variables during the recursion step.

Return Value

The function returns a subset of the set *AllConstraints* (page 817), containing the constraints found.

See also:

The functions *ConstraintVariables* (page 582) and *ReferencedIdentifiers* (page 595).

Listing 3.1: A common model query example

```
SelectedIdentifiers := AllParameters ; ! Or some other selection.

put outf ;

outf.pagewidth := 255 ; ! Wide
put "type":20, "  ", "name":32, "  ", "dim  ", "unit":20, "  ",
     "range":20, "  ", "Text", / ;
put "-"*20,    "  ", "-"*32,    "  ", "---  ", "-"*20,    "  ", "-"*40, / ;

for ( si ) do                                ! For each selected identifier
    put IdentifierType( si ):20, "  "              ! Type
            si:32, "  ",                              ! name
            "(", IdentifierDimension( si ):1:0, ")  ", ! dimension
            IdentifierUnit( si ):20, "  ",             ! unit
            IdentifierElementRange( si ):20, "  ",     ! range
            IdentifierText( si ), /                    ! Documenting text.
endfor ;

putclose ;
```

3.2 Model Edit Functions

AIMMS supports the following functions for model editing:

3.2.1 me::AllowedAttribute

The function *me::AllowedAttribute* (page 599) returns 1 if the attribute is allowed for the runtime id.

```
me::AllowedAttribute(
        runtimeId,  ! (input) an element
        attr        ! (input) an element
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

attr An element in the set *AllAttributeNames* (page 793)

Return Value

Returns 1 if the attribute `attr` of runtime identifier `runtimeId` is allowed. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

The code:

```
_bp_isAllowedAttribute := me::AllowedAttribute('webui_
↪runtime::AllImplicitPublicIdentifiers','text');
display _bp_isAllowedAttribute ;
```

produces in the listing file:

```
_bp_isAllowedAttribute := 1 ;
```

References

- *IsRuntimeIdentifier* (page 594)
- *me::SetAttribute* (page 618)
- *me::Create* (page 606).

Generic references for model edit functions can be found on the index page

3.2.2 me::ChangeType

The procedure *me::ChangeType* (page 600) changes the type of a runtime identifier.

```
me::ChangeType(
        runtimeId,   ! (input) an element
        newType      ! (input) an element
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

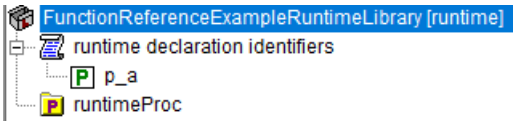
newType An element in the set *AllIdentifierTypes* (page 800).

Return Value

Returns 1 if the change type operation is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

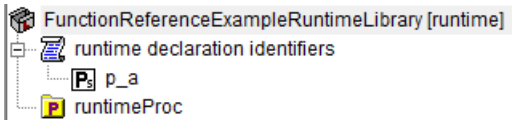
Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeParameter` have value `frerl::p_a`, then the code:

```
me::ChangeType(
    runtimeId :  ep_functionReferenceExampleRuntimeParameter,
    newType   :  'string parameter');
```

Changes that library to the following:



Illustrating the type change of runtime identifier `frerl::p_a` from `parameter` to `string parameter`.

References

- *me::Create* (page 606)
- *me::Move* (page 613)

Generic references for model edit functions can be found on the index page

3.2.3 me::ChangeTypeAllowed

The function *me::ChangeTypeAllowed* (page 601) returns 1 if the type of runtime identifier `runtimeId` can be changed into type `newType`.

```
me::ChangeTypeAllowed(
        runtimeId,   ! (input) an element
        newType      ! (input) an element
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

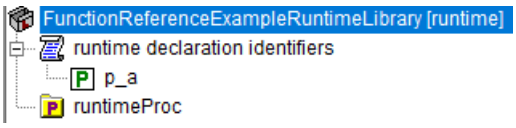
newType An element in the set *AllIdentifierTypes* (page 800).

Return Value

Returns 1 if the identifier `runtimeId` can be changed into `newType`. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeParameter` have value `frerl::p_a`, then the code:

```
_bp_changeTypeAllowed := me::ChangeTypeAllowed( ep_
↪functionReferenceExampleRuntimeParameter, 'quantity' );
display _bp_changeTypeAllowed ;
```

produces in the listing file:

```
_bp_changeTypeAllowed := 0 ;
```

Illustrating that a parameter cannot be changed into a quantity.

References

- *me::ChangeType* (page 600)
- `IdentifierTYpe`.

Generic references for model edit functions can be found on the index page

3.2.4 me::Children

The procedure *me::Children* (page 602) returns the number of children of a runtime identifier and fills an output parameter with those children.

```
me::Children(
        runtimeId,          ! (input) an element
        runtimeChildren(i)  ! (output) indexed element parameter.
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

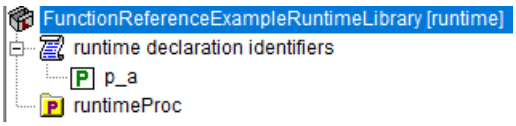
runtimeChildren The children in the runtime identifier tree. This parameter needs to be an output parameter indexed over a (subset of) the set *Integers* (page 812).

Return Value

This procedure returns the number of children of `runtimeId`. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeLib` have value `FunctionReferenceExampleRuntimeLibrary`, then the code:

```
me::Children(
        runtimeId       :  ep_functionReferenceExampleRuntimeLib,
        runtimeChildren :  _ep_children);
block where single_column_display := 1, listing_number_precision := 6 ;
        display _ep_children ;
endblock ;
```

produces in the listing file:

```
_ep_children := data
{ 1 : 'frerl::runtime_declaration_identifiers',
  2 : 'frerl::runtimeProc'                     } ;
```

Illustrating navigating from one runtime identifier to another using model edit functions.

References

- *me::Parent* (page 614)
- *me::GetAttribute* (page 610).

Generic references for model edit functions can be found on the index page

3.2.5 me::ChildTypeAllowed

The function *me::ChildTypeAllowed* (page 603) returns 1 if a child of type `newType` can be added as a child to runtime identifier `runtimeId`..

```
me::ChildTypeAllowed(
        runtimeId,   ! (input) an element
        newType      ! (input) an element
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

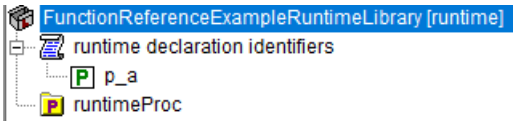
newType An element in the set *AllIdentifierTypes* (page 800).

Return Value

Returns 1 if the identifier of type `newType` can be added as a child to identifier `runtimeId`. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeParameter` have value `frerl::p_a`, then the code:

```
_bp_childTypeAllowed := me::ChildTypeAllowed(
    runtimeId :  ep_functionReferenceExampleRuntimeParameter,
    newType   :  'parameter');
display _bp_childTypeAllowed ;
```

Produces the following in the listing file:

```
_bp_childTypeAllowed := 0 ;
```

Illustrating that a parameter cannot have, as child, another parameter.

References

- *me::Create* (page 606)
- *me::Children* (page 602)

Generic references for model edit functions can be found on the index page

3.2.6 me::Compile

The procedure *me::Compile* (page 605) compiles a runtime identifier and all runtime identifiers below that identifier. If that runtime identifier is a runtime library, all procedures can be run and set / parameter definitions can be evaluated provided there are no errors.

```
me::Compile(
            runtimeId  ! (input) an element
            )
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

Return Value

Returns 1 if the compilation operation is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library defined as:

```
LibraryModule FunctionReferenceExampleRuntimeLibrary {
    Prefix: frerl;
    DeclarationSection runtime_declaration_identifiers {
        Parameter p_a;
    }
    Procedure runtimeProc {
        Body: {
            display p_a;
        }
    }
}
```

Then the code:

```
me::Compile( ep_functionReferenceExampleRuntimeLib );
apply( ep_functionReferenceExampleRuntimeProc );
```

Produces the following in the listing file:

```
frerl::p_a := 0 ;
```

This is an illustration of the best practice to compile an entire runtime library, before executing a procedure instide that library.

References

- *me::SetAttribute* (page 618)
- *me::Create* (page 606).
- *me::IsRunnable* (page 612)
- The `APPLY statement`, see The APPLY Operator of the Language Reference.

Generic references for model edit functions can be found on the index page

3.2.7 me::Create

The function *me::Create* (page 606) creates a runtime identifier.

```
me::Create(
        name,     ! (input) a string
        newType,  ! (input) an element
        parentId, ! (input) an element
        pos       ! (optional) an integer
)
```

Arguments

name A string that is valid name for a runtime identifier.

newType An element in the set *AllIdentifierTypes* (page 800).

parentId An element in the set *AllSymbols* (page 787) referencing a runtime identifier.

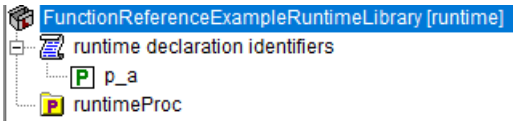
pos 1 is the first position, and 0 means "place at end", the default is 0.

Return Value

Returns an element in *AllSymbols* (page 787) if successful or the empty element otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

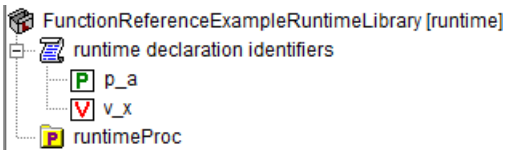
Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeDeclSec` have value `frerl::runtime_declaration_identifiers`, then the code:

```
me::Create(
    name     :  "v_x",
    newType  :  'variable',
    parentId :  ep_functionReferenceExampleRuntimeDeclSec,
    pos      :  0);
```

inserts a variable at the end of the declaration section, thus changing that library to the following:



Illustrating that a runtime library can grow at runtime.

References

- *me::Delete* (page 608)
- *me::CreateLibrary* (page 607)
- *me::Move* (page 613)
- *me::SetAttribute* (page 618).

Generic references for model edit functions can be found on the index page

3.2.8 me::CreateLibrary

The function *me::CreateLibrary* (page 607) creates a new runtime library.

```
me::CreateLibrary(
        libraryName,    ! (input) a string
        prefixName      ! (optional) a string
)
```

Arguments

libraryName The name of the new runtime library.

prefixName The name of the new prefix, when not specified one is generated from the `libraryName`.

Return Value

The function returns an element in the set *AllIdentifiers* (page 821) referencing the library when successful and the empty element upon failure. In the latter case at least one error has been raised.

Example

The code:

```
ep_functionReferenceExampleRuntimeLib := me::CreateLibrary(
        libraryName :  sp_functionReferenceExampleLibName,
        prefixName  :  sp_functionReferenceExampleLibPrefix );
_ep_type := IdentifierType(ep_functionReferenceExampleRuntimeLib);
display _ep_type ;
```

Creates a runtime library, and produces in the listing file:

```
_ep_type := 'LibraryModule' ;
```

References

- *me::ImportLibrary* (page 611)
  - *me::Create* (page 606)

Generic references for model edit functions can be found on the index page

3.2.9 me::Delete

The procedure *me::Delete* (page 608) a runtime identifier and all runtime identifiers below that identifier.

```
me::Delete(
        runtimeId  ! (input) an element
)
```

Arguments

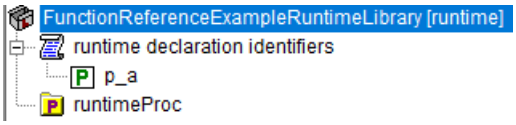
runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

Return Value

Returns 1 if the delete operation is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

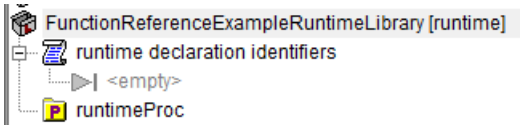
Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeParameter` have value `frerl::p_a`, then the code:

```
me::Delete( ep_functionReferenceExampleRuntimeParameter );
```

deletes that parameter from the declaration section, thus changing that library to the following:



Illustrating that a runtime library can shrink at runtime.

References

- *me::Children* (page 602)
- *me::GetAttribute* (page 610).

Generic references for model edit functions can be found on the index page

3.2.10 me::ExportNode

The procedure *me::ExportNode* (page 609) writes a section to file.

```
me::ExportNode(
        esection,   ! (input) section element.
        filename)   ! (input) a string
```

Arguments

esection An element in the set *AllIdentifiers* (page 821) referencing a runtime library or a section in a runtime library.

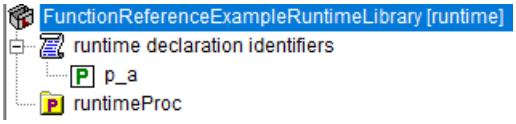
filename The name of file to which the section is written. The filename should have the `.ams` extension.

Return Value

The procedure returns 1 if the file is written successfully. If the procedure fails to write the file it returns 0 after raising errors.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let `ep_functionReferenceExampleRuntimeLib` be the element parameter that holds the root of that library as value, then the code:

```
me::ExportNode(
    esection :  ep_functionReferenceExampleRuntimeLib,
    filename :  "test.ams");
```

writes the contents of the runtime library to the file `test.ams`.

References

- *me::CreateLibrary* (page 607),
- *me::ImportLibrary* (page 611)
- *me::ImportNode* (page 612).
  - *me::ExportNode* (page 609) plays an essential role in Create a static AIMMS Library from a runtime library

Generic references for model edit functions can be found on the index page

3.2.11 me::GetAttribute

The function *me::GetAttribute* (page 610) returns the contents of an attribute as a string.

```
me::GetAttribute(
        runtimeId,   ! (input) an element
        attr         ! (input) an element
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

attr An element in the set *AllAttributeNames* (page 793)

Return Value

Returns the contents of the attribute `attr` of runtime identifier `runtimeId` as a string. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library defined as:

```
LibraryModule FunctionReferenceExampleRuntimeLibrary {
    Prefix: frerl;
    DeclarationSection runtime_declaration_identifiers {
        Parameter p_a;
    }
    Procedure runtimeProc {
        Body: {
            display p_a;
        }
    }
}
```

Then the code:

```
_sp_body := me::GetAttribute(ep_functionReferenceExampleRuntimeProc,'body');
display _sp_body ;
```

produces the following in the listing file:

```
_sp_body := "display p_a;\n" ;
```

References

- *AttributeToString* (page 573)
- *me::SetAttribute* (page 618)
- *me::Create* (page 606).

Generic references for model edit functions can be found on the index page

3.2.12 me::ImportLibrary

The function *me::ImportLibrary* (page 611) reads a runtime library from an `.ams` file.

```
me::ImportLibrary(
        filename)    ! (input) a string
```

Arguments

filename The name of file that contains a runtime library.

Return Value

The function returns an element in the set *AllIdentifiers* (page 821) referencing the library when successful and the empty element upon failure. In the latter case at least one error has been raised.

See also:

The functions *me::CreateLibrary* (page 607), *me::ImportNode* (page 612) and *me::ExportNode* (page 609).

3.2.13 me::ImportNode

The procedure *me::ImportNode* (page 612) reads a section from file.

```
me::ImportNode(
        esection,  ! (input) section element.
        filename)  ! (input) a string
```

Arguments

esection An element in the set *AllIdentifiers* (page 821) referencing a section in a runtime library.

filename The name of file that contains a runtime library. The filename should have the `.ams` extension.

Return Value

The procedure returns 1 if the file is read successfully. If the procedure fails to read the file it returns 0 after raising errors.

See also:

The functions *me::CreateLibrary* (page 607) and *me::ExportNode* (page 609).

3.2.14 me::IsRunnable

The function *me::IsRunnable* (page 612) determines whether or not the runtime identifier resides in a runtime library for which all procedures are runnable and all definitions can be evaluated.

```
me::IsRunnable(
        runtimeId   ! (input) an element
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

Return Value

The function returns 1 iff `runtimeId` resides in a runtime library where all procedures are runnable and all definitions can be evaluated. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

See also:

The functions *me::Compile* (page 605) and `me::IsReadonly`.

3.2.15 me::Move

The procedure *me::Move* (page 613) renames a runtime identifier. In addition, when the move changes the namespace of the runtime identifier all text within the runtime library referencing that runtime identifier will be adapted accordingly.

```
me::Move(
        runtimeId,  ! (input) an element
        parentid,   ! (input) an element
        pos         ! (input) integer
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

parentid An element in the set *AllIdentifiers* (page 821) referencing a runtime identifeir in the same runtime library.

pos An integer position in the section. 1 is the first position, and 0 means "place at end".

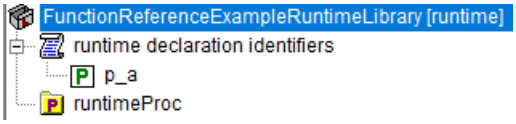
Return Value

Returns 1 if the move operation is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Note: The name change file is not supported for runtime libraries.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:

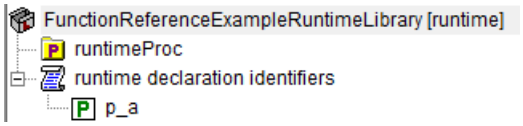


Let

- `ep_functionReferenceExampleRuntimeLib` refer to the root of the runtime library, and
- `ep_functionReferenceExampleRuntimeProc` refer to a procedure inside the library, then the code:

```
me::Move(
    runtimeId :  ep_functionReferenceExampleRuntimeProc,
    parentId  :  ep_functionReferenceExampleRuntimeLib ,
    pos       :  1);
```

moves the procedure to the front of the library, as shown below:



Illustrating that a runtime library can be reorganized at runtime.

References

- *me::ChangeType* (page 600)
- *me::Rename* (page 615).
  - *me::Move* (page 613) plays a central role in Runtime functions with arguments.

Generic references for model edit functions can be found on the index page

3.2.16 me::Parent

The function *me::Parent* (page 614) returns the parent of a runtime identifier.

```
me::Parent(
        runtimeId    ! (input) an element
)
```

Arguments

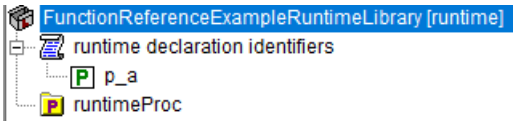
runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

Return Value

The function returns an element in the set *AllIdentifiers* (page 821) referencing the parent of the referenced identifier or the empty element if the referenced identifier is a runtime library. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let

- `ep_functionReferenceExampleRuntimeLib` refer to the root of the runtime library, and
- `ep_functionReferenceExampleRuntimeProc` refer to a procedure inside the library, then the code:

```
_ep_parent := me::Parent( ep_functionReferenceExampleRuntimeProc );

display _ep_parent ;
```

produces the following in the listing file:

```
_ep_parent := 'FunctionReferenceExampleRuntimeLibrary' ;
```

References

- *me::Children* (page 602)
- *me::GetAttribute* (page 610).

Generic references for model edit functions can be found on the index page

3.2.17 me::Rename

The procedure *me::Rename* (page 615) renames a runtime identifier. In addition, all text within the runtime library referencing that runtime identifier will be adapted accordingly.

```
me::Rename(
        runtimeId,   ! (input) an element
        newname      ! (input) a string
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

newname A string.

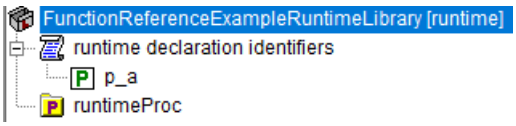
Return Value

Returns 1 if the rename operation is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Note: The name change file is not supported for runtime libraries.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:

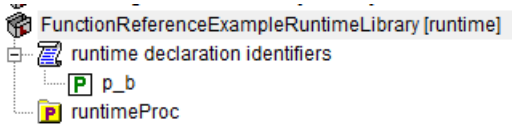


Let

- `ep_functionReferenceExampleRuntimeParameter` refer to the parameter in the runtime library then the code:

```
me::Rename(
        runtimeId :  ep_functionReferenceExampleRuntimeParameter,
        newName   :  "p_b");
```

renames the parameter as shown below:



References

- *me::ChangeType* (page 600)
- *me::Move* (page 613)

Generic references for model edit functions can be found on the index page

3.2.18 me::SetAnnotation

The procedure *me::SetAnnotation* (page 617) changes an annotation of a runtime identifier.

```
me::SetAnnotation(
        runtimeId,   ! (input) an element
        key,         ! (input) name of annotation
        value        ! (input) a string expression
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

key A string that is a valid annotation key.

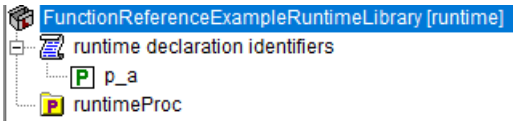
value The text to be assigned.

Return Value

Returns 1 if the text assignment to the attribute is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let

- `ep_functionReferenceExampleRuntimeParameter` refer to the parameter in the runtime library, then the code:

```
me::SetAnnotation(
    runtimeId :  ep_functionReferenceExampleRuntimeParameter,
    key       :  "webui::TooltipIdentifier",
    value     :  "chapterModel::sectionModelEdit::funcMeSetAnnotation::sp_
↪tooltipRuntimeIdentifier");
me::Compile( ep_functionReferenceExampleRuntimeLib );
GetAnnotationValues("webui::TooltipIdentifier",_sp_rtd);
```

(continues on next page)

(continued from previous page)

```
_sp_annot := _sp_rtd( ep_functionReferenceExampleRuntimeParameter );
display _sp_annot ;
```

produces the following in the listing file:

```
_sp_annot := "chapterModel::sectionModelEdit::funcMeSetAnnotation::sp_
↪tooltipRuntimeIdentifier" ;
```

References

- *me::ChangeType* (page 600)
- *me::Rename* (page 615).

Generic references for model edit functions can be found on the index page

3.2.19 me::SetAttribute

The procedure *me::SetAttribute* (page 618) changes the contents of an attribute of a runtime identifier.

```
me::SetAttribute(
        runtimeId,   ! (input) an element
        attr,        ! (input) an element
        txt          ! (input) a string expression
)
```

Arguments

runtimeId An element in the set *AllIdentifiers* (page 821) referencing a runtime identifier.

attr An element in the set *AllAttributeNames* (page 793)

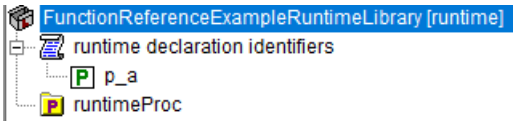
txt The text to be assigned. Using the empty string will effectively delete the attribute from the runtime identifier.

Return Value

Returns 1 if the text assignment to the attribute is successful, 0 otherwise. In the latter case error(s) have been raised. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

Example

Viewing a small runtime library with prefix `frerl` in the model explorer:



Let

- `ep_functionReferenceExampleRuntimeParameter` refer to the parameter in the runtime library, then the code:

```
me::SetAttribute(
        runtimeId :  ep_functionReferenceExampleRuntimeParameter,
        attr      :  'text',
        txt       :  "just another runtime identifier");
_sp_text := me::GetAttribute(
        runtimeId :  ep_functionReferenceExampleRuntimeParameter,
        attr      :  'text');
display _sp_text ;
```

produces the following in the listing file:

```
_sp_text := "just another runtime identifier" ;
```

References

- *me::ChangeType* (page 600)
- *me::Rename* (page 615).

Generic references for model edit functions can be found on the index page

3.2.20 References

1. Working with the Set AllIdentifiers of the Language Reference.
2. Runtime Libraries and the Model Edit Functions
3. Model query functions
4. Runtime functions with arguments
5. Create a static AIMMS Library from a runtime library

3.2.21 Use in AIMMS repository Libraries

The model edit functions and runtime libraries are extensively used in the following AIMMS repository libraries:

1. AIMMS WebUI
2. AIMMS DataExchange
3. AIMMS Collaborative Data Management.

3.2.22 Applications of model editing

The following how-to's illustrate using model edit functions in increasingly interesting manner:

1. Retrieve Value of Dynamic Identifier
2. Repeat Data with Model Query and Model Edit
3. Formulas as Data

CHAPTER

FOUR

DATA MANAGEMENT

4.1 Case Management

If your project has set the option `Data_Management_style` to `Disk_files_and_folders`, AIMMS supports a set of data management functions, that allow you to modify the default data management behavior.

There are two groups of functions. The *Core* functions and the GUI/IDE related functions.

The core functions allow you to save data to and load data from case files located on your system. These core functions do not keep track of whether a specific case file is the current one, nor do they check whether current data needs to be saved. These core functions are:

- *CaseFileLoad* (page 622)
- *CaseFileMerge* (page 624)
- *CaseFileSave* (page 625)
- *CaseFileGetContentType* (page 628)
- *CaseCompareIdentifier* (page 623)
- *CaseCreateDifferenceFile* (page 626)
- *CaseFileURLtoElement* (page 636)
- *CaseFileSectionExists* (page 629)
- *CaseFileSectionGetContentType* (page 630)
- *CaseFileSectionLoad* (page 631)
- *CaseFileSectionMerge* (page 632)
- *CaseFileSectionRemove* (page 633)
- *CaseFileSectionSave* (page 633)

The GUI/IDE related data management functions can be used to create a specific GUI for your own (modified) data management. They allow you to re-use some of the default data management features. For example the selecting of case files using dialog boxes, and the concept of a current case.

- *CaseFileSetCurrent* (page 635)
- *CaseCommandLoadAsActive* (page 634)
- *CaseCommandLoadIntoActive* (page 637)
- *CaseCommandMergeIntoActive* (page 637)
- *CaseCommandNew* (page 638)

- *CaseCommandSave* (page 638)
- *CaseCommandSaveAs* (page 639)
- *CaseDialogConfirmAndSave* (page 640)
- *CaseDialogSelectForLoad* (page 640)
- *CaseDialogSelectForSave* (page 641)
- *CaseDialogSelectMultiple* (page 642)
- *DataManagementExit* (page 643)

4.1.1 CaseFileLoad

With the function *CaseFileLoad* (page 622), you can load the data of an existing case file into memory. All identifiers read from the case file will replace the corresponding data of the identifier in the current model.

```
CaseFileLoad(
    url,                          ! (input) a scalar string expression
    [keepUnreferencedRuntimeLibs] ! (optional) 0 or 1
    )
```

Arguments

url A string referencing the url of the case file that should be loaded. This url can point to a file on your local file system, or to a network location.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the data is loaded, but not referenced in the case file, should be kept in memory or destroyed during the data load. The default is 0, indicating that the runtime libraries not referenced in the case file should be destroyed during the case load.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.
- Data stored in user sections of the case file, will not be read by *CaseFileLoad* (page 622).

Example

Given two cases `data/caseA.data` and `data/caseB.data` containing:

	caseA	caseB	caseB merged to caseA
p_num	77	88	88
p_dat(a)	11	11	11
p_dat(b)	22	20	20
p_dat(c)	33		33
p_dat(d)		44	44

Then the code:

```
CaseFileLoad(
    url                          :  "data/caseA.data",
    keepUnreferencedRuntimeLibs :  0);
block where single_column_display := 1 ;
    display p_num, p_dat ;
endblock ;
```

produces in the listing file:

```
chapterData::sectionCaseManagement::p_num := 77 ;


chapterData::sectionCaseManagement::p_dat := data
{ a : 11,
  b : 22,
  c : 33 } ;
```

See also:

The procedure *CaseFileMerge* (page 624).

4.1.2 CaseCompareIdentifier

With the function *CaseCompareIdentifier* (page 623) you can determine whether or not two cases differ with respect to a certain identifier.

```
CaseCompareIdentifier(
        FirstCase,    ! (input) element in the set AllCases
        SecondCase,   ! (input) element in the set AllCases
        Identifier,   ! (input) element in the set AllIdentifiers
        Suffix        ! (optional) element in the set AllSuffixNames
        Mode          ! (optional) element in the set AllCaseComparisonModes
        )
```

Arguments

FirstCase An element in the set *AllCases* (page 842)

SecondCase An element in the set *AllCases* (page 842)

Identifier An element in the set *AllIdentifiers* (page 821), refering to the specific identifier that you want to compare.

Suffix An element in the set *AllSuffixNames* (page 809) with respect to which you want to compare the data.

Mode An element in the *AllCaseComparisonModes* (page 794) with respect to how you want to compare the data.

Return Value

- For numerical identifiers the function returns the differences between the values of the identifier in both cases, based on the mode. It can be the minimum, maximum, average, sum or count of all differences.
- For non-numerical identifiers the function counts the number of differences between the identifier in both cases.

4.1.3 CaseFileMerge

With the function *CaseFileMerge* (page 624), you can merge the data of an existing case file with the current data in memory. When merging, the current data in memory will only be overwritten by the non-defaults of the identifiers read from the case file.

```
CaseFileMerge(
    url,                          ! (input) a scalar string expression
    [keepUnreferencedRuntimeLibs] ! (optional) 0 or 1
    )
```

Arguments

url A string referencing the url of the case file that should be merged. This url can point to a file on your local file system, or to a network location.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the data is loaded, but not referenced in the case file, should be kept in memory or destroyed during the data load. For a merge, the default is 1, indicating that the runtime libraries not referenced in the case will be retained during the case merge.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.
- Data stored in user sections of the case file will not be read by *CaseFileMerge* (page 624).

Example

Given two cases `data/caseA.data` and `data/caseB.data` containing:

	caseA	caseB	caseB merged to caseA
p_num	77	88	88
p_dat(a)	11	11	11
p_dat(b)	22	20	20
p_dat(c)	33		33
p_dat(d)		44	44

Then the code:

```
CaseFileLoad(
    url                          :  "data/caseA.data",
    keepUnreferencedRuntimeLibs :  0);
CaseFileMerge(
    url                          :  "data/caseB.data",
    keepUnreferencedRuntimeLibs :  0);
block where single_column_display := 1 ;
    display p_num, p_dat ;
endblock ;
```

produces in the listing file:

```
chapterData::sectionCaseManagement::p_num := 88 ;


chapterData::sectionCaseManagement::p_dat := data
{ a : 11,
  b : 20,
  c : 33,
  d : 44 } ;
```

See also:

The procedure *CaseFileLoad* (page 622)

4.1.4 CaseFileSave

The function *CaseFileSave* (page 625) saves a specific subset of identifiers to a case file. If the file already exists, it is completely overwritten.

```
CaseFileSave(
    url,                          ! (input) a scalar string expression
    contents                      ! (input) a subset of AllIdentifiers
    )
```

Arguments

url A string referencing the url of the case file in which you want to save the data. This url can point to a file on your local file system, or to a network location.

contents A subset of *AllIdentifiers* (page 821) containing all the identifiers that must be saved. Preferrably, this set is an element of *AllCaseFileContentTypes* (page 848) such that, when reading back the case file, the content type can be determined correctly.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function will only save the data to the specified file. It does not change the value of `CurrentCase` or `CurrentCaseFileContentType`, nor does it mark the current data as being saved.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.
- When you save using *CaseFileSave* (page 625) to an existing `.data` file with sections, the sections are removed.

Example

The code:

```
1    read from file "caseFileInputData/caseC.txt" ;
2    CaseFileSave( "data/caseC.data", s_CaseManagementData );
```

Reads data from a text file, and then saves it in binary format to `data/caseC.data`.

See also:

The functions *CaseFileSectionSave* (page 633) and *CaseFileLoad* (page 622)

4.1.5 CaseCreateDifferenceFile

With the procedure *CaseCreateDifferenceFile* (page 626) you can create an AIMMS input file containing the differences between the current data and a reference case.

```
CaseCreateDifferenceFile(
        referenceCase,            ! (input) element in the set AllCases
        outputFilename,           ! (input) scalar string expression
        diffTypes,                ! (input)  indexed element parameter
        absoluteTolerance,        ! (optional) scalar expression
        relativeTolerance,        ! (optional) scalar expression
        outputPrecision,          ! (optional) scalar expression
        respectDomainCurrentCase  ! (optional) scalar expression
        )
```

Arguments

referenceCase An element in the set *AllCases* (page 842) specifying the case to which the current data should be compared.

outputFilename A string expression specifying the name of the file the differences are written to.

diffTypes An element parameter indexed over (a subset of) *AllIdentifiers* (page 821) with range the predeclared set *AllDifferencingModes* (page 797).

absoluteTolerance A scalar expression specifying the absolute tolerance when comparing numerical values. The range of this argument is $[0, inf)$, the default is the value of the option `equality_absolute_tolerance`.

relativeTolerance A scalar expression specifying the relative tolerance when comparing numerical values. The range of this argument is $[0, 1]$, the default is the value of the option `equality_relative_tolerance`.

outputPrecision A scalar expression specifying how many decimals should be printed. The range of the argument is $\{0 \ldots 20\}$, the default is the value of the option `listing_precision`.

respectDomainCurrentCase A scalar expression specifying whether or not the current domain should be taken into account. When 0: The current domain is not taken into account and all differences are written to the output file. When 1: The current domain is taken into account; the differences are filtered according to the domain of the identifier.

Return Value

This procedure returns 0 upon failure, 1 upon success. When successful all differences between the current model data and the data in the reference case are written to a file.

Note:

- In a READ statement, when reading from an input file that was generated by *CaseCreateDifferenceFile* (page 626) function with `diffTypes` equal to `elementReplacement`, `elementAddition` or `elementMultiplication`, the file is always read in merge mode, so that the `diffTypes` can be applied in a sensible way.

Example

Given two cases `data/caseA.data` and `data/caseB.data` containing:

	caseA	caseB	caseB merged to caseA
p_num	77	88	88
p_dat(a)	11	11	11
p_dat(b)	22	20	20
p_dat(c)	33		33
p_dat(d)		44	44

Then the code:

```
CaseFileLoad(
    url                          :  "data/caseA.data",
    keepUnreferencedRuntimeLibs :  0);
_ep_diffTypes(i_caseManagementData) := 'elementReplacement' ;
CaseFileURLtoElement( "data/caseB.data", _ep_refCase, 1 );
CaseCreateDifferenceFile(
    referenceCase            :  _ep_refCase,
    outputFilename           :  "log/casediff.txt",
    diffTypes                :  _ep_diffTypes,
    absoluteTolerance        :  0,
    relativeTolerance        :  0,
    outputPrecision          :  0,
    respectDomainCurrentCase :  0);
```

Will produce the following contents in `log/casediff.txt`:

```
chapterData::sectionCaseManagement::s_chars += data
{ c } ;
chapterData::sectionCaseManagement::s_chars -= data
{ d } ;

chapterData::sectionCaseManagement::p_num := 77 ;

chapterData::sectionCaseManagement::p_dat :=$ data
{ b : 22,
  c : 33,
  d :  0 } ;
```

4.1.6 CaseFileGetContentType

The procedure *CaseFileGetContentType* (page 628) retrieves the subset reference that was used when saving the case file.

```
CaseFileGetContentType(
        url,             ! (input) a scalar string expression
        contents         ! (output) a scalar element parameter into the
                         !          set AllSubsetsOfAllIdentifiers
    )
```

Arguments

url A string referencing the url of an existing case file from which you want to retrieve the contents information. This url can point to a file on your local file system, or to a network location.

contents An element parameter in `AllSubsetsOfAllIdentifiers`. On return it holds the reference to the subset that was used when saving the case file.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

Example

The code:

```
CaseFileGetContentType(
        url         :  "data/caseA.data",
        contentType :  _ep_contentType);
display _ep_contentType ;
```

produces in the listing file:

```
_ep_contentType := 'chapterData::sectionCaseManagement::s_CaseManagementData' ;
```

See also:

The function *CaseFileSave* (page 625).

4.1.7 CaseFileSectionExists

The function *CaseFileSectionExists* (page 629) returns whether a user section exists in a given case file.

```
CaseFileSectionExists(
    url,                          ! (input) a scalar string expression
    sectionName                   ! (input) a scalar string expression
    )
```

Arguments

url A string referencing the url an existing case file. This url can point to a file on your local file system, or to a network location.

sectionName The name of the user section. Any leading or trailing spaces in the name are ignored, and an empty string is not allowed. The length of the name is limited to 27 characters.

Return Value

The procedure returns 1 if the section exists or 0 if the section does not exist. If any other error occurs, the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

See also:

The functions *CaseFileSectionSave* (page 633), *CaseFileSectionLoad* (page 631), *CaseFileSectionMerge* (page 632), *CaseFileSectionRemove* (page 633)

4.1.8 CaseFileSectionGetContentType

The procedure *CaseFileSectionGetContentType* (page 630) retrieves the subset reference that was used when saving a user section in a case file.

```
CaseFileSectionGetContentType(
        url,                ! (input) a scalar string expression
        sectionName,        ! (input) a scalar string expression
        contents            ! (output) a scalar element parameter in the
                            !          set AllSubsetsOfAllIdentifiers
)
```

Arguments

url A string referencing the url of an existing case file from which you want to retrieve the contents information. This url can point to a file on your local file system, or to a network location.

sectionName The name of the user section. Any leading or trailing spaces in the name are ignored, and an empty string is not allowed.

contents An element parameter in `AllSubsetsOfAllIdentifiers`. Upon return, it holds the reference to the subset that was used when saving the user section in the case file.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

See also:

The functions *CaseFileSectionSave* (page 633), *CaseFileGetContentType* (page 628)

4.1.9 CaseFileSectionLoad

With the function *CaseFileSectionLoad* (page 631), you can load the data of a user section in an existing case file into memory. All identifiers stored in the case file section will replace the corresponding data of the identifier in the current model.

```
CaseFileSectionLoad(
    url,                          ! (input) a scalar string expression
    sectionName,                  ! (input) a scalar string expression
    [keepUnreferencedRuntimeLibs] ! (optional) 0 or 1
)
```

Arguments

url A string referencing the url of the case file that should be loaded. This url can point to a file on your local file system, or to a network location.

sectionName The name of the user section from which you want to load the data. Any leading or trailing spaces in the name are ignored, and an empty string is not allowed. The length of the name is limited to 27 characters.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the data is loaded, but not referenced in the case file, should be kept in memory or destroyed during the data load. The default is 0, indicating that the runtime libraries not referenced in the case file should be destroyed during the case load.

Return Value

The procedure returns 1 on success. If any other error occur, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

See also:

The functions *CaseFileLoad* (page 622), *CaseFileSectionSave* (page 633), *CaseFileSectionMerge* (page 632), *CaseFileSectionExists* (page 629), *CaseFileSectionRemove* (page 633)

4.1.10 CaseFileSectionMerge

With the function *CaseFileSectionMerge* (page 632), you can merge the data of a user section in an existing case file with the current data in memory. When merging, the current data in memory will only be overwritten by the non-defaults of the identifiers stored in the case file section.

```
CaseFileSectionMerge(
    url,                          ! (input) a scalar string expression
    sectionName,                  ! (input) a scalar string expression
    [keepUnreferencedRuntimeLibs] ! (optional) 0 or 1
)
```

Arguments

url A string referencing the url of the case file that should be merged. This url can point to a file on your local file system, or to a network location.

sectionName The name of the user section from which you want to load the data. Any leading or trailing spaces in the name are ignored, and an empty string is not allowed. The length of the name is limited to 27 characters.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the data is loaded, but not referenced in the case file, should be kept in memory or destroyed during the data load. The default is 0, indicating that the runtime libraries not referenced in the case file should be destroyed during the case load.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

See also:

The functions *CaseFileMerge* (page 624), *CaseFileSectionSave* (page 633), *CaseFileSectionLoad* (page 631), *CaseFileSectionExists* (page 629), *CaseFileSectionRemove* (page 633)

4.1.11 CaseFileSectionRemove

The function *CaseFileSectionRemove* (page 633) can remove a user section from a specified existing case file.

```
CaseFileSectionRemove(
    url,                          ! (input) a scalar string expression
    sectionName                   ! (input) a scalar string expression
)
```

Arguments

url A string referencing the url of an existing case file. This url can point to a file on your local file system, or to a network location.

sectionName The name of the user section to remove. Any leading or trailing spaces in the name are ignored, and an empty string is not allowed. The length of the name is limited to 27 characters.

Return Value

The function returns 1 if the section was successfully removed or did not exist at all. It returns 0 if the section exists, but could not be removed. In case of any other error, the function returns -1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

See also:

The functions *CaseFileSectionSave* (page 633), *CaseFileSectionLoad* (page 631), *CaseFileSectionMerge* (page 632), *CaseFileSectionExists* (page 629)

4.1.12 CaseFileSectionSave

Beside the main data area in a case file, which is written using the function CaseFileSave, you can store additional data in user defined sections of the case file. To save data in a user section, you call the function *CaseFileSectionSave* (page 633).

```
CaseFileSectionSave(
    url,                          ! (input) a scalar string expression
    sectionName,                  ! (input) a scalar string expression
    contents                      ! (input) a subset of AllIdentifiers
)
```

Arguments

url A string referencing the url of an existing case file in which you want to save the additional data. This url can point to a file on your local file system, or to a network location.

sectionName The name of the section in which you want to write additional data. If the section does not yet exist, it is created. Otherwise, the existing contents of the section is replaced by the newly saved data. Any leading or trailing spaces in the name are ignored, and an empty string is not allowed. The length of the name is limited to 27 characters.

contents A subset of *AllIdentifiers* (page 821) containing all the identifiers that must be saved.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and CurrentErrorMessage will contain a proper error message.

Note:

- This function is only applicable if the project option Data_Management_style is set to Disk_files_and_folders.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.
- You cannot use this function to create a new case file. A new case file can only be created using CaseFileSave.

See also:

The functions *CaseFileSave* (page 625), *CaseFileSectionLoad* (page 631), *CaseFileSectionMerge* (page 632), *CaseFileSectionExists* (page 629), *CaseFileSectionRemove* (page 633)

4.1.13 CaseCommandLoadAsActive

The procedure *CaseCommandLoadAsActive* (page 634) executes the same code that is behind the menu command **Data-Load Case-As Active** in the IDE by default (please note that you can override items in the **Data** menu using the options listed under `Project - Data manager - Using disk files and folders - Data menu overrides`). It shows a dialog box in which the user can select a case file, and subsequently tries to load the data from that file. If the previously active case needs to be saved, a confirmation dialog box will be displayed first. Afterwards, the active case will reference the selected case file.

```
CaseCommandLoadAsActive
```

Return Value

The procedure returns 1 on success, or 0 if the user cancelled the operation in one of the dialog boxes. If any other error occurs, the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseCommandLoadIntoActive* (page 637), *CaseCommandMergeIntoActive* (page 637), *CaseCommandNew* (page 638), *CaseCommandSave* (page 638), *CaseCommandSaveAs* (page 639)

4.1.14 CaseFileSetCurrent

The procedure *CaseFileSetCurrent* (page 635) sets the predefined element parameter *CurrentCase* (page 846) and, as a result, updates the corresponding field in the status bar of the IDE.

```
CaseFileSetCurrent(
    url      ! (input) a scalar string expression
    )
```

Arguments

url A string referencing the url of the case file that should be loaded. This url can point to a file on your local file system, or to a network location. If you specify the empty string, the element parameter `CurrentCase` will be emptied.

Return Value

The procedure returns 1 on success. If any other error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

4.1.15 CaseFileURLtoElement

For each case file that has been accessed during an AIMMS session, a new element is created in the predefined set *AllCases* (page 842). The predefined string parameter *CaseFileURL* (page 850) is updated accordingly. When working with a selection of case files, for example in a multiple case view, or in statements with the case dot notation, you should actually create a subset of *AllCases* (page 842). In that process, it may be useful to find the corresponding element in *AllCases* (page 842) given the url of a case file.

```
CaseFileURLtoElement(
    url,                          ! (input) a scalar string expression
    caseFileElement,              ! (output) element in AllCases
    [checkURLExists]              ! (optional) 0 or 1
    )
```

Arguments

url A string referencing the url of a case file. This url can point to an existing file on your local file system, or to a network location. The given url does not need to be present in *AllCases* (page 842) a priori.

caseFileElement On return, this element parameter is set to the element in *AllCases* (page 842) that corresponds to the given url. In other words, the following condition will be true: `CaseFileUrl(caseFileElement) = url`.

checkURLExists (optional) If this value is set to 1 then the procedure always returns 0 if the specified url cannot be found in the underlying file system. If set to 0 and the underlying file does not exist, the procedure returns 1 if the corresponding element already existed in *AllCases* (page 842). The default value is 0.

Return Value

The procedure returns 1 on success. If any error occurs, the procedure returns 0 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If your application is linked to the AIMMS PRO server, the url can also point to a case file stored at the server.

- If *url* exists, but is not in *CaseFileURL* (page 850), an element will be added to *AllCases* (page 842).
- If *url* does not exist, but there is a corresponding entry *CaseFileURL* (page 850), the procedure returns is 1 if checkURLExists is set to 0 and it returns 0 if checkURLExists is set to 1.

See also:

The procedures *CaseDialogSelectMultiple* (page 642)

4.1.16 CaseCommandLoadIntoActive

The procedure *CaseCommandLoadIntoActive* (page 637) executes the same code that is behind the menu command **Data-Load Case-Into Active** in the IDE by default (please note that you can override items in the **Data** menu using the options listed under `Project - Data manager - Using disk files and folders - Data menu overrides`). It shows a dialog box in which the user can select a case file, and subsequently tries to load the data from that file. The command changes the data for the active case. It does not set the active case to the selected case, though.

```
CaseCommandLoadIntoActive
```

Return Value

The procedure returns 1 on success, or 0 if the user cancelled the operation in one of the dialog boxes. If any other error occurs, the procedure returns -1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseCommandLoadAsActive* (page 634), *CaseCommandMergeIntoActive* (page 637), *CaseCommandNew* (page 638), *CaseCommandSave* (page 638), *CaseCommandSaveAs* (page 639)

4.1.17 CaseCommandMergeIntoActive

The procedure *CaseCommandMergeIntoActive* (page 637) executes the same code that is behind the menu command **Data-Load Case-Merging into Active** in the IDE by default (please note that you can override items in the **Data** menu using the options listed under `Project - Data manager - Using disk files and folders - Data menu overrides`). It shows a dialog box in which the user can select a case file, and subsequently tries to merge the data from that file. The command changes the data for the active case. It does not set the active case to the selected case, though.

```
CaseCommandMergeIntoActive
```

Return Value

The procedure returns 1 on success, or 0 if the user cancelled the operation in one of the dialog boxes. If any other error occurs, the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseCommandLoadAsActive* (page 634), *CaseCommandLoadIntoActive* (page 637), *CaseCommandNew* (page 638), *CaseCommandSave* (page 638), *CaseCommandSaveAs* (page 639)

4.1.18 CaseCommandNew

The procedure *CaseCommandNew* (page 638) executes the same code that is behind the menu command **Data-New Case** in the IDE by default (please note that you can override items in the **Data** menu using the options listed under `Project - Data manager - Using disk files and folders - Data menu overrides`). If the data of the currently active case needs to be saved, a confirmation dialog box will be displayed first. Afterwards, the active case will not refer to any case file.

```
CaseCommandNew
```

Return Value

The procedure returns 1 on success, or 0 if the user cancelled the operation in one of the dialog boxes. If any other error occurs, the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.
- An alternative for calling *CaseCommandNew* (page 638) is calling *CaseFileSetCurrent* (page 635) with an empty string. The latter will not check whether the current case should be saved first.

See also:

The procedures *CaseCommandLoadAsActive* (page 634), *CaseCommandLoadIntoActive* (page 637), *CaseCommandMergeIntoActive* (page 637), *CaseCommandSave* (page 638), *CaseCommandSaveAs* (page 639)

4.1.19 CaseCommandSave

The procedure *CaseCommandSave* (page 638) executes the same code that is behind the menu command **Data-Save Case** in the IDE by default (please note that you can override items in the **Data** menu using the options listed under `Project - Data manager - Using disk files and folders - Data menu overrides`). If there is no active case yet, this procedure behaves the same as `CaseCommandSaveAs`. Otherwise, the active data is saved to the active case file.

```
CaseCommandSave
```

Return Value

The procedure returns 1 on success, or 0 if the user cancelled the operation in one of the dialog boxes. If any other error occurs, the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseCommandLoadAsActive* (page 634), *CaseCommandLoadIntoActive* (page 637), *CaseCommandMergeIntoActive* (page 637), *CaseCommandNew* (page 638), *CaseCommandSaveAs* (page 639)

4.1.20 CaseCommandSaveAs

The procedure *CaseCommandSaveAs* (page 639) executes the same code that is behind the menu command **Data-Save Case As** in the IDE by default (please note that you can override items in the **Data** menu using the options listed under `Project - Data manager - Using disk files and folders - Data menu overrides`). It shows a dialog box in which the user can select a (new) case file, and subsequently tries to save the data to that case file. Afterwards, the active case will reference the selected case file.

```
CaseCommandSaveAs
```

Return Value

The procedure returns 1 on success, or 0 if the user cancelled the operation in one of the dialog boxes. If any other error occurs, the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.

- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseCommandLoadAsActive* (page 634), *CaseCommandLoadIntoActive* (page 637), *CaseCommandMergeIntoActive* (page 637), *CaseCommandNew* (page 638), *CaseCommandSave* (page 638)

4.1.21 CaseDialogConfirmAndSave

The procedure *CaseDialogConfirmAndSave* (page 640) shows and handles the standard confirmation dialog box, in which the user is asked whether he wants to save the currently active data before continuing.

```
CaseDialogConfirmAndSave
```

Return Value

The procedure returns 1 if the user chooses not to save the data, or if the user chooses to save the data and the save was executed successfully. It returns 0 if the user cancelled any of the dialog boxes. If any other error occurs, the procedure returns -1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This procedure is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This procedure returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.
- This procedure does not check whether the data needs to be saved; that check should be made by the calling code, prior to calling this procedure.
- If the user confirms to save the data, the function `CaseDialogSave` is called. If no active case file exists, this implies that the `CaseDialogSaveAs` is called instead.

See also:

The procedure `DataChangeMonitorAnyChange`

4.1.22 CaseDialogSelectForLoad

The procedure *CaseDialogSelectForLoad* (page 640) shows the case file selection dialog box. This dialog box allows the user to select an existing case file. The procedure only results in the url of the selected case file, it does not actually load any data from the case file.

```
CaseDialogSelectForLoad(
    url       ! (input/output) a scalar string parameter
    )
```

Arguments

url A string representing the case file to be loaded. On entry, the string is used to initialize the dialog box to the correct folder location. On return, the string will contain the reference to the selected case file.

Return Value

The procedure returns 1 if the user selected an existing url, and 0 if the user cancelled the dialog box.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseDialogSelectForSave* (page 641), *CaseFileLoad* (page 622)

4.1.23 CaseDialogSelectForSave

The procedure *CaseDialogSelectForSave* (page 641) shows the case file selection dialog box. This dialog box allows the user to select an existing or a new case file. If the selected file already exists, an overwrite confirmation dialog box is displayed. The procedure only results in the url of the selected case file, it does not actually create the file or replace the existing contents. If the predefined set *AllCaseFileContentTypes* (page 848) contains multiple elements, then the dialog box also allows the user to select the specific contents that he wants to save.

```
CaseDialogSelectForSave(
    url,           ! (input/output) a scalar string parameter
    contentType    ! (input/output) an element in AllCaseFileContentTypes
    )
```

Arguments

url A string representing the case file to be saved. On entry, the string is used to initialize the dialog box to the correct folder location. On return, the string will contain the reference to the selected case file.

contentType An element parameter in *AllCaseFileContentTypes* (page 848). On return, this element parameter will contain the element that the user selected.

Return Value

The procedure returns 1 if the user selected an existing or new url, and 0 if the user cancelled the dialog box.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The procedures *CaseDialogSelectForLoad* (page 640), *CaseFileSave* (page 625)

4.1.24 CaseDialogSelectMultiple

The procedure *CaseDialogSelectMultiple* (page 642) shows a case file selection dialog box in which you can select multiple case files. The result is a subset of *AllCases* (page 842) that can be used in multiple case views, or in execution statements with the case dot notation.

```
CaseDialogSelectMultiple(
    selectedCaseFiles     ! (input/output) a subset of AllCases
    )
```

Arguments

selectedCaseFiles A subset of *AllCases* (page 842). On entry, this subset is used to initalize the selection in the dialog box. On return, it contains the subset that has been selected by the user.

Return Value

The procedure returns 1 if the user selected a set of case files, and 0 if the user cancelled the dialog box.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- This function returns 0 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.
- You can use any subset of *AllCases* (page 842) as an argument to this function, but if you want to use it for a multiple case view in one of your pages, you should use the predefined set *CurrentCaseSelection* (page 847).
- If the subset should have the selected cases in the order as specified in the dialog, you must make sure that the given subset has the attribute `Order by` set to `user`.

See also:

The procedure *CaseFileURLtoElement* (page 636), the string parameter *CaseFileURL* (page 850) and the set *AllCases* (page 842).

4.1.25 DataManagementExit

The function *DataManagementExit* (page 643) checks whether any data should be saved according to the active data management style. If any of the data needs saving, a dialog box is displayed, in which the user can select to save the data, not to save the data, or to cancel the current operation.

```
DataManagementExit
```

Return Value

The procedure returns 1 if the current data does not need to be saved, or if the user explicitly decided to save or not to save the data. If the user cancelled the dialog box, or if the saving of the data resulted in an error, the return value is 0.

Note:

- This function is applicable if the project option `Data_Management_style` is set to either `Disk_files_and_folders` or `Single_Data_Manager_file`.
- When the project option `Data_Management_style` is set to `Disk_files_and_folders`, the "dirty" status can be cleared using the following statement: `DataChangeMonitorReset(DataManagementMonitorID, AllIdentifiers)`
- This function is used as the default content of the procedure `MainTermination`, such that upon project close the data management can check whether any data needs to be saved first.
- This function always returns 1 if the IDE is not loaded, for example when running the component version of AIMMS, or when running with the command line option `--as-server`.

See also:

The predeclared identifier `DataManagementMonitorID` and the intrinsic function *DataChangeMonitorReset* (page 646)

4.2 Data Change Monitor Functions

To keep track of which data has been changed during a session, you can define one or more Data Change Monitors. The following functions are for creating and maintaining these monitors:

4.2.1 DataChangeMonitorCreate

With the function *DataChangeMonitorCreate* (page 643), you can create a new data change monitor. With a data change monitor, you can determine whether any identifiers in a subset of *AllIdentifiers* (page 821) have been changed since the latest call to *DataChangeMonitorCreate* (page 643) or DataChangeMonitorReset. To check for any changes, you can use *DataChangeMonitorHasChanged* (page 645).

```
DataChangeMonitorCreate(
    ID,                     ! (input) a scalar string expression
    monitoredIdentifiers,   ! (input) subset of AllIdentifiers
    [excludeNonSaveables]   ! (optional) 0 or 1
    )
```

Arguments

ID A string identifying a (new) data change monitor.

monitoredIdentifiers The subset of identifiers that you want to monitor for this data change monitor.

excludeNonSaveables (optional) If the data change monitor is used to monitor whether or not a subset of identifiers needs to be saved, it is unnecessary to include identifiers that have the Nosave property. If you set this argument to 1, these identifiers will automatically be excluded from the given subset of identifiers. The default of this argument is 1. This exclusion is applied also on any subset that is passed in later calls to DataChangeMonitorReset.

Return Value

The function returns 1 upon success. If there already exists a data change monitor for the given ID, the function returns 0. In case of any other error, it returns -1. If the return value is 0 or -1 CurrentErrorMessage will contain a proper error message.

Note:

- The newly created monitor is reset automatically, so there is no need to call the function DataChangeMonitorReset immediately after creation.
- If your project uses the Data management style 'Disk files and folders', AIMMS itself uses a data change monitor to keep track of whether the active data needs to be saved before exiting, or before loading any new data. The ID of this internal data change monitor is given by the predeclared string parameter DataManagementMonitorID.

See also:

The functions *DataChangeMonitorHasChanged* (page 645), *DataChangeMonitorReset* (page 646), *DataChangeMonitorDelete* (page 644).

4.2.2 DataChangeMonitorDelete

With the function *DataChangeMonitorDelete* (page 644), you can delete a data change monitor that was created using the function DataChangeMonitorCreate.

```
DataChangeMonitorDelete(
    ID                    ! (input) a scalar string expression
    )
```

Arguments

ID A string identifying an existing data change monitor.

Return Value

The function returns 1 upon success. If there exists no data change monitor for the given ID, the function returns 0. In case of any other error, it returns −1 and CurrentErrorMessage will contain a proper error message.

See also:

The functions *DataChangeMonitorCreate* (page 643), *DataChangeMonitorReset* (page 646), *DataChangeMonitorHasChanged* (page 645).

4.2.3 DataChangeMonitorHasChanged

The function *DataChangeMonitorHasChanged* (page 645) returns whether the data of any identifier that is monitored by the specified data change monitor has been changed since a previous call to DataChangeMonitorCreate or DataChangeMonitorReset.

```
DataChangeMonitorHasChanged(
    ID                    ! (input) a scalar string expression
    )
```

Arguments

ID A string identifying an existing data change monitor.

Return Value

The function returns 1 if any of the identifiers monitored by the data change monitor has been changed since a previous call to either DataChangeMonitorCreate or DataChangeMonitorReset. If none of the identifiers has been changed, the function returns 0. In case of any other error, it returns −1 and CurrentErrorMessage will contain a proper error message. If the monitored set contains identifiers that were not present in that set at the previous call to either DataChangeMonitorCreate or DataChangeMonitorReset, these identifiers are assumed to be changed, and the function returns 1 as well.

Note:

- Calling *DataChangeMonitorHasChanged* (page 645) does not reset the data change monitor.

See also:

The functions *DataChangeMonitorCreate* (page 643), *DataChangeMonitorReset* (page 646), *DataChangeMonitorDelete* (page 644).

4.2.4 DataChangeMonitorReset

The function *DataChangeMonitorReset* (page 646) assigns a new set of identifiers to an existing data change monitor and resets the monitor to the 'unchanged' status.

```
DataChangeMonitorReset(
    ID,                     ! (input) a scalar string expression
    monitoredIdentifiers    ! (input) subset of AllIdentifiers
    )
```

Arguments

ID A string identifying an existing data change monitor.

monitoredIdentifiers The subset of identifiers that should be monitored by the data change monitor.

Return Value

The function returns 1 upon success. If there exists no data change monitor for the given `ID`, the function returns 0. In case of any other error it returns -1 and `CurrentErrorMessage` will contain a proper error message.

See also:

The functions *DataChangeMonitorCreate* (page 643), *DataChangeMonitorHasChanged* (page 645), *DataChangeMonitorDelete* (page 644).

4.3 Database Functions

AIMMS supports the following database related functions:

4.3.1 CloseDataSource

With the procedure *CloseDataSource* (page 646) you can temporarily close the connection to a data source. AIMMS automatically opens the connection to a data source if needed, and closes the connection when the project is exited.

```
CloseDataSource(
     Datasource          ! (input) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

Note: When *CloseDataSource* (page 646) is called during a transaction that was explicitly started by calling *StartTransaction* (page 650) the transaction is rolled back before actually closing the data source. *CurrentErrorMessage* (page 835) contains a message telling it did so.

4.3.2 CommitTransaction

By default, AIMMS places a transaction around any *single* WRITE statement to a database table. In this way, AIMMS makes sure that the complete WRITE statement can be rolled back in the event of a database error during the execution of that WRITE statement. With the procedure *CommitTransaction* (page 647) you can commit all the changes to the database (through WRITE statements or SQL queries) made since the last call to StartTransaction.

```
CommitTransaction
```

Arguments

None

Return Value

The procedure returns 1 if the transaction was committed successfully, or 0 otherwise.

See also:

The procedures *StartTransaction* (page 650), *RollbackTransaction* (page 649).

4.3.3 DirectSQL

With the procedure *DirectSQL* (page 647) you can directly execute SQL statements within a data source.

```
DirectSQL(
     Datasource,          ! (input) a string expression
     SQLstatement         ! (input) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

SQLstatement A string containing the SQL statement that must be executed within the data source.

Return Value

The procedure returns 1 if the SQL statement is executed successfully, or 0 if the execution failed. In case of failure, the corresponding error message can be obtained through the predefined string parameter *CurrentErrorMessage* (page 835).

Note:

- If the SQL statement also produces a result set, then this set is ignored by AIMMS.
- Note that the SQL dialect used by, for instance, Oracle, SQL Server and Microsoft Access may differ. If a call to *DirectSQL* (page 647) fails because of such differences, you should inspect `CurrentErrorMessage` for further details.

See also:

Calling stored procedures and executing SQL queries through AIMMS `DATABASE PROCEDURES` is discussed in Executing Stored Procedures and SQL Queries of the Language Reference.

4.3.4 LoadDatabaseStructure

The AIMMS `Read ...From Table ...` and `Write ...To Table ...` statements offer a very flexible way to connect to data tables stored in an ODBC compliant database. The AIMMS execution engine queries the structure of the corresponding database tables in order to check whether the connection between the table in the database and the AIMMS identifiers can be set up in a valid way, and, if so, how to handle the statements efficiently. Retrieving structural information may cost a significant amount of time, depending on the number of tables, the quality of the network and the quality of the ODBC database driver implementation in providing this information. Although AIMMS already buffers this information for each table after first use, retrieving this information anew each AIMMS run might still be prohibitively expensive in some cases. Therefore, AIMMS offers intrinsic database functions to empower the app developer with caching this information outside AIMMS. With the procedure *LoadDatabaseStructure* (page 648) you can load the cached database table structure information.

```
LoadDatabaseStructure(
    Filename          ! (input) a string expression
    )
```

Arguments

Datasource A string containing the name of the file containing the database table structure information.

Return Value

The procedure returns 1 if the database table structure information is successfully loaded, or 0 otherwise.

See also:

The procedure *SaveDatabaseStructure* (page 649)

4.3.5 RollbackTransaction

By default, AIMMS places a transaction around any *single* `WRITE` statement to a database table. In this way, AIMMS makes sure that the complete `WRITE` statement can be rolled back in the event of a database error during the execution of that `WRITE` statement. With the procedure *RollbackTransaction* (page 649) you can rollback (undo) all the changes to the database (through `WRITE` statements or SQL queries) made since the last call to `StartTransaction`.

```
RollbackTransaction
```

Arguments

None

Return Value

The procedure returns 1 if the transaction was rolled back successfully, or 0 otherwise.

See also:

The procedures *StartTransaction* (page 650), *RollbackTransaction* (page 649).

4.3.6 SaveDatabaseStructure

With the procedure *SaveDatabaseStructure* (page 649) you can save the database table structure information such that this information can be quickly retrieved in subsequent AIMMS sessions. This procedure only stores the structure information for tables that are currently connected. In order to connect to a database table, you should either run a read or write statement, or open the mapping wizard of that database tables. Please note that calling *CloseDataSource* (page 646) will close all these connections, so this should not be called before the call to *SaveDatabaseStructure* (page 649).

```
SaveDatabaseStructure(
     Filename           ! (input) a string expression
     )
```

Arguments

Filename A string containing the name of a data source.

Return Value

The procedure returns 1 if the database table structure is succesfully saved to file `Filename`, or 0 otherwise.

See also:

The procedure *LoadDatabaseStructure* (page 648).

4.3.7 StartTransaction

By default, AIMMS places a transaction around any *single* `WRITE` statement to a database table. In this way, AIMMS makes sure that the complete `WRITE` statement can be rolled back in the event of a database error during the execution of that `WRITE` statement. With the procedure *StartTransaction* (page 650) you can manually initiate a database transaction which can contain multiple `READ`, `WRITE` statements and SQL queries.

```
StartTransaction(
     IsolationLevel          ! (optional) an element expression
     )
```

Arguments

IsolationLevel Element value into the set *AllIsolationLevels* (page 801), indicating the isolation level at which the transaction has to take place. If omitted, defaults to `'ReadCommitted'`.

Return Value

The procedure returns 1 if the transaction was started successfully, or 0 otherwise.

Note: You cannot call *StartTransaction* (page 650) recursively, i.e. you must call `CommitTransaction` or `RollbackTransaction` prior to the next call to *StartTransaction* (page 650).

See also:

The procedures *CommitTransaction* (page 647) and *RollbackTransaction* (page 649).

4.3.8 TestDataSource

With the procedure *TestDataSource* (page 650) you can test for the presence of a data source on a host computer, before reading or writing to it. If you try to read or write to a non-existing data source, AIMMS will generate error messages which may be confusing for your end users.

```
TestDataSource(
     Datasource,          ! (input) a string expression
     interactive,         ! (input/optional) an integer, default 1
     timeout              ! (input/optional) unit: seconds, default 30
     )
```

Arguments

Datasource A string containing the name of a data source.

interactive When non-zero: if additional (logon) information is required a window is popped up. When zero: if additional (logon) information is required, the procedure will return immediately with the value 0.

timeout When the timeout is expired the procedure TestDataSource will return with the value 0.

Return Value

The procedure returns 1 if the data source is present, or 0 otherwise. If the result is 0, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *TestDatabaseTable* (page 651) and *TestDatabaseColumn* (page 653).

4.3.9 TestDatabaseTable

With the procedure *TestDatabaseTable* (page 651) you can check whether a given table name exists in a specific data source.

```
TestDatabaseTable(
     Datasource,          ! (input) a string expression
     Tablename            ! (input) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

Tablename A string containing the name of a table in *Datasource*.

Return Value

The procedure returns 1 if the database table is present in the given data source, or 0 otherwise. If the result is 0, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: The *Tablename* argument of the procedure *TestDatabaseTable* (page 651) is case sensitive if the ODBC driver is case sensitive.

See also:

The procedures *TestDataSource* (page 650) and *TestDatabaseColumn* (page 653).

4.3.10 GetDataSourceProperty

With the function *GetDataSourceProperty* (page 652) you can retrieve some meta-data about a datasource. This is useful, when you don't know beforehand what kind of datasource will be linked with your AIMMS project. It allows you to provide datasource-specific SQL Queries in your project, which you can then call based upon what datasource is actually linked to your project. For example, you can determine with this function that the actual datasource is an Oracle database, and then execute some Oracle-specific SQL Queries.

```
GetDataSourceProperty(
     Datasource,          ! (input) a string expression
     Property,            ! (input) an element in the set
                              AllDataSourceProperties
     )
```

Arguments

Datasource A string containing the name of a data source.

Property An element parameter in the set *AllDataSourceProperties* (page 796).

Return Value

The function returns a string with the requested datasource property in it.

Note: The actual string which is returned depends on the datasource used. As an example of the datasource dependency of the function: retrieving the property `SQL_DATA_SOURCE_NAME` may return `"null"` for a MySQL ODBC datasource, while it returns the actual name of your datasource when you retrieve it for an Oracle database. This means that you should experiment with the return values a bit, to make sure that you understand what values to expect for your specific datasource(s).

4.3.11 SQLNumberOfColumns

With the function *SQLNumberOfColumns* (page 652) you can determine the number of columns of a database table.

```
SQLNumberOfColumns(
     Datasource,          ! (input) a string expression
     TableName,           ! (input) a string expression
     Owner                ! (input/optional) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

TableName A string containing the name of the database table for which the number of columns must be determined.

Owner A string containing the owner of the database table for which the number of columns must be determined. If the datasource doesn't support the owner concept, but the owner argument is specified, an error will be raised.

Return Value

The function returns the number of columns in the specified database table. If the database table doesn't exist, an error is raised.

See also:

The functions *SQLNumberOfViews* (page 655), *SQLNumberOfTables* (page 654) and *SQLColumnData* (page 656).

4.3.12 TestDatabaseColumn

With the procedure *TestDatabaseColumn* (page 653) you can check whether a given column is present in a database table on a specific datasource.

```
TestDatabaseColumn(
     Datasource,          ! (input) a string expression
     TableName            ! (input) a string expression
     ColumnName           ! (input) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

TableName A string containing the name of a table in *Datasource*.

ColumnName A string containing the name of a column in the *TableName*.

Return Value

The procedure returns 1 if the column name is present in the given database table, or 0 otherwise. If the result is 0, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: The *TableName* and *ColumnName* arguments of the procedure *TestDatabaseColumn* (page 653) are case sensitive if the ODBC driver is case sensitive.

See also:

The procedures *TestDataSource* (page 650) and *TestDatabaseTable* (page 651).

4.3.13 SQLNumberOfDrivers

With the function *SQLNumberOfDrivers* (page 654) you can determine the number of installed ODBC drivers on your system.

```
SQLNumberOfDrivers(
    DatabaseInterface,    ! (input) an element expression
    )
```

Arguments

DatabaseInterface Element value into the set AllDatabaseInterfaces. Currently, this set contains only the value `'ODBC'`.

Return Value

The function returns the number of installed ODBC drivers on your system (using `'ODBC'` as argument). In case none are installed, the value `0` is returned. In case of an error, `-1` is returned.

Note: This function should be used in combination with the function *SQLDriverName* (page 657), to determine all ODBC drivers installed on your system.

See also:

The functions *SQLDriverName* (page 657) and *SQLCreateConnectionString* (page 658).

4.3.14 SQLNumberOfTables

With the function *SQLNumberOfTables* (page 654) you can determine the number of tables in a datasource.

```
SQLNumberOfTables(
     Datasource,          ! (input) a string expression
     Owner                ! (input/optional) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

owner A string containing the owner for which the number of tables must be determined. If the datasource doesn't support the owner concept, but the owner argument is specified, an error will be raised.

Return Value

The function returns the number of tables in the specified datasource. If there are no tables for the specified datasource and owner, 0 is returned. If an error occurs when determining the number of tables, -1 is returned and an error message is displayed in the error window.

See also:

The functions *SQLNumberOfViews* (page 655), *SQLNumberOfColumns* (page 652) and *SQLTableName* (page 659).

4.3.15 SQLNumberOfViews

With the function *SQLNumberOfViews* (page 655) you can determine the number of views in a datasource.

```
SQLNumberOfViews(
     Datasource,          ! (input) a string expression
     Owner                ! (input/optional) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

Owner A string containing the owner for which the number of views must be determined. If the datasource doesn't support the owner concept, but the owner argument is specified, an error will be raised.

Return Value

The function returns the number of views in the specified datasource. If there are no views for the specified datasource and owner, 0 is returned. If an error occurs when determining the number of views, -1 is returned and an error message is displayed in the error window.

See also:

The functions *SQLNumberOfTables* (page 654), *SQLNumberOfColumns* (page 652) and *SQLViewName* (page 659).

4.3.16 SQLColumnData

With the function *SQLColumnData* (page 656) you can determine the characteristics of a certain column of a database table.

```
SQLColumnData(
     Datasource,           ! (input) a string expression
     TableName,            ! (input) a string expression
     ColumnNumber,         ! (input) an integer expression
     Owner,                ! (input/optional) a string expression
     ColumnCharacteristic ! (input/optional) an element in set AllData-
                             ColumnCharacteristics, with default
                             value 'Name'
     )
```

Arguments

Datasource A string containing the name of a data source.

TableName A string containing the name of the database table of the column for which to retrieve a characteristic.

ColumnNumber An integer containing the number of the column for which to retrieve a characteristic. The maximum value of this argument can be obtained by calling the function `SQLNumberOfColumns` prior to calling this function. The minimum value of this argument is 1.

Owner A string containing the owner of the database table. If the datasource doesn't support the owner concept, but the owner argument is specified, an error will be raised.

ColumnCharacteristic An element in the set *AllDataColumnCharacteristics* (page 795), which contains all possible characteristics to obtain for a column.

Return Value

The function returns the specified characteristic, as a string value. This means that also the numerical characteristics (`'Width'`, `'NumberOfDecimals'` and (possibly) `'DefaultValue'`) are returned as string values. So, if you want to use these results in their numeric form, please use the function `Val`.

Note: Typically, this function will be used in a construction like the following, to ensure that the right `ColumnNumber` argument is passed:

```
NumberOfColumns := SQLNumberOfColumns("MyDataSource", "MyTable");

ColCount := 1;
while ColCount <= NumberOfColumns do
    for IndexDataColumnCharacteristics do
        Characteristic := SQLColumnData(MyDataSource, "MyTable", ColCount, "",
                                        IndexDataColumnCharacteristics);
        ! Do something with the characteristic
    endfor;
    ColCount += 1;
endwhile;
```

See also:

The functions *SQLNumberOfColumns* (page 652) and *Val* (page 44).

4.3.17 SQLDriverName

With the function *SQLDriverName* (page 657) you can determine the name of a certain ODBC driver on your system. This function is designed to be used in conjunction with the *SQLNumberOfDrivers* (page 654) function.

```
SQLDriverName(
     DatabaseInterface,    ! (input) an element expression
     DriverNo,             ! (input) an integer expression
     )
```

Arguments

DatabaseInterface Element value into the set AllDatabaseInterfaces. Currently, this set contains only the value 'ODBC'.

DriverNo An integer containing the number of the ODBC driver for which you want to retrieve the name. To determine the maximum value of this argument, please use the function *SQLNumberOfDrivers* (page 654) prior to calling this function. The minimum value of this argument is 1.

Return Value

The function returns the name of the ODBC driver (specified by the `DatabaseInterface` argument), with the number as specified through the `DriverNo` argument. If you specify a number outside of the correct range, AIMMS will display an error message.

Note: Typically, this function can best be used in a construction like the following:

```
NumberOfDrivers := SQLNumberOfDrivers('ODBC');

while LoopCount <= NumberOfDrivers do
    DriverName := SQLDriverName('ODBC', LoopCount);
    ! Do something with the retrieved table name here...
endwhile;
```

The retrieved name of an ODBC driver, can be used as argument in the function *SQLCreateConnectionString* (page 658).

See also:

The functions *SQLNumberOfDrivers* (page 654) and *SQLCreateConnectionString* (page 658).

4.3.18 SQLCreateConnectionString

The function *SQLCreateConnectionString* (page 658) assists you in creating a *connection string*, which can be used to specify the `Data source` attribute of database tables, functions or procedures. Using a connection string to connect to a data source, makes it possible to keep your database passwords hidden.

```
SQLCreateConnectionString(
     DatabaseInterface,                ! (input) an element expression
     DriverName,                       ! (input) a string expression
     [ServerName],                     ! (optional) a string expression
     [DatabaseName],                   ! (optional) a string expression
     [UserId],                         ! (optional) a string expression
     [Password],                       ! (optional) a string expression
     [AdditionalConnectionParameters]  ! (optional) a string expression
     )
```

Arguments

DatabaseInterface Element value into the set AllDatabaseInterfaces. Currently, this set contains only the value `'ODBC'`.

DriverName A string containing the name of the ODBC driver to which you want to connect using the resulting connection string. See the functions *SQLNumberOfDrivers* (page 654) and *SQLDriverName* (page 657) on how to obtain the driver/provider name.

ServerName (optional) A string containing the name of the server on which the data source to connect to is hosted.

DatabaseName (optional) A string containing the name of the database to which you want to connect.

UserId (optional) A string containing the user id with which to login on the datasource.

Password A string containing the password to use when logging in on the datasource. The password will not be part of the resulting string, but will be stored internally, making it possible to communicate by means of the connectionstring without revealing the credentials.

AdditionalConnectionParameters (optional) A string containing any additional connection parameters to be passed to the data source using the resulting connection string. These additional parameters should be specified in the form `KEYWORD=VALUE`, and these keyword/value pairs must be separated by semi-colons. Different drivers/providers accept different keywords. Please refer to the documentation of your ODBC driver for more information.

Return Value

The function returns a connection string, which can be used to connect to a data source on your system.

Note: The returned connection string can be used as the data source attribute of database related identifiers in AIMMS. Also, it can be used in database related functions (e.g. `SQLDirect`) as the `Datasource` argument.

See also:

The functions *SQLNumberOfDrivers* (page 654) and *SQLDriverName* (page 657).

4.3.19 SQLTableName

With the function *SQLTableName* (page 659) you can determine the name of a certain table in a datasource. This function is designed to be used in conjunction with the `SQLNumberOfTables` function.

```
SQLTableName(
     Datasource,          ! (input) a string expression
     TableNo,             ! (input) an integer expression
     Owner                ! (input/optional) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

TableNo An integer containing the number of the table for which you want to retrieve the name. To determine the maximum value of this argument, please use the function `SQLNumberOfTables` prior to calling this function. The minimum value of this argument is 1.

Owner A string containing the owner of the table for which the name must be determined. If the datasource doesn't support the owner concept, but the owner argument is specified, an error will be raised.

Return Value

The function returns the name of the table, with the number as specified through the `TableNo` argument.

Note: Typically, this function can best be used in a construction like the following:

```
NumberOfTables := SQLNumberOfTables("MyDataSource");

while LoopCount <= NumberOfTables do
    TableName := SQLTableName("MyDataSource", LoopCount);
    ! Do something with the retrieved table name here...
endwhile;
```

See also:

The functions *SQLNumberOfTables* (page 654) and *SQLViewName* (page 659).

4.3.20 SQLViewName

With the function *SQLViewName* (page 659) you can determine the name of a certain view in a datasource. This function is designed to be used in conjunction with the SQLNumberOfViews function.

```
SQLViewName(
     Datasource,          ! (input) a string expression
     TableNo,             ! (input) an integer expression
     Owner                ! (input/optional) a string expression
     )
```

Arguments

Datasource A string containing the name of a data source.

ViewNo An integer containing the number of the view for which you want to retrieve the name. To determine the maximum value of this argument, please use the function SQLNumberOfViews prior to calling this function. The minimum value of this argument is 1.

Owner A string containing the owner of the view for which the name must be determined. If the datasource doesn't support the owner concept, but the owner argument is specified, an error will be raised.

Return Value

The function returns the name of the view, with the number as specified through the ViewNo argument.

Note: Typically, this function can best be used in a construction like the following:

```
NumberOfViews := SQLNumberOfViews("MyDataSource");

while LoopCount <= NumberOfViews do
    ViewName := SQLViewName("MyDataSource", LoopCount);
    ! Do something with the retrieved view name here...
endwhile;
```

See also:

The functions *SQLNumberOfViews* (page 655) and *SQLTableName* (page 659).

4.4 Spreadsheet Functions

Warning: As detailed in the AIMMS Product Lifecycle, Spreadsheet Functions are deprecated.

One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

AIMMS supports the following functions for reading from and writing to Excel and OpenOffice Calc workbooks:

4.4.1 Spreadsheet::ColumnName

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The function *Spreadsheet::ColumnName* (page 660) returns the name of the Excel or OpenOffice Calc column with the given number.

```
Spreadsheet::ColumnName(
        ColumnNumber  ! (input) scalar numerical expression
        )
```

Arguments

ColumnNumber A scalar integer expression representing the column number for which to determine the name.

Return Value

The function returns a string representing the column name corresponding to the *ColumnNumber*. If it fails, AIMMS issues an error message and execution is halted.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelColumnName`, which has become deprecated as of AIMMS 3.12.

See also:

The function *Spreadsheet::ColumnNumber* (page 661).

4.4.2 Spreadsheet::ColumnNumber

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The function *Spreadsheet::ColumnNumber* (page 661) returns the number of the Excel or OpenOffice Calc column with the given name.

```
Spreadsheet::ColumnNumber(
        ColumnName  ! (input) scalar string expression
        )
```

Arguments

ColumnName A scalar string expression representing the column name for which to determine the number.

Return Value

The function returns an integer representing the column number corresponding to the *ColumnName*. If it fails, AIMMS issues an error message and execution is halted.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelColumnNumber`, which has become deprecated as of AIMMS 3.12.

See also:

The function *Spreadsheet::ColumnName* (page 660).

4.4.3 Spreadsheet::SetActiveSheet

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::SetActiveSheet* (page 662) sets the active sheet for the given Excel or OpenOffice Calc workbook.

```
Spreadsheet::SetActiveSheet(
        Workbook,       ! (input) scalar string expression
        Name            ! (input) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Name A scalar string expression representing the sheet to be selected as the active sheet.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling this procedure explicitly before other procedures, the optional sheet argument can be omitted in those procedures.
- A call to another procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelSetActiveSheet`, which has become deprecated as of AIMMS 3.12.

4.4.4 Spreadsheet::SetVisibility

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::SetVisibility* (page 663) turns the visibility mode of the given Excel or OpenOffice Calc workbook on or off.

```
Spreadsheet::SetVisibility(
        Workbook,       ! (input) scalar string expression
        Visibility      ! (input) scalar element expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Visibility A scalar element expression in the pre-defined AIMMS set *OnOff* (page 814) specifying whether to show or hide the specified workbook.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- If the workbook is not yet open, it will be opened.
- Upto AIMMS 3.11 this function was known as `ExcelSetVisibility`, which has become deprecated as of AIMMS 3.12.

4.4.5 Spreadsheet::AssignValue

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::AssignValue* (page 663) writes a value or formula from AIMMS to an Excel or OpenOffice Calc cell or range of cells.

```
Spreadsheet::AssignValue(
        Workbook,       ! (input) scalar string expression
        Value,          ! (input) scalar expression
        Range,          ! (input) scalar string expression
        [Sheet]         ! (optional) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Value A scalar numerical, string, element-valued or unit-valued expression containing the value to be written to the spreadsheet.

Range A scalar string expression containing the range in the spreadsheet to which the *Value* should be written.

Sheet The sheet to which the *Value* should be written. Default is the active sheet.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelAssignValue`, which has become deprecated as of AIMMS 3.12.

4.4.6 Spreadsheet::SetOption

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::SetOption* (page 664) sets a global option that has an effect in all subsequent calls to the spreadsheet functions. Currently the following options are supported:

- `CalendarElementsAsStrings` By default elements in an AIMMS Calendar are communicated to the spreadsheet in a special date format, which is independent of the current time slot format in AIMMS. If this option is set to 1, the elements are communicated as a string, using the time slot format of the calendar.
- `WriteInfValueAsString` By default a value of INF or -INF in AIMMS is passed to the spreadsheet as a huge numeric number (1e150 and -1e150 respectively). If you set this option to 1, these values are written as a string "INF" or "-INF". Please be aware that in this case the cell will not have a numerical content which may cause problems in other code that is using the spreadsheet.

```
Spreadsheet::SetOption(
        Name,       ! (input) scalar string expression
        Value       ! (input) scalar expression
        )
```

Arguments

Name A scalar string representing the name of the option.

Value A scalar expression representing the new value for the option.

Return Value

The procedure returns 1 on success, or 0 otherwise.

4.4.7 Spreadsheet::SetUpdateLinksBehavior

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

This procedure specifies how Excel or OpenOffice Calc workbooks containing links to other workbooks should be opened. In the Excel case, such links can be either links to external workbooks or to remote workbooks. In the Calc case, this distinction is not made. If you do not call this procedure before using an Excel workbook containing links, you are prompted whether you want the links to be updated or not. In the OpenOffice case, you will get the default behavior as specified in the update setting*, if no Calc dialogs are required. This procedure is designed to give the AIMMS user control over the Excel and Calc behavior regarding links.

```
ExcelSetUpdateLinksBehavior(
        UpdateLinksBehavior  ! (input) scalar integer expression
        )
```

Arguments

UpdateLinksBehavior A scalar expression that sets the behavior of Excel or Calc when a workbook is opened. Possible values are:

- 0: (Excel) Excel prompts the user (the Excel default behavior).
- 1: (Excel) Do not update any links.
- 2: (Excel) Only update external links.
- 3: (Excel) Only update remote links
- 4: (Excel) Update both external and remote links
- 5: (Calc) Do not update any links.
- 6: (Calc) If the update setting in Calc* is 'Always', all links are updated. Otherwise, no links are updated (the Calc default behavior).
- 7: (Calc) Always update the links.

Argument values 0 to 4 are for Excel workbooks, values 5 to 7 are for OpenOffice Calc workbooks.
* This setting is called *Update links when opening* and can be found in the Calc menu, under Tools - Options - OpenOffice.org Calc - General.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter `CurrentErrorMessage` (page 835) contains a description of what went wrong.

Note:

- When the procedure is called, the setting remains valid for all consequent workbooks that will be opened, until the procedure is called again with a different setting.
- In case you use both Excel and Calc workbooks with links in your AIMMS application, you should call this function twice: once with an argument to control the Excel behavior, and once with an argument to control the Calc behavior. The setting of the first call will be remembered when you do the second call. For example: first call `Spreadsheet::SetUpdateLinksBehavior(1)`, to specify that Excel workbooks should not update their links, and then call `Spreadsheet::SetUpdateLinksBehavior(7)`, to specify that Calc workbooks should always update their links upon opening.
- Upto AIMMS 3.11 this function was known as `ExcelSetUpdateLinksBehavior`, which has become deprecated as of AIMMS 3.12.

4.4.8 Spreadsheet::AssignSet

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::AssignSet* (page 666) writes the elements of an AIMMS set into the given range of an Excel or OpenOffice Calc workbook.

```
Spreadsheet::AssignSet(
        Workbook,       ! (input) scalar string expression
        Set,            ! (input) set identifier
        Range,          ! (input) scalar string expression
        [Sheet]         ! (optional) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Set The AIMMS set to be written to the spreadsheet.

Range A scalar string expression containing the range in the sheet to which the *Set* should be written.

Sheet The sheet to which the *Set* should be written. Default is the active sheet.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelAssignSet`, which has become deprecated as of AIMMS 3.12.

4.4.9 Spreadsheet::RetrieveSet

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::RetrieveSet* (page 667) fills an AIMMS set based on the data in the given range of an Excel or OpenOffice Calc workbook.

```
Spreadsheet::RetrieveSet(
        Workbook,       ! (input) scalar string expression
        Set,            ! (output) set identifier
        Range,          ! (input) scalar string expression
        [Sheet],        ! (optional) scalar string expression
        [Mode]          ! (optional) scalar element expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Set The set to be filled.

Range The range in the workbook based on which the *Set* must be filled.

Sheet The sheet from which the data should be read. Default is the active sheet.

Mode Element in the pre-defined set *MergeReplace* (page 813). In *replace* mode, the AIMMS set is emptied before being filled. In *merge* mode, the new elements are added to the existing set. By default, the set is filled in replace mode.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelRetrieveSet`, which has become deprecated as of AIMMS 3.12.

4.4.10 Spreadsheet::RetrieveValue

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::RetrieveValue* (page 668) reads the value of an Excel or OpenOffice Calc cell into a scalar AIMMS parameter.

```
Spreadsheet::RetrieveValue(
        Workbook,       ! (input) scalar string expression
        Parameter,      ! (output) scalar identifier
        Range,          ! (input) scalar string expression
        [Sheet]         ! (optional) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Parameter A scalar numerical parameter, string parameter, element parameter or unit parameter to which the value from the *Range* will be written.

Range A scalar string expression containing a reference to the cell in the sheet from which the value will be read.

Sheet The sheet from which the value should be read. Default is the active sheet.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelRetrieveValue`, which has become deprecated as of AIMMS 3.12.

4.4.11 Spreadsheet::AssignParameter

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::AssignParameter* (page 669) writes data from the given parameter into the range of the Excel or OpenOffice Calc workbook.

```
Spreadsheet::AssignParameter(
        Workbook,        ! (input) scalar string expression
        Parameter,       ! (input) identifier
        Range,           ! (input) scalar string expression
        [Sheet],         ! (optional) scalar string expression
        [Sparse],        ! (optional) scalar binary expression
        [Transposed]     ! (optional) scalar binary expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Parameter The AIMMS identifier to be written to the spreadsheet. This can be a numerical parameter, an element parameter, a string parameter, a unit parameter or a variable. The dimension of this identifier can be 0, 1, or 2.

Range The range in the workbook into which the *parameter* must be written.

Sheet The sheet to which the *Value* should be written. Default is the active sheet.

Sparse If this argument is 1 (its default value), the default values of the parameter will be represented as empty cells in the sheet, instead of the real default value.

Transposed If this argument is 1, the parameter will be transposed before being displayed. The argument does not have any effect on scalar and one-dimensional data. The default value of this argument is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelAssignParameter`, which has become deprecated as of AIMMS 3.12.

4.4.12 Spreadsheet::RetrieveParameter

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::RetrieveParameter* (page 670) reads data from the given range in the Excel or OpenOffice Calc workbook into the specified AIMMS parameter.

```
Spreadsheet::RetrieveParameter(
        Workbook,        ! (input) scalar string expression
        Parameter,       ! (output) identifier
        Range,           ! (input) scalar string expression
        [Sheet],         ! (optional) scalar string expression
        [Transposed]     ! (optional) scalar binary expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Parameter The AIMMS identifier to be filled with spreadsheet data. This can be a numerical parameter, an element parameter, a string parameter, a unit parameter or a variable. The dimension of the parameter can be 0, 1 or 2.

Range The range in the workbook based on which the *parameter* must be filled.

Sheet The sheet in which the *Range* lies. Default is the active sheet.

Transposed If this argument is 1, the parameter is read transposed from the sheet. The argument does not have any effect on scalar and one-dimensional data. The default value for this argument is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelRetrieveParameter`, which has become deprecated as of AIMMS 3.12.

4.4.13 Spreadsheet::AssignTable

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::AssignTable* (page 671) writes tabular data to the specified Excel or OpenOffice Calc workbook.

```
Spreadsheet::AssignTable(
        Workbook,        ! (input) scalar string expression
        Parameter,       ! (input) identifier
        DataRange,       ! (input) scalar string expression
        [RowsRange],     ! (optional) scalar string expression
        [ColumnsRange],  ! (optional) scalar string expression
        [Sheet],         ! (optional) scalar string expression
        [Sparse],        ! (optional) scalar binary expression
        [RowMode],       ! (optional) scalar integer expression
        [ColumnMode]     ! (optional) scalar integer expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Parameter The AIMMS parameter to be written to the spreadsheet. This can be a numerical parameter, an element parameter, a string parameter, a unit parameter or a variable. The identifier must have a dimension greater than or equal to 1.

DataRange The range in the workbook into which the *Parameter* must be written.

RowsRange The range in the workbook into which the row labels must be written. The row labels are the elements of the sets that are identified by the first indices of *Parameter*. If the *RowsRange* is an $m \times n$-matrix, then the row labels are the elements of the sets of the first m indices of *Parameter*.

ColumnsRange The range in the workbook into which the column labels must be written. The column labels are the elements of the sets that are identified by the remaining indices of *Parameter* (the indices after those that constitute the *RowsRange*).

Sheet The sheet to which the *Parameter* should be written. Default is the active sheet.

Sparse If this argument is 1 (the default value), the default values of the *Parameter* will be represented as empty cells in the sheet, instead of the real default value.

RowMode Possible values are:

- 0: `SPARSE_OUTPUT`: Only those rows will be shown in the workbook, for which there exists at least one non-default data value. If no default data value exists for the row, neither the row labels nor the row data are displayed.
- 1: `DENSE_OUTPUT`: All rows (both the labels and the data) are shown in the workbook, even if all data values for a particular row are equal to the default value.
- 2: `USER_INPUT`: The row labels for which the data must be transferred to the workbook, must already be present in the workbook. This way, they serve as input to *Spreadsheet::AssignTable* (page 671).

- 3: NON_EXISTING: Use this mode to specify that no row labels must be printed, i.e. all indices should be represented by column labels. In this case the *RowsRange* argument does not need to be specified.

ColumnMode Possible values are:

- 0: SPARSE_OUTPUT: Only those columns will be shown in the workbook, for which there exists at least one non-default data value. If no default data value exists for the column, neither the column labels nor the column data are displayed.
- 1: DENSE_OUTPUT: All columns (both the labels and the data) are shown in the workbook, even if all data values for a particular column are equal to the default value.
- 2: USER_INPUT: The column labels for which the data must be transferred to the workbook, must already be present in the workbook. This way, they serve as input to *Spreadsheet::AssignTable* (page 671).
- 3: NON_EXISTING: Use this mode to specify that no column labels must be printed, i.e. all indices should be represented by row labels. In this case the *ColumnsRange* argument does not need to be specified.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as ExcelAssignTable, which has become deprecated as of AIMMS 3.12.

4.4.14 Spreadsheet::ClearRange

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::ClearRange* (page 673) empties the specified range in the specified sheet.

```
Spreadsheet::ClearRange(
        Workbook,               ! (input) scalar string expression
        Range,                  ! (input) scalar string expression
        [Sheet],                ! (optional) scalar string expression
        [IncludeCellFormatting] ! (optional) scalar binary expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Range A scalar string expression containing a reference to the range in the sheet that should be emptied.

Sheet The sheet from which the value should be read. Default is the active sheet. If the range is a uniquely named range, no active sheet needs to be set, since named ranges already contain a reference to a sheet.

IncludeCellFormatting When set to 1, the formatting of the cell (e.g. font size, color, ...) is also cleared. If set to 0, only the value of the cell is cleared.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelClearRange`, which has become deprecated as of AIMMS 3.12.

4.4.15 Spreadsheet::RetrieveTable

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::RetrieveTable* (page 674) reads tabular data from the specified Excel or OpenOffice Calc workbook.

```
Spreadsheet::RetrieveTable(
        Workbook,                 ! (input) scalar string expression
        Parameter,                ! (output) identifier
        DataRange,                ! (input) scalar string expression
        [RowsRange],              ! (optional) scalar string expression
        [ColumnsRange],           ! (optional) scalar string expression
        [Sheet]                   ! (optional) scalar string expression
        [AutomaticallyExtendSets] ! (optional) scalar binary expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Parameter The AIMMS parameter in which the data read from the spreadsheet will be stored. This can be a numerical parameter, an element parameter, a string parameter, a unit parameter or a variable. The identifier must have a dimension greater than or equal to 1.

DataRange The range in the workbook from which the data must be read.

RowsRange The range in the workbook from which the row labels must be read. The row labels will be added to the sets that are identified by the first indices of *Parameter*. If the *RowsRange* is an $m \times n$-matrix (*m columns, n rows), then the row labels are the elements of the sets of the first :math:`m` indices of Parameter.*

ColumnsRange The range in the workbook from which the column labels must be read. The column labels will be added to the sets that are identified by the remaining indices of *Parameter* (the indices after those that constitute the *RowsRange*).

Sheet The sheet to which the *Parameter* should be written. Default is the active sheet.

AutomaticallyExtendSets Indicates whether AIMMS should automatically extend the domain set of an identifier if necessary. If not, an error will be generated. The default value of this argument is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelRetrieveTable`, which has become deprecated as of AIMMS 3.12.

4.4.16 Spreadsheet::AddNewSheet

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::AddNewSheet* (page 675) adds a new empty sheet to the specified Excel or OpenOffice Calc workbook.

```
Spreadsheet::AddNewSheet(
        Workbook,       ! (input) scalar string expression
        Name,           ! (input) scalar string expression
```

(continues on next page)

(continued from previous page)

```
        [SetAsActive],   ! (optional) scalar binary expression
        [Hidden]         ! (optional) scalar binary expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Name The name to assign to the new sheet.

SetAsActive If this parameter is 1, the sheet is set as the active sheet. The default value of this argument is 1.

Hidden If this parameter is 1, the sheet is created as a hidden sheet. The default value of this argument is 0.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelAddNewSheet`, which has become deprecated as of AIMMS 3.12.

4.4.17 Spreadsheet::CopyRange

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::CopyRange* (page 676) copies the contents of a complete Excel or OpenOffice Calc range to another Excel/Calc range.

```
Spreadsheet::CopyRange(
        Workbook,          ! (input) scalar string expression
        SourceRange,       ! (input) scalar string expression
        DestinationRange,  ! (input) scalar string expression
        [SourceSheet],     ! (optional) scalar string expression
        [DestinationSheet] ! (optional) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

SourceRange A scalar string expression containing a reference to the range in the spreadsheet that should be copied from.

DestinationRange A scalar string expression containing a reference to the range in the spreadsheet that should be copied to.

SourceSheet The sheet containing the *SourceRange*. Default is the active sheet. If the source range is a uniquely named range, no active sheet needs to be set, since named ranges already contain a reference to a sheet.

DestinationSheet The sheet containing the *DestinationRange*. Default is the active sheet. If the destination range is a uniquely named range, no active sheet needs to be set, since named ranges already contain a reference to a sheet.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet arguments can be omitted in this procedure. The active sheet will then be used both for the source and the destination sheet of *Spreadsheet::CopyRange* (page 676).
- In case that the active sheet was not set before the call to this function, the active sheet is set to the *SourceSheet* argument, if supplied. If the *SourceSheet* argument is not supplied, the active sheet is set to the *DestinationSheet* argument, if supplied. Otherwise, the active sheet is not changed.
- Upto AIMMS 3.11 this function was known as `ExcelCopyRange`, which has become deprecated as of AIMMS 3.12.

4.4.18 Spreadsheet::DeleteSheet

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::DeleteSheet* (page 677) deletes the given sheet from the specified Excel or OpenOffice Calc workbook.

```
Spreadsheet::DeleteSheet(
        Workbook,       ! (input) scalar string expression
        Name            ! (input) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Name The name of the sheet to be deleted.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelDeleteSheet`, which has become deprecated as of AIMMS 3.12.

4.4.19 Spreadsheet::GetAllSheets

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::GetAllSheets* (page 678) obtains the names of all sheets currently present in the specified Excel or OpenOffice Calc workbook.

```
Spreadsheet::GetAllSheets(
        Workbook,       ! (input) scalar string expression
        SheetNames      ! (input) 1-dimensional string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Name A 1-dimensional string parameter, which after successful execution will contain all present sheet names of the supplied workbook. The root set of the index should be a subset of Integers.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note: None.

4.4.20 Spreadsheet::RunMacro

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::RunMacro* (page 679) executes an Excel or OpenOffice Calc macro.

```
Spreadsheet::RunMacro(
        Workbook,           ! (input) scalar string expression
        Name,               ! (input) scalar string expression
        [MacroArgument01], ! (optional) scalar expression
        ...
        [MacroArgument30], ! (optional) scalar expression
        [Sheet]             ! (optional) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

Name

The name of the macro to be executed. Please note that in the Excel case you need to specify the fully qualified name here. If, for example, you have a macro called `ThisWorkbook.MyMacro`, only specifying `MyMacro` isn't sufficient. For the full name of an Excel macro, please refer to your Excel workbook and look under *Tools - Macro - Macros...*. Only in case you have created a so-called Visual Basic *Module* in your Excel workbook, you can just use the short name of your macro. Furthermore, it's also possible to call macro's which are located in a different workbook than the workbook it should be applied upon. In such cases, use the `WorkbookContainingMacro!MacroName` format for the name of the macro. Also, you have to make sure that the workbook containing the macro is opened before the call to RunMacro, since only macro's in opened workbooks can be found by Excel.

For OpenOffice Calc macros, you'll also need to specify the full path of a macro, for example `"TheLibrary.TheModule.TheMacroToCall"`. Please note that Calc macros can be stored at either document scope, or at application scope. In the former case, the macros are stored within your document(i.e. `.ods` file), allowing you to distribute them easily to other users. In the latter case, the macros are stored in the Calc application on your machine, making it a bit harder to share your macros with other users, but enabling you to create macros that can be applied to all your workbooks.

By default, AIMMS assumes that the `Name` argument specifies a macro stored at document scope, since that is the more likely scenario for AIMMS use in combination with Calc. In case you want to call a macro at application scope, the `Name` argument should start with `"Global."` (case sensitive), for example `"Global.TheLibrary.TheDocument.TheMacroToCall"`.

AIMMS does not support the calling of the OpenOffice standard macros (those are the macros under the `OpenOffice.org Macros` branch in the macro tree in OpenOffice).

MacroArgument01...MacroArgument30 A list of arguments to be passed to the macro. A maximum of 30 arguments is allowed. Only scalar arguments are supported. The scalar values can be of any type (numerical parameter, string parameter, element parameter, unit parameter, literal or variable). Furthermore, only input arguments are allowed.

Sheet The sheet on which the macro should be applied. Please note: in a macro, it is possible to specify on which sheet certain actions should be performed. Clearly, in that case the Sheet argument does not influence this.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- Element parameters that are passed as macro argument are usually passed to the workbook as strings, except when their range is a subset of integers.
- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelRunMacro`, which has become deprecated as of AIMMS 3.12.

4.4.21 Spreadsheet::CloseWorkbook

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::CloseWorkbook* (page 680) closes the specified Excel or OpenOffice Calc workbook. Internally, AIMMS keeps the workbook open from the moment that a procedure is applied on it for the first time. This is good for performance. Nevertheless, the user can specify that he is finished with the workbook and that the workbook can be closed. If a workbook is not closed explicitly, and changes have been made to it, the user is asked whether or not to save it just before closing the AIMMS project.

```
Spreadsheet::CloseWorkbook(
        Workbook,          ! (input) scalar string expression
        SaveBeforeClose    ! (input) scalar binary expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

SaveBeforeClose If this argument is 1, the workbook is saved before it is closed.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelCloseWorkbook`, which has become deprecated as of AIMMS 3.12.

4.4.22 Spreadsheet::CreateWorkbook

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::CreateWorkbook* (page 681) creates a new Excel or OpenOffice Calc workbook. In the Calc case, the workbook contains three empty sheets. In the Excel case, it is dependant of an Excel setting how many sheets the workbook contains. The first sheet is automatically set as the active sheet.

```
Spreadsheet::CreateWorkbook(
        WorkbookName,   ! (input) scalar string expression
        [SheetName]     ! (optional) scalar string expression
        )
```

Arguments

WorkbookName The name under which the workbook will be known in AIMMS. In later calls to other procedures, *WorkbookName* has to be specified as the *Workbook* argument. When the workbook should eventually be saved in a particular path, then this path can be included in this argument. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is used.

SheetName The name of the first sheet of the new workbook. If this argument is omitted, the sheet will be determined by the spreadsheet application ("Sheet1" in the English version). This sheet will automatically be set as the active sheet.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelCreateWorkbook`, which has become deprecated as of AIMMS 3.12.

4.4.23 Spreadsheet::SaveWorkbook

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::SaveWorkbook* (page 682) saves the specified Excel or OpenOffice Calc workbook. The workbook is saved with the name under which it is known in AIMMS, unless the *SaveAsName* argument is specified. Only when the *SaveAsName* argument is specified, or when dealing with a workbook that has never been saved before (i.e. created by a call to `Spreadsheet::CreateWorkbook`), and a workbook with the same name already exists on disk, the user is prompted with the question whether or not to overwrite the existing file.

```
Spreadsheet::SaveWorkbook(
        Workbook,       ! (input) scalar string expression
        [SaveAsName]    ! (optional) scalar string expression
        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

SaveAsName The (new) name to be used for saving the workbook.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- Upto AIMMS 3.11 this function was known as `ExcelSaveWorkbook`, which has become deprecated as of AIMMS 3.12.

4.4.24 Spreadsheet::Print

Warning: *Spreadsheet Functions* (page 660) are deprecated. One may use the AIMMS Excel Library - AXLL or the Data Exchange Library.

The procedure *Spreadsheet::Print* (page 682) makes it possible to print an Excel or OpenOffice Calc sheet from AIMMS.

```
Spreadsheet::Print(
        Workbook,          ! (input) scalar string expression
        Range,             ! (input) scalar string expression
        [Sheet],           ! (optional) scalar string expression
        [ShowPreview],     ! (optional) scalar binary expression
        [NumberOfCopies], ! (optional) scalar integer expression
        [Collate],         ! (optional) scalar binary expression
        [ActivePrinter]    ! (optional) scalar string expression

        )
```

Arguments

Workbook A scalar string expression representing the Excel or Calc workbook. If this argument ends in `.ods`, OpenOffice Calc is used. Otherwise, Excel is

Range The range to be printed.

Sheet The sheet on which the range lies.

ShowPreview If this argument is 1, Excel or Calc shows a print preview window before printing. The visibility mode of the workbook should be 'On' in this case. The default value of this argument is 0. In the preview window, you can decide whether to actually print or to cancel the printing.

NumberOfCopies The number of copies to print. The default value of this argument is 1.

Collate If this argument is 1, and more than one copy of the sheet is printed, the printed sheets are collated neatly. The default value of this argument is 1.

ActivePrinter The user can specify the name of the printer to be used for printing the sheet. The default printer is used by default.

Return Value

The procedure returns 1 on success, or 0 otherwise. In case of an error the pre-defined AIMMS parameter *CurrentErrorMessage* (page 835) contains a description of what went wrong.

Note:

- By calling the procedure *Spreadsheet::SetActiveSheet* (page 662) you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- Upto AIMMS 3.11 this function was known as `ExcelPrint`, which has become deprecated as of AIMMS 3.12.

The functions operate on OpenOffice Calc workbooks, if the `WorkbookName` argument ends in `.ods`. In all other cases, the functions operate on Excel workbooks.

4.5 XML Functions

AIMMS supports the following functions for reading and writing XML files:

4.5.1 GenerateXML

The procedure *GenerateXML* (page 684) generates XML output data for a given set of AIMMS identifiers.

```
GenerateXML(
        XMLFile,          ! (input) scalar string expression
        IdentifierSet,    ! (input) set expression
        Merge,            ! (optional) 0 or 1
        SchemaFile        ! (optional) scalar string expression
        )
```

Arguments

XMLFile Name of the file to which the generated XML must be written.

IdentifierSet A subset of the predefined set *AllIdentifiers* (page 821), containing the set of identifiers for which XML output must be generated.

Merge (optional) Indicates whether or not the contents of the file can be merged within another XML file.

SchemaFile (optional) If this argument is specified, a schema corresponding to the generated XML data will be written to the specified file name. A namespace will be generated for this schema file, and added to the `xmlns` attribute of the root element of the generated XML file.

Return Value

The procedure returns 1 on success. or 0 on failure.

Note: Notice that the `Merge` attribute does *not* mean that the generated XML will be appended to the specified XML file. The latter will *always* be overwritten. If the *Merge* argument is non-zero, AIMMS will omit the XML header from the generated file, allowing you to merge its contents into another XML document.

See also:

The procedures *ReadGeneratedXML* (page 684), *ReadXML* (page 685), *WriteXML* (page 686). Generating XML data is discussed in full detail in Reading and Writing AIMMS-generated XML Data of the Language Reference.

4.5.2 ReadGeneratedXML

The procedure *ReadGeneratedXML* (page 684) reads the contents of an AIMMS-generated XML data file.

```
ReadGeneratedXML(
        XMLFile,       ! (input) scalar string expression
        merge          ! (optional) 0 or 1
        )
```

Arguments

XMLFile Name of the AIMMS-generated XML file to read.

merge (optional) With this optional argument (default 0), you can choose whether you want to merge the data included in the XML file with the existing data, or overwrite any existing data (default)

Return Value

The procedure returns 1 if the XML file is read successfully, or 0 otherwise.

See also:

The procedures *GenerateXML* (page 684), *ReadXML* (page 685), *WriteXML* (page 686). Generating XML data is discussed in full detail in Reading and Writing AIMMS-generated XML Data of the Language Reference.

4.5.3 ReadXML

The procedure *ReadXML* (page 685) you can read an XML data file according to a given user-defined XML format.

```
ReadXML(
        XMLFile,          ! (input) scalar string expression
        MappingFile,      ! (input) scalar string expression
        merge,            ! (optional) 0 or 1
        SchemaFile        ! (optional) scalar string expression
        )
```

Arguments

XMLFile The name of the file from which the XML data must be read

MappingFile The name of the file containing the mapping between the user-defined XML format and the identifiers in your model.

merge (optional) With this optional argument (default 0), you can choose whether you want to merge the data included in the XML file with the existing data, or overwrite any existing data (default)

SchemaFile If you specify the name of a schema file through this argument, AIMMS will validate the contents of the XML data file against this schema prior to reading it into AIMMS.

Return Value

The procedure returns 1 if successful, or 0 otherwise.

Note: The namespace defined in the schema file (if specified) must match the namespace specified in the `xmlns` attribute of the root element in the XML data file.

See also:

The procedures *GenerateXML* (page 684), *ReadGeneratedXML* (page 684), *WriteXML* (page 686). Reading user-defined XML data is discussed in full detail in Reading and Writing User-Defined XML Data of the Language Reference.

4.5.4 WriteXML

With the procedure *WriteXML* (page 686) you write an XML data file according to a given user-defined XML format.

```
WriteXML(
        XMLFile,          ! (input) scalar string expression
        MappingFile,      ! (input) scalar string expression
        Merge             ! (optional) 0 or 1
        )
```

Arguments

XMLFile The name of the file to which the XML data must be written

MappingFile The name of the file containing the mapping between the user-defined XML format and the identifiers in your model.

Merge (optional) Indicates whether or not the contents of the file can be merged within another XML file.

Return Value

The procedure returns 1 if successful, or 0 otherwise.

Note: Notice that the `merge` attribute does *not* mean that the generated XML will be appended to the specified XML file. The latter will *always* be overwritten. If the *merge* argument is non-zero, AIMMS will omit the XML header from the generated file, allowing you to merge its contents into another XML document.

See also:

The procedures *GenerateXML* (page 684), *ReadGeneratedXML* (page 684), *ReadXML* (page 685). Writing user-defined XML data is discussed in full detail in Reading and Writing User-Defined XML Data of the Language Reference.

CHAPTER

FIVE

USER INTERFACE RELATED FUNCTIONS

5.1 Dialog Functions

AIMMS supports the following functions for simple interaction with the end user.

5.1.1 DialogAsk

The procedure *DialogAsk* (page 687) displays a small dialog box containing a message and two or three buttons. Usually these buttons are an **OK** and **Cancel**, or **Yes**, **No** and **Cancel**, but they can contain any text you want. The procedure returns the number of the button that is pressed by the user.

```
DialogAsk(
        message,        ! (input) string expression
        button1,        ! (input) string expression
        button2,        ! (input) string expression
        [button3]       ! (optional) string expression
        [title]         ! (optional) title of dialog box
        )
```

Arguments

message A scalar string expression containing the text you want to display in the dialog box.

button1 A scalar string expression containing the text of the first button.

button2 A scalar string expression containing the text of the second button.

button3 (optional) A scalar string expression containing the text of the third button. If this argument is omitted then the dialog box will only show two buttons.

title A scalar string expression containing the text that you want to appear in the title of the dialog box.

Return Value

The procedure returns the number of the button that is pressed: 1 for the first button, 2 for the second button or 3 for the third button.

Note: If the user presses the **Esc** key, or closes the dialog box via the [x] in the top right corner, then this is interpreted as pressing the last button in the dialog box (which is usually the **Cancel** button).

See also:

The procedures *DialogMessage* (page 693), *DialogError* (page 688).

5.1.2 DialogError

The procedure *DialogError* (page 688) displays a small dialog box containing a specified error message and an **OK** button. The execution will be halted until the user presses the **OK** button.

```
DialogError(
        message,          ! (input) string expression
        [title]           ! (optional) title of dialog box
        )
```

Arguments

message A scalar string expression containing the text you want to display in the dialog box.

title A scalar string expression containing the text that you want to appear in the title of the dialog box.

Note: The procedures `DialogMessage` and *DialogError* (page 688) only differ in the icon that is displayed at the left side of the dialog box.

See also:

The procedures *DialogMessage* (page 693), *DialogAsk* (page 687), *DialogProgress* (page 694).

5.1.3 DialogGetColor

The procedure *DialogGetColor* (page 688) displays a standard Windows color selection dialog box. The procedure returns the color (RGB values) selected by the user.

```
DialogGetColor(
        r,            ! (input/output) scalar numerical parameter
        g,            ! (input/output) scalar numerical parameter
        b             ! (input/output) scalar numerical parameter
        )
```

Arguments

r A scalar numerical paramter containing the red value of the selected color.

g A scalar numerical paramter containing the green value of the selected color.

b A scalar numerical paramter containing the blue value of the selected color.

Return Value

The procedure returns 1 if the user completed the color selection dialog box successfully, or 0 otherwise.

5.1.4 DialogGetDate

The procedure *DialogGetDate* (page 689) displays a standard Windows date selection dialog box. The procedure returns the date (in the specified format) selected by the user.

```
DialogGetDate(
        title,          ! (input) string expression
        format,         ! (input) string expression
        date,           ! (input/output) scalar string parameter
        [nr_rows,]      ! (optional) integer expression
        [nr_columns]    ! (optional) integer expression
        )
```

Arguments

title A scalar string expression containing the text you want to display in the title of the dialog box.

format A scalar string expression containing the date format of the *date* argument.

date A scalar string parameter in which the selected date is returned according to the date format specified in *format*.

nr_rows (optional) A scalar integer expression in the range $1, \ldots, 3$ containing the number of rows to be displayed in the date selectiond dialog box.

nr_columns (optional) A scalar integer expression in the range $1, \ldots, 4$ containing the number of columns to be displayed in the date selectiond dialog box.

Return Value

The procedure returns 1 if the user completed the date selection dialog box successfully, or 0 otherwise.

Note: If the *date* argument contains a valid date according to the format specified in *date-format*, AIMMS will set the initial date in the date selection dialog box equal to the specified date.

See also:

The date format specification components are discussed in full detail in Date-specific Components of the Language Reference.

5.1.5 DialogGetElement

The procedure *DialogGetElement* (page 689) displays a dialog box in which the user can select an element from a list of set elements.

```
DialogGetElement(
        title,          ! (input) string expression
        reference       ! (input/output) scalar element parameter
        )
```

Arguments

title A scalar string expression containing the text you want to display as title of the dialog box.

reference A scalar element parameter. When creating the dialog box, the range set of this parameter is used to fill the list with elements, and the current value of the element parameter will be initially selected. On return, this parameter will refer to the selected element.

Return Value

The procedure returns 1 if the user has pressed the **OK** button, and 0 if he has pressed the **Cancel** button.

See also:

The procedures *DialogGetElementByText* (page 691), *DialogGetElementByData* (page 690), *DialogGetNumber* (page 691).

5.1.6 DialogGetElementByData

The procedure *DialogGetElementByData* (page 690) is an extension of the procedure `DialogGetElementByText`. Instead of only showing a list box with only a single string per element, this procedure allows you to show a list box with multiple columns of text per element. The text that is displayed in each column is specified via a 2-dimensional string parameter. The first dimension of this parameter corresponds to the rows of the list box, the second dimension corresponds to the column in the listbox.

```
DialogGetElementByData(
        title,          ! (input) string expression
        reference,      ! (input/output) scalar element parameter
        element_data    ! (input) 2-dimensional string parameter
        )
```

Arguments

title A scalar string expression containing the text you want to display as title of the dialog box.

reference A scalar element parameter. When creating the dialog box, the range set of this parameter is used to fill the list with elements, and the current value of the element parameter will be initially selected. On return, this parameter will refer to the selected element.

element_data A 2-dimensional string parameter. The first index in its domain should matches the range set of the element parameter *reference*, the second index defines the number of columns that are shown. Instead of the element names, the dialog box will display multiple columns of text derived from this parameter.

Return Value

The procedure returns 1 if the user has pressed the **OK** button, and 0 if he has pressed the **Cancel** button.

See also:

The procedures *DialogGetElement* (page 689), *DialogGetElementByText* (page 691).

5.1.7 DialogGetElementByText

The procedure *DialogGetElementByText* (page 691) displays a dialog box in which the user can select an element from a set. However, other than `DialogGetElement`, this procedure does not show a list of element names but a list of strings, which are given as a separate argument to the procedure.

```
DialogGetElementText(
        message,        ! (input) string expression
        reference,      ! (input/output) scalar element parameter
        element_text    ! (input) 1-dimensional string parameter
        )
```

Arguments

message A scalar string expression containing the text you want to display as title of the dialog box.

reference A scalar element parameter. When creating the dialog box, the range set of this parameter is used to fill the list with elements, and the current value of the element parameter will be initially selected. On return, this parameter will refer to the selected element.

element_text A 1-dimensional string parameter, with a domain that matches the range set of the element parameter *reference*. Instead of the element names, the dialog box will display the corresponding strings of this parameter.

Return Value

The procedure returns 1 if the user has pressed the **OK** button, and 0 if he has pressed the **Cancel** button.

See also:

The procedures *DialogGetElement* (page 689), *DialogGetElementByData* (page 690).

5.1.8 DialogGetNumber

The procedure *DialogGetNumber* (page 691) displays a small dialog box in which the user can enter a single numerical value. The dialog box remains on the screen (and thus halts the execution) until the user presses either the **OK** or the **Cancel** button.

```
DialogGetNumber(
        message,        ! (input) string expression
        reference,      ! (input/output) scalar numerical identifier
        [decimals,]     ! (optional) integer
        [title]         ! (optional) string expression
        )
```

Arguments

message A scalar string expression containing the text you want to display in front of the edit field.

reference A scalar identifier. When creating the dialog box, its value is used to fill the edit field. After the user presses the **OK** button, the edited value is returned through this argument.

decimals A integer expression to indicate the number of decimals that is displayed initially.

title A scalar string expression containing the text that you want to appear in the title of the dialog box.

Return Value

The procedure returns 1 if the user has pressed the **OK** button, and 0 if he has pressed the **Cancel** button.

See also:

The procedures *DialogGetString* (page 693), *DialogGetElement* (page 689).

5.1.9 DialogGetPassword

The procedure *DialogGetPassword* (page 692) displays a small dialog box in which the user can enter a password string. In the dialog box the string is presented by a sequence of asterisks. The dialog box remains on the screen (and thus halts the execution) until the user presses either the **OK** or the **Cancel** button.

```
DialogGetPassword(
        message,        ! (input) string expression
        password,       ! (input/output) scalar string parameter
        [title]         ! (optional) string expression
        )
```

Arguments

message A scalar string expression containing the text you want to display in front of the edit field.

password A scalar string valued identifier containing the password. When creating the dialog box, its value is used to fill the edit field. After the user presses the **OK** button, the edited password string is returned through this argument.

title A scalar string expression containing the text that you want to appear in the title of the dialog box.

Return Value

The procedure returns 1 if the user has pressed the **OK** button, and 0 if he has pressed the **Cancel** button.

See also:

The procedure *DialogGetString* (page 693).

5.1.10 DialogGetString

The procedure *DialogGetString* (page 693) displays a small dialog in which the user can enter a text string. The dialog remains on the screen (and thus halts the execution) until the user presses either the **OK** or the **Cancel** button.

```
DialogGetString(
        message,        ! (input) string expression
        reference,      ! (input/output) scalar string parameter
        [title]         ! (optional) string expression
        )
```

Arguments

message A scalar string expression containing the text you want to display in front of the edit field.

reference A scalar string valued identifier. When creating the dialog, its value is used to fill the edit field. After the user presses the **OK** button, the edited string is returned through this argument.

title A scalar string expression containing the text that you want to appear in the title of the dialog box.

Return Value

The procedure returns 1 if the user has pressed the **OK** button, and 0 if he has pressed the **Cancel** button.

See also:

The procedures *DialogGetNumber* (page 691), *DialogGetPassword* (page 692), *DialogGetElement* (page 689).

5.1.11 DialogMessage

The procedure *DialogMessage* (page 693) displays a small dialog box containing a specified informational message and an **OK** button. The execution will be halted until the user presses the **OK** button.

```
DialogMessage(
        message,        ! (input) string expression
        [title]         ! (optional) string expression
        )
```

Arguments

message A scalar string expression containing the text you want to display in the dialog box.

title A scalar string expression containing the text that you want to appear in the title of the dialog box.

Note: The procedures *DialogMessage* (page 693) and `DialogError` only differ in the icon that is displayed at the left side of the dialog box

---

See also:

The procedures *DialogError* (page 688), *DialogAsk* (page 687).

5.1.12 DialogProgress

The procedure *DialogProgress* (page 694) displays a small dialog box containing a specified message and a progress bar that can indicate how much of a specific task has already been processed. This dialog box will not halt the execution, and you can call the procedure sequentially during a timely task to change either the displayed message or the length of the progress bar.

```
DialogProgress(
        message,         ! (input) string expression
        [percentage]     ! (optional) integer expression
        )
```

Arguments

message A scalar string expression containing the text you want to display in the dialog box.

percentage (optional) A scalar value between 0 and 100. It is used to set the length of the progress bar at the bottom of the dialog box. If this argument is omitted then the progress bar is not displayed.

Note: The progress dialog box does not adjust the length of the progress bar itself, so you must do it yourself by sequentially calling the procedure with an increasing percentage. The progress dialog box is automatically removed from the screen if the execution terminates. If you want to remove the dialog box yourself, then you should call *DialogProgress* (page 694) with an empty message string: `DialogProgress("")`.

See also:

The procedures *DialogMessage* (page 693), *DialogError* (page 688), *DialogAsk* (page 687).

5.1.13 StatusMessage

With the procedure *StatusMessage* (page 695) you can display a short message in the status bar at the bottom of the AIMMS window.

```
StatusMessage(
        message          ! (input) string expression
        )
```

Arguments

message A scalar string expression containing the text you want to display in the status bar.

Note: If you have set the status bar to be hidden (via the project options), then the message will not be visible to the user.

See also:

The procedures *DialogMessage* (page 693), *DialogProgress* (page 694).

5.2 Page Functions

AIMMS supports the following functions for opening, closing, and manipulating the pages in the interface:

5.2.1 PageClose

With the procedure *PageClose* (page 695) you can close a page that is currently open.

```
PageClose(
        page             ! (optional) string expression
        )
```

Arguments

page (optional) A string expression representing the name of the page that you want to close. This name is the unique name as it appears in the Page Manager tree. If you omit this argument, then *PageClose* (page 695) closes the currently active page.

Return Value

The procedure returns 1 if the page is closed successfully, or a 0 otherwise.

Note: The active page can be obtained by *PageGetActive* (page 698).

See also:

The procedures *PageOpen* (page 705), *PageGetActive* (page 698), and *PageOpenSingle* (page 705).

5.2.2 PageCopyTableToClipboard

With the procedure *PageCopyTableToClipboard* (page 696) you can copy (part of) a specific table on a specific page to the clipboard, so that you subsequently can paste it in any other application.

```
PageCopyTableToClipboard(
        pageName,          ! (input) scalar string expression
        tag,               ! (input) scalar string expression
        includeHeaders,    ! (input) scalar numerical expression
        selectionOnly      ! (input) scalar numerical expression
        )
```

Arguments

pageName A string expression representing the name of the page containing the table.

tag A string expression representing the tag name of the table for which you want to copy the current displayed data. This can be a Composite Table, a Pivot Table or an standard Table object.

includeHeaders A scalar numerical expression to control whether or not the headers should be copied as well. If `includeHeaders` is not equal to 0 then the headers are included.

selectionOnly A scalar numerical expression to control whether the entire table or only the currently selected cells should be copied. If `selectionOnly` is not equal to 0 then only the currently selected cells (with or without the corresponding headers, based on the value of `includeHeaders`) are copied.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0 and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: You can specify a unique tag name for each page object via the object properties.

See also:

The procedure *PageCopyTableToExcel* (page 697).

5.2.3 PageCopyTableToExcel

With the procedure *PageCopyTableToExcel* (page 697) you can copy (part of) a specific table on a specific page directly to a range in Excel.

```
PageCopyTableToExcel(
        pageName,          ! (input) scalar string expression
        tag,               ! (input) scalar string expression
        includeHeaders,    ! (input) scalar numerical expression
        selectionOnly,     ! (input) scalar numerical expression
        ExcelWorkbook,     ! (input) scalar string expression
        Range,             ! (input) scalar string expression
        [Sheet]            ! (optional) scalar string expression
        )
```

Arguments

pageName A string expression representing the name of the page containing the table.

tag A string expression representing the tag name of the table for which you want to copy the current displayed data. This can be a Composite Table, a Pivot Table or an standard Table object.

includeHeaders A scalar numerical expression to control whether or not the headers should be copied as well. If `includeHeaders` is not equal to 0 then the headers are included.

selectionOnly A scalar numerical expression to control whether the entire table or only the currently selected cells should be copied. If `selectionOnly` is not equal to 0 then only the currently selected cells (with or without the corresponding headers, based on the value of `includeHeaders`) are copied.

ExcelWorkbook A scalar string expression representing the Excel workbook.

Range A scalar string expression containing the (named) range in the Excel sheet to which the table should be copied.

Sheet The sheet to which the table should be copied. Default is the active sheet.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0 and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- By calling the procedure `ExcelSetActiveSheet` you can set the active sheet, after which the optional sheet argument can be omitted in procedures like this one.
- A call to this procedure with a specified sheet argument does not change the active sheet, except when the workbook does not have an active sheet yet.
- When the dimensions of the specified range do no match the dimensions of the table on the clipboard, then the standard Excel rules for pasting are applied. That is:
  - if the range is only one column wide, then the range will automatically be expanded horizontally to match the number of columns on the clipboard,
  - else if the number of columns in the range is smaller than the number of columns on the clipboard then only the first columns that fit will be copied,
  - else if the number of columns in the range is larger than the number of columns on the clipboard, the range is made smaller.

  A similar algorithm is used for the number of rows. So if you want to make sure that the entire contents of the copied table is pasted in Excel, you can best specify a range of exactly one cell.
- You can specify a unique tag name for each page object via the object properties.

See also:

The procedure *PageCopyTableToClipboard* (page 696).

5.2.4 PageGetActive

With the procedure *PageGetActive* (page 698) you can retrieve the name of the currently active page.

```
PageGetActive(
        page            ! (output) scalar string identifier
        )
```

Arguments

page A string identifier to hold the name of the page that is currently active. If the same page name is used in more than one (library) project, then the prefix of the library project (or `::` in case of the main project) will be prepended.

Return Value

The procedure returns 1 on success, or 0 if there is no currently active page.

See also:

The procedures *PageGetFocus* (page 700) and *PageClose* (page 695).

5.2.5 PageGetAll

With the procedure *PageGetAll* (page 699) you can retrieve the names of all pages and/or templates in your project

```
PageGetAll(
        page_set,           ! (output) an (empty) root set
        IncludePages,       ! (optional, default 1) scalar expression
        IncludeTemplates,   ! (optional, default 1) scalar expression
        ExcludeHidden,      ! (optional, default 0) scalar expression
        ExcludePrintables   ! (optional, default 0) scalar expression
        )
```

Arguments

page_set A root set, that on return will contain the names of all the requested pages.

IncludePages A scalar numerical expression to indicate whether the returned set should contain the names of pages in your project.

IncludeTemplates A scalar numerical expression to indicate whether the returned set should contain the names of templates in your project.

ExcludeHidden A scalar numerical expression to indicate whether hidden pages should be part of the returned set. If ExcludeHidden is set to 1 then the returned set will not contain any page that is currenlty hidden.

ExcludePrintables A scalar numerical expression to indicate whether print pages or print templates should be part of the returned set. Print pages/templates are those pages/templates that are especially created for printing (i.e. in the Template Manager they are placed as children of a root print template). If ExcludePrintables is set to 1 then the returned set will not contain any printable page or template.

Return Value

The procedure returns 1 on success, and 0 on failure.

See also:

The procedures *PageGetNext* (page 701), *PageGetPrevious* (page 703), *PageGetChild* (page 699), *PageGetParent* (page 702), *PageGetNextInTreeWalk* (page 702).

5.2.6 PageGetChild

The procedure *PageGetChild* (page 699) retrieves the name of the first child page for a specific page in the Page Manager tree.

```
PageGetChild(
        page,               ! (input) scalar string expression
        childpage,          ! (output) scalar string identifier
        IncludeHiddenPages  ! (optional) scalar numerical expression
        )
```

Arguments

page A string expression containing the name of a (parent) page in the Page Manager tree.

childpage A scalar string identifier to hold the name of the first child page beneath the given parent page (if any).

IncludeHiddenPages A scalar numerical expression to indicate whether hidden pages should be taken into account. If IncludeHiddenPages is set to 1 then the resulting child page may be a page that is currently hidden, otherwise these hidden pages are skipped. The default is 0.

Return Value

The procedure returns 1 on success, or 0 if the given page name does not exist or if the page does not have any child pages.

See also:

The procedures *PageGetParent* (page 702), *PageGetNext* (page 701), *PageGetPrevious* (page 703), *PageGetNextInTreeWalk* (page 702), *PageGetAll* (page 699).

5.2.7 PageGetFocus

With the procedure *PageGetFocus* (page 700) you can retrieve the name of the currently active page.

```
PageGetFocus(
        page,           ! (output) scalar string identifier
        tag,            ! (output) scalar string identifier
        [fullPathTag]   ! (optional) 0 or 1
        )
```

Arguments

page A string identifier to hold the name of the currently active page. If the same page name is used in more than one (library) project, then the prefix of the library project (or `::` in case of the main project) will be prepended.

tag A string identifier to hold the tag name of the object that currently has the keyboard input focus.

fullPathTag (optional) If this value is set to 0, then returned tag will be the simple tag name of the object that has focus. If this value is set to 1 (the default), then the returned tag name will also contain the tags of Tabbed or Indexed Page objects in which the object with focus is contained. See the remarks below.

Return Value

The procedure returns 1 on success, or 0 if there is no currently active page or if no object has the input focus.

Note: You can specify a unique tag name for each page object via the object properties. If no tag name has been given explicitly, then the type of object is returned ("Table", "Bar Chart", etc.) If an object with tag "X" is displayed in a tabbed page object with tag "T", then the full path tag name will be "T::X". If an object with tag "X" is displayed in an indexed page object with tag "IP" on a row and column that corresponds with elements "rowi" and "colj", then the full path tag name will be "IP('rowi','colj')::X".

See also:

The procedures *PageSetFocus* (page 707), *PageGetActive* (page 698).

5.2.8 PageGetNext

The procedure *PageGetNext* (page 701) retrieves the name of the next page for a specific page in the Page Manager tree. The next page is the page that has the same parent page, and is positioned directly below the given page.

```
PageGetNext(
        page,                       ! (input) scalar string expression
        nextpage,                   ! (output) scalar string identifier
        IncludeHiddenPages  ! (optional) scalar numerical expression
        )
```

Arguments

page A string expression containing the name of a (child) page in the Page Manager tree.

nextpage A scalar string identifier to hold the name of the next page of the given page (if it exists).

IncludeHiddenPages A scalar numerical expression to indicate whether hidden pages should be taken into account. If IncludeHiddenPages is set to 1 then the resulting page may be a page that is currently hidden, otherwise these hidden pages are skipped. The default is 0.

Return Value

The procedure returns 1 on success, or 0 if the given page name does not exist or if the page does not have a next page.

See also:

The procedures *PageGetPrevious* (page 703), *PageGetChild* (page 699), *PageGetParent* (page 702), *PageGetNextInTreeWalk* (page 702), *PageGetAll* (page 699).

5.2.9 PageGetNextInTreeWalk

The procedure *PageGetNextInTreeWalk* (page 702) retrieves the name of the next page for a specific page in the Page Manager tree by traversing the tree in a depth-first manner: This procedure will try to find the next page of a page first by searching for child nodes of the selected page. If the page has no child nodes, it will look for a next page on the same level. If there also isn't a next page in the same level, it will try to find a next page for the parent nodes. This procedure includes hidden pages and ignores separators.

```
PageGetNextInTreeWalk(
        page,           ! (input) scalar string expression
        nextpage,        ! (output) scalar string identifier
       IncludeHiddenPages   ! (optional) scalar numerical expression
        )
```

Arguments

page A string expression containing the name of a (child) page in the Page Manager tree.

nextpage A scalar string identifier to hold the name of the next page of the given page (if it exists).

IncludeHiddenPages A scalar numerical expression to indicate whether hidden pages should be taken into account. If IncludeHiddenPages is set to 1 then the resulting parent page may be a page that is currently hidden, otherwise these hidden pages are skipped. The default is 0.

Return Value

The procedure returns 1 on success, or 0 if the given page name does not exist or if the page does not have a next page.

See also:

The procedures *PageGetNext* (page 701), *PageGetPrevious* (page 703), *PageGetChild* (page 699), *PageGetParent* (page 702), *PageGetAll* (page 699).

5.2.10 PageGetParent

The procedure *PageGetParent* (page 702) retrieves the name of the parent page for a specific page in the Page Manager tree.

```
PageGetParent(
        page,                       ! (input) scalar string expression
        parentpage,          ! (output) scalar string identifier
        IncludeHiddenPages   ! (optional) scalar numerical expression
        )
```

Arguments

page A string expression containing the name of a (child) page in the Page Manager tree.

parentpage A scalar string identifier to hold the name of the parent page of the given page (if it exists).

IncludeHiddenPages A scalar numerical expression to indicate whether hidden pages should be taken into account. If IncludeHiddenPages is set to 1 then the resulting parent page may be a page that is currently hidden, otherwise these hidden pages are skipped. The default is 0.

Return Value

The procedure returns 1 on success, or 0 if the given page name does not exist or if the page does not have a parent page.

See also:

The procedures *PageGetChild* (page 699), *PageGetNext* (page 701), *PageGetPrevious* (page 703), *PageGetNextInTreeWalk* (page 702), *PageGetAll* (page 699).

5.2.11 PageGetPrevious

The procedure *PageGetPrevious* (page 703) retrieves the name of the previous page for a specific page in the Page Manager tree. The previous page is the page that has the same parent page, and is positioned directly above the given page.

```
PageGetPrevious(
        page,                       ! (input) scalar string expression
        previouspage,        ! (output) scalar string identifier
        IncludeHiddenPages   ! (optional) scalar numerical expression
        )
```

Arguments

page A string expression containing the name of a (child) page in the Page Manager tree.

previouspage A scalar string identifier to hold the name of the previous page of the given page (if it exists).

IncludeHiddenPages A scalar numerical expression to indicate whether hidden pages should be taken into account. If IncludeHiddenPages is set to 1 then the resulting page may be a page that is currently hidden, otherwise these hidden pages are skipped. The default is 0.

Return Value

The procedure returns 1 on success, or 0 if the given page name does not exist or if the page does not have a previous page.

See also:

The procedures *PageGetNext* (page 701), *PageGetChild* (page 699), *PageGetParent* (page 702), *PageGetNextInTreeWalk* (page 702), *PageGetAll* (page 699).

5.2.12 PageGetTitle

The procedure *PageGetTitle* (page 704) retrieves the title of a specific page in the Page Manager tree.

```
PageGetTitle(
        pageName,       ! (input) scalar string expression
        pageTitle       ! (output) scalar string identifier
        )
```

Arguments

pageName A string expression containing the name of a page in the Page Manager tree.

pageTitle A scalar string identifier to hold the title of the given page.

Return Value

The procedure returns 1 on success, or 0 otherwise.

5.2.13 PageGetUsedIdentifiers

The procedure *PageGetUsedIdentifiers* (page 704) returns a subset of *AllIdentifiers* (page 821) containing all identifiers used on a specified page.

```
PageGetUsedIdentifiers(
        page,           ! (input) scalar string expression
        identifier_set  ! (output) subset of all identifiers
        )
```

Arguments

page A string expression containing the name of a page in the Page Manager tree.

identifier_set A subset of all identifers containing all the identifiers used in the page.

Return Value

The procedure returns 1 on success, or 0 if the given page name does not exist.

See also:

The procedure *IdentifierGetUsedInformation* (page 722).

5.2.14 PageOpen

With the procedure *PageOpen* (page 705) you can open any page that is defined in the Page Manager. If the page is already open, then the procedure will make this page the active page. The *PageOpen* (page 705) procedure does not halt the execution, unless the page to open is defined as a dialog page. In the latter case, the execution is halted until the user closes the page.

```
PageOpen(
        page              ! (input) string expression
        )
```

Arguments

page A string expression representing the name of the page that you want to open. This name is the unique name as it appears in the Page Manager tree.

Return Value

The procedure returns 1 if the page is opened successfully. If the procedure fails to open the page it returns 0, and the pre-defined parameter *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *PageOpenSingle* (page 705), *PageClose* (page 695).

5.2.15 PageOpenSingle

The procedure *PageOpenSingle* (page 705) is similar to PageOpen, except that after successfull opening the page *PageOpenSingle* (page 705) makes sure that all other currently opened pages are closed.

```
PageOpenSingle(
              page         ! (input) string expression
              )
```

Arguments

page A string expression representing the name of the page that you want to open. This name is the unique name as it appears in the Page Manager tree.

Return Value

The procedure returns 1 if the page is opened successfully. If the procedure fails to open the page it returns 0, and the pre-defined parameter *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *PageOpen* (page 705), *PageClose* (page 695).

5.2.16 PageRefreshAll

Normally, the data on all open pages is refreshed automatically each time AIMMS has finished executing a procedure. Via a call to *PageRefreshAll* (page 706) you can refresh the data on all pages at any time during a procedure run (for example to show intermediate results).

```
PageRefreshAll
```

Arguments

None

Note:

- Pages that you open from within a procedure will always show the data that is available at that moment, so it is not necessary to call *PageRefreshAll* (page 706) for a newly opened page.
- At the end of an button action, AIMMS will automatically refresh all pages.

See also:

The procedure *PageOpen* (page 705).

5.2.17 PageSetCursor

With the procedure *PageSetCursor* (page 706) you have maximum control over where you want to set the current keyboard input focus. Similar to `PageSetFocus` you can specify which page object should get the focus, but additionally you can specify the data element that should be highlighted within the focus object.

```
PageSetCursor(
        page             ! (input) scalar string expression
        tag,             ! (input) scalar string expression
        scalar_reference, ! (input) scalar identifier
        )
```

Arguments

page A string expression representing the name of the page in which you want to set the input focus.

tag A string expression representing the tag name of the object that should get the keyboard input focus.

scalar_reference A scalar data element that matches the element that you want to highlight within the object.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0 and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Example

If you are displaying a variable `Transport` in a table with tag "TransportTable" on page "Results", then you can set the focus and cursor to a specific cell in this table using the following procedure call:

```
PageSetCursor("Results", "TransportTable", Transport('Amsterdam','Rotterdam'));
```

Note: You can specify a unique tag name for each page object via the object properties.

See also:

The procedure *PageSetFocus* (page 707).

5.2.18 PageSetFocus

With the procedure *PageSetFocus* (page 707) you can set the keyboard input focus to a specific object within a specific page. If the page is not open, then the procedure will first try to open the page.

```
PageSetFocus(
            page,       ! (input) scalar string expression
            tag         ! (input) scalar string expression
            )
```

Arguments

page A string expression representing the name of the page in which you want to set the input focus.

tag A string expression representing the tag name of the object that should get the keyboard input focus.

Return Value

The procedure returns 1 on success. If it fails to set the focus to the specified object, then the return value is 0 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: You can specify a unique tag name for each page object via the object properties.

See also:

The procedures *PageSetCursor* (page 706), *PageGetFocus* (page 700).

5.2.19 PivotTableDeleteState

With the procedure *PivotTableDeleteState* (page 708) you can delete a specific state in either the Developer or End User state file.

```
PivotTableDeleteState(
            statename,   ! (input) scalar string expression
            statesource ! (input) scalar string expression
        )
```

Arguments

statename A string expression representing the name of the state to be deleted.

statesource A string expression representing the type of state to be deleted. Possible values are:

- DeveloperState: Delete the specified state from the *developer* state file.
- UserState: Delete the specified state from the *user* state file.
- Both: Delete the state from both the *developer* and *user* state file

Return Value

The procedure returns 1 on success. If it fails to delete the specified state, then the return value is 0 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- When running in End User mode, you cannot delete states from the developer state file.

See also:

- The Pivot Table example that comes with the AIMMS installation includes a library that uses this new function. It includes a right-mouse menu that can be assigned to a Pivot Table, after which the user can save, load, or delete states for that Pivot Table. You can include this library in your own project as well.
- The functions *PivotTableReloadState* (page 708), *PivotTableSaveState* (page 709).

5.2.20 PivotTableReloadState

With the procedure *PivotTableReloadState* (page 708) you can reload the state of a specific pivot table from either the developer or user state file.

```
PivotTableReloadState(
            page,        ! (input) scalar string expression
            tag,         ! (input) scalar string expression
            statesource ! (input) scalar string expression
        )
```

Arguments

page A string expression representing the name of the page that contains the pivot table.

tag A string expression representing the tag that identifies the pivot table.

statesource A string expression representing the type of state to be reloaded. Possible values are:

- DeveloperState: Reload the pivot table with a state that is present in the *developer* state file.
- UserState: Reload the pivot table with a state that is present in the *user* state file.
- None: Reload the pivot table as if no state was available.

Return Value

The procedure returns 1 on success. If it fails to reload the state for the specified object, then the return value is 0 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- You can specify a unique tag name for each page object on the **Misc** tab of the object properties dialog box.
- The name of the state is specified by the *Specific State Name* property on the **General** tab of the pivot table properties dialog box.
- This procedure will only reload the state when the **Save Layout/State - By Developer** property (or **Save Layout/State - By End User** when running in end-user mode) on the **general** tab of the pivot table properties dialog box, has been set to a value other than *No*.

See also:

- The Pivot Table example that comes with the AIMMS installation includes a library that uses this new function. It includes a right-mouse menu that can be assigned to a Pivot Table, after which the user can save, load, or delete states for that Pivot Table. You can include this library in your own project as well.
- The functions *PivotTableSaveState* (page 709), *PivotTableDeleteState* (page 708).

5.2.21 PivotTableSaveState

With the procedure *PivotTableSaveState* (page 709) you can save the state of a specific pivot table to either the developer or user state file.

```
PivotTableSaveState(
            page,        ! (input) scalar string expression
            tag,         ! (input) scalar string expression
            statesource ! (input) scalar string expression
        )
```

Arguments

page A string expression representing the name of the page that contains the pivot table.

tag A string expression representing the tag that identifies the pivot table.

statesource A string expression representing the type of state to be saved. Possible values are:

- DeveloperState: Save the specified state to the *developer* state file.
- UserState: Save the specified state to the *user* state file.

Return Value

The procedure returns 1 on success. If it fails to save the state for the specified object, then the return value is 0 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- When running in end-user mode, it is not possible to save a *developer* state.
- You can specify a unique tag name for each page object on the **Misc** tab of the object properties dialog box.
- The name of the state is specified by the *Specific State Name* property on the **General** tab of the pivot table properties dialog box.
- This procedure will only save the state when the **Save Layout/State - By Developer** property (or **Save Layout/State - By End User** when running in end-user mode) on the **general** tab of the pivot table properties dialog box, has been set to a value other than *No*.

See also:

- The Pivot Table example that comes with the AIMMS installation includes a library that uses this new function. It includes a right-mouse menu that can be assigned to a Pivot Table, after which the user can save, load, or delete states for that Pivot Table. You can include this library in your own project as well.
- The functions *PivotTableDeleteState* (page 708), *PivotTableReloadState* (page 708).

5.2.22 PrintEndReport

With the procedure PageEndReport you finish the printing of a report that was started via a call to PrintStartReport.

```
PrintEndReport
```

Arguments

None

Return Value

The procedure returns 1 on success, or 0, if there was no current report.

See also:

The procedures *PrintStartReport* (page 713), *PrintPage* (page 711).

5.2.23 PrintPage

With the procedure *PrintPage* (page 711) you can print a single print page. If the page contains a data object for which the available data does not fit onto a single printed sheet, AIMMS will print as many sheets as needed.

```
PrintPage(
        page,                                   ! (input) scalar string expression
        [filename,]                             ! (optional) scalar string expression
        [from_pagenr,]                          ! (optional) integer
        [to_pagenr,]                            ! (optional) integer
        [UseDefaultBitmapPrintSettings]     ! (optional) integer
        )
```

Arguments

page A string expression representing the name of the page that you want to print. This name is the unique name as it appears in the Page Manager tree.

filename (optional) If this file name is specified, then AIMMS will print to the specific file and not directly to the printer. If this argument is omitted, then AIMMS will print according to the settings of the currently selected printer.

from_pagenr (optional) If the objects on the page result in multiple printed sheets, then with this argument you can specify the first sheet to print. If omitted, then printing will start at the first sheet (from_pagenr = 1).

to_pagenr (optional) If the objects on the page result in multiple printed sheets, then with this argument you can specify the last sheet to print. If omitted, then printing continues until the last sheet.

UseDefaultBitmapPrintSettings (optional) When printing a non-print page, the page is printed by creating an exact bitmap copy of the page as it appears on the screen. By default (if the argument equals 0), a dialog will appear in which you can specify which scale should be applied such that it fits on one or more sheets. By settings this argument to 1, this dialog box will be skipped and the bitmap

print will use the standard settings of the dialog box. If the page to print is designed as a print page, then this argument is ignored.

Return Value

The procedure returns the actual number of pages printed if the print page is printed successfully. If the procedure fails to print the page it returns 0, and the pre-defined parameter *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *PrintPageCount* (page 712), *PrintStartReport* (page 713).

5.2.24 PrintPageCount

The procedure *PrintPageCount* (page 712) will return how many sheets of paper are needed to print a single print page in the interface.

```
PrintPageCount(
        page            ! (input) scalar string expression
        )
```

Arguments

page A string expression representing the name of the page that you want to print. This name is the unique name as it appears in the Page Manager tree.

Return Value

The procedure returns the number of sheets needed, or 0 if the page cannot be printed.

See also:

The procedure *PrintPage* (page 711).

5.2.25 PrinterGetCurrentName

With the procedure *PrinterGetCurrentName* (page 712) you can retrieve the name of the currently selected printer.

```
PrinterGetCurrentName(
        printerName     ! (ouput) scalar string parameter
        )
```

Arguments

printerName On return this string parameter will contain the name of the currently selected printer.

Return Value

The procedure returns 1 if it did retrieve a printer name successfully. If it return 0, something is wrong with the printer setup and `printerName` will be empty.

Example

You can use the procedure *PrinterGetCurrentName* (page 712) to create a PDF preview mode for the pages that you want to print:

```
PrinterGetCurrentName(currentPrinter);
        if FindString(currentPrinter,"PDF") then
                PrintStartReport("Report", "output.pdf");
                PrintPage("MyPrintPage");
                PrintEndReport;
                ! if there is a PDF viewer installed (like AcrobatReader), you␣
↪can now open the document with it:
                OpenDocument("output.pdf");
        endif;
```

Note: To change the current printer, you can use the menu item `File - Print Setup` or make a call to the procedure *PrinterSetupDialog* (page 714).

See also:

The procedures *PrinterSetupDialog* (page 714).

5.2.26 PrintStartReport

With the procedure *PrintStartReport* (page 713) you start printing a report that consists of the printing of multiple pages (using the procedure `PrintPage`). The advantage of printing in the form of a report is that all print request until `PrintEndReport` arrive at the printer as a single print job, and that the pages are numbered correctly.

```
PrintStartReport(
        title,          ! (input) scalar string expression
        [filename]      ! (optional) scalar string expression
        )
```

Arguments

title A string expression representing the title of the report. This title is used in the communication to the printer as the name of the print job.

filename (optional) If this file name is specified, then AIMMS will print to the specific file and not directly to the printer. If this argument is omitted, then AIMMS will print according to the settings of the currently selected printer.

Return Value

The procedure returns 1 on success. If the procedure fails, then the pre-defined parameter *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: A successful call to *PrintStartReport* (page 713) must be followed by a call to `PrintEndReport`, otherwise nothing is printed, and your printer may hang.

See also:

The procedures *PrintEndReport* (page 710), *PrintPage* (page 711).

5.2.27 PrinterSetupDialog

With the procedure *PrinterSetupDialog* (page 714) you can open the standard printer setup dialog. This same dialog is also available via the menu command `File - Print Setup`.

```
PrinterSetupDialog
```

Arguments

None

Return Value

If the setup dialog is cancelled, the procedure *PrinterSetupDialog* (page 714) returns 0. Otherwise it will return 1.

Example

You can use the procedure *PrinterSetupDialog* (page 714) to make sure that a user selects a PDF printer:

```
isPDFPrinter := 0;
Repeat
        PrinterGetCurrentName(currentPrinter);
        if FindString(currentPrinter,"PDF") then
                isPDFPrinter := 1;
                break;
```

(continues on next page)

(continued from previous page)

```
        endif;
        DialogMessage("Please select a PDF printer.");
        break when PrinterSetupDialog() = 0;
EndRepeat;
```

See also:

The procedures *PrinterGetCurrentName* (page 712).

5.2.28 ShowMessageWindow

With the procedure *ShowMessageWindow* (page 715) you programmatically open or close the AIMMS message window.

```
ShowMessageWindow(
     [do_show]        ! (optional) scalar expression
     )
```

Arguments

do_show (optional) A scalar 0-1 expression, indicating whether the message window should be opened (value is 1) or should be closed (value is 0). The default is 1.

See also:

The procedure *ShowProgressWindow* (page 715).

5.2.29 ShowProgressWindow

With the procedure *ShowProgressWindow* (page 715) you programmatically open or close the AIMMS progress window.

```
ShowProgressWindow(
     [do_show]        ! (optional) scalar expression
     )
```

Arguments

do_show (optional) A scalar 0-1 expression, indicating whether the progress window should be opened (value is 1) or should be closed (value is 0). The default is 1.

See also:

The procedure *ShowMessageWindow* (page 715).

5.3 User Colors

5.3.1 UserColorAdd

With the procedure *UserColorAdd* (page 716) you can programmatically add a new color to the set of user colors.

```
UserColorAdd(
     color_name,          ! (input) scalar string expression
     red,                 ! (input) scalar numerical expression
     green,               ! (input) scalar numerical expression
     blue                 ! (input) scalar numerical expression
     )
```

Arguments

color_name A string expression holding the name of the user color to add.

red An integer value in the range $0 \ldots 255$ indicating the red component in the RGB value of the color.

green An integer value in the range $0 \ldots 255$ indicating the green component in the RGB value of the color.

blue An integer value in the range $0 \ldots 255$ indicating the blue component in the RGB value of the color.

Return Value

The procedure returns 1 if the color could be added successfully, or 0 if the color already exists.

Note: Only project colors, i.e. colors added through the **Tools-User Colors** dialog box, are persistent. User colors that are added to a project using the procedure *UserColorAdd* (page 716) do not persist, and, therefore, have to be added during the initialization of every project session.

See also:

UserColorDelete (page 716), *UserColorGetRGB* (page 717), *UserColorModify* (page 717). User colors are discussed in full detail in Setting colors within the model.

5.3.2 UserColorDelete

With the procedure *UserColorDelete* (page 716) you can programmatically delete a color from the set of user colors.

```
UserColorDelete(
     color_name           ! (input) scalar string expression
     )
```

Arguments

color_name A string expression holding the name of the user color to delete.

Return Value

The procedure returns 1 if the color could be deleted successfully, or 0 if the color does not exist, or is contained in the fixed set of project colors.

Note: You can only delete user colors that have been added using the procedure `UserColorAdd`. Colors added through the **Tools-User Colors** dialog box are fixed and cannot be deleted or modified.

See also:

UserColorAdd (page 716), *UserColorGetRGB* (page 717), *UserColorModify* (page 717). User colors are discussed in full detail in Setting colors within the model.

5.3.3 UserColorGetRGB

With the procedure *UserColorGetRGB* (page 717) you can programmatically obtain the RGB values of a color in the set of user colors.

```
UserColorGetRGB(
     color_name,          ! (input) scalar string expression
     red,                 ! (output) scalar numerical parameter
     green,               ! (output) scalar numerical parameter
     blue                 ! (output) scalar numerical parameter
     )
```

Arguments

color_name A string expression holding the name of the user color to query.

red An scalar parameter that, on return, holds the red component in the RGB value of the color.

green An scalar parameter that, on return, holds the green component in the RGB value of the color.

blue An scalar parameter that, on return, holds the blue component in the RGB value of the color.

Return Value

The procedure returns 1 if the color exists in the set of user colors, or 0 if the color does not exist.

See also:

UserColorAdd (page 716), *UserColorDelete* (page 716), *UserColorModify* (page 717). User colors are discussed in full detail in Setting colors within the model.

5.3.4 UserColorModify

With the procedure *UserColorModify* (page 717) you can programmatically modify an existing color in the set of user colors.

```
UserColorModify(
     color_name,           ! (input) scalar string expression
     red,                 ! (input) scalar numerical expression
     green,               ! (input) scalar numerical expression
     blue                 ! (input) scalar numerical expression
     )
```

Arguments

color_name A string expression holding the name of the user color to modify.

red An integer value in the range $0 \ldots 255$ indicating the red component in the RGB value of the color.

green An integer value in the range $0 \ldots 255$ indicating the green component in the RGB value of the color.

blue An integer value in the range $0 \ldots 255$ indicating the blue component in the RGB value of the color.

Return Value

The procedure returns 1 if the color could be modified successfully, and 0 if the color does not exist, or is contained in the fixed set of project colors.

Note: You can only modify user colors that have been added using the procedure `UserColorAdd`. Colors added through the **Tools-User Colors** dialog box are fixed and cannot be deleted or modified.

See also:

UserColorAdd (page 716), *UserColorDelete* (page 716), *UserColorGetRGB* (page 717). User colors are discussed in full detail in Setting colors within the model.

CHAPTER

SIX

DEVELOPMENT SUPPORT

6.1 Profiler and Debugger

6.1.1 DebuggerBreakPoint

The procedure *DebuggerBreakPoint* (page 719) breaks execution and activates the debugger when needed.

```
DebuggerBreakPoint(
     [only_if_active]       ! (optional, default 0) scalar binary expression
     )
```

Arguments

only_if_active When this argument equals 1, execution is only stopped when the debugger is active. If this argument equals 0 the execution is always stopped and the debugger is activated if necessary.

Note:

- The debugger and profiler are exclusive. When the profiler is active, this procedure has no effect.
- This procedure has no effect in end-user mode because the debugger is not available in end-user mode.

6.1.2 ProfilerPause

The procedure *ProfilerPause* (page 719) temporarily disables measuring the execution time of statements and definitions.

```
ProfilerPause
```

Note:

- This procedure is the programmatic counterpart of the **Profiler** - **Pause** menu command.
- This procedure only has effect when the profiler has been activated.

See also:

The procedure *ProfilerContinue* (page 720) and *ProfilerRestart* (page 720).

6.1.3 ProfilerStart

The procedure *ProfilerStart* (page 720) starts measuring the execution time of statements and definitions.

```
ProfilerStart
```

Note: When the option `profiler_store_data` has been set to `On` profiling information is stored in the predefined identifier `ProfilerData`.

See also:

The procedures *ProfilerPause* (page 719), *ProfilerContinue* (page 720) and *ProfilerRestart* (page 720) and the predefined identifier *ProfilerData* (page 789).

6.1.4 ProfilerContinue

The procedure *ProfilerContinue* (page 720) continues measuring the execution time of statements and definitions.

```
ProfilerContinue
```

Note:

- This procedure is the programmatic counterpart of the **Profiler** - **Continue** menu command.
- This procedure only has effect when the profiler has been activated.

See also:

The procedure *ProfilerPause* (page 719) and *ProfilerRestart* (page 720).

6.1.5 ProfilerRestart

The procedure *ProfilerRestart* (page 720) clears the execution time measurement data of all statements and definitions.

```
ProfilerRestart
```

Note:

- This procedure is the programmatic counterpart of the **Profiler** - **Restart** menu command.
- This procedure only has effect when the profiler has been activated.

See also:

The procedure *ProfilerContinue* (page 720) and *ProfilerPause* (page 719).

6.1.6 ProfilerCollectAllData

With the procedure *ProfilerCollectAllData* (page 721) you can retrieve the current results of the profiler into a parameter in your model. This procedure is especially usefull when you want to investigate timings of a model that runs server-side, without the IDE. Data will be retrieved for procedures and functions, and for parameter and sets that have a definition.

```
ProfilerCollectAllData(
     ProfilerData,           ! (output) a 3-dimensional identifier
     GrossTimeThreshold,     ! (optional) scalar numerical parameter
     NetTimeThreshold        ! (optional) scalar numerical parameter
     )
```

Arguments

ProfilerData A three dimensional identifier where the indices represent (1) the identifiers, (2) the line numbers and (3) the specific profiler value. The first index should be an index in (a subset of) the predeclared set *AllIdentifiers* (page 821), only for identifiers in this set the profiling data will be retrieved. The second index should be an index in a subset of `Integers`. The third index should be an index in (a subset of) the predeclared set *AllProfilerTypes* (page 804).

GrossTimeThreshold An optional value, in seconds, which filters out all the profiler measurements where the gross time is smaller.

NetTimeThreshold An optional value, in seconds, which filters out all the profiler measurements where the net time is smaller.

Note: The procedure will only produce results when the profiler is currently active and some execution has already taken place. The subset of integers that is used for the line number will automatically be extended with all the line numbers that have actual measurements. So this set may be left empty when calling the procedure. For a procedure or function the timings of each individual statement is retrieved and stored using the corresponding line number. Besides that, the total timings of the procedure or function is stored as an entry with line number 0.

Example

With these declarations

```
Set Lines {
    Index: line;
    Subset of: Integers;
}
Parameter Results {
    IndexDomain: (IndexIdentifiers,line,IndexProfilerTypes);
}
```

the procedure call

```
ProfilerCollectAllData(Results, GrossTimeThreshold: 0.5);
```

fills the parameter `Results` with all profiler measurements for which the gross time is larger than 0.5 seconds.

See also:

The procedure *ProfilerStart* (page 720).

6.2 Application Information

6.2.1 IdentifierGetUsedInformation

With the procedure *IdentifierGetUsedInformation* (page 722) you can obtain information on whether an identifier in the model is still referenced in either a page, a user menu or a case type/data category.

```
IdentifierGetUsedInformation(
     identifier             ! (input) element parameter
     isUsedInPages,         ! (output) scalar numerical identifier
     isUsedInMenus,         ! (output) scalar numerical identifier
     isUsedInDataCategories ! (output) scalar numerical identifier
     )
```

Arguments

identifier The identifier, given as element in the set *AllIdentifiers* (page 821), whose usage info you want to retrieve. Please note that local identifiers (declared inside procedures or functions) are not taken into account by this function.

isUsedInPages On return this value is set to 1 if the identifier is referenced in either a page, template or print page. It is set to 0 otherwise.

isUsedInMenus On return this value is set to 1 if the identifier is referenced in a menu item or submenu of a user menu. It is set to 0 otherwise.

isUsedInDataCategories On return this value is set to 1 if the identifier is referenced in either a data category or case type. It is set to 0 otherwise.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note: The function only indicates whether the identifier is used in either of the three GUI areas. To figure out in which specific page, menu or data category the identifier is used you can use the drag-and-find feature of the IDE: if you drag an identifier from the Model Explorer, holding down both the Control and Shift key, and drop it on either the Page Manager, Template Manager, Menu Builder or Data Management Setup tree, all items that reference the identifier will be highlighted.

See also:

The procedure *PageGetUsedIdentifiers* (page 704).

6.2.2 IdentifierMemory

With the function *IdentifierMemory* (page 723) you can determine the total amount of memory occupied by the identifier.

```
IdentifierMemory(
     Identifier,          ! (input) scalar element parameter
     IncludePermutations ! (optional, default 1) scalar binary expression
     )
```

Arguments

Identifier An element expression in the set *AllIdentifiers* (page 821) specifying the identifier for which the amount of occupied memory should be determined.

IncludePermutations An 0-1 value indicating whether the amount of memory occupied by permutations of the identifier should also be included in the total memory determination.

Return Value

The function reports the sum of the memory occupied by the identifier, its suffixes and the associated hidden identifiers (that are introduced as temporary identifiers by the AIMMS compiler/execution engine. The unit of measurement for this function is bytes.

Note: The return value of this function differs from the value reported in the 'Memory Usage' column of the **Identifier Cardinalities** dialog box because in the **Identifier Cardinalities** dialog box the value for hidden identifiers and suffixes are reported separately.

6.2.3 IdentifierMemoryStatistics

With the procedure *IdentifierMemoryStatistics* (page 723) you can obtain a report containing the statistics collected by AIMMS' memory manager for a single or multiple high dimensional identifiers.

```
IdentifierMemoryStatistics(
     IdentSet,            ! (input) a set of identifiers
     OutputFileName,      ! (input) scalar string expression
     AppendMode,          ! (optional, default 0) scalar numerical expression
     MarkerText           ! (optional) scalar string expression
     ShowLeaksOnly        ! (optional) scalar expression
     ShowTotals           ! (optional) scalar expression
     ShowSinceLastDump    ! (optional) scalar expression
     ShowMemPeak          ! (optional) scalar expression
     ShowSmallBlockUsage ! (optional) scalar expression
```

(continues on next page)

(continued from previous page)

```
    doAggregate         ! (optional, default 0) scalar expression
    )
```

Arguments

IdentSet A subset of *AllIdentifiers* (page 821) whose memory statistics are to be reported.

OutputFileName A string expression holding the name of the file to which the statistics must be written.

AppendMode An 0-1 value indicating whether the file must be overwritten or whether the statistics must be appended to an existing file.

MarkerText A string printed at the top of the memory statistics report.

ShowLeaksOnly A 0-1 value that is only used internally by AIMMS. The value specified doesn't influence the memory statistics report.

ShowTotals A 0-1 value indicating whether the report should include detailed information about the total memory use in AIMMS' own memory management system until the moment of calling *IdentifierMemoryStatistics* (page 723).

ShowSinceLastDump A 0-1 value indicating whether the report should include basic and detailed information about the memory use in AIMMS' own memory management system since the previous call to *IdentifierMemoryStatistics* (page 723).

ShowMemPeak A 0-1 value indicating whether the report should include detailed information about the memory use in AIMMS' own memory management system, when the memory consumption was at its peak level prior to calling *IdentifierMemoryStatistics* (page 723).

ShowSmallBlockUsage A 0-1 value indicating whether the detailed information about the `MemoryStatistics` memory use in AIMMS' own memory management system is included at all in the memory statistics report. Setting this value to 0 results in a report with only the most basic statistical information about the memory use.

doAggregate A 0-1 value (default 0) indicating whether a single aggregated report is to be presented or multiple individual reports.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- The procedure prints a report of the statistics collected by AIMMS' memory manager since the last call to *IdentifierMemoryStatistics* (page 723).
- AIMMS will only collect memory statistics if the option `memory_statistics` is `on`.

6.2.4 MemoryInUse

With the function *MemoryInUse* (page 724) you can obtain the current amount of memory in use as it is reported by the operating system.

```
MemoryInUse()
```

Return Value

This function returns the amount of memory in use in [Mb].

Note: See also the functions *MemoryStatistics* (page 725), *IdentifierMemory* (page 723), *GMP::Instance::GetMemoryUsed* (page 396)

6.2.5 MemoryStatistics

With the procedure *MemoryStatistics* (page 725) you can obtain a report containing the statistics collected by AIMMS' memory manager.

```
MemoryStatistics(
   OutputFileName,       ! (input) scalar string expression
   AppendMode,           ! (optional, default 0) scalar numerical expression
   MarkerText,           ! (optional, default empty) scalar string expression
   ShowLeaksOnly,        ! (optional, default 0) scalar numerical expression
   ShowTotals,           ! (optional, default 1) scalar numerical expression
   ShowSinceLastDump,    ! (optional, default 1) scalar numerical expression
   ShowMemPeak,          ! (optional, default 0) scalar numerical expression
   ShowSmallBlockUsage,  ! (optional, default 0) scalar numerical expression
   GlobalOnly            ! (optional, default 0) scalar numerical expression
 )
```

Arguments

OutputFileName A string expression holding the name of the file to which the statistics must be written color to modify.

AppendMode An 0-1 value indicating whether the file must be overwritten or whether the statistics must be appended to an existing file.

MarkerText A string printed at the top of the memory statistics report.

ShowLeaksOnly A 0-1 value that is only used internally by AIMMS. The value specified doesn't influence the memory statistics report.

ShowTotals A 0-1 value indicating whether the report should include detailed information about the total memory use in AIMMS' own memory management system until the moment of calling MemoryStatistics.

ShowSinceLastDump A 0-1 value indicating whether the report should include basic and detailed information about the memory use in AIMMS' own memory management system since the previous call to MemoryStatistics.

ShowMemPeak 0-1 value indicating whether the report should include detailed information about the memory use in AIMMS' own memory management system, when the memory consumption was at its peak level prior to calling MemoryStatistics.

ShowSmallBlockUsage 0-1 value indicating whether the detailed information about the memory use in AIMMS' own memory management system is included at all in the memory statistics report. Setting this value to 0 results in a report with only the most basic statistical information about the memory use.

GlobalOnly A 0-1 value indicating whether only memory used by the global memory manager (i.e. the 'main' memory manager of AIMMS, as opposed to seperate memory manager for individual higher-dimensional identifiers) is reported in the memory statistics file.

Return Value

The procedure prints a report of the statistics collected by AIMMS' memory manager since the last call to *MemoryStatistics* (page 725).

Note: AIMMS will only collect memory statistics if the option `memory_statistics` is `on`.

6.2.6 ShowHelpTopic

With the procedure *ShowHelpTopic* (page 726) you can jump to a specific help topic in a help file.

```
ShowHelpTopic(
     topic,           ! (input) scalar string
     [helpfile]       ! (optional) scalar string
     )
```

Arguments

topic A string representing the help topic to jump to.

helpfile (optional) A string representing the help file to open. If not specified, then AIMMS will use the help file that is specified in the project options.

Note: AIMMS supports the following help file formats: WinHelp or WinHelp2000 (`*.hlp`), compiled HTML Help (`*.chm`), and Acrobat Reader (`*.pdf`).

CHAPTER

SEVEN

SYSTEM INTERACTION

7.1 Error Handling Functions

AIMMS supports the following functions for error handling:

7.1.1 errh::Adapt

The procedure *errh::Adapt* (page 727) adapts an error with the specified information.

```
errh::Adapt(
   err,       ! (input) an element
   severity,  ! (optional input) an element
   message,   ! (optional input) a string
   category,  ! (optional input) an element
   code       ! (optional input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

severity An element in the set *errh::AllErrorSeverities* (page 861).

message A string describing the problem and possibly suggestions for repairing the problem.

category An element in the set *errh::AllErrorCategories* (page 860), indicating the problem category to which the error belongs.

code An element with root set *errh::ErrorCodes* (page 859). The element will be added to the set *errh::ErrorCodes* (page 859) if needed.

Return Value

Returns 1 if adapting the error is successful, 0 otherwise. In the latter case additional error(s) have been raised.

Note: When `err` does not reference an error in the set *errh::PendingErrors* (page 860) an additional error will be raised. If the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
1 block
2     pr_divideByZero();
3 onerror _ep_err do
4     errh::Adapt(
5         err      :  _ep_err,
6         message  :  "Don't worry, be happy" );
7 endblock ;
```

Lines 4-6 adapt the message of the error thrown by `pr_divideByZero` to `"Don't worry, be happy"`

References

- *errh::Severity* (page 740)
- *errh::Message* (page 737)
- *errh::Category* (page 729)
- *errh::Code* (page 730)

7.1.2 errh::Attribute

The function *errh::Attribute* (page 728) returns the attribute of the identifier or node in which the error occurred.

```
errh::Attribute(
        err,   ! (input) an element
        loc    ! (optional input) an integer, default 1.
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

loc An integer in the range `{ 1 .. errh::NumberOfLocations(err) }`.

Return Value

Returns an element in *AllAttributeNames* (page 793) if the information is available and the empty element otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _ep_attr := errh::Attribute(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_ep_attr = 'body'
```

See also:

The functions *errh::Node* (page 739), *errh::Line* (page 735) and *errh::NumberOfLocations* (page 739).

7.1.3 errh::Category

The function *errh::Category* (page 729) returns the error category to which the error belongs.

```
errh::Category(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns an element in *errh::AllErrorCategories* (page 860) if the information is available and the empty element otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
    block
            pr_divideByZero();
    onerror _ep_err do
            _ep_cat := errh::Category(_ep_err);
            errh::MarkAsHandled( _ep_err, 1 );
    endblock ;
```

Afterwards:

```
_ep_cat = 'Math'
```

See also:

The function *errh::Code* (page 730), *errh::InsideCategory* (page 733).

7.1.4 errh::Code

The function *errh::Code* (page 730) returns the identification code of the format string.

```
errh::Code(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns an element in *errh::ErrorCodes* (page 859) if the information is available and the empty element otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _ep_code := errh::Code(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_ep_code = 'EXE_PARALLEL_ENGINE'
```

See also:

The function *errh::Category* (page 729) and the procedure *errh::Adapt* (page 727). The predeclared identifier *errh::PendingErrors* (page 860).

7.1.5 errh::Column

The function *errh::Column* (page 731) returns the column number within the line in the file in which the error occured during reading from file.

```
errh::Column(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns a column number if the information is available and 0 otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
1 block
2     pr_divideByZero();
3 onerror _ep_err do
4     _p_col := errh::Column(_ep_err);
5     errh::MarkAsHandled( _ep_err, 1 );
6 endblock ;
```

Afterwards:

```
_p_col = 11
```

See also:

The functions *errh::Line* (page 735) and *errh::Filename* (page 732).

7.1.6 errh::CreationTime

The function *errh::CreationTime* (page 731) returns the creation time of the error.

```
errh::CreationTime(
        err,   ! (input) an element
        fmt    ! (optional) a format string.
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

fmt A string that holds the date and time format used in the returned string. Valid format strings are described in Format of Time Slots and Periods When this argument is not given, or if `fmt` is not a valid string format, the full reference date format "`%c%y-%m-%d %H:%M:%S`" will be used.

Return Value

Returns the creation time of the error as a string.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _sp_crea := errh::CreationTime(_ep_err, "%c%y-%m-%d %H:%M:%S");
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

See also:

The function *CurrentToString* (page 97).

7.1.7 errh::Filename

The function *errh::Filename* (page 732) returns the file in which the error occurred during reading from file

```
errh::Filename(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns a string containing the filename in which the error occurred, if that error occurred during reading from file and the empty string otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_readError();
onerror _ep_err do
    _sp_file := errh::Filename(_ep_err );
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_sp_file = "./verifiedData/readInError.txt"
```

When that was the file being read.

See also:

The functions *errh::Line* (page 735) and *errh::Column* (page 731).

7.1.8 errh::InsideCategory

The function *errh::InsideCategory* (page 733) returns 1 if the error is inside the given category.

```
errh::InsideCategory(
        err,   ! (input) an element
        cat    ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

cat An element in the set *errh::AllErrorCategories* (page 860) referencing an error.

Return Value

Returns 1 if `err` in inside the category `cat` and 0 otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _bp_insideCat := errh::InsideCategory(_ep_err, 'Solver' );
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_bp_insideCat = 0
```

Because the error is unrelated to a solver.

See also:

The functions *errh::Code* (page 730) and *errh::Category* (page 729).

7.1.9 errh::IsMarkedAsHandled

The function *errh::IsMarkedAsHandled* (page 734) returns 1 if the error is marked as handled and 0 otherwise.

```
errh::IsMarkedAsHandled(
        err   ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns 1 if the error is marked as handled and 0 otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _bp_isMarkedAsHandled := errh::IsMarkedAsHandled(_ep_err );
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_bp_isMarkedAsHandled = 0
```

Because the error is marked as handled just after.

See also:

The function *errh::MarkAsHandled* (page 736).

7.1.10 errh::Line

The function *errh::Line* (page 735) returns the line number in the file or attribute in which the error occured.

```
errh::Line(
        err,   ! (input) an element
        loc    ! (optional input) an integer, default 1.
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

loc An integer in the range `{ 1 .. errh::NumberOfLocations(err) }`.

Return Value

Returns a line number if the information is available and 0 otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _p_line := errh::Line(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_p_line = 2
```

See also:

The function *errh::Column* (page 731), *errh::Filename* (page 732), *errh::Attribute* (page 728), *errh::Node* (page 739) and *errh::NumberOfLocations* (page 739).

7.1.11 errh::MarkAsHandled

The procedure *errh::MarkAsHandled* (page 736) marks or unmarks an error as handled.

```
errh::MarkAsHandled(
        err,     ! (input) an element
        actually ! (optional input), default 1.
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

actually When 1, the error `err` is marked as handled, when 0, the mark is cleared.

Return Value

Returns a line number if the information is available and 0 otherwise.

Note: When `err` doesn't reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Which amounts to ignoring the division by zero error.

See also:

The function *errh::IsMarkedAsHandled* (page 734).

7.1.12 errh::Message

The function *errh::Message* (page 737) returns a description of the error.

```
errh::Message(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns a string if the information is available and the empty string otherwise.

Note: When `err` does not reference an element in CrossRef2or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _sp_msg := errh::Message(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_sp_msg =  "Division by zero error with 1 / 0."
```

See also:

The procedure *errh::Adapt* (page 727).

7.1.13 errh::Multiplicity

The function *errh::Multiplicity* (page 738) returns the number of occurrences of this error.

```
errh::Multiplicity(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns the number of occurrences of this error.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _p_mult := errh::Multiplicity(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_p_mult = 1
```

See also:

The functions *errh::Code* (page 730), *errh::Category* (page 729), *errh::Message* (page 737) and *errh::Severity* (page 740).

7.1.14 errh::Node

The function *errh::Node* (page 739) returns the identifier or node in which the error occurred.

```
errh::Node(
        err,   ! (input) an element
        loc    ! (optional input) an integer, default 1.
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

loc An integer in the range `{ 1 .. errh::NumberOfLocations(err) }`.

Return Value

Returns an element in *AllSymbols* (page 787) if the information is available and the empty element otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _ep_node := errh::Node(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_ep_node = 'chapterSystem::sectionError::pr_divideByZero'
```

See also:

The functions *errh::Attribute* (page 728), *errh::Line* (page 735) and *errh::NumberOfLocations* (page 739).

7.1.15 errh::NumberOfLocations

The function *errh::NumberOfLocations* (page 739) returns the number of locations stored to which this error is relevant. The relevant locations are file (if any) that is being read, and the procedures currently active.

```
errh::NumberOfLocations(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns the number locations for which this error is relevant.

Note: When `err` doesn't reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _p_noLoc := errh::NumberOfLocations(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_p_noLoc >= 1
```

Depending on the number of procedures on the running stack.

See also:

The functions *errh::Node* (page 739), *errh::Attribute* (page 728) and *errh::Line* (page 735).

7.1.16 errh::Severity

The function *errh::Severity* (page 740) returns the severity of the error.

```
errh::Severity(
        err  ! (input) an element
)
```

Arguments

err An element in the set *errh::PendingErrors* (page 860) referencing an error.

Return Value

Returns an element in *errh::AllErrorSeverities* (page 861) if the information is available and the empty element otherwise.

Note: When `err` does not reference an element in *errh::PendingErrors* (page 860) or when the current filter is the filter `To Global Collector` an additional error will be raised.

Example

```
block
    pr_divideByZero();
onerror _ep_err do
    _ep_sev := errh::severity(_ep_err);
    errh::MarkAsHandled( _ep_err, 1 );
endblock ;
```

Afterwards:

```
_ep_sev = 'error'
```

See also:

The procedures *errh::Adapt* (page 727) and *errh::MarkAsHandled* (page 736).

See also:

- Our online training about error handling following this link.
- More documentation about using those functions in Raising and Handling Warnings and Errors

7.2 Option Manipulation

7.2.1 OptionGetDefaultString

With the procedure *OptionGetDefaultString* (page 741) you can obtain the string representation of the current value of an AIMMS option, as displayed in the AIMMS **Options** dialog box.

```
OptionGetDefaultString(
     OptionName,          ! (input) scalar string expression
     DefaultString        ! (output) scalar string parameter
     )
```

Arguments

OptionName A string expression holding the name of the option.

DefaultString A scalar string parameter that, on return, contains the string representation of the default value of the option.

Return Value

The procedure returns 1 if the option exists, or 0 if the name refers to a non-existent option.

See also:

OptionGetValue (page 743), *OptionGetKeywords* (page 742), *OptionGetString* (page 742).

7.2.2 OptionGetKeywords

With the procedure *OptionGetKeywords* (page 742) you can obtain set of string keywords, as displayed in the AIMMS **Options** dialog box, that correspond to the numerical (integer) values of an option.

```
OptionGetKeywords(
    OptionName,          ! (input) scalar string expression
    Keywords             ! (output) a 1-dimensional string parameter
    )
```

Arguments

OptionName A string expression holding the name of the option.

Keywords A 1-dimensional string parameter that, on return, contains the keywords corresponding to the set of possible (integer) option values.

Return Value

The procedure returns 1 if the option exists, and 0 if the *OptionName* refers to a non-existent option or if the domain set of the 1-dimensional string parameter is too small.

Note: The domain set of the 1-dimensional parameter passed as the *Keywords* argument must have sufficient elements to hold the string keywords of the (integer) option values from the lower bound up to and including the upper bound.

See also:

OptionGetValue (page 743), *OptionGetString* (page 742), *OptionSetString* (page 744).

7.2.3 OptionGetString

With the procedure *OptionGetString* (page 742) you can obtain the string representation of the current value of an AIMMS option, as displayed in the AIMMS **Options** dialog box.

```
OptionGetString(
     OptionName,          ! (input) scalar string expression
     CurrentString        ! (output) scalar string parameter
     )
```

Arguments

OptionName A string expression holding the name of the option.

CurrentString A scalar string parameter that, on return, contains the string representation of the current value of the option.

Return Value

The procedure returns 1 if the option exists, or 0 if the name refers to a non-existent option.

Note:

- Options for which strings are displayed in the AIMMS **Options** dialog box, are represented by numerical (integer) values internally. To obtain the numerical option value, or to obtain the mapping between numerical option values and the corresponding string keywords, you can use the procedures *OptionGetValue* (page 743) and *OptionGetKeywords* (page 742).
- The procedure *OptionGetString* (page 742) can also be used to set a solver specific option by prefixing the option name by the name of the solver followed by a double colon `::`, e.g., 'CPLEX 22.1::LP method'.

See also:

OptionGetValue (page 743), *OptionGetKeywords* (page 742), *OptionSetString* (page 744).

7.2.4 OptionGetValue

With the procedure *OptionGetValue* (page 743) you can obtain the current value of an AIMMS option, as well as its lower and upper bound and default value.

```
OptionGetValue(
     OptionName,          ! (input) scalar string expression
     Lower,               ! (output) scalar numerical parameter
     Current,             ! (output) scalar numerical parameter
     Default,             ! (output) scalar numerical parameter
     Upper                ! (output) scalar numerical parameter
     )
```

Arguments

OptionName A string expression holding the name of the option.

Lower A scalar parameter that, on return, contains the lower bound of the possible option values.

current A scalar parameter that, on return, contains the current (numerical) value of the option.

Default A scalar parameter that, on return, contains the default (numerical) value of the option.

Upper A scalar parameter that, on return, contains the upper bound of the possible option values.

Return Value

The procedure returns 1 if the option exists, or 0 if the name refers to a non-existent option or to an option that does not take a number as value.

Note:

- Options for which strings are displayed in the AIMMS **Options** dialog box, are also represented by numerical (integer) values. To obtain the corresponding option keywords, you can use the procedures *OptionGetString* (page 742) and *OptionGetKeywords* (page 742).
- The procedure *OptionGetValue* (page 743) can also be used to set a solver specific option by prefixing the option name by the name of the solver followed by a double colon `::`, e.g., 'CPLEX 22.1::Integrality'.
- You can modify option values programmatically using the `OPTION` statement (see also The OPTION and PROPERTY Statements of the Language Reference), or using the procedures *OptionSetValue* (page 745) and *OptionSetString* (page 744).

See also:

OptionGetString (page 742), *OptionGetKeywords* (page 742), *OptionSetValue* (page 745), *OptionSetString* (page 744).

7.2.5 OptionSetString

With the procedure *OptionSetString* (page 744) you can set the value of a string-valued AIMMS option. You must use the values as displayed in the AIMMS **Options** dialog box.

```
OptionSetString(
     OptionName,      ! (input) scalar string expressionN
     NewString        ! (input) scalar string expression
     )
```

Arguments

OptionName A string expression holding the name of the option.

NewString A scalar string expression representing the string representation of the value to be assigned to the option.

Return Value

The procedure returns 1 if the value can be assigned to the option, or 0 if the name refers to a non-existent option, or the value to a non-existent option value.

Note:

- Options for which strings are displayed in the AIMMS **Options** dialog box, are represented by numerical (integer) values internally. To obtain the numerical option value, or to obtain the mapping between numerical option values and the corresponding string keywords, you can use the procedures *OptionGetValue* (page 743) and *OptionGetKeywords* (page 742).
- The procedure *OptionSetString* (page 744) can also be used to set a solver specific option by prefixing the option name by the name of the solver followed by a double colon `::`, e.g., 'CPLEX 22.1::LP method'.

See also:

OptionSetValue (page 745), *OptionGetValue* (page 743), *OptionGetKeywords* (page 742).

7.2.6 OptionSetValue

With the procedure *OptionSetValue* (page 745) you can set the value of a numeric AIMMS option. The value assigned to the option must be contained in the option range displayed in the AIMMS **Options** dialog box.

```
OptionSetValue(
     OptionName,          ! (input) scalar string expression
     NewValue             ! (input) scalar numeric expression
     )
```

Arguments

OptionName A string expression holding the name of the option.

NewValue A scalar numeric expression representing the new value to be assigned to the option.

Return Value

The procedure returns 1 if the option exists and the value can be assigned to the option, or 0 otherwise.

Note:

- Options for which strings are displayed in the AIMMS **Options** dialog box, are also represented by numerical (integer) values. To obtain the corresponding option keywords, you can use the procedures *OptionGetString* (page 742) and *OptionGetKeywords* (page 742).
- The procedure *OptionSetValue* (page 745) can also be used to set a solver specific option by prefixing the option name by the name of the solver followed by a double colon `::`, e.g., 'CPLEX 22.1::Integrality'.
- You can also modify option values using the `OPTION` statement (see also The OPTION and PROPERTY Statements of the Language Reference).

See also:

OptionGetString (page 742), *OptionGetKeywords* (page 742), *OptionSetString* (page 744).

7.3 Licensing Functions

AIMMS supports the following licensing functions:

7.3.1 LicenseExpirationDate

The procedure *LicenseExpirationDate* (page 746) returns the expiration date of the current AIMMS license.

```
LicenseExpirationDate(
                      date    ! (output) a scalar string parameter
                      )
```

Arguments

date A scalar string parameter that, on return, contains the expiration date of the current AIMMS license.

Return Value

The procedure returns 1 on success, and 0 on failure.

Note: The date returned by the procedure has the standard date format `"YYYY-MM-DD"`, or holds the text `"No expiration date"` if the current AIMMS license has no expiration date.

See also:

The procedures *LicenseStartDate* (page 747), *LicenseMaintenanceExpirationDate* (page 746).

7.3.2 LicenseMaintenanceExpirationDate

The procedure *LicenseMaintenanceExpirationDate* (page 746) returns the maintenance expiration date of the current AIMMS license.

```
LicenseMaintenanceExpirationDate(
                                 date    ! (output) a scalar string parameter
                                 )
```

Arguments

date A scalar string parameter that, on return, contains the maintenance expiration date of the current AIMMS license.

Return Value

The procedure returns 1 on success, and 0 on failure.

Note: The date returned by the procedure has the standard date format `"YYYY-MM-DD"`, or holds the text `"No maintenance expiration date"` if the current AIMMS license has no maintenance expiration date.

See also:

The procedures *LicenseStartDate* (page 747), *LicenseExpirationDate* (page 746).

7.3.3 LicenseNumber

The procedure *LicenseNumber* (page 747) returns the license number of the current AIMMS license.

```
LicenseNumber(
             license    ! (output) a scalar string parameter
             )
```

Arguments

license A scalar string parameter that, on return, contains the current license number.

Return Value

The procedure returns 1 on success, and 0 on failure.

Note: The procedure will return the license number as a string of the form `015.090.010.007` if you are using an AIMMS 3 license, or as a string of the form `1234.56` if you are using an AIMMS 2 license.

See also:

The procedure *LicenseType* (page 748).

7.3.4 LicenseStartDate

The procedure *LicenseStartDate* (page 747) returns the start date of the current AIMMS license.

```
LicenseStartDate(
                date    ! (output) a scalar string parameter
                )
```

Arguments

date A scalar string parameter that, on return, contains the start date of the current AIMMS license.

Return Value

The procedure returns 1 on success, and 0 on failure.

Note: The date returned by the procedure has the standard date format "YYYY-MM-DD", or holds the text "No start date" if the current AIMMS license has no start date.

See also:

The procedures *LicenseExpirationDate* (page 746), *LicenseMaintenanceExpirationDate* (page 746).

7.3.5 LicenseType

The procedure *LicenseType* (page 748) returns the type and size of the current AIMMS license.

```
LicenseType(
           type,    ! (output) a scalar string parameter
           size     ! (output) a scalar string parameter
           )
```

Arguments

type A scalar string parameter that, on return, contains the type of the current license.

size A scalar string parameter that, on return, contains the size of the current license.

Return Value

The procedure returns 1 on success, and 0 on failure.

Note: Upon success, the *type* argument contains the license type description (e.g. `"Economy"`) and the *size* argument contains a description of the license size (e.g. `"Large"`).

See also:

The procedure *LicenseNumber* (page 747).

7.3.6 ProjectDeveloperMode

The function *ProjectDeveloperMode* (page 749) indicates whether a project is opened in developer or end-user mode.

```
ProjectDeveloperMode
```

Arguments

None

Return Value

The function returns 1 if the project is opened in developer mode, or 0 if the project is opened in end-user mode.

7.3.7 SecurityGetGroups

With the procedure *SecurityGetGroups* (page 749) you can fill a set with group names from the user database that is linked to the project.

```
SecurityGetGroups(
     group_set          ! (output) an (empty) root set
     )
```

Arguments

group_set A root set, that on return will contain elements that represent all group names from the user database.

Return Value

The procedure returns 1 on success, and 0 on failure.

See also:

The procedure *SecurityGetUsers* (page 750).

7.3.8 SecurityGetUsers

With the procedure *SecurityGetUsers* (page 750) you can fill a set with user names from the user database that is linked to the project. You can filter which users are included in the set based upon their group or authorization level.

```
SecurityGetUsers(
    user_set,        ! (output) an (empty) root set
    [group,]         ! (optional) scalar string
    [level]          ! (optional) element of the set AllAuthorizationLevels
    )
```

Arguments

user_set A root set, that on return will contain elements that represent the user names from the user database.

group (optional) A string representing a group name from the user database. If specified, then only the users that belong to this group are returned.

level (optional) An element of the set *AllAuthorizationLevels* (page 779). If specified, then only the users that have the specified authorization level are returned.

Return Value

The procedure returns 1 on success, and 0 on failure.

See also:

The procedure *SecurityGetGroups* (page 749).

7.3.9 SolverGetControl

A single use local license allows you to run two concurrent AIMMS sessions. At any time, however, only one of these sessions can make use of a solver. Prior to executing a SOLVE statement, AIMMS will determine whether the solver is already locked by another session. If this is the case, AIMMS will abort the SOLVE statement with a runtime error. If the solver is not locked, AIMMS locks the solver for the duration of SOLVE statement by default. With the procedure *SolverGetControl* (page 750) you can programmatically lock the solver for a prolonged period of time, for instance, during an algorithm requiring multiple solves.

```
SolverGetControl
```

Arguments

None

Return Value

The procedure returns 1 if the solver was successfully locked, or 0 otherwise.

Note:

- AIMMS also supports multi-session local licenses that allow you to run multiple concurrent solves, and twice that number of concurrent AIMMS sessions.
- This procedure has no effect if you are connecting to an AIMMS network license server. In that case every session requires a separate floating network license.

See also:

The procedure *SolverReleaseControl* (page 751).

7.3.10 SolverReleaseControl

A single use local license allows you to run two concurrent AIMMS sessions. At any time, however, only one of these sessions can make use of a solver. Prior to executing a SOLVE statement, AIMMS will determine whether the solver is already locked by another session. If this is the case, AIMMS will abort the SOLVE statement with a runtime error. If the solver is not locked, AIMMS locks the solver for the duration of SOLVE statement by default. With the procedure *SolverReleaseControl* (page 751) you can unlock a solver previously locked by a call to the procedure SolverGetControl.

```
SolverReleaseControl
```

Arguments

None

Return Value

The procedure returns 1 if successful, or 0 if the solver was not currently locked by this session.

Note:

- AIMMS also supports multi-session local licenses that allow you to run multiple concurrent solves, and twice that number of concurrent AIMMS sessions.
- This procedure has no effect if you are connecting to an AIMMS network license server. In that case every session requires a separate floating network license.

See also:

The procedure *SolverGetControl* (page 750).

7.4 Environment Functions

AIMMS supports the following system setting functions, which give access to, or allow modification of, various system settings:

7.4.1 AimmsRevisionString

The procedure *AimmsRevisionString* (page 752) returns the revision number of the current AIMMS executable.

```
AimmsRevisionString(
                    Version        ! (output) a scalar string parameter
                    NumberOfFields ! (optional) a scalar numerical expression)
```

Arguments

Version A scalar string parameter that, on return, contains the current revision number.

NumberOfFields A scalar integer expression indicating the number of fields displayed in the revision string.

Return Value

The procedure returns 1 on success, and 0 on failure.

Note: The revision string returned by the procedure has the format "`x.y.b.r`" where *x* represents the major AIMMS version number (e.g. 3), *y* represents the minor AIMMS version number (e.g. 0), where *b* represents the build number (e.g. 476) of the current executable, and where *r* represents the internal revision number.

7.4.2 EnvironmentGetString

With the procedure *EnvironmentGetString* (page 752) you can obtain the string representation of an environment setting, either set by the process calling AIMMS or by AIMMS itself.

```
EnvironmentGetString(
     Key,          ! (input) scalar string expression
     Value         ! (output) scalar string parameter
     )
```

Arguments

Key A string expression holding the name of the environment variable.

Value A scalar string parameter that, on return, contains the string representation of the current value of the environment variable.

Return Value

The procedure returns 1 if the variable `Key` is available, and 0 otherwise.

Note:

- The environment variables defined by AIMMS itself are: `AIMMSROOT`, `AIMMSBIN`, `AIMMSSOLVERS`, `AIMMSCFG`, `AIMMSHELP`, `AIMMSDOC`, `AIMMSUSERDLL`, `AIMMSLOG`, `AIMMSPROJECT`, `AIMMSMODULES`, and `AIMMSTUTORIAL`.
- Examples of environment variables available on a Windows system are `COMPUTERNAME`, `OS`, `PATH`, `TEMP`, `TMP`, and `USERNAME`. Entering the MSDOS command `set` on an MSDOS prompt will present you with the set of available environment variables on a Windows system. Via the control panel tool `system` and then going to `Advanced system settings` - `Advanced` tab - `Environment variables` button, you can manipulate the set of environment variables.
- On Linux systems a distinction is made between the variables kept to a process itself, and those exported to the environment of all its child processes. In a bash shell you can obtain the collection of variables set via the bash `set` command, and the subset of all exported environment variables via the bash `env` command. In order to make a variable available to the environment, you will have to explicitly place it in the environment, via an `export` command. In several system wide bash scripts, `/etc/bashrc`, or user startup bash scripts, `~/.bashrc`, export commands such as:

```
export HOSTNAME
export OSTYPE
```

  can be found in order to make these useful environment variables available to all processes executed.

See also:

EnvironmentSetString (page 753).

7.4.3 EnvironmentSetString

With the function *EnvironmentSetString* (page 753) you can set environment variables.

```
EnvironmentSetString(
    Key,          ! (input) scalar string expression
    Value         ! (input) scalar string parameter
    )
```

Arguments

Key A string expression holding the name of the environment variable.

Value A scalar string parameter that contains the string representation of the value of you want to assign to the environment variable.

Return Value

The function returns 1 upon success, or 0 otherwise.

Note:

- With *EnvironmentSetString* (page 753) you can change the value for existing environment variables as well as create new environment variables.
- Note that the function *EnvironmentSetString* (page 753) will only change the values of variables in the environment associated with the AIMMS process.

See also:

EnvironmentGetString (page 752).

7.4.4 GeoFindCoordinates

The procedure *GeoFindCoordinates* (page 754) can be used to find the latitude/longitude coordinates for a given address. The procedure uses the free OpenStreetMap (OSM) geocoding service. You are advised to carefully read the OSM geocoder usage policy before using this procedure in your application.

```
GeoFindCoordinates(
     address,              ! (input) scalar string expression
     latitude,             ! (output) scalar numerical parameter
     longitude,            ! (output) scalar numerical parameter
     email,                ! (optional) scalar string parameter
     url                   ! (optional) scalar string parameter
     )
```

Arguments

address A string representing the address for which the latitude and longitude coordinates have to be found.

latitude A scalar numerical parameter that will contain the latitude coordinate of the specified address upon success.

longitude A scalar numerical parameter that will contain the longitude coordinate of the specified address upon success.

email An optional string representing the email address that the OSM organization will use to contact you in the event of problems (as mentioned in their usage policy).

url An optional string representing the url of an alternative (e.g. your own) OSM geocoder server. If not specified, the public OSM geocoder server is being used.

Return Value

The procedure returns 1 on success, and 0 if the specified address could not be found. On failure, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Example

The following calls to the procedure *GeoFindCoordinates* (page 754) return valid latitude and longitude coordinates

```
GeoFindCoordinates("Netherlands", Latitude, Longitude, "me@mycompany.com");
GeoFindCoordinates("Haarlem, Netherlands", Latitude, Longitude);
GeoFindCoordinates("2034 Haarlem, Netherlands", Latitude, Longitude);
GeoFindCoordinates("Schipholweg, Haarlem, Netherlands", Latitude, Longitude);

GeoFindCoordinates("US", Latitude, Longitude);
GeoFindCoordinates("Kirkland, WA, US", Latitude, Longitude);
GeoFindCoordinates("Lake Washington Boulevard NE, Kirkland, US", Latitude,␣
↪Longitude);
GeoFindCoordinates("5400 Carillon Point, Kirkland, US", Latitude, Longitude);

GeoFindCoordinates("Singapore", Latitude, Longitude);
GeoFindCoordinates("Chulia Street, Singapore", Latitude, Longitude);

GeoFindCoordinates("Shanghai, China", Latitude, Longitude);
GeoFindCoordinates("Middle Huaihai Road, Shanghai, China", Latitude,␣
↪Longitude);
```

assumed that *Latitude* and *Longitude* are declared as numerical parameters in your model.

Note:

- With the introduction of AIMMS 3.9.5 and AIMMS 3.10 PR, this procedure has been disabled because Microsoft discontinued support to the Virtual Earth geocoder service that was used to locate the address. In AIMMS 3.11 FR2, the *GeoFindCoordinates* (page 754) procedure was enabled again by using the OSM geocoding service instead.
- *'One of the hard things about geocoding is parsing addresses into something intelligible'* (see the OpenStreetMap wiki for details on address formats). As a result, you may need to slightly play around with the address format in order for the geocoder to correctly parse your address.
- To discourage *'bulk geocoding'* (see the OSM usage policy for more details), AIMMS inserts a small delay in case the time between two consecutive geocoding requests is smaller than a second.

7.4.5 TestInternetConnection

With the procedure *TestInternetConnection* (page 755) you can verify whether an internet connection to a given URL is possible.

```
TestInternetConnection(
     url                  ! (input) scalar string expression
     )
```

Arguments

url A string representing the address of the internet site AIMMS will try to reach.

Return Value

The procedure returns 1 on success, and 0 if AIMMS could not establish a connection to the specified address (by pinging). On failure, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: This procedure will only check whether the host as specified in the url can be reached, not whether a certain service is running nor whether a certain internet page exists.

7.5 Invoking Actions

7.5.1 Delay

With the procedure *Delay* (page 756) you can block the execution of your model for the indicated delay time. You can use this procedure, for instance, when you want to display intermediate results on a page using the procedure PageRefreshAll.

```
Delay(
     delaytime          ! (input) scalar expression
     )
```

Arguments

delaytime The number of seconds that the execution should be blocked.

See also:

The procedure *PageRefreshAll* (page 706).

7.5.2 Execute

With the *Execute* (page 756) procedure you can start another application.

```
Execute(
     executable,          ! (input) scalar string expression
     [commandline,]       ! (optional) scalar string expression
     [workdir,]           ! (optional) scalar string expression
     [wait,]              ! (optional) 0 or 1
     [minimized]          ! (optional) 0 or 1
     )
```

Arguments

executable A string representing the name of the program that you want to execute.

commandline (optional) A string representing the arguments that you want to pass to the program.

workdir (optional) A string representing the directory where the program should start in. If omitted, then the current project directory is used. Please note that this argument does not specify the folder where the executable is located. Rather, it specifies the folder that the executable should use as its working folder.

wait (optional) This argument indicates whether or not AIMMS will wait for the program to finish. The default value is 0 (not wait).

minimized (optional) This argument indicates whether or not the program should run in a minimized state. The default is 0 (not minimized).

Note:

- When running on Linux and the program is located in the AIMMS project folder, the program name must be prefixed with `./`.
- On Linux, ensure the program has the permission to execute. To ensure `myprog` is executable, whereby `myprog` is in the current directory:

```
1 Execute(
2     Executable  :  "chmod",
3     CommandLine :  " a+rx ./myprog ");
```

- On Linux, avoid using redirects in the command line (using chacters >, <, and |.
- As a general rule, you should not wait for interactive windowed applications. Waiting for the termination of a program is necessary when the program does some form of external data processing which is required for the execution of your model.

See also:

The procedure *OpenDocument* (page 758).

7.5.3 ExitAimms

With the procedure *ExitAimms* (page 757) you can exit the current AIMMS session from within a procedure.

```
ExitAimms(
         [interactive]       ! (optional) 0 or 1
         )
```

Arguments

interactive (optional) This optional argument is still present for compatibility, but does no longer have any effect. You should use MainTermination to specify whether or not AIMMS should display a confirmation dialog box before closing the current project.

Note:

The procedure does not immediately exit AIMMS, but it will try to exit as soon as the execution of the current procedure has finished. If existing, the logoff procedure and the procedure `MainTermination` will be executed as normal.

Please note that calling the pre-definded function `ExitAimms()` from within WebUI (for example, as part of an action behind a button widget) is currently not supported and will result in an error. In fact, calling `ExitAimms()` only works for the main AIMMS thread itself and not for any of the other AIMMS contexts (of which WebUI is just one example). Exiting only from the underlying AIMMS session itself is not deemed as a proper behavior for an application with Web-based User Interface.

7.5.4 OpenDocument

The procedure *OpenDocument* (page 758) uses the current association of Windows to open documents, run programs, etc. Its procedureality is similar to that of the **Run** command in the **Start Menu** of Windows. You can use it, for instance, to display an HTML file using the default web browser, open a Word document, or initiate an e-mail session.

```
OpenDocument(
        document          ! (input) string expression
        )
```

Arguments

document A string expression representing the document or program you want to open.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Example

```
OpenDocument( "http://www.aimms.com" );
OpenDocument( "mailto:info@aimms.com" );
OpenDocument( "anyfile.doc" );
OpenDocument( "c:\\windows" );
```

See also:

The procedure *Execute* (page 756).

7.5.5 ScheduleAt

With the procedure *ScheduleAt* (page 759) you schedule a specific procedure to be run at a specified moment in time.

```
ScheduleAt(
     starttime,          ! (input) scalar string expression
     procedure           ! (input) element of the set AllProcedures
     )
```

Arguments

starttime A string representing the time at which you want to start the execution of the specified procedure. This time must be respresent using AIMMS' standard time format: "YYYY-MM-DD hh:mm:ss".

procedure An element in the set *AllProcedures* (page 825). This procedure cannot have any arguments.

Return Value

The procedure returns 1 on success, and 0 if AIMMS could not schedule the procedure at the specified start time. On failure, the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: If at the specified start time AIMMS is busy running some other task, then the procedure will start as soon as AIMMS has finished this task. If you want to run a procedure at regular intervals, then you can re-schedule the procedure from within the scheduled procedure itself.

7.5.6 SessionArgument

With the procedure *SessionArgument* (page 759) you can retrieve the string value of any user defined command line argument, that was specified during startup of AIMMS.

```
SessionArgument(
     argno,              ! (input) integer number
     argument            ! (output) string valued parameter
     )
```

Arguments

argno An integer greater or equal to 1, representing the argument that you want retrieve. If the argument does not exist, then the procedure returns 0.

argument A string valued parameter, to hold the string of the requested command line argument.

Return Value

The procedure returns 1 on success, and 0 if the request argument number does not exist.

Note: When you open an AIMMS project from the command line, AIMMS allows you to add an arbitrary number of additional arguments directly after the project name. The procedure *SessionArgument* (page 759) gives you access to these arguments. You can use these arguments, for instance, to specify a varying data source name from which you want to read data into your model, or run your project in different modes.

7.6 File and Directory Functions

AIMMS supports the following functions for accessing disk files and directories:

7.6.1 DirectoryCopy

The procedure *DirectoryCopy* (page 760) copies one or more directories to a new or other directory.

```
DirectoryCopy(
        source,         ! (input) scalar string expression
        destination,    ! (input) scalar string expression
        [confirm]       ! (optional) 0 or 1
        )
```

Arguments

source A scalar string expression representing the directories(s) you want to copy. The string may contain wild-card characters such as '*' and '?', allowing you to copy a whole group of directories at once.

destination A scalar string expression representing the destination directory.

confirm (optional) An integer value that indicates whether you want to let the user confirm any copy operation that would overwrite existing files. If this argument is omitted, then the default behavior is that files are overwritten without any notice.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: If the destination name does not exist, AIMMS will create a directory with the specified name with the same contents as the source directory. If the destination directory does already exists as a directory, AIMMS will copy the contents of the source directory into a directory with the same name as the source directory contained in the destination directory.

See also:

The procedures *DirectoryMove* (page 766), *FileCopy* (page 769), *DirectoryExists* (page 762).

7.6.2 DirectoryCreate

The procedure *DirectoryCreate* (page 761) creates a new directory on your disk.

```
DirectoryCreate(
        directoryname      ! (input) scalar string expression
        )
```

Arguments

directoryname A scalar string expression representing the new directory name. If the name does not contain a full path, then the it is assumed to be relative to the current project directory.

Return Value

The procedure returns 1 if the directory is created successfully. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: If the new directory path contains references to non-existing directories, then the procedure tries to create each of these directories.

See also:

The procedures *DirectoryExists* (page 762), *DirectoryDelete* (page 761).

7.6.3 DirectoryDelete

The procedure *DirectoryDelete* (page 761) deletes a directory from your disk. If this directory contains files, then these files are deleted as well.

```
DirectoryDelete(
        directory,                ! (input) scalar string expression
        [delete_readonly_files]   ! (optional, default 0) scalar expression
        )
```

Arguments

directory A scalar string expression representing the directory you want to delete.

delete_readonly_files A scalar expression indicating whether read-only files must be deleted without further notice (value $\neq$ 0), or whether the procedure should fail on read-only files (value 0).

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *DirectoryExists* (page 762), *FileDelete* (page 770).

7.6.4 DirectoryExists

With the procedure *DirectoryExists* (page 762) you can check whether a specific directory name currently exists.

```
DirectoryExists(
        directoryname      ! (input) scalar string expression
        )
```

Arguments

directoryname A scalar string expression representing a valid directory name. The file name may contain a partial path relative to the project directory, or a full path.

Return Value

The procedure returns 1 if the given directory name exists, or 0 otherwise.

Note: Note that if you want use some static directory name in your model, then you have to specify two slashes behind each directory, as in `"c:\\windows\\temp"`.

See also:

The procedure *DirectoryDelete* (page 761).

7.6.5 DirectoryGetCurrent

The procedure *DirectoryGetCurrent* (page 763) retrieves the full path of the current project directory.

```
DirectoryGetCurrent(
        directoryname      ! (output) scalar string parameter
        )
```

Arguments

directory A scalar string parameter, that on return will contain the path of the current project directory. The string is always terminated by a directory slash \.

Return Value

The procedure returns 1.

See also:

The procedure *DirectorySelect* (page 767).

7.6.6 DirectoryGetFiles

The procedure *DirectoryGetFiles* (page 763) creates a list of filenames present in a directory.

```
DirectoryGetFiles(
        directory,      ! (input) scalar string expression
        filter,         ! (input) scalar string expression
        filenames,      ! (output) a one-dimensional string parameter
        recursive,      ! (optional) default 0
        attributeFilter ! (optional) default: empty set
        )
```

Arguments

directory A scalar string expression representing the directory you want to search. The empty string is interpreted as the current directory.

filter The pattern file names should match. The empty string is interpreted as all files.

filenames A one-dimensional string parameter indexed over a subset of the predeclared set *Integers* (page 812). This parameter will be filled with the names of the files matching the pattern as specified in the first argument.

recursive An optional scalar expression. When zero the procedure *DirectoryGetFiles* (page 763) doesn't work recursively; it scans only the directory specified, not its subdirectories. When non-zero, these subdirectories will also be searched.

attributeFilter files that have one of the specified attributes will not be included in the result. This argument is a subset of *AllFileAttributes* (page 799).

Return Value

The procedure returns the number of files found on success, which may be 0. If it fails, then it returns -1, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Example

Using the declarations

```
Set FileNumbers {
    SubsetOf     :  Integers;
    Index        :  fn;
}
```

```
StringParameter FileNames {
    IndexDomain  :  (fn);
}
```

the statements

```
DirectoryGetFiles("log", "*.err", Filenames);
display Filenames ;
```

will result in

```
FileNames := data { 1 : "aimms.err" } ;
```

to be printed in the listing file.

Note:

- The `directory` argument can specify either a relative or an absolute folder path.
- Devices, hidden files, system files, hidden subdirectories and system subdirectories are not searched. On Linux systems, files and subdirectories that start with a '.' are considered hidden files and are not returned in the result.

See also:

- The procedure *DirectoryGetSubdirectories* (page 765) to find the names of the subdirectories in a particular directory.
- The procedures *DirectoryGetCurrent* (page 763) and *DirectorySelect* (page 767) to obtain the current directory and to select a particular directory.

7.6.7 DirectoryGetSubdirectories

The procedure *DirectoryGetSubdirectories* (page 765) creates a list of subdirectory names present in a directory.

```
DirectoryGetSubdirectories(
        directory,          ! (input) scalar string expression
        filter,             ! (input) scalar string expression
        subdirectorynames,  ! (output) a one-dimensional string parameter
        recursive,          ! (optional) default 0
        attributeFilter     ! (optional) default: empty set
        )
```

Arguments

directory A scalar string expression representing the directory you want to search. The empty string is interpreted as the current directory.

filter The pattern file names should match. The empty string is interpreted as all files.

subdirectorynames A one-dimensional string parameter indexed over a subset of the predeclared set *Integers* (page 812). This parameter will be filled with the names of the folders matching the pattern as specified in the first argument.

recursive An optional scalar expression. When zero the procedure *DirectoryGetSubdirectories* (page 765) doesn't work recursively; it scans only the directory specified, not its subdirectories. When non-zero, these subdirectories will also be searched.

attributeFilter files that have one of the specified attributes will not be included in the result. This argument is a subset of *AllFileAttributes* (page 799).

Return Value

The procedure returns the number of subdirectories found on success, which may be 0. If it fails, then it returns -1, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Example

Using the declarations

```
Set FolderNumbers {
    SubsetOf    :  Integers;
    Index       :  fn;
}
```

```
StringParameter FolderNames {
   IndexDomain  :  (fn);
}
```

the statements

```
DirectoryGetSubdirectories("", "*.*", FolderNames,
    recursive: 1, attributeFilter: { 'Executable'} );
display FolderNames ;
```

will result in

```
FolderNames := data { 1 : "backup",  2 : "log" } ;
```

to be printed in the listing file.

Note:

- The `directory` argument can specify either a relative or an absolute folder path.
- Hidden and system files and subdirectories are not searched, nor are devices. On Linux systems, files and subdirectories that start with a '.' are considered hidden files and are not searched. The names "." and ".." are never included in the result.

See also:

- The procedure *DirectoryGetFiles* (page 763) to find the names of the files in a particular directory.
- The procedures *DirectoryGetCurrent* (page 763) and *DirectorySelect* (page 767) to obtain the current directory and to select a particular directory.

7.6.8 DirectoryMove

The procedure *DirectoryMove* (page 766) moves one or more directories to either a new name (a rename) or to another directory.

```
DirectoryMove(
        source,         ! (input) scalar string expression
        destination,    ! (input) scalar string expression
        confirm         ! (optional) 0 or 1
        )
```

Arguments

source A scalar string expression representing the file(s) you want to move. The string may contain wildcard characters such as '*' and '?', allowing you to move a whole group of directories at once.

destination A scalar string expression representing the destination directory.

confirm (optional) An integer value that indicates whether or not you want to let the user confirm any move operation that would overwrite existing files. If this argument is omitted, then the default behavior is that files are overwritten without any notice.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note: If the destination name does not exist, AIMMS will move the source directory to the specified position. If the destination directory does already exists as a directory, AIMMS will move the source directory into the (existing) destination directory, retaining the original name of the source directory.

See also:

The procedures *DirectoryCopy* (page 760), *FileMove* (page 772), *DirectoryExists* (page 762).

7.6.9 DirectoryOfLibraryProject

Via the procedure *DirectoryOfLibraryProject* (page 767) the name of the folder (directory) where a library is located can be obtained. This is the folder where the corresponding *Project.xml* file exists.

```
DirectoryOfLibraryProject(
    libraryname,  ! (input) a scalar string
    directoryname ! (output) a scalar string
)
```

Arguments

libraryname The name of the library or the name of the library prefix.

directoryname The full path to the folder where the *Project.xml* file of the library is located.

Return value

Returns 1 on success, 0 if the library cannot be found in the current AIMMS application.

7.6.10 DirectorySelect

With the procedure *DirectorySelect* (page 767) you can let the user select an existing directory using Windows' standard directory selection dialog box.

```
DirectorySelect(
        directoryname,    ! (input/output) scalar string parameter
        [directory,]      ! (optional input) scalar string expression
        [title]           ! (optional input) scalar string expression
        )
```

Arguments

directoryname A scalar string parameter. On return this parameter will represent the selected directory name. If the selected directory is a sub directory below the current project directory, then the directory name will be presented using a relative path. In other cases the directory name is presented using a full path specification. In both cases, the returned directory string is terminated by a \ character.

directory (optional) A scalar string representing an existing directory. The dialog box will initially select this directory. If omitted, then the current project directory will be used.

title (optional) A scalar string that is used as the title of the selection dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user did select a directory. If some error occurs or if the user presses the **Cancel** button, then the procedure returns 0.

Note: If *DirectorySelect* (page 767) returns 0, then the first argument may not contain a valid directory path. So you must always check the return value, and, if it is 0, either abort the current procedure or continue with some default directory name.

See also:

The procedures *FileSelect* (page 774), *DirectoryGetCurrent* (page 763).

7.6.11 FileAppend

The procedure *FileAppend* (page 768) appends the contents of one file to the end of another file. Both files must be text files.

```
FileAppend(
        filename,       ! (input) scalar string expression
        appendname      ! (input) scalar string expression
        )
```

Arguments

filename A scalar string expression representing the file name to which you want to append the contents of the second file.

appendname A scalar string expression representing the file name that you want to append.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- If the first file (the file to which you append) does not exist, then this file will be created. The contents of the appended file will always start on a new line in the resulting file.
- When appending files with different character encodings, the result is unpredictable.

See also:

The procedures *FileCopy* (page 769), *FileExists* (page 771).

7.6.12 FileCopy

The procedure *FileCopy* (page 769) copies one or more files to a new name or to another directory.

```
FileCopy(
        source,      ! (input) scalar string expression
        destination, ! (input) scalar string expression
        [confirm]    ! (optional) 0 or 1
        )
```

Arguments

source A scalar string expression representing the file(s) you want to copy. The string may contain wildcard characters such as '*' and '?', allowing you to copy a whole group of files at once.

destination A scalar string expression representing the destination file name or destination directory.

confirm (optional) An integer value that indicates whether or not you want to let the user confirm any copy operation that would overwrite existing files. If this argument is omitted, then the default behavior is that files are overwritten without any notice.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *FileMove* (page 772), *DirectoryCopy* (page 760), *FileExists* (page 771).

7.6.13 FileDelete

The procedure *FileDelete* (page 770) deletes one or more files from your disk.

```
FileDelete(
        filename,                ! (input) scalar string expression
        [delete_readonly_files]  ! (optional, default 0) scalar expression
        )
```

Arguments

filename A scalar string expression representing the file(s) you want to delete. The string may contain wild-card characters such as '*' and '?', allowing you to delete a whole group of files at once.

delete_readonly_files A scalar expression indicating whether read-only files must be deleted without further notice (value $\neq$ 0), or whether the procedure should fail on a read-only file (value 0).

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *FileExists* (page 771), *DirectoryDelete* (page 761).

7.6.14 FileEdit

The procedure *FileEdit* (page 770) opens a specific file in the internal AIMMS text file editor. Optionally, you can set the cursor on a specific piece of text within the file.

```
FileEdit(
        filename,     ! (input) scalar string expression
        find,         ! (optional) scalar string expression
        encoding      ! (optional) scalar element expression
        )
```

Arguments

filename A scalar string expression representing the file name that you want to edit.

find (optional) A scalar string expression that is used to position the cursor over a specific piece of text in the file. If this argument is omitted (or if the specified text cannot be found), then the cursor will be positioned at the top of the file.

encoding (optional) A scalar element expression that results in an element of *AllCharacterEncodings* (page 782). If this argument is not specified, the value of the option `default_input_character_encoding` is used.

Return Value

The procedure returns 1 on success, and 0 if it could not open the file in the editor.

Note: If you want to use another external text editor to edit a specific file, then you can use the procedure *Execute* (page 756).

See also:

The procedures *FileView* (page 777), *Execute* (page 756).

7.6.15 FileExists

With the procedure *FileExists* (page 771) you can check whether a specific file name currently exists.

```
FileExists(
        filename      ! (input) scalar string expression
        )
```

Arguments

filename A scalar string expression representing a valid file name. The file name may contain a partial path relative to the project directory, or a full path.

Return Value

The procedure returns 1 if the given file name exists, and 0 otherwise.

Note: Note that if you want use some static file name in your model, then you have to specify two slashes behind each directory, as in `"c:\\windows\\temp\\filename.dat"`

See also:

The procedure *FileDelete* (page 770)

7.6.16 FileGetSize

The procedure *FileGetSize* (page 771) retrieves the size on disk of an existing file.

```
FileGetSize(
        filename,     ! (input) scalar string expression
        fileSize      ! (output) scalar numerical identifier
        )
```

Arguments

filename A scalar string expression representing an existing file name.

fileSize A scalar identifier to hold the size of the file, or -1 if the size could not be retrieved.

Return Value

The procedure returns 1 on success. If it failed to retrieve the file size, then it returns 0 and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedure *FileExists* (page 771).

7.6.17 FileMove

The procedure *FileMove* (page 772) moves one or more files to either a new name (a rename) or to another directory.

```
FileMove(
        source,       ! (input) scalar string expression
        destination,  ! (input) scalar string expression
        [confirm]     ! (optional) 0 or 1
        )
```

Arguments

source A scalar string expression representing the file(s) you want to move. The string may contain wildcard characters such as '*' and '?', allowing you to move a whole group of files at once.

destination A scalar string expression representing the destination file name or destination directory.

confirm (optional) An integer value that indicates whether or not you want to let the user confirm any move operation that would overwrite existing files. If this argument is omitted, then the default behavior is that files are overwritten without any notice.

Return Value

The procedure returns 1 on success. If it fails, then it returns 0, and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedures *FileCopy* (page 769), *DirectoryMove* (page 766), *FileExists* (page 771).

7.6.18 FilePrint

The procedure *FilePrint* (page 773) prints a specific text file using the currently selected printer.

```
FilePrint(
        filename,      ! (input) scalar string expression
        encoding       ! (optional) scalar element expression
        )
```

Arguments

filename A scalar string expression representing the text file that you want to print.

encoding (optional) A scalar element expression that results in an element of *AllCharacterEncodings* (page 782). If this argument is not specified, the value of the option `default_input_character_encoding` is used.

Return Value

The procedure returns 1 on success, and 0 if it could not print the file.

Note: The file is printed using the paper and font settings that are specified in the **Text Printing** dialog box, which is accessible from the `Settings` menu.

See also:

The procedure *FileEdit* (page 770).

7.6.19 FileRead

With the procedure *FileRead* (page 773) you can read the contents of a file into a string parameter.

```
FileRead(
        filename,      ! (input) scalar string expression
        encoding       ! (optional) scalar element expression
        )
```

Arguments

filename A scalar string expression representing a valid file name. The file name may contain a partial path relative to the project directory, or a full path.

encoding (optional) A scalar element expression that results in an element of *AllCharacterEncodings* (page 782). If this argument is not specified, the value of the option `default_input_character_encoding` is used.

Return Value

The procedure returns a string containing the contents of the file.

Note:

- This procedure will not automatically reread a file when its contents has changed. It is therefore better to use it in a procedure than in a parameter definition.
- In case the file does not exist, no error message will be returned and the result will be the empty string. In case there is any doubt the file exists it is advised to first check using the procedure `FileExists`.

See also:

The procedure *FileExists* (page 771).

7.6.20 FileSelect

With the procedure *FileSelect* (page 774) you can let the user select an existing file name using Windows' standard file selection dialog box. Usually you use this procedure to select some input file (i.e. a file for reading), because other than `FileSelectNew`, this procedure only allows the user to select existing files.

```
FileSelect(
        filename,         ! (input/output) scalar string identifier
        [directory,]      ! (optional) scalar string expression
        [extension,]      ! (optional) scalar string expression
        [title]           ! (optional) scalar string expression
        )
```

Arguments

filename A scalar string identifier holding the file name that the user selected. If on entry this strings represents a valid file name, then this file name is used to initialize the dialog box.

directory (optional) A scalar string representing an existing directory. The dialog box will initially only show the files that are located in this directory. If this argument is omitted, then the current project directory will be used.

extension (optional) A scalar string representing a file extension. The dialog box will initially only show those files that match this extension. If this argument is omitted, then all files are shown.

title (optional) A scalar string that is used as the title of the selection dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user actually has selected a file. If some error occurs or if the user presses the **Cancel** button, the procedure returns 0.

Note: If *FileSelect* (page 774) returns 0, then the first argument may not contain a valid file name. So you must always check the return value, and, if it is 0, either abort the current procedure or continue with some default file name.

See also:

The procedure *FileSelectNew* (page 775).

7.6.21 FileSelectNew

With the procedure *FileSelectNew* (page 775) the user can select a new (or existing) file using Windows' file selection dialog box. Usually it is used to select an output file (i.e. for writing), because other than `FileSelect`, this procedure allows you to specify new file names. If an existing file name is selected, a warning will be displayed. The procedure does not create any files on disk or make any changes to existing files. It only returns the file name selected by the user.

```
FileSelectNew(
        filename,         ! (input/output) scalar string identifier
        [directory,]      ! (optional) scalar string expression
        [extension,]      ! (optional) scalar string expression
        [title]           ! (optional) scalar string expression
        )
```

Arguments

filename A scalar string identifier holding the file name that the user specified. If on entry this strings represents a valid file name, then this file name is used to initialize the dialog box.

directory (optional) A scalar string representing an existing directory. The dialog box will initially only show the files that are located in this directory. If this argument is omitted, then the current project directory will be used.

extension (optional) A scalar string representing a file extension. The dialog box will initially only show those files that match this extension. If this argument is omitted, then all files are shown.

title (optional) A scalar string that is used as the title of the selection dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user actually has selected a file. If some error occurs or if the user presses the **Cancel** button, the procedure returns 0.

Note: If *FileSelectNew* (page 775) returns 0, then the first argument may not contain a valid file name. So you must always check the return value, and, if it is 0, either abort the current procedure or continue with some default file name.

See also:

The procedure *FileSelect* (page 774).

7.6.22 FileTime

The procedure *FileTime* (page 776) retrieves the last modification time of an existing file.

```
FileTime(
        filename,     ! (input) scalar string expression
        file_time     ! (output) scalar string identifier
        )
```

Arguments

filename A scalar string expression representing an existing file name.

file_time A scalar string identifier to hold the file modification time of the specified file. This time is represented using AIMMS' standard date and time format: "YYYY-MM-DD hh:mm:ss"

Return Value

The procedure returns 1 on success. If it failed to retrieve the file time, then it returns 0 and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

See also:

The procedure *FileExists* (page 771).

7.6.23 FileTouch

The procedure *FileTouch* (page 776) changes the modification time of a file.

```
FileTouch(
        filename,     ! (input) scalar string expression
        [newtime]     ! (optional) scalar string expression
        )
```

Arguments

filename A scalar string expression representing an existing file name.

newtime This time is represented using AIMMS' standard date and time format: "YYYY-MM-DD hh:mm:ss". If omitted the modification time of the file is set to the current time.

Return Value

The procedure returns 1 on success. If it failed to set the file time, then it returns 0 and the pre-defined identifier *CurrentErrorMessage* (page 835) will contain a proper error message.

7.6.24 FileView

The procedure *FileView* (page 777) opens a specific file in the internal AIMMS text file viewer. Optionally, you can highlight a specific piece of text within the file.

```
FileView(
        filename,      ! (input) scalar string expression
        find,          ! (optional) scalar string expression
        encoding       ! (optional) scalar element expression
        )
```

Arguments

filename A scalar string expression representing the file name that you want to edit.

find (optional) A scalar string expression that is used to position the cursor over a specific piece of text in the file. If this argument is omitted (or if the specified text cannot be found), then the cursor will be positioned at the top of the file.

encoding (optional) A scalar element expression that results in an element of *AllCharacterEncodings* (page 782). If this argument is not specified, the value of the option `default_input_character_encoding` is used.

Return Value

The procedure returns 1 on success, and 0 if it could not open the file in the viewer.

Note: If you want to use another external text editor to view a specific file, then you can use the procedure *Execute* (page 756).

See also:

The procedures *FileEdit* (page 770), *Execute* (page 756).

CHAPTER

EIGHT

PREDEFINED IDENTIFIERS

8.1 System Settings Related Identifiers

The following collection of predefined identifiers contains system-related information. The contents of these identifiers typically corresponds to data entered in system dialog boxes, such as the **Solver Configuration** dialog box, the **User Colors** dialog box and the **User** and **Authorization Level Setup** dialog boxes.

8.1.1 AllAuthorizationLevels

The predefined set *AllAuthorizationLevels* (page 779) contains the names of all authorization levels associated with an AIMMS project.

```
Set AllAuthorizationLevels {
    Index       :  IndexAuthorizationLevels;
}
```

Definition

The contents of the set *AllAuthorizationLevels* (page 779) is the collection of all authorization levels defined for a particular project through the **Authorization Level Setup** dialog box.

Updatability

The contents of the set can only be modified through the **Authorization Level Setup** dialog box.

Note: The set *AllAuthorizationLevels* (page 779) is typically used in the index domains of parameters used in the model and graphical end-user interface to define accessibility rights for groups of users with the same authorization level. By referring to the data slice determined by the value of element parameter *CurrentAuthorizationLevel* (page 788), AIMMS will use the accessibility rights associated with the authorization level of the current user. The use of authorization levels in AIMMS directly is deprecated, as user authentication and authorization during deployment is now arranged via AIMMS PRO (cf. Section Project Security).

8.1.2 AllAvailableCharacterEncodings

The predefined set *AllAvailableCharacterEncodings* (page 779) contains the names of all character encodings available during the current AIMMS session.

```
Set AllAvailableCharacterEncodings {
    SubsetOf   :  AllCharacterEncodings;
    Index      :  IndexAvailableCharacterEncodings;
}
```

Definition

The contents of the set *AllAvailableCharacterEncodings* (page 779) is the collection of all character encodings available during the current AIMMS session.

Updatability

The contents of the set can not be modified and is determined at AIMMS startup.

See also:

- Paragraph Text files in the preliminaries of the language reference 18.
- The encoding attribute of files, see 496.
- The set of all character encodings known to AIMMS: *AllCharacterEncodings* (page 782).

8.1.3 ASCIICharacterEncodings

The predefined set *ASCIICharacterEncodings* (page 780) contains the names of ASCII character encodings. Here an ASCII character encoding is an encoding whereby code point 33 thru 126 are the same as the US-ASCII encoding.

```
Set ASCIICharacterEncodings {
    SubsetOf   :  AllCharacterEncodings;
    Index      :  IndexAvailableCharacterEncodings;
}
```

Definition

The contents of the set *ASCIICharacterEncodings* (page 780) is the collection of ASCII character encodings.

Updatability

The contents of the set can not be modified and is determined at AIMMS startup.

See also:

- Paragraph Text files in the preliminaries of the language reference 18.
- The encoding attribute of files, see 496.
- The set of all character encodings known to AIMMS: *AllCharacterEncodings* (page 782).

8.1.4 ASCIIUnicodeCharacterEncodings

The predefined set *ASCIIUnicodeCharacterEncodings* (page 781) is the union of ASCIICharacterEncodings and UnicodeCharacterEncodings.

```
Set ASCIIUnicodeCharacterEncodings {
    SubsetOf   :  AllCharacterEncodings;
    Index      :  IndexAvailableCharacterEncodings;
}
```

Definition

The contents of the set *ASCIIUnicodeCharacterEncodings* (page 781) is the union of ASCIICharacterEncodings and UnicodeCharacterEncodings.

Updatability

The contents of the set can not be modified and is determined at AIMMS startup.

See also:

- Paragraph Text files in the preliminaries of the language reference 18.
- The encoding attribute of files, see 496.
- The set of all character encodings known to AIMMS: *AllCharacterEncodings* (page 782).

8.1.5 UnicodeCharacterEncodings

The predefined set *UnicodeCharacterEncodings* (page 781) contains the names of Unicode character encodings.

```
Set UnicodeCharacterEncodings {
    SubsetOf   :  AllCharacterEncodings;
    Index      :  IndexAvailableCharacterEncodings;
}
```

Definition

The contents of the set *UnicodeCharacterEncodings* (page 781) is the collection of Unicode character encodings.

Updatability

The contents of the set can not be modified and is determined at AIMMS startup.

See also:

- Paragraph Text files in the preliminaries of the language reference 18.
- The encoding attribute of files, see 496.
- The set of all character encodings known to AIMMS: *AllCharacterEncodings* (page 782).

8.1.6 AllCharacterEncodings

The predefined set *AllCharacterEncodings* (page 782) contains the names of all character encodings known to AIMMS.

```
Set AllCharacterEncodings {
    Index       :  IndexCharacterEncodings;
}
```

Definition

The contents of the set *AllCharacterEncodings* (page 782) is the collection of all character encodings known to AIMMS.

Updatability

The contents of the set can not be modified; it has the following fixed contents:

```
AllCharacterEncodings := data
{ ASMO-708      , ! Arabic

  BIG5          , ! Chinese used in Taiwan, Hong Kong, and
                  ! Macau for Traditional Chinese characters.

  CP737         , ! Greek
  CP875         , ! EBCDIC Greek Modern
  CP932         , ! Windows SHift JIS, Japan
  CP949         , ! Windows Korean

  EUC-CN        , ! Extended Unix Code, simplified Chinese
  EUC-JP        , ! Extended Unix Code, Japanese

  GB2312        , ! Chinese national standard
  GB18030       , ! Chinese national standard
```

(continues on next page)

(continued from previous page)

```
IBM037       , ! EBCDIC with Latin-1
IBM273       , ! EBCDIC German
IBM277       , ! EBCDIC Danish
IBM278       , ! EBCDIC Finnish
IBM280       , ! EBCDIC Italian
IBM284       , ! EBCDIC Spanish
IBM285       , ! EBCDIC British
IBM290       , ! EBCDIC Japanese
IBM297       , ! EBCDIC French
IBM420       , ! EBCDIC Arabic
IBM423       , ! EBCDIC Greek
IBM424       , ! EBCDIC Hebrew
IBM437       , ! EBCDIC Latin-1 (PC)
IBM500       , ! EBCDIC Latin-1 International
IBM775       , ! EBCDIC Polish
IBM850       , ! IBM ASCII Latin 1
IBM852       , ! IBM ASCII Latin 2
IBM855       , ! IBM ASCII Cyrillic
IBM857       , ! IBM ASCII Turkish
IBM860       , ! IBM DOS Portuguese
IBM861       , ! IBM DOS Icelandic
IBM862       , ! IBM DOS Hebrew
IBM863       , ! IBM DOS French Canadian
IBM864       , ! IBM DOS Arabic
IBM865       , ! IBM DOS Nordic
IBM866       , ! IBM DOS Cyrillic
IBM869       , ! IBM DOS Greek
IBM870       , ! IBM EBCDIC Latin 2
IBM871       , ! IBM EBCDIC Iceland
IBM880       , ! IBM EBCDIC Cyrillic Russian
IBM905       , ! IBM EBCDIC Turkish
IBM1026      , ! IBM EBCDIC Turkish Latin 5

ISO-2022-KR  , ! ISO 2022 Korean
ISO-8859-1   , ! ASCII based Latin-1 (West  European)
ISO-8859-2   , ! ASCII based Latin-2 (East  European)
ISO-8859-3   , ! ASCII based Latin-3 (South European)
ISO-8859-4   , ! ASCII based Latin-4 (North European)
ISO-8859-5   , ! ASCII based Latin/Cyrillic
ISO-8859-6   , ! ASCII based Latin/Arabic
ISO-8859-7   , ! ASCII based Latin/Greek
ISO-8859-9   , ! ASCII based Latin-5 Turkish
ISO-8859-13  , ! ASCII based Latin-7 Baltic Rim
ISO-8859-15  , ! ASCII based Latin-9 Western European

JOHAB        , ! Korean

KOI8-R       , ! Cyrillic 8 bit Russian
KOI8-U       , ! Cyrillic 8 bit Ukrainian

US-ASCII     , ! 7 bit ASCII
```

(continues on next page)

(continued from previous page)

```
UTF-16BE     , ! Unicode 2 byte, Big endian
UTF-16LE     , ! Unicode 2 byte, Little endian
UTF-32BE     , ! Unicode 4 byte, Big endian
UTF-32LE     , ! Unicode 4 byte, Little endian

UTF8         , ! Unicode multi-byte and preferred!

WINDOWS-874  , ! ASCII Windows Thai
WINDOWS-1250 , ! ASCII Windows Latin Central European
WINDOWS-1251 , ! ASCII Windows Cyrillic
WINDOWS-1252 , ! ASCII Windows Latin Wetern European
WINDOWS-1253 , ! ASCII Windows Greek
WINDOWS-1254 , ! ASCII Windows Turkish
WINDOWS-1255 , ! ASCII Windows Hebrew
WINDOWS-1256 , ! ASCII Windows Arabic
WINDOWS-1257 , ! ASCII Windows Latin Baltic
WINDOWS-1258 } ! ASCII Windows Vietnamese
```

Note: Not all character encodings enumerated above may be available on your system. The subset of available character encodings is *AllAvailableCharacterEncodings* (page 779). The set *AllCharacterEncodings* (page 782) is the range for the options:

- `aim_input_character_encoding` used for reading and writing of model text files,
- `ascii_case_character_encoding` used for reading cases created by the ASCII flavor of AIMMS 3.13 and older,
- `default_input_character_encoding` used during a `read from file` statement,
- `default_output_character_encoding` used during a `write to file` and `put` statements, and
- `external_string_character_encoding` used for communicating strings to external DLLs.

See also:

- Paragraph Text files in the preliminaries of the Language Reference 18.
- The encoding attribute of files, see Character encoding used in text files of the Language Reference.
- The set of character encodings available to the current AIMMS session: *AllAvailableCharacterEncodings* (page 779).

8.1.7 AllColors

The predefined set *AllColors* (page 784) contains the names of all users colors associated with an AIMMS project.

```
Set AllColors {
    Index      :  IndexColors;
}
```

Definition

The contents of the set *AllColors* (page 784) is the collection of all user colors defined for a particular project through the **User Colors** dialog box.

Updatability

The contents of the set can only be modified through the **User Colors** dialog box, or programmatically through the functions *UserColorAdd* (page 716) and *UserColorDelete* (page 716).

Note: The set *AllColors* (page 784) is typically used to allow programmatic assignment of colors to data displayed in the graphical end-user interface in a data-driven manner.

See also:

The use of user colors is explained in full detail in Setting colors within the model.

8.1.8 AllIntrinsics

The predefined set *AllIntrinsics* (page 785) contains the names of all standard AIMMS functions and operators.

```
Set AllIntrinsics {
    SubsetOf   :  AllSymbols;
    Index      :  IndexIntrinsics;
}
```

Definition

The contents of the set *AllIntrinsics* (page 785) is the collection of all standard functions and operators.

Updatability

The contents of the set cannot be modified.

See also:

The set *AllSymbols* (page 787).

8.1.9 AllKeywords

The predefined set *AllKeywords* (page 785) contains the names of all keywords in AIMMS.

```
Set AllKeywords {
    SubsetOf   :  AllSymbols;
    Index      :  IndexKeywords;
}
```

Definition

The contents of the set *AllKeywords* (page 785) is the collection of all keywords.

Updatability

The contents of the set cannot be modified.

See also:

The set *AllSymbols* (page 787).

8.1.10 AllOptions

The predefined set *AllOptions* (page 786) contains the names of all options available in AIMMS.

```
Set AllOptions {
    Index       :  IndexOptions;
}
```

Definition

The contents of the set *AllOptions* (page 786) is the collection of all options available in AIMMS from the language and through the **Options** dialog box.

Updatability

The contents of the set can only be modified through the **Solver Configuration** dialog box. By adding or removing solvers the corresponding solver options will be added or removed in the set *AllOptions* (page 786).

Note: In the set *AllOptions* (page 786), the solver specific options are prefixed by the solver name and version.

See also:

Options in AIMMS is described in detail in Basic Concepts of the 'User's Guide'.

8.1.11 AllPredeclaredIdentifiers

The predefined set *AllPredeclaredIdentifiers* (page 786) contains the names of all predeclared identifiers in AIMMS.

```
Set AllPredeclaredIdentifiers {
    SubsetOf    :  AllSymbols;
    Index       :  IndexPredeclaredIdentifiers;
}
```

Definition

The contents of the set *AllPredeclaredIdentifiers* (page 786) is the collection of all predeclared identifier names.

Updatability

The contents of the set cannot be modified.

See also:

The set *AllSymbols* (page 787).

8.1.12 AllSolvers

The predefined set *AllSolvers* (page 787) contains the names of all types of solvers associated with the AIMMS system installed on a particular computer.

```
Set AllSolvers {
    Index       :  IndexSolvers;
}
```

Definition

The contents of the set *AllSolvers* (page 787) is the collection of all types of solvers linked to a particular AIMMS system through the **Solver Configuration** dialog box.

Updatability

The contents of the set can only be modified through the **Solver Configuration** dialog box.

Note: The set *AllSolvers* (page 787) can be used in applications to test whether one or more solvers are available, as illustrated in the AIMMS example `Economic Exchange Equilibrium`.

See also:

- Solver configuration is discussed in full detail in Solver configuration.
- The parameter *CurrentSolver* (page 790).
- The functions *GMP::Instance::CreateSolverSession* (page 378) and *GMP::Instance::GetSolver* (page 406)

8.1.13 AllSymbols

The predefined set *AllSymbols* (page 787) contains the names of identifiers, predeclared identifiers, keywords, and intrinsics.

```
Set AllSymbols {
    Index      :  IndexSymbols;
    Definition :  {
         AllPredeclaredIdentifiers + AllIdentifiers +
         AllKeywords + AllIntrinsics
    }
}
```

Definition

The contents of the set *AllSymbols* (page 787) is the collection of all identifiers, predeclared identifiers, keywords, and intrinsics.

Updatability

The contents of the set can only be modified by adding or deleting identifiers in the **Model Explorer**.

See also:

The sets *AllIdentifiers* (page 821), *AllPredeclaredIdentifiers* (page 786), *AllKeywords* (page 785), and *AllIntrinsics* (page 785).

8.1.14 CurrentAuthorizationLevel

The predefined element parameter *CurrentAuthorizationLevel* (page 788) refers to the authorization level assigned to the user currently logged on to an AIMMS project.

```
ElementParameter CurrentAuthorizationLevel {
    Range      :  AllAuthorizationLevels;
}
```

Definition

The contents of the element parameter *CurrentAuthorizationLevel* (page 788) is the authorization level assigned to the user currently logged on to a project, as assigned by the User Administrator in the **User Setup** dialog box.

Updatability

The contents of *CurrentAuthorizationLevel* (page 788) can only be modified by logging on to the project as another user through the **File-Authorization-User** menu, or by directly modifying the authorization level through the **File-Authorization-Level** menu.

Note: The element parameter *CurrentAuthorizationLevel* (page 788) is typically used refer to the slice of model-defined data that defines access rights to various parts of the model or end-user interface of a model. By referring to the data slice determined by the value of element parameter *CurrentAuthorizationLevel* (page 788), AIMMS will use the accessibility rights associated with the authorization level of the current logged on user. The use of authorization levels in AIMMS directly is deprecated, as user authentication and authorization during deployment is now arranged via AIMMS PRO (cf. Section Project Security).

See also:

The set *AllAuthorizationLevels* (page 779).

8.1.15 ProfilerData

The predefined parameter *ProfilerData* (page 789) can be used by AIMMS to store profiling information about the execution of procedures and the updating of definitions.

```
Parameter ProfilerData {
    IndexDomain  :  ( IndexIdentifiers, IndexprofilerTypes );
}
```

Note:

- Profiling information is only stored in the parameter *ProfilerData* (page 789) if the profiler has been activated and if the option `profiler_store_data` has been set to `On`.
- The number of reported hits is an postive integer and all reported profiling times are measured in seconds.

See also:

The function *ProfilerStart* (page 720) and the predefined identifier *AllProfilerTypes* (page 804).

8.1.16 CurrentGroup

The predefined string parameter *CurrentGroup* (page 789) contains the name of the user group associated with the user currently logged on to an AIMMS project.

```
StringParameter CurrentGroup;
```

Definition

The contents of the string parameter *CurrentGroup* (page 789) is the name of the user group associated with the user currently logged on to a project. User groups are defined by the User Administrator in the **User Setup** dialog box.

Updatability

The contents of *CurrentGroup* (page 789) can only be modified by logging on to the project as another user through the **File-Authorization-User** menu, or directly through the **File-Authorization-Group** menu.

Note: The string parameter *CurrentGroup* (page 789) only contains data when the project has been linked to a user database. The use of User Groups in AIMMS directly is deprecated, as user authentication and authorization during deployment is now arranged via AIMMS PRO (cf. Section Project Security).

See also:

The function *SecurityGetGroups* (page 749).

8.1.17 CurrentSolver

The predefined element parameter *CurrentSolver* (page 790) contains, for every mathematical programming type, the name of the solver that AIMMS will currently use to solve models of that type.

```
ElementParameter CurrentSolver {
    IndexDomain  :  IndexMathematicalProgrammingTypes;
    Range        :  AllSolvers;
}
```

Definition

The contents of the element parameter *CurrentSolver* (page 790) are, for all types of mathematical programs, the names of the currently active solver for solving mathematical programs of each type, as set through the **Solver Configuration** dialog box.

Updatability

The value of *CurrentSolver* (page 790) can also be modified programmatically from within an AIMMS model, and then determines the solver that will be used to solve subsequent problems of the specified type. Modifying the values of *CurrentSolver* (page 790) will, however, not modify the (default) settings in the **Solver Configuration** dialog box, that will be loaded at startup.

Note:

- The procedure *GMP::Instance::Solve* (page 424) takes *CurrentSolver* (page 790) into account unless a solver has been assigned using the procedure *GMP::Instance::SetSolver* (page 423).

- The procedures *GMP::SolverSession::Execute* (page 530) and *GMP::SolverSession::AsynchronousExecute* (page 528) take *CurrentSolver* (page 790) into account unless a solver has been assigned using the function *GMP::Instance::CreateSolverSession* (page 378) or the procedure *GMP::Instance::SetSolver* (page 423).

See also:

- The sets *AllMathematicalProgrammingTypes* (page 802) and *AllSolvers* (page 787).
- Solver configuration is discussed in full detail in Solver configuration.

8.1.18 AimmsStringConstants

The predefined string parameter *AimmsStringConstants* (page 791) contains the constituents that determine the running version of AIMMS. It is used to determine which installation of AIMMS is running.

```
StringParameter AimmsStringConstants {
    IndexDomain :  ( IndexAimmsStringConstantElements );
}
```

This string parameter contains the following elements:

Platform AIMMS supports the platform "Windows", and the platform "Linux".

Architecture The architecture for 32 bit systems is known as "x86", and the architecture for 64 bit systems is known as "x64".

Flavor AIMMS comes only in a single flavor: "utf8". Up to AIMMS 3.13, AIMMS came in the single byte per character flavor, abbreviated to "asc", and it came in the two byte per character flavor, abbreviated to "uni". For the Linux platform only the asc flavor was available.

Example

```
StringParameter myDllName {
    Definition : {
        AimmsStringConstants('Architecture') + "\\" +
        AimmsStringConstants('Flavor') + "\\" +
        "myDll.dll"
    }
}
```

A possible outcome of myDllName is x86\asc\myDll.dll.

See also:

The function *EnvironmentGetString* (page 752) and the predeclared set *AllAimmsStringConstantElements* (page 791).

8.1.19 AllAimmsStringConstantElements

The predefined set *AllAimmsStringConstantElements* (page 791) contains the elements for which the predeclared string parameter *AimmsStringConstants* (page 791) has a value.

```
Set AllAimmsStringConstantElements {
    Index      :  IndexAimmsStringConstantElements;
}
```

Definition

This set is fixed to { Platform, Architecture, Flavor }.

8.1.20 CurrentUser

The predefined string parameter *CurrentUser* (page 792) contains the name of the user currently logged on to an AIMMS project.

```
StringParameter CurrentUser;
```

Definition

The contents of the string parameter *CurrentUser* (page 792) is the name of the user currently logged on to a project. Project users are defined by the User Administrator in the **User Setup** dialog box.

Updatability

The contents of *CurrentUser* (page 792) can only be modified by logging on to the project as another user through the **File-Authorization-User** menu.

Note: The string parameter *CurrentUser* (page 792) only contains data when the project has been linked to a user database. The use of User Groups in AIMMS directly is deprecated, as user authentication and authorization during deployment is now arranged via AIMMS PRO (cf. Section Project Security).

See also:

The function *SecurityGetUsers* (page 750).

8.2 Language Related Identifiers

The following collection of predefined identifiers define various sets containing similar keywords from the AIMMS language. These sets are mostly used for the specification of accurate prototypes of intrinsic AIMMS functions.

8.2.1 AggregationTypes

The predefined set *AggregationTypes* (page 792) contains the collection of all possible aggregation types supported by the *Aggregate* (page 91) and *DisAggregate* (page 101) functions.

```
Set AggregationTypes {
    Index      :  IndexAggregationTypes;
    Definition :  {
        data { summation, average,
               maximum, minimum,
               interpolation }
        }
    }
```

Definition

The set *AggregationTypes* (page 792) contains the collection of all possible aggregation types supported by the *Aggregate* (page 91) and *DisAggregate* (page 101) functions.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *AggregationTypes* (page 792) can be used as the *type* argument of the *Aggregate* (page 91) and *DisAggregate* (page 101) functions.

See also:

The functions *Aggregate* (page 91) and *DisAggregate* (page 101). Time-dependent aggregation and disaggregation is discussed in full detail in Data Conversion of Time-Dependent Identifiers of the Language Reference.

8.2.2 AllAttributeNames

The predefined set *AllAttributeNames* (page 793) contains the names of all possible identifier attributes.

```
Set AllAttributeNames {
    Index      :  IndexAttributeNames;
}
```

Definition

The predefined set *AllAttributeNames* (page 793) contains the names of all possible identifier attributes.

Updatability

The contents of the set cannot be modified.

See also:

- The sets *AllIdentifierTypes* (page 800), and *AllSuffixNames* (page 809).
- Model edit functions, see Runtime Libraries and the Model Edit Functions of the Language Reference.
- The functions *me::AllowedAttribute* (page 599) and *IdentifierAttributes* (page 587).

8.2.3 AllBasicValues

The predefined set *AllBasicValues* (page 794) contains the names of all basic values available in AIMMS.

```
Set AllBasicValues {
    Index      :  IndexBasicValues;
    Definition :  data { NonBasic, Basic, SuperBasic };
}
```

Definition

The set *AllBasicValues* (page 794) contains the names of all basic values in AIMMS.

Updatability

The contents of the set cannot be modified.

8.2.4 AllCaseComparisonModes

The predefined set *AllCaseComparisonModes* (page 794) contains the collection of all possible modes supported by the *CaseCompareIdentifier* (page 623) function.

```
Set AllCaseComparisonModes {
    Index      :  IndexCaseComparisonModes;
    Definition :  {
        data { min, max, sum
               average, count }
      }
}
```

Definition

The predefined set *AllCaseComparisonModes* (page 794) contains the collection of all possible modes supported by the *CaseCompareIdentifier* (page 623) function.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *AllCaseComparisonModes* (page 794) can be used as the *mode* argument of the *CaseCompareIdentifier* (page 623) function.

See also:

The function *CaseCompareIdentifier* (page 623).

8.2.5 AllColumnTypes

The predefined set *AllColumnTypes* (page 795) contains the names of all column types available in the GMP library of AIMMS.

```
Set AllColumnTypes {
    Index      :  IndexColumnTypes;
    Definition :  data { integer, continuous };
}
```

Definition

The set *AllColumnTypes* (page 795) contains the names of all column types available in the GMP library of AIMMS.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *AllColumnTypes* (page 795) can be used as the *type* argument of the *GMP::Column::SetType* (page 347) function.

See also:

The function *GMP::Column::SetType* (page 347). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.6 AllDataColumnCharacteristics

The predefined set *AllDataColumnCharacteristics* (page 795) contains all possible column properties, which can be queried using the function *SQLColumnData* (page 656).

```
Set AllDataColumnCharacteristics {
    Index      :  IndexDataColumnCharacteristics;
    Definition :  {
        data { Name, DataType, Width,
               NumberOfDecimals, IsPrimaryKey,
               Nullable, DefaultValue, Remark }
    }
}
```

Definition

The set *AllDataColumnCharacteristics* (page 795) contains all possible column properties, which can be queried using the function *SQLColumnData* (page 656). They are:

- `Name` : The name of the column.
- `DataType` : The data type of the column.
- `Width` : The column width.
- `NumberOfDecimals` : The number of decimals of the column. Only applicable for numeric columns.
- `IsPrimaryKey` : Specfies whether the column is part of the primary key for the database table. Returns `"Yes"` or `"No"`.
- `Nullable` : Specifies whether the column is nullable or not. Returns `"Yes"` or `"No"`.
- `DefaultValue` : The default value of the column.
- `Remark` : The remark associated with the column.

Updatability

The contents of the set cannot be modified.

See also:

The function *SQLColumnData* (page 656).

8.2.7 AllDataSourceProperties

The predefined set *AllDataSourceProperties* (page 796) contains all datasource properties, which can be queried using the function `GetDataSourceProperty`.

```
Set AllDataSourceProperties {
    Index      :  IndexDataSourceProperties;
    Definition :
         data { SQL_DATA_SOURCE_NAME, SQL_DATA_SOURCE_READ_ONLY,
                SQL_DBMS_NAME, SQL_DBMS_VER, SQL_DRIVER_NAME,
```

(continues on next page)

(continued from previous page)

```
                SQL_DM_VER, SQL_DRIVER_VER, SQL_KEYWORDS,
                SQL_SERVER_NAME }
    }
}
```

Definition

The set *AllDataSourceProperties* (page 796) contains all datasource properties, which can be queried using the function `GetDataSourceProperty`. They are:

- `SQL_DATA_SOURCE_NAME` : The name of the datasource.
- `SQL_DATA_SOURCE_READ_ONLY` : The read-only status of the datasource. Returns `"Yes"` or `"No"`.
- `SQL_DBMS_NAME` : The name of the database system (e.g., returns `"Oracle"` for an Oracle database).
- `SQL_DBMS_VER` : The version of the database system.
- `SQL_DRIVER_NAME` : The actual DLL of the ODBC driver for the datasource.
- `SQL_DM_VER` : The version of the ODBC driver manager.
- `SQL_DRIVER_VER` : The version of the ODBC driver for the datasource.
- `SQL_KEYWORDS` : A comma-separated list of all reserved keywords of the datasource.
- `SQL_SERVER_NAME` : The datasource-specific server name.

Updatability

The contents of the set cannot be modified.

See also:

The function *GetDataSourceProperty* (page 652).

8.2.8 AllDifferencingModes

The predefined set *AllDifferencingModes* (page 797) contains the collection of all possible differencing modes supported by the *CaseCreateDifferenceFile* (page 626) function.

```
Set AllDifferencingModes {
    Index      :  IndexDifferencingModes;
    Definition :  {
         data { blockReplacement, elementReplacement,
                elementAddition, elementMultiplication  }
    }
}
```

Definition

The predefined set *AllDifferencingModes* (page 797) contains the collection of all possible differencing modes supported by the *CaseCreateDifferenceFile* (page 626) function:

- `blockReplacement`: When there are differences between the reference case and the current case for an identifier the data of that identifier in the current case is entirely displayed.
- `elementReplacement`: When there are differences between the reference case and the current case for an identifier the differing elements in the current case are displayed. This may include defaults for elements deleted.
- `elementAddition`: When there are differences between the reference case and the current case for an identifier the differences between elements in the current case and reference case are displayed.
- `elementMultiplication`: When there are differences between the reference case and the current case for an identifier the relative differences between elements in the current case and reference case are displayed.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *AllDifferencingModes* (page 797) can be used as the *diffTypes* argument of the *CaseCreateDifferenceFile* (page 626) function.

See also:

The function *CaseCreateDifferenceFile* (page 626).

8.2.9 AllExecutionStatuses

The predefined set *AllExecutionStatuses* (page 798) contains the names of all execution statuses associated with asynchronous solves.

```
Set AllExecutionStatuses {
    Index        :  IndexExecutionStatus;
}
```

Definition

The set *AllExecutionStatuses* (page 798) contains the names of all execution statuses associated with asynchronous solves. The execution status of an asynchronous solve can be queried using the function `GMP::SolverSession::ExecutionStatus`.

See also:

The function *GMP::SolverSession::ExecutionStatus* (page 531).

8.2.10 AllGMPExtensions

The predefined set *AllGMPExtensions* (page 798) contains the collection of all possible extensions in the GMP library of AIMMS.

```
Set AllGMPExtensions {
    Index      :  IndexGMPExtensions;
    Definition :  {
         data { DualObjective, DualDefinition,
                DualLowerBound, DualUpperBound }
    }
}
```

Definition

The predefined set *AllGMPExtensions* (page 798) contains the collection of all possible extensions in the GMP library of AIMMS.

Updatability

The contents of the set cannot be modified.

Note: Together with the suffixes `.ExtendedConstraint` and `.ExtendedVariable`, element parameters into *AllGMPExtensions* (page 798) can be used as the `extension` argument of a constraint, a variable, and a mathematical program.

See also:

The set *AllSuffixNames* (page 809). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.11 AllFileAttributes

The predefined set *AllFileAttributes* (page 799) contains the attributes which can be used in the filtering of files.

```
Set AllFileAttributes {
    Index      :  IndexFileAttributes;
    Definition :  data { Hidden, ReadOnly, Executable };
}
```

Definition

The predefined set *AllFileAttributes* (page 799) contains the attributes the intrinsic functions *DirectoryGetFiles* (page 763), and DirectoryGetSubDirectories use to filter their result. They are:

- Hidden: the file or subdirectory is normally not visible when querying the folder in which it resides,
- ReadOnly: the file or subdirecotry is read only,
- Executable: the file is executable (this attribute is ignored for DirectoryGetSubdirectories).

Updatability

The contents of the set cannot be modified.

See also:

The functions *DirectoryGetFiles* (page 763), and DirectoryGetSubDirectories.

8.2.12 AllIdentifierTypes

The predefined set *AllIdentifierTypes* (page 800) contains the names of all possible identifier types.

```
Set AllIdentifierTypes {
    Index      :  IndexIdentifierTypes;
}
```

Definition

The predefined set *AllIdentifierTypes* (page 800) contains the names of all possible identifier types.

Updatability

The contents of the set can not be modified; it has the following fixed contents:

```
AllIdentifierTypes := data
{ set                       ,
  calendar                  ,
  horizon                   ,
  index                     ,
  parameter                 ,
  'element parameter'       ,
  'string parameter'        ,
  'unit parameter'          ,
  variable                  ,
  'element variable'        ,
  'complementarity variable',
  constraint                ,
  arc                       ,
  node                      ,
```

(continues on next page)

(continued from previous page)

```
  'uncertainty variable'    ,
  'uncertainty constraint'  ,
  activity                  ,
  resource                  ,
  'mathematical program'    ,
  macro                     ,
  assertion                 ,
  'database table'          ,
  'database procedure'      ,
  file                      ,
  procedure                 ,
  function                  ,
  quantity                  ,
  convention                ,
  LibraryModule             ,
  module                    ,
  section                   ,
  declaration               } ;
```

See also:

- The sets *AllAttributeNames* (page 793) and *AllSuffixNames* (page 809).
- Model edit functions, see Runtime Libraries and the Model Edit Functions of the Language Reference.
- The functions *me::ChangeType* (page 600) and *IdentifierType* (page 591).

8.2.13 AllIsolationLevels

The predefined set *AllIsolationLevels* (page 801) contains the supported isolation levels for a database transaction, as started through the procedure *StartTransaction* (page 650).

```
Set AllIsolationLevels {
    Index      :  IndexIsolationLevels;
    Definition :  {
        data { ReadUncommitted, ReadCommitted,
               RepeatableRead, Serializable }
    }
}
```

Definition

The predefined set *AllIsolationLevels* (page 801) contains the supported isolation levels for a database transaction. They are:

- `ReadUncommitted`: a transaction operating at this level can see uncommitted changes made by other transactions,
- `ReadCommitted` (default): a transaction operating at this level cannot see changes made by other transactions until those transactions are committed,
- `RepeatableRead`: a transaction operating at this level is guaranteed not to see any changes made by other transactions in values it has already read during the transaction, and

- `Serializable`: a transaction operating at this level guarantees that all concurrent transactions interact only in ways that produce the same effect as if each transaction were entirely executed one after the other.

Updatability

The contents of the set cannot be modified.

Note: Not all database servers may support all of these isolation levels, and may cause the call to `StartTransaction` to fail.

See also:

The function *StartTransaction* (page 650).

8.2.14 AllMathematicalProgrammingTypes

The predefined set *AllMathematicalProgrammingTypes* (page 802) contains the list of mathematical programming types supported by AIMMS.

```
Set AllMathematicalProgrammingTypes {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexMathematicalProgrammingTypes;
}
```

Definition

The set *AllMathematicalProgrammingTypes* (page 802) contains the list of mathematical programming types supported by AIMMS.

Updatability

The contents of the set *AllMathematicalProgrammingTypes* (page 802) is completely under the control of AIMMS, and cannot be modified.

Note: Element parameters into the set *AllMathematicalProgrammingTypes* (page 802) can be used in the declaration of mathematical programs or as part of the `SOLVE` statement to dynamically modify the type of a mathematical program. The predefined identifier *CurrentSolver* (page 790) defines the active solver for each mathematical programming type.

See also:

The set *AllValueKeywords* (page 809), *CurrentSolver* (page 790). Mathematical programs are discussed in full detail in MathematicalProgram Declaration and Attributes of the Language Reference, the `SOLVE` statement in The SOLVE Statement

8.2.15 AllMatrixManipulationDirections

The predefined set *AllMatrixManipulationDirections* (page 802) contains the list of optimization directions supported by the GMP library of AIMMS.

```
Set AllMatrixManipulationDirections {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexMatrixManipulationDirections;
}
```

Definition

The set *AllMatrixManipulationDirections* (page 802) contains the list of optimization directions supported by the GMP library of AIMMS.

Updatability

The contents of the set *AllMatrixManipulationDirections* (page 802) is completely under the control of AIMMS, and cannot be modified.

Note: Element parameters into the set *AllMatrixManipulationDirections* (page 802) can be used as the *direction* argument of the *GMP::Instance::SetDirection* (page 419) function.

See also:

The set *AllValueKeywords* (page 809), the function *GMP::Instance::SetDirection* (page 419). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.16 AllMatrixManipulationProgrammingTypes

The predefined set *AllMatrixManipulationProgrammingTypes* (page 803) contains the collection of mathematical programming types that can be used in conjunction with the GMP library of AIMMS.

```
Set AllMatrixManipulationProgrammingTypes {
    SubsetOf   :  AllMathematicalProgrammingTypes;
    Index      :  IndexMatrixManipulationProgrammingTypes;
}
```

Definition

The predefined set *AllMatrixManipulationProgrammingTypes* (page 803) contains the collection of mathematical programming types that can be used in conjunction with the GMP library of AIMMS.

Updatability

The contents of the set *AllMatrixManipulationProgrammingTypes* (page 803) is completely under the control of AIMMS, and cannot be modified.

Note: Element parameters into the set *AllMatrixManipulationDirections* (page 802) can be used as the *type* argument of the *GMP::Instance::SetMathematicalProgrammingType* (page 420) function.

See also:

The set *AllMathematicalProgrammingTypes* (page 802), the function *GMP::Instance::SetMathematicalProgrammingType* (page 420). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.17 AllProfilerTypes

The predefined set *AllProfilerTypes* (page 804) contains the names of all types of profiler data that can be stored in the predefined identifier `ProfilerData`.

```
Set AllProfilerTypes {
    Index      :  IndexprofilerTypes;
}
```

Definition

The set *AllProfilerTypes* (page 804) currently contains the profiler types '*hits*', '*gross time*', and '*net time*'.

See also:

The function *ProfilerStart* (page 720) and the predefined parameter *ProfilerData* (page 789).

8.2.18 AllConstraintProgrammingRowTypes

The predefined set *AllConstraintProgrammingRowTypes* (page 804) contains the collection of all possible row types available to be used by the Constraint Programming global constraint *cp::Count* (page 272).

```
Set AllConstraintProgrammingRowTypes {
    SubsetOf   :  AllRowTypes;
    Index      :  IndexConstraintProgrammingRowTypes;
    Definition :  data { '<=', '=', '>=', '<', '>', '<>' };
}
```

Definition

The set *AllConstraintProgrammingRowTypes* (page 804) contains the collection of all possible row types available as relation operator to the function *cp::Count* (page 272).

Updatability

The contents of the set cannot be modified.

See also:

The function *cp::Count* (page 272) and the set *AllRowTypes* (page 805)

8.2.19 AllRowColumnStatuses

The predefined set *AllRowColumnStatuses* (page 805) contains the collection of all possible row and column statuses.

```
Set AllRowColumnStatuses {
    Index      :  IndexRowColumnStatuses;
    Definition :  data { Active, Deactivated, Deleted, NotGenerated, PresolveDeleted };
}
```

Definition

The set *AllRowColumnStatuses* (page 805) contains the collection of all possible row and column statuses available in the GMP library of AIMMS.

Updatability

The contents of the set cannot be modified.

Note: The functions *GMP::Column::GetStatus* (page 333) and *GMP::Row::GetStatus* (page 465) return an element in *AllRowColumnStatuses* (page 805).

See also:

The functions *GMP::Column::GetStatus* (page 333) and *GMP::Row::GetStatus* (page 465). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.20 AllRowTypes

The predefined set *AllRowTypes* (page 805) contains the collection of all possible row types and is the superset of .

```
Set AllRowTypes {
    Index      :  IndexRowTypes;
    Definition :  data { '<=', '=', '>=', ranged, '<', '>', '<>' };
}
```

Definition

The set *AllRowTypes* (page 805) contains the collection of all possible row types available in AIMMS.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *AllRowTypes* (page 805) can be used as the *type* argument of the *GMP::Row::SetType* (page 476) function.

See also:

The function *GMP::Row::SetType* (page 476). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.21 AllMathematicalProgrammingRowTypes

The predefined set *AllMathematicalProgrammingRowTypes* (page 806) contains the collection of all possible row types available in the GMP library of AIMMS.

```
Set AllMathematicalProgrammingRowTypes {
    Index      :  IndexMathematicalProgrammingRowTypes;
    Definition :  data { '<=', '=', '>=', ranged };
}
```

Definition

The set *AllMathematicalProgrammingRowTypes* (page 806) contains the collection of all possible row types available in the GMP library of AIMMS.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *AllMathematicalProgrammingRowTypes* (page 806) can be used as the *type* argument of the *GMP::Row::SetType* (page 476) function.

See also:

The function *GMP::Row::SetType* (page 476) and the super set *AllRowTypes* (page 805). The GMP library is discussed in more detail in Implementing Advanced Algorithms for Mathematical Programs of the Language Reference.

8.2.22 AllSolutionStates

The predefined set *AllSolutionStates* (page 807) contains the names of possible values of the program and solver status of a mathematical program.

```
Set AllSolutionStates {
    Index      :  IndexSolutionStates;
}
```

Definition

The set *AllSolutionStates* (page 807) contains the names of all possible values of the `ProgramStatus` and `SolverStatus` suffixes of a mathematical program.

Updatability

The contents of the set cannot be modified.

Note: The suffixes `ProgramStatus` and `SolverStatus` of a mathematical program take their values in the set *AllSolutionStates* (page 807).

See also:

The program status and solver status are discussed in more detail in Suffices and Callbacks of the Language Reference.

8.2.23 AllSolverInterrupts

The predefined set *AllSolverInterrupts* (page 807) contains the names of all causes for a callback.

```
Set AllSolverInterrupts {
    Index      :  IndexSolverInterrupts;
    Definition :  {
        data { AddCut, Branch, Candidate, Heuristic, Incumbent,
               Iterations, StatusChange, AddLazyConstraint,
```

(continues on next page)

(continued from previous page)

```
                Finished, Time }
    }
}
```

Definition

The set *AllSolverInterrupts* (page 807) contains the names of all causes for a callback.

Updatability

The contents of the set cannot be modified.

Note: If you have installed the same callback procedure for several callbacks, you can call the function `GMP::SolverSession::GetCallbackInterruptStatus`, which returns an element into the set *AllSolverInterrupts* (page 807), to obtain the particular callback for which your callback procedure was called.

See also:

The routines *GMP::Instance::SetCallbackAddCut* (page 410), *GMP::Instance::SetCallbackAddLazyConstraint* (page 411), *GMP::Instance::SetCallbackBranch* (page 412), *GMP::Instance::SetCallbackCandidate* (page 413), *GMP::Instance::SetCallbackHeuristic* (page 414), *GMP::Instance::SetCallbackIncumbent* (page 415), *GMP::Instance::SetCallbackStatusChange* (page 417), *GMP::Instance::SetCallbackTime* (page 417), and *GMP::SolverSession::GetCallbackInterruptStatus* (page 541).

8.2.24 AllStochasticGenerationModes

The predefined set *AllStochasticGenerationModes* (page 808) defines the modes in which *GMP::Instance::GenerateStochasticProgram* (page 387) may generate a stochastic programming problem.

```
Set AllStochasticGenerationModes {
    Index      :  IndexStochasticGenerationModes;
    Definition :  {
        data { CreateNonAnticipativityConstraints,
               SubstituteStochasticVariables }
    }
}
```

Definition

The predefined set *AllStochasticGenerationModes* (page 808) defines the set of elements CreateNonAnticipativityConstraints and SubstituteStochasticVariables.

Updatability

The contents of the set *AllStochasticGenerationModes* (page 808) cannot be modified.

See also:

- Stochastic programming is discussed in Stochastic Programming of the Language Reference.
- The intrinsic function *GMP::Instance::GenerateStochasticProgram* (page 387).

8.2.25 AllSuffixNames

The predefined set *AllSuffixNames* (page 809) contains the names of all existing suffixes of all identifier types.

```
Set AllSuffixNames {
    Index      :  IndexSuffixNames;
}
```

Definition

The set *AllSuffixNames* (page 809) contains the names of all possible suffixes for the entire collection of identifier types.

Updatability

The contents of the set cannot be modified.

See also:

- The set *AllIdentifiers* (page 821).
- The functions *ScalarValue* (page 32), *ActiveCard* (page 45), *Card* (page 46), *CaseCompareIdentifier* (page 623), and *GMP::Solution::SendToModelSelection* (page 504).

8.2.26 AllValueKeywords

The predefined set *AllValueKeywords* (page 809) serves as the root set of various other predefined sets containing AIMMS keywords.

```
Set AllValueKeywords {
    Index      :  IndexValueKeywords;
    Definition :  {
        AllMathematicalProgrammingTypes +
        AllMatrixManipulationDirections +
```

(continues on next page)

(continued from previous page)

```
        AllViolationTypes + YesNo +
        ContinueAbort + MergeReplace + OnOff +
        DiskWindowVoid + MaximizingMinimizing
    }
}
```

Definition

The set *AllValueKeywords* (page 809) contains keywords used in various other predefined sets containing AIMMS keywords.

Updatability

The contents of the set *AllValueKeywords* (page 809) is completely under the control of AIMMS, and cannot be modified.

Note: The set *AllValueKeywords* (page 809) is, in general, of little direct use in an AIMMS application.

See also:

The sets *AllMathematicalProgrammingTypes* (page 802), *AllMatrixManipulationDirections* (page 802), *AllViolationTypes* (page 810), *YesNo* (page 815), *ContinueAbort* (page 811), *DiskWindowVoid* (page 811), *MaximizingMinimizing* (page 813), *MergeReplace* (page 813), *OnOff* (page 814).

8.2.27 AllViolationTypes

The predefined set *AllViolationTypes* (page 810) contains the collection of all violation types for which violation penalties can be specified in a mathematical program declaration.

```
Set AllViolationTypes {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexViolationTypes;
    Definition :  data { Lower, Upper, Definition };
}
```

Definition

The set *AllViolationTypes* (page 810) contains the violation types for which violation penalties can be specified in a mathematical program declaration.

Updatability

The contents of the set *AllViolationTypes* (page 810) is completely under the control of AIMMS, and cannot be modified.

Note: The set *AllViolationTypes* (page 810) is typically used in the index domain of identifiers specified in the `ViolationPenalties` attribute of a `MathematicalProgram`.

See also:

The sets *AllMathematicalProgrammingTypes* (page 802), *AllMatrixManipulationDirections* (page 802), *ContinueAbort* (page 811), *DiskWindowVoid* (page 811), *MaximizingMinimizing* (page 813), *MergeReplace* (page 813), *OnOff* (page 814). The `ViolationPenalties` attribute of a mathematical programs is discussed in Infeasibility Analysis of the Language Reference.

8.2.28 ContinueAbort

The predefined set *ContinueAbort* (page 811) defines the set of possible return statuses of solver callback procedures.

```
Set ContinueAbort {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexContinueAbort;
    Definition :  data { continue, abort };
}
```

Definition

The set *ContinueAbort* (page 811) defines the set of possible return statuses of solver callback procedures.

Updatability

The contents of the set cannot be modified.

Note: The elements of the set *ContinueAbort* (page 811) can be assigned to the `CallbackReturnStatus` suffix of a mathematical program upon return of a solver callback procedure.

See also:

The set *AllValueKeywords* (page 809). Solver callback procedures are discussed in Suffices and Callbacks of the Language Reference.

8.2.29 DiskWindowVoid

The predefined set *DiskWindowVoid* (page 811) defines the set of possible devices of file identifiers.

```
Set DiskWindowVoid {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexDiskWindowVoid;
    Definition :  data { disk, window, void };
}
```

Definition

The predefined set *DiskWindowVoid* (page 811) defines the set of possible devices which can be entered in the Device attribute of a File identifier.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into the set *DiskWindowVoid* (page 811) can be entered in the Device attribute of File identifiers to allow dynamic device changes for a file.

See also:

The set *AllValueKeywords* (page 809). File identifiers are discussed in The File Declaration of the Language Reference.

8.2.30 Integers

The predefined set *Integers* (page 812) defines the range of allowed integer set elements in AIMMS.

```
Set Integers {
    Index      :  IndexIntegers;
    Definition :  {
        { (-2^30+5) .. (2^30+2) }
    }
}
```

Definition

The set *Integers* (page 812) defines the range of integers that can possibly serve as integer set elements in AIMMS.

Updatability

The contents of the set cannot be modified.

Note: Subsets of the sets *Integers* (page 812) are frequently used to enumerate objects within a model. Datafiles (i.e. cases and datasets) in AIMMS are enumerated as subsets of the set *Integers* (page 812).

See also:

The sets *AllDataFiles* (page 845), *AllCases* (page 842), *AllDataSets* (page 845). Integer sets are discussed in Integer Sets of the Language Reference.

8.2.31 MaximizingMinimizing

The predefined set *MaximizingMinimizing* (page 813) defines the set of possible optimization directions of mathematical programs.

```
Set MaximizingMinimizing {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexMaximizingMinimizing;
    Definition :  data { maximize, minimize };
}
```

Definition

The predefined set *MaximizingMinimizing* (page 813) defines the set of possible optimization directions that can be entered in the `Direction` attribute of mathematical programs.

Updatability

The contents of the set cannot be modified.

Note: Element parameters into the set *MaximizingMinimizing* (page 813) can be entered in the `Direction` attribute of mathematical programs to allow dynamic choices of the optimization direction.

See also:

The set *AllValueKeywords* (page 809). Mathematical programs are discussed in more detail in MathematicalProgram Declaration and Attributes of the Language Reference.

8.2.32 MergeReplace

The predefined set *MergeReplace* (page 813) defines the set of modes for the READ, WRITE and SOLVE statements.

```
Set MergeReplace {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexMergeReplace;
    Definition :  data { merge, replace };
}
```

Definition

The predefined set *MergeReplace* (page 813) defines the set of modes for the READ, WRITE and SOLVE statements as specified through the IN MERGE/REPLACE MODE clause.

Updatability

The contents of the set *MergeReplace* (page 813) cannot be modified.

Note: Element parameters into the set *MergeReplace* (page 813) can be used to dynamically indicate the mode of a READ, WRITE or SOLVE statement.

See also:

The set *AllValueKeywords* (page 809). The SOLVE statement is discussed in The SOLVE Statement of the Language Reference, the READ and WRITE statements in Syntax of the READ and WRITE Statements

8.2.33 OnOff

The predefined set *OnOff* (page 814) defines the set of possibilities the PageMode suffix of File identifiers.

```
Set OnOff {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexOnOff;
    Definition :  data { on, off };
}
```

Definition

The set *OnOff* (page 814) defines the set of possibilities the PageMode suffix of File identifiers.

Updatability

The contents of the set *OnOff* (page 814) cannot be modified.

Note: Element parameters into the set *OnOff* (page 814) assigned to be `PageMode` suffix of a `File` identifier can be used to dynamically change the page mode of a file.

See also:

The set *AllValueKeywords* (page 809). The `PageMode` suffix of `FILE` identifiers is discussed in full detail in Structuring a Page in Page Mode

8.2.34 TimeSlotCharacteristics

The predefined set *TimeSlotCharacteristics* (page 815) contains the collection of timeslot characteristic which can be used in conjunction with the function *TimeSlotCharacteristic* (page 110).

```
Set TimeSlotCharacteristics {
    Index      :  IndexTimeSlotCharacteristics;
    Definition :  {
        data { century, year, quarter, month
        weekday, yearday, monthday
        week, weekyear, weekcentury
        hour, minute, second, tick }
    }
}
```

Definition

The set *TimeSlotCharacteristics* (page 815) contains the collection of timeslot characteristic which can be used in conjunction with the function *TimeSlotCharacteristic* (page 110).

Updatability

The contents of the set cannot be modified.

Note: Element parameters into *TimeSlotCharacteristics* (page 815) can be used as the *characteristic* argument of the *TimeSlotCharacteristic* (page 110) function.

See also:

The function *TimeSlotCharacteristic* (page 110). The use of the function `TimeSlotCharacteristic` is explained in more detail in Creating Timetables of the Language Reference.

8.2.35 YesNo

The predefined set *YesNo* (page 815) defines the set of elements Yes and No.

```
Set YesNo {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexYesNo;
    Definition :  data { yes, no };
}
```

Definition

The predefined set *YesNo* (page 815) defines the set of elements Yes and No.

Updatability

The contents of the set *YesNo* (page 815) cannot be modified.

Note: The set *YesNo* (page 815) is not used by AIMMS anymore.

See also:

The set *AllValueKeywords* (page 809).

8.3 Model Related Identifiers

The following collection of predefined identifiers contains information regarding the model associated with the AIMMS project at hand. The identifiers listed here contain either the complete set of model identifiers or the set of all identifiers of a specific type.

8.3.1 AllAssertions

The predefined set *AllAssertions* (page 816) contains the names of all assertions within an AIMMS model.

```
Set AllAssertions {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexAssertions;
}
```

Definition

The contents of the set *AllAssertions* (page 816) is the collection of all assertion names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting assertions in the **Model Explorer**.

Note: The set *AllAssertions* (page 816) or subsets thereof are typically used in the ASSERT statement in an AIMMS model.

See also:

The sets *AllIdentifiers* (page 821). Assertions are discussed in Assertions of the Language Reference.

8.3.2 AllConstraints

The predefined set *AllConstraints* (page 817) contains the names of all constraints within an AIMMS model.

```
Set AllConstraints {
    SubsetOf   :  AllVariablesConstraints;
    Index      :  IndexConstraints;
}
```

Definition

The contents of the set *AllConstraints* (page 817) is the collection of all *symbolic* constraint names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting constraints in the **Model Explorer**.

Note: The set *AllConstraints* (page 817) or subsets thereof are typically used in the Constraints attribute of MathematicalProgram declared within an AIMMS model.

See also:

The sets *AllIdentifiers* (page 821), *AllVariables* (page 831). Constraints are discussed in Constraint Declaration and Attributes, mathematical programs in MathematicalProgram Declaration and Attributes of the Language Reference.

8.3.3 AllConventions

The predefined set *AllConventions* (page 817) contains the names of all conventions defined within an AIMMS model.

```
Set AllConventions {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexConventions;
}
```

Definition

The contents of the set *AllConventions* (page 817) is the collection of all conventions defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting conventions in the **Model Explorer**.

Note: Element parameters into the set *AllConventions* (page 817) are typically used in the main model node to allow dynamic selection of the unit convention to which the model is subject.

See also:

The sets *AllIdentifiers* (page 821), *AllQuantities* (page 826). Conventions are discussed in full detail in Globally Overriding Units Through Conventions of the Language Reference.

8.3.4 AllDatabaseTables

The predefined set *AllDatabaseTables* (page 818) contains the names of all database tables declared within an AIMMS model.

```
Set AllDatabaseTables {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexDatabaseTables;
}
```

Definition

The contents of the set *AllDatabaseTables* (page 818) is the collection of all database tables declared within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting database tables in the **Model Explorer**.

See also:

The sets *AllIdentifiers* (page 821). Database tables are discussed in The DatabaseTable Declaration of the Language Reference.

8.3.5 AllDefinedParameters

The predefined set *AllDefinedParameters* (page 819) contains the names of all defined parameters within an AIMMS model.

```
Set AllDefinedParameters {
    Subsetof   :  AllParameters;
    Index      :  IndexDefinedParameters;
}
```

Definition

The contents of the set *AllDefinedParameters* (page 819) is the collection of all parameters names with a non-empty `Definition` attribute within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting definitions of parameters declared in the **Model Explorer**.

See also:

The sets *AllParameters* (page 825). Parameters are discussed in Parameter Declaration and Attributes of the Language Reference.

8.3.6 AllDefinedSets

The predefined set *AllDefinedSets* (page 819) contains the names of all defined sets within an AIMMS model.

```
Set AllDefinedSets {
    SubsetOf   :  AllSets;
    Index      :  IndexDefinedSets;
}
```

Definition

The contents of the set *AllDefinedSets* (page 819) is the collection of all set names with a non-empty `Definition` attribute within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting definitions of sets declared in the **Model Explorer**.

See also:

The sets *AllSets* (page 827). Sets are discussed in Set Declaration and Attributes of the Language Reference.

8.3.7 AllFiles

The predefined set *AllFiles* (page 820) contains the names of all files declared within an AIMMS model.

```
Set AllFiles {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexFiles;
}
```

Definition

The contents of the set *AllFiles* (page 820) is the collection of all file identifiers defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting file identifiers in the **Model Explorer**.

Note: The set *AllFiles* (page 820) is the range of the element parameter *CurrentFile* (page 836).

See also:

The element parameter *CurrentFile* (page 836). Files are discussed in The File Declaration of the Language Reference.

8.3.8 AllFunctions

The predefined set *AllFunctions* (page 820) contains the names of all functions defined within an AIMMS model.

```
Set AllFunctions {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexFunctions;
}
```

Definition

The contents of the set *AllFunctions* (page 820) is the collection of all function names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting functions in the **Model Explorer**.

Note: Elements of the set *AllFunctions* (page 820) are typically used in conjunction with the APPLY statement, to allow data-driven evaluation of functional expressions.

See also:

The sets *AllIdentifiers* (page 821). Functions are discussed in Internal Functions of the Language Reference, the APPLY statement in The APPLY Operator.

8.3.9 AllGMPEvents

The predefined set *AllGMPEvents* (page 821) contains all GMP Events.

```
Set AllGMPEvents {
    SubsetOf     :  AllSolverSessionCompletionObjects;
    Index        :  IndexGMPEvents;
}
```

Definition

The set *AllGMPEvents* (page 821) contains all GMP events used by the functions GMP::Event::Create, GMP::Event::Delete, GMP::Event::Reset, and GMP::Event::Set.

See also:

The functions *GMP::Event::Create* (page 357), *GMP::Event::Delete* (page 357), *GMP::Event::Reset* (page 358), and *GMP::Event::Set* (page 358), and the predeclared identifier *AllSolverSessionCompletionObjects* (page 828).

8.3.10 AllIdentifiers

The predefined set *AllIdentifiers* (page 821) contains the names of all identifiers declared within an AIMMS model.

```
Set AllIdentifiers {
    SubsetOf   :  AllSymbols;
    Index      :  IndexIdentifiers, SecondIndexIdentifiers;
}
```

Definition

The contents of the set *AllIdentifiers* (page 821) is the collection of all identifier and section names declared within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting identifiers in the **Model Explorer**.

Note: Subsets of *AllIdentifiers* (page 821) are occassionally used in READ, WRITE or DISPLAY statements to indicate the set of identifiers to be read or written, as well as in data control statements such as EMPTY and CLEANUP. It also serves as the root set of the other (typed) identifier sets, which can be used throughout an AIMMS project.

See also:

The set *AllSymbols* (page 787). Data control statements are discussed in Data Control, the READ and WRITE statements in Syntax of the READ and WRITE Statements, and the DISPLAY statement in The DISPLAY Statement of the Language Reference. Working with the set *AllIdentifiers* (page 821) is described in more detail in Working with the Set AllIdentifiers.

8.3.11 AllIndices

The predefined set *AllIndices* (page 822) contains the names of all indices defined within an AIMMS model.

```
Set AllIndices {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexIndices;
}
```

Definition

The contents of the set *AllIndices* (page 822) is the collection of all indices defined within a particular model.

Updatability

The contents of the set can only be modified by adding indices to or deleting indices from sets within the **Model Explorer**.

See also:

The sets *AllSets* (page 827), *AllIdentifiers* (page 821). Sets and their corresponding indices are discussed in Set Declaration and Attributes of the Language Reference.

8.3.12 AllIntegerVariables

The predefined set *AllIntegerVariables* (page 823) contains the names of all integer variables within an AIMMS model.

```
Set AllIntegerVariables {
    SubsetOf   :  AllVariables;
    Index      :  IndexIntegerVariables;
}
```

Definition

The contents of the set *AllIntegerVariables* (page 823) is the collection of all *symbolic* variable names with as range a subset of `Integers` defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting integer variables in the **Model Explorer**.

See also:

The sets *AllVariables* (page 831), *Integers* (page 812).

8.3.13 AllMacros

The predefined set *AllMacros* (page 823) contains the names of all macros within an AIMMS model.

```
Set AllMacros {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexMacros;
}
```

Definition

The contents of the set *AllMacros* (page 823) is the collection of all *symbolic* macro names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting macros in the **Model Explorer**.

See also:

Macros are discussed in MACRO Declaration and Attributes of the Language Reference.

8.3.14 AllMathematicalPrograms

The predefined set `AllMathematicalprograms` contains the names of all mathematical programs within an AIMMS model.

```
Set AllMathematicalPrograms {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexMathematicalPrograms;
}
```

Definition

The contents of the set *AllMathematicalPrograms* (page 824) is the collection of all *symbolic* mathematical programs defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting mathematical in the **Model Explorer**.

See also:

- Mathematical programs in MathematicalProgram Declaration and Attributes of the Language Reference.
- The functions *GMP::Instance::Generate* (page 384), *GMP::Instance::GenerateStochasticProgram* (page 387), and *GMP::Instance::GetSymbolicMathematicalProgram* (page 406).

8.3.15 AllNonLinearConstraints

The predefined set *AllNonLinearConstraints* (page 824) contains the names of all non-linear constraints within an AIMMS model.

```
Set AllNonLinearConstraints {
    SubsetOf   :  AllConstraints;
    Index      :  IndexNonLinearConstraints;
}
```

Definition

The contents of the set *AllNonLinearConstraints* (page 824) is the collection of all *symbolic* non-linear constraint names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting non-linear constraints in the **Model Explorer**.

See also:

The set *AllConstraints* (page 817).

8.3.16 AllParameters

The predefined set *AllParameters* (page 825) contains the names of all parameters within an AIMMS model.

```
Set AllParameters {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexParameters;
}
```

Definition

The contents of the set *AllParameters* (page 825) is the collection of all *symbolic* parameter names declared within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting parameters in the **Model Explorer**.

Note: Subsets of *AllParameters* (page 825) are occassionally used in `READ`, `WRITE` or `DISPLAY` statements to indicate the set of parameters to be read or written, as well as in data control statements such as `EMPTY` and `CLEANUP`.

See also:

The sets *AllDefinedParameters* (page 819), *AllIdentifiers* (page 821). Data control statements are discussed in Data Control, the `READ` and `WRITE` statements in Syntax of the READ and WRITE Statements, and the `DISPLAY` statement in The DISPLAY Statement of the Language Reference.

8.3.17 AllProcedures

The predefined set *AllProcedures* (page 825) contains the names of all procedures defined within an AIMMS model.

```
Set AllProcedures {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexProcedures;
}
```

Definition

The contents of the set *AllProcedures* (page 825) is the collection of all procedure names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting procedures in the **Model Explorer**.

Note: Elements of the set *AllProcedures* (page 825) are typically used in conjunction with the APPLY statement, to allow data-driven procedural execution.

See also:

The sets *AllIdentifiers* (page 821). Procedures are discussed in Internal Procedures of the Language Reference, the APPLY statement in The APPLY Operator.

8.3.18 AllQuantities

The predefined set *AllQuantities* (page 826) contains the names of all quantities defined within an AIMMS model.

```
Set AllQuantities {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexQuantities;
}
```

Definition

The contents of the set *AllQuantities* (page 826) is the collection of all quantities defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting quantities in the **Model Explorer**.

See also:

The sets *AllIdentifiers* (page 821), *AllConventions* (page 817). Quantities are discussed in full detail in The Quantity Declaration of the Language Reference.

8.3.19 AllSections

The predefined set *AllSections* (page 827) contains the names of all sections within an AIMMS model.

```
Set AllSections {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexSections;
}
```

Definition

The contents of the set *AllSections* (page 827) is the collection of all section names defined within a particular model tree.

Updatability

The contents of the set can only be modified by adding or deleting sections in the **Model Explorer**.

Note: Section names contained in *AllSections* (page 827) are occassionally used in READ, WRITE or DISPLAY statements to indicate the set of identifiers to be read or written, as well as in data control statements such as EMPTY and CLEANUP.

See also:

The set *AllIdentifiers* (page 821). Model sections are discussed in Creating and Managing a Model. Data control statements are discussed in Section Data Control, the READ and WRITE statements in Syntax of the READ and WRITE Statements, and the DISPLAY statement in The DISPLAY Statement of the Language Reference.

8.3.20 AllSets

The predefined set *AllSets* (page 827) contains the names of all sets within an AIMMS model.

```
Set AllSets {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexSets;
}
```

Definition

The contents of the set *AllSets* (page 827) is the collection of all set names declared within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting sets in the **Model Explorer**.

Note: Subsets of *AllSets* (page 827) are occassionally used in READ, WRITE or DISPLAY statements to indicate the set of sets to be read or written, as well as in data control statements such as EMPTY, CLEANUP and CLEANUPDEPENDENTS.

See also:

The sets *AllDefinedSets* (page 819), *AllIdentifiers* (page 821). Data control statements are discussed in Data Control, the READ and WRITE statements in Syntax of the READ and WRITE Statements, and the DISPLAY statement in The DISPLAY Statement of the Language Reference.

8.3.21 AllSolverSessionCompletionObjects

The predefined set *AllSolverSessionCompletionObjects* (page 828) is the root set containing both *AllGMPEvents* (page 821) and *AllSolverSessions* (page 828).

```
Set AllSolverSessionCompletionObjects {
    Index        :  IndexSolverSessionCompletionObjects;
    Definition   :  AllGMPEvents + AllSolverSessions;
}
```

Definition

The set *AllExecutionStatuses* (page 798) is the root set containing both *AllGMPEvents* (page 821) and *AllSolverSessions* (page 828).

See also:

The predeclared identifiers *AllGMPEvents* (page 821) and *AllSolverSessions* (page 828).

8.3.22 AllSolverSessions

The predefined set *AllSolverSessions* (page 828) contains the names of all solver sessions associated with generated mathematical programs in your model.

```
Set AllSolverSessions {
    SubsetOf     :  AllSolverSessionCompletionObjects;
    Index        :  IndexSolverSessions;
}
```

Definition

The set *AllSolverSessions* (page 828) contains the names of all solver sessions associated with generated mathematical programs in your model. Solver sessions are created through the SOLVE statement, and the functions GMP::Instance::Solve and GMP::Instance::CreateSolverSession.

Updatability

The contents of *AllSolverSessions* (page 828) can only be modified programmatically through the SOLVE statement, and the functions GMP::Instance::Solve, GMP::Instance::CreateSolverSession and GMP::Instance::DeleteSolverSession.

See also:

The functions *GMP::Instance::Solve* (page 424), *GMP::Instance::CreateSolverSession* (page 378) and *GMP::Instance::DeleteSolverSession* (page 381), and the predeclared identifier *AllSolverSessionCompletionObjects* (page 828).

8.3.23 AllStochasticConstraints

The predefined set *AllStochasticConstraints* (page 829) contains the names of all constraints within an AIMMS which references in its definition a parameter or variable with the property Stochastic set.

```
Set AllStochasticConstraints {
    SubsetOf   :  AllConstraints;
    Index      :  IndexStochasticConstraints;
}
```

Definition

The contents of the set *AllStochasticConstraints* (page 829) is the collection of all constraints which reference a parameter or variable with the property Stochastic set within a particular model.

Updatability

The contents of the set can only be modified by setting or clearing the property Stochastic of the referenced variables and parameters in the definition of constraints declared in the **Model Explorer**.

See also:

- Stochastic programming is discussed in Stochastic Programming of the Language Reference.
- The intrinsic function *GMP::Instance::GenerateStochasticProgram* (page 387).
- The sets *AllConstraints* (page 817), *AllStochasticParameters* (page 829) and *AllStochasticVariables* (page 830).
- Constraints are discussed in Stochastic Programming of the Language Reference.

8.3.24 AllStochasticParameters

The predefined set *AllStochasticParameters* (page 829) contains the names of all parameters within an AIMMS model with the property Stochastic set.

```
Set AllStochasticParameters {
    SubsetOf   :  AllParameters;
    Index      :  IndexStochasticParameters;
}
```

Definition

The contents of the set *AllStochasticParameters* (page 829) is the collection of all parameters with the property Stochastic set within a particular model.

Updatability

The contents of the set can only be modified by setting or clearing the property Stochastic of parameters declared in the **Model Explorer**.

See also:

- Stochastic programming is discussed in Stochastic Programming of the Language Reference.
- The intrinsic function *GMP::Instance::GenerateStochasticProgram* (page 387).
- The sets *AllParameters* (page 825), *AllStochasticVariables* (page 830) and *AllStochasticConstraints* (page 829).
- Parameters are discussed in Parameter Declaration and Attributes of the Language Reference.

8.3.25 AllStochasticVariables

The predefined set *AllStochasticVariables* (page 830) contains the names of all variables within an AIMMS model with the property Stochastic set.

```
Set AllStochasticVariables {
    SubsetOf   :  AllVariables;
    Index      :  IndexStochasticVariables;
}
```

Definition

The contents of the set *AllStochasticVariables* (page 830) is the collection of all variables with the property Stochastic set within a particular model.

Updatability

The contents of the set can only be modified by setting or clearing the property `Stochastic` of variables declared in the **Model Explorer**.

See also:

- Stochastic programming is discussed in Stochastic Programming of the Language Reference.
- The intrinsic function *GMP::Instance::GenerateStochasticProgram* (page 387).
- The sets *AllVariables* (page 831), *AllStochasticParameters* (page 829) and *AllStochasticConstraints* (page 829).
- Variables are discussed in Stochastic Programming of the Language Reference.

8.3.26 AllUpdatableIdentifiers

The predefined set *AllUpdatableIdentifiers* (page 831) contains the names of the identifiers that are, in principle, updatable.

```
Set AllUpdatableIdentifiers {
    SubsetOf     :  AllIdentifiers;
    Index        :  IndexUpdatableIdentifiers;
    InitialData  :  {
        ( AllSets - AllDefinedSets ) +
        ( AllParameters - AllDefinedParameters )
    }
}
```

Definition

The set *AllUpdatableIdentifiers* (page 831) contains the names of the model identifiers that are, in principle, considered updatable by AIMMS.

Updatability

The contents of *AllUpdatableIdentifiers* (page 831) can be modified programmatically from within an AIMMS model. The set cannot be updated from within the end-user interface.

Note:

- The set *AllUpdatableIdentifiers* (page 831) determines which identifiers are updatable *in principle*. Which identifiers in *AllUpdatableIdentifiers* (page 831) can *actually* be modified within the graphical end-user interface is determined by the set *CurrentInputs* (page 837).
- By default, variables are considered not updatable by AIMMS. If you want to allow your end-users to update some or all variables from within the end-user interface, you can accomplish this by adding these variables to both the sets *AllUpdatableIdentifiers* (page 831) and *CurrentInputs* (page 837).

See also:

The sets *AllIdentifiers* (page 821), *CurrentInputs* (page 837).

8.3.27 AllVariables

The predefined set *AllVariables* (page 831) contains the names of all variables within an AIMMS model.

```
Set AllVariables {
    SubsetOf   :  AllVariablesConstraints;
    Index      :  IndexVariables;
}
```

Definition

The contents of the set *AllVariables* (page 831) is the collection of all *symbolic* variable names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting variables in the **Model Explorer**.

Note: The set *AllVariables* (page 831) or subsets thereof are typically used in the `Variables` attribute of `MathematicalPrograms` declared within an AIMMS model.

See also:

The sets *AllIdentifiers* (page 821), *AllConstraints* (page 817). Variables are discussed in Variable Declaration and Attributes, mathematical programs in MathematicalProgram Declaration and Attributes of the Language Reference.

8.3.28 AllVariablesConstraints

The predefined set *AllVariablesConstraints* (page 832) contains the names of all variables and constraints within an AIMMS model.

```
Set AllVariablesConstraints {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexVariablesConstraints;
}
```

Definition

The contents of the set *AllVariablesConstraints* (page 832) is the collection of all *symbolic* variable and constraint names defined within a particular model.

Updatability

The contents of the set can only be modified by adding or deleting variables and/or constraints in the **Model Explorer**.

Note: The set *AllVariablesConstraints* (page 832) or subsets thereof are typically used in the index domain of parameters entered in the ViolationPenalties attribute of a MathematicalProgram declared within an AIMMS model.

See also:

The sets *AllIdentifiers* (page 821), *AllVariables* (page 831), *AllConstraints* (page 817). The ViolationPenalties attribute of a mathematical programs is discussed in Infeasibility Analysis of the Language Reference.

8.4 Execution State Related Identifiers

The following collection of predefined identifiers contains information about the current state of the AIMMS execution engine.

8.4.1 AllGeneratedMathematicalPrograms

The predefined set *AllGeneratedMathematicalPrograms* (page 833) contains the names of all generated mathematical programs associated with the symbolic mathematical programs in an AIMMS model.

```
Set AllGeneratedMathematicalPrograms {
    Index       :  IndexGeneratedMathematicalPrograms;
    Parameter   :  CurrentGeneratedMathematicalProgram;
}
```

Definition

- The contents of the set *AllGeneratedMathematicalPrograms* (page 833) is the collection of all generated mathematical programs associated with symbolic mathematical programs in your model, and generated through the SOLVE statement, or the functions GMP::Instance::Generate and GMP::Instance::CreateDual.
- The element parameter CurrentGeneratedMathematicalProgram refers to the currently active generated mathematical program instance.

Updatability

The contents of the set can only be modified through the SOLVE statement, and the functions GMP::Instance::Generate, GMP::Instance::Copy, GMP::Instance::Rename, GMP::Instance::Delete and GMP::Instance::CreateDual.

See also:

The function *GMP::Instance::Generate* (page 384), *GMP::Instance::Copy* (page 364), *GMP::Instance::Rename* (page 408), *GMP::Instance::Delete* (page 378) and *GMP::Instance::CreateDual* (page 370).

8.4.2 AllProgressCategories

The predefined set *AllProgressCategories* (page 834) contains the names of all created progress categories.

```
Set AllProgressCategories {
    Index       :  IndexProgressCategories;
}
```

Definition

The contents of the set *AllProgressCategories* (page 834) is the collection of all progress categories created by the functions *GMP::Instance::CreateProgressCategory* (page 377) and *GMP::SolverSession::CreateProgressCategory* (page 529). These progress categories are used by the GMP::ProgressWindow functions.

Updatability

The contents of the set can only be modified through the functions *GMP::Instance::CreateProgressCategory* (page 377), *GMP::SolverSession::CreateProgressCategory* (page 529) and *GMP::ProgressWindow::DeleteCategory* (page 436).

8.4.3 AllStochasticScenarios

The predefined set *AllStochasticScenarios* (page 834) contains the names of all stochastic scenarios.

```
Set AllStochasticScenarios {
    Index       :  IndexStochasticScenarios;
}
```

Definition

The contents of the set *AllStochasticScenarios* (page 834) is the collection of all stochastic scenarios.

Updatability

The contents of the set can be modified in the model.

See also:

- Stochastic programming is discussed in Stochastic Programming of the Language Reference.
- The intrinsic function *GMP::Instance::GenerateStochasticProgram* (page 387).

8.4.4 CurrentAutoUpdatedDefinitions

The predefined set *CurrentAutoUpdatedDefinitions* (page 835) contains the names of the defined identifiers whose values are updated automatically upon change of their input values when displayed in the graphical end-user interface.

```
Set CurrentAutoUpdatedDefinitions {
    SubsetOf     :  AllIdentifiers;
    Index        :  IndexCurrentAutoUpdatedDefinitions;
    InitialData  :  AllDefinedSets + AllDefinedParameters;
}
```

Definition

The set *CurrentAutoUpdatedDefinitions* (page 835) contains the names of the defined identifiers whose values are updated automatically upon change of their input values when displayed in the graphical end-user interface.

Updatability

The contents of *CurrentAutoUpdatedDefinitions* (page 835) can be modified programmatically from within an AIMMS model. The set cannot be modified from within the end-user interface.

Note: By default, all defined parameters and sets are immediately updated in a graphical display whenever their input values are modified. In some cases, however, this behavior can be unwanted, for instance if each single data change by an end-user leads to a long re-evaluation of a defined identifier which is also displayed on the same page. In such cases, you can remove the defined identifier at hand from the set *CurrentAutoUpdatedDefinitions* (page 835) and explicitly update the identifier when you see fit, either by calling the UPDATE statement, or by updating the identifier on page entry, upon data change, or through a button action.

See also:

The sets *AllIdentifiers* (page 821), *CurrentInputs* (page 837). The UPDATE statement and the set *CurrentAutoUpdatedDefinitions* (page 835) are discussed in more detail in Nonprocedural Execution of the Language Reference.

8.4.5 CurrentErrorMessage

The predefined string parameter *CurrentErrorMessage* (page 835) contains a description of the last runtime error that occurred during the execution of an AIMMS model.

```
StringParameter CurrentErrorMessage;
```

Definition

The string parameter *CurrentErrorMessage* (page 835) contains a description of the last runtime error that occurred during the execution of an AIMMS model. It also contains the error message associated with errors occurring in AIMMS interface functions.

Updatability

The value of *CurrentErrorMessage* (page 835) can be modified programmatically from within an AIMMS model. Its value cannot be modified from within the end-user interface.

Note:

- AIMMS never clears the contents *CurrentErrorMessage* (page 835), but only updates its value whenever an error occurs.
- When AIMMS is called through the AIMMS API, *CurrentErrorMessage* (page 835) is the only way to retrieve a description of the last AIMMS runtime error when an execution request failed.

See also:

Error handling in the AIMMS API is discussed in more detail in Passing Errors and Messages of the Language Reference.

8.4.6 CurrentFile

The predefined element parameter *CurrentFile* (page 836) contains the name of the file identifier to which output is currently directed.

```
ElementParameter CurrentFile {
    Range        :  AllFiles;
}
```

Definition

The element parameter *CurrentFile* (page 836) contains the name of the file identifier to which output from the PUT and DISPLAY statements is currently directed.

Updatability

The value of *CurrentFile* (page 836) can be modified both programmatically from within the AIMMS model and from within the end-user interface. As a result, the output from subsequent PUT and DISPLAY statements will be redirected to the newly specified file identifier.

Note: Output redirection can equivalently be accomplished using the PUT statement. The name of the physical file or window associated with a file identifier can be retrieved through the string parameter *CurrentFileName* (page 837).

See also:

The string parameter *CurrentFileName* (page 837). The PUT statement is discussed in The PUT Statement of the Language Reference, the DISPLAY statement in The DISPLAY Statement

8.4.7 CurrentFileName

The predefined string parameter *CurrentFileName* (page 837) contains the file name associated with the file identifier to which output is currently directed.

```
StringParamter CurrentFileName;
```

Definition

The string parameter *CurrentFileName* (page 837) contains the file name associated with the file identifier (as specified in its Name attribute) to which output from the PUT and DISPLAY statements is currently directed.

Updatability

The value of *CurrentFileName* (page 837) is only for display purposes. It can be modified programmatically from within the AIMMS model, but the output from PUT and DISPLAY will always be sent to the file or window whose name is specified in the Name attribute of the corresponding file identifier.

Note: The physical file name associated with a file identifier can be changed dynamically, by entering a string parameter in the Name attribute of the file identifier. The file identifier to which output is currently directed can be retrieved through the element parameter *CurrentFile* (page 836).

See also:

The element parameter *CurrentFile* (page 836). File identifiers are discussed in The File Declaration of the Language Reference.

8.4.8 CurrentInputs

The predefined set *CurrentInputs* (page 837) contains the names of the identifiers which can actually be modified from within the graphical end-user interface.

```
Set CurrentInputs {
    SubsetOf     :  AllUpdatableIdentifiers;
    Index        :  IndexCurrentInputs;
    InitialData  :  AllUpdatableIdentifiers;
}
```

Definition

The set *CurrentInputs* (page 837) contains the names of the model identifiers that can actually modified from within the graphical end-user interface of AIMMS.

Updatability

The contents of *CurrentInputs* (page 837) can be modified programmatically from within an AIMMS model. The set cannot be updated from within the end-user interface.

Note:

- The set *AllUpdatableIdentifiers* (page 831) determines which identifiers are updatable *in principle*. Therefore, you can only add identifiers to *CurrentInputs* (page 837) which are already contained in the set *AllUpdatableIdentifiers* (page 831)
- By default, variables are considered not updatable by AIMMS, and cannot be modified from within the end-user interface. If you want to allow your end-users to update some or all variables from within the end-user interface, you can accomplish this by adding these variables to both the sets *AllUpdatableIdentifiers* (page 831) and *CurrentInputs* (page 837).
- Please be careful when changing the content of this set, because it has a side-effect which may be overlooked easily. For example, when executing the following statement:

  ```
  CurrentInputs := 'MyIdentifier';
  ```

  you are not only assigning your identifier to the set, **but also totally replacing the previous content of the set!** In order to prevent this, you should use the following statement instead of the one above:

  ```
  CurrentInputs := CurrentInputs - 'Main_My_Model' + 'MyIdentifier'
  ```

  (if your model is called 'My Model')

See also:

The sets *AllIdentifiers* (page 821), *CurrentInputs* (page 837).

8.4.9 CurrentMatrixBlockSizes

The predefined parameter *CurrentMatrixBlockSizes* (page 838) contains the number of non-zeros for the last mathematical program generated.

```
Parameter CurrentMatrixBlockSizes {
    IndexDomain  :  (IndexConstraints, IndexVariables);
}
```

Definition

The parameter *CurrentMatrixBlockSizes* (page 838) contains the number of non-zeros for the last mathematical program generated. The parameter counts the non-zeros in all generated rows of a particular *symbolic* constraint with respect to all generated columns of a particular *symbolic* variable.

Note:

- You can use the parameter *CurrentMatrixBlockSizes* (page 838), for example, to analyze which constraint-variable sub-block of the generated matrix accounts for a number of non-zeros in a mathematical program that appears to be unnaturally high.
- The parameters *CurrentMatrixRowCount* (page 840), *CurrentMatrixColumnCount* (page 839) and *CurrentMatrixBlockSizes* (page 838) are only set when the AIMMS option **Solvers General - Matrix Generation - Matrix_Block_Sizes** is set to on.

See also:

The sets *CurrentMatrixColumnCount* (page 839), *CurrentMatrixRowCount* (page 840).

8.4.10 CurrentMatrixColumnCount

The predefined parameter *CurrentMatrixColumnCount* (page 839) contains the number of columns for the last mathematical program generated.

```
Parameter CurrentMatrixColumnCount {
    IndexDomain  :  IndexVariables;
}
```

Definition

The parameter *CurrentMatrixColumnCount* (page 839) contains the number of columns for the last mathematical program generated. The parameter counts the columns generated for each individual *symbolic* variable.

Note:

- You can use the parameter *CurrentMatrixColumnCount* (page 839), for example, to analyze which symbolic variable accounts for a number of columns in a mathematical program that appears to be unnaturally high.

- The parameters *CurrentMatrixRowCount* (page 840), *CurrentMatrixColumnCount* (page 839) and *CurrentMatrixBlockSizes* (page 838) are only set when the AIMMS option **Solvers General - Matrix Generation - Matrix_Block_Sizes** is set to on.

See also:

The sets *CurrentMatrixRowCount* (page 840), *CurrentMatrixBlockSizes* (page 838).

8.4.11 CurrentMatrixRowCount

The predefined parameter *CurrentMatrixRowCount* (page 840) contains the number of rows for the last mathematical program generated.

```
Parameter CurrentMatrixRowCount {
    IndexDomain  :  IndexConstraints;
}
```

Definition

The parameter *CurrentMatrixRowCount* (page 840) contains the number of rows for the last mathematical program generated. The parameter counts the rows generated for each individual *symbolic* constraint.

Note:

- You can use the parameter *CurrentMatrixRowCount* (page 840), for example, to analyze which symbolic constraint accounts for a number of rows in a mathematical program that appears to be unnaturally high.
- The parameters *CurrentMatrixRowCount* (page 840), *CurrentMatrixColumnCount* (page 839) and *CurrentMatrixBlockSizes* (page 838) are only set when the AIMMS option **Solvers General - Matrix Generation - Matrix_Block_Sizes** is set to on.

See also:

The sets *CurrentMatrixColumnCount* (page 839), *CurrentMatrixBlockSizes* (page 838).

8.4.12 CurrentPageNumber

The predefined parameter *CurrentPageNumber* (page 840) contains current page number used by AIMMS when printing print pages.

```
Parameter CurrentPageNumber;
```

Definition

The predefined parameter *CurrentPageNumber* (page 840) contains current page number used by AIMMS when printing print pages.

Updatability

AIMMS will automatically reset the value *CurrentPageNumber* (page 840) to 1 at the following times:

- before printing a print page using the **File-Print** menu,
- before printing a print page using the *PrintPage* (page 711) function outside of a pair of calls to the functions *PrintStartReport* (page 713) and *PrintEndReport* (page 710), and
- just after calling the function *PrintStartReport* (page 713).

The value of *CurrentPageNumber* (page 840) can be modified programmatically from within the AIMMS model.

Note: According to the list of rules above, modifying the value of *CurrentPageNumber* (page 840) will only have an effect of the page numbers printed on print pages within a pair of calls to *PrintStartReport* (page 713) and *PrintEndReport* (page 710).

See also:

The functions *PrintPage* (page 711), *PrintStartReport* (page 713), *PrintEndReport* (page 710). Print pages are discussed in Print configuration

8.4.13 ODBCDateTimeFormat

The predefined string parameter *ODBCDateTimeFormat* (page 841) defines, for each identifier within an AIMMS model, the date-time conversion string.

```
StringParameter ODBCDateTimeFormat {
    IndexDomain  :  IndexIdentifiers;
}
```

Definition

The string parameter *ODBCDateTimeFormat* (page 841) defines, for each identifier within an AIMMS model, the date-time format string, which AIMMS will use in converting AIMMS data to date-time columns in a database table and vice versa.

Updatability

The data of *ODBCDateTimeFormat* (page 841) can be modified both from within the model and the end-user interface.

Note: The use of *ODBCDateTimeFormat* (page 841) to convert AIMMS data to date-time columns and vice versa, are not necessary for columns which are mapped onto AIMMS calendars. In that case, AIMMS is able to determine the conversion itself based on the timeslot format specified for the calendar.

See also:

The use of *ODBCDateTimeFormat* (page 841) is discussed in more detail in Dealing with Date-Time Values of the Language Reference. The format to which values of *ODBCDateTimeFormat* (page 841) should comply are discussed in Format of Time Slots and Periods

8.5 Case Management Related Identifiers

You can setup the case management of your AIMMS project either to use a single data manager file with cases and datasets, or to use separate folders and case files on disk. Both styles of case management have their own collection of predefined identifiers.

The following collection of predefined identifiers contains data regarding the case types, data categories, cases and datasets associated with a particular AIMMS project, that uses the style `Single_Data_Manager_file`:

- *AllCases* (page 842)
- *AllCaseTypes* (page 843)
- *AllDataCategories* (page 844)
- *AllDataFiles* (page 845)
- *AllDataSets* (page 845)
- *CurrentCase* (page 846)
- *CurrentCaseSelection* (page 847)
- *CurrentDataSet* (page 847)
- *CurrentDefaultCaseType* (page 849)

The following collection of predefined identifiers contains data regarding the case files and types of case files associated with a particular AIMMS project, that uses the style `Disk_files_and_folders`:

- *AllCases* (page 842)
- *CurrentCase* (page 846)
- *CurrentCaseSelection* (page 847)
- *CurrentCaseFileContentType* (page 849)
- *AllCaseFileContentTypes* (page 848)
- *CaseFileURL* (page 850)

8.5.1 AllCases

The predefined set *AllCases* (page 842) contains the references to all cases that are currently available in the AIMMS project.

```
Set AllCases {
    SubsetOf   :  AllDataFiles;
    Index      :  IndexCases;
}
```

Definition

The set *AllCases* (page 842) is used in both data management styles `Single_Data_Manager_file` and `Disk_files_and_folders`. When using `Single_Data_Manager_file`, the contents of the set *AllCases* (page 842) is the collection of (integer) references to all cases stored within the data manager file currently loaded within an AIMMS project. When using `Disk_files_and_folders`, the contents of the set *AllCases* (page 842) is the collection of (integer) references to all case files that have been referenced thus far. Each newly opened or saved case file is automatically added to this set.

Updatability

The contents of the set can only be modified implicitly by using the various features of the Data Management tool, by executing any of the Data menu commands or by using the specific case or dataset functions.

Note: If the data management style is set to `Single_Data_Manager_file`.

- Further information about the integer case references can be obtained through the functions *DataFileGetAcronym* (page 949), *DataFileGetDescription* (page 951), *DataFileGetGroup* (page 951), *DataFileGetName* (page 952), *DataFileGetOwner* (page 953), *DataFileGetPath* (page 953) and *DataFileGetTime* (page 954).
- The integer case references stored in the set *AllCases* (page 842) are only guaranteed to be unique within a single AIMMS session, and, furthermore, only within the context of a single data manager file associated with a project. As a consequence, additional case information retrieved through the functions listed above must be refreshed after opening another data manager file.

If the data management style is set to `Disk_files_and_folders`.

- The corresponding location on disk of any element in the set *AllCases* (page 842) can be obtained through the predeclared identifier *CaseFileURL* (page 850).
- The integer case references stored in the set *AllCases* (page 842) are only guaranteed to be unique within a single AIMMS session and depend on the order in which case files are accessed.

See also:

The set *AllDataFiles* (page 845). Accessing cases from within an AIMMS model is discussed in full detail in Managing multiple case selections.

8.5.2 AllCaseTypes

The predefined set *AllCaseTypes* (page 843) contains the names of all case types declared within an AIMMS project.

```
Set AllCaseTypes {
    Index       :  IndexCaseTypes;
}
```

Definition

The contents of the set *AllCaseTypes* (page 843) is the collection of all case types defined within the **Data Management Setup** tool of a project.

Updatability

The contents of the set can only be modified by adding or deleting case types in the **Data Management Setup** tool.

Note:

- The function *CaseGetType* (page 920) returns the case type of a case as an element of the set *AllCaseTypes* (page 843). The identifiers and data categories associated with a case type can be obtained through the functions *CaseTypeContents* (page 946) and *CaseTypeCategories* (page 946). The default case type of a case when saving it is set through the predefined element parameter *CurrentDefaultCaseType* (page 849).
- This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

8.5.3 AllDataCategories

The predefined set *AllDataCategories* (page 844) contains the names of all data categories declared within an AIMMS project.

```
Set AllDataCategories {
    Index       :  IndexDataCategories;
}
```

Definition

The contents of the set *AllDataCategories* (page 844) is the collection of all data categories defined within the **Data Management Setup** tool of a project.

Updatability

The contents of the set can only be modified by adding or deleting data categories in the **Data Management Setup** tool.

Note:

- The function *DatasetGetCategory* (page 935) returns the data category of a dataset as an element of the set *AllDataCategories* (page 844). The identifiers associated with a data category can be obtained through the function *DataCategoryContents* (page 947).
- This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

8.5.4 AllDataFiles

The predefined set *AllDataFiles* (page 845) contains the references to all data files stored in the data manager file currently loaded within an AIMMS project.

```
Set AllDataFiles {
    Index      :  IndexDataFiles;
    Definition :  AllCases + AllDataSets;
}
```

Definition

The contents of the set *AllDataFiles* (page 845) is the collection of (integer) references to all data files (i.e. cases and datasets) stored within the data manager file currently loaded within an AIMMS project.

Updatability

The contents of the set can only be modified by adding or deleting cases and dataset in the **Data Manager** or through the **Data** menu, or using various case and dataset interface functions.

Note:

- Elements of the set *AllDataFiles* (page 845) are more commonly referenced through its subsets *AllCases* (page 842) and *AllDataSets* (page 845).
- Further information about the integer data file references can be obtained through the functions *DataFileGetAcronym* (page 949), *DataFileGetDescription* (page 951), *DataFileGetGroup* (page 951), *DataFileGetName* (page 952), *DataFileGetOwner* (page 953), *DataFileGetPath* (page 953) and *DataFileGetTime* (page 954).
- The integer data file references stored in the set *AllDataFiles* (page 845) are only guaranteed to be unique within a single AIMMS session, and, furthermore, only within the context of a single data manager file associated with a project. As a consequence, additional data file information retrieved through the functions listed above must be refreshed after opening another data manager file.

See also:

The sets *AllCases* (page 842), *AllDataSets* (page 845).

8.5.5 AllDataSets

The predefined set *AllDataSets* (page 845) contains the references to all datasets stored in the data manager file currently loaded within an AIMMS project.

```
Set AllDataSets {
    SubsetOf   :  AllDataFiles;
    Index      :  IndexDataSets;
}
```

Definition

The contents of the set *AllDataSets* (page 845) is the collection of (integer) references to all datasets stored within the data manager file currently loaded within an AIMMS project.

Updatability

The contents of the set can only be modified by adding or deleting datasets in the **Data Manager**, by saving cases in the **Data** menu, or through the functions *DatasetCreate* (page 933), *DatasetDelete* (page 933) and *DatasetSaveAs* (page 942).

Note:

- Further information about the integer dataset references can be obtained through the functions *DataFileGetAcronym* (page 949), *DataFileGetDescription* (page 951), *DataFileGetGroup* (page 951), *DataFileGetName* (page 952), *DataFileGetOwner* (page 953), *DataFileGetPath* (page 953) and *DataFileGetTime* (page 954).
- The integer dataset references stored in the set *AllDataSets* (page 845) are only guaranteed to be unique within a single AIMMS session, and, furthermore, only within the context of a single data manager file associated with a project. As a consequence, additional case information retrieved through the functions listed above must be refreshed after opening another data manager file.
- This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

8.5.6 CurrentCase

The predefined element parameter *CurrentCase* (page 846) contains a reference to the currently active case within an AIMMS project.

```
ElementParameter CurrentCase {
    Range      :  AllCases;
}
```

Definition

The element parameter *CurrentCase* (page 846) contains an (integer) case reference (as an element of *AllCases* (page 842)) to the currently active case within an AIMMS project, or is empty if the active case is not named.

Updatability

The element parameter *CurrentCase* (page 846) is used in both data management styles `Single_Data_Manager_file` and `Disk_files_and_folders`. When using `Single_Data_Manager_file`, the value of *CurrentCase* (page 846) can only be modified by actively loading another case either in the **Data Manager**, through the **Data** menu, or using the functions *CaseLoadCurrent* (page 921) and *CaseSaveAs* (page 927). When using `Disk_files_and_folders`, the value of *CurrentCase* (page 846) can only be modified by actively loading or saving a case through the **Data** menu, or by using the functions *CaseFileSetCurrent* (page 635), *CaseCommandLoadAsActive* (page 634), `CaseComandSave`, `CaseComandSaveAs` or *CaseCommandNew* (page 638).

See also:

The set *AllCases* (page 842), the element parameter *CurrentDataSet* (page 847). Loading and saving cases is discussed in full detail in Working with Cases.

8.5.7 CurrentCaseSelection

The predefined set *CurrentCaseSelection* (page 847) contains the current multiple case selection within an AIMMS project.

```
Set CurrentCaseSelection {
    SubsetOf   :  AllCases;
    Index      :  IndexCurrentCaseSelection;
}
```

Definition

The contents of the set *CurrentCaseSelection* (page 847) is the collection of (integer) case references (as elements of *AllCases* (page 842)) that is currently part of the multiple case selection.

Updatability

The contents of the set can be modified through the **Data-Multiple Cases** menu, by calling the function *CaseSelectMultiple* (page 928), or programmatically through a direct assignment within the model.

See also:

The set *AllCases* (page 842). Working with multiple cases is discussed in full detail in Managing multiple case selections.

8.5.8 CurrentDataSet

The predefined element parameter *CurrentDataSet* (page 847) contains a reference to the current actively loaded dataset(s) within an AIMMS project.

```
ElementParameter CurrentDataSet {
    IndexDomain  :  IndexDataCategories;
    Range        :  AllDataSets;
}
```

Definition

The element parameter *CurrentDataSet* (page 847) contains, for every data category, an (integer) dataset reference (as an element of *AllDataSets* (page 845)) to the current actively loaded dataset within the active case, or is empty if there no named dataset loaded as active for a particular data category.

Updatability

The value of the element parameter *CurrentDataSet* (page 847) can only be modified by actively actively loading datasets into the active case either in the **Data Manager**, through the **Data** menu, or using the functions *DatasetLoadCurrent* (page 936) and *DatasetSaveAs* (page 942).

Note: This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

8.5.9 AllCaseFileContentTypes

The predefined set *AllCaseFileContentTypes* (page 848) contains the references to all case file content types that can be used within a particular AIMMS project.

```
Set AllCaseFileContentTypes {
    SubsetOf   :  AllSubsetsOfAllIdentifiers;
    Index      :  IndexCaseFileContentTypes;
}
```

Definition

An element in the set *AllCaseFileContentTypes* (page 848) is a subset of *AllIdentifiers* (page 821). Such a subset defines the identifiers that are stored in a case file.

Updatability

The contents of this set can be freely modified. By default, it only contains the element *AllIdentifiers* (page 821). If your project uses multiple types of case files with different content, you should replace the default content of this set with all content types applicable to your project.

Note:

- This predeclared identifier is only relevant if the project option `Data_Management_style` is set to `Disk_files_and_folders`.
- If this set contains more than one element, the dialog box for saving a case file will show an additional drop down box, in which the user can select the case content type to be used for saving.

See also:

The set `AllSubsetsOfAllIdentifiers`.

8.5.10 CurrentCaseFileContentType

The predefined element parameter *CurrentCaseFileContentType* (page 849) contains the references to a case file content, corresponding to the most recently loaded or saved case file.

```
ElementParameter CurrentCaseFileContentType {
    Range        :  AllCaseFileContentTypes;
}
```

Definition

The value of *CurrentCaseFileContentType* (page 849) is a references to a subset of *AllIdentifiers* (page 821), which corresponds to the data that is stored in the most recently loaded or saved case file. This subset is also used to determine whether any data needs to be saved for the current case, before loading another case file.

Updatability

The value of the element parameter can be freely modified. The standard case management functionality updates the value itself whenever a case file is loaded or saved.

Note:

- This predeclared identifier is only relevant if the project option `Data_Management_style` is set to `Disk_files_and_folders`.

See also:

The set *AllCaseFileContentTypes* (page 848).

8.5.11 CurrentDefaultCaseType

The predefined element parameter *AllCaseTypes* (page 843) contains the name of the current default case type.

```
ElementParameter CurrentDefaultCaseType {
    Range      :  AllCaseTypes;
}
```

Definition

The value of the element parameter *CurrentDefaultCaseType* (page 849), if non-empty, restricts the selection of visible cases to the cases of the specified case type in the **Load Case** dialog box. In addition, a non-empty value of *CurrentDefaultCaseType* (page 849) presets the case type to the specified case type in the **Save Case** dialog box, and removes the end-user's capability to modify the case type interactively.

Updatability

The value of the element parameter can be modified both in the model and in the graphical end-user interface.

Note: This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

8.5.12 CaseFileURL

The string parameter *CaseFileURL* (page 850) holds the url (i.e. the full path name) of the file that corresponds to each element in *AllCases* (page 842).

```
StringParameter CaseFileURL {
    IndexDomain  :  AllCases;
}
```

Definition

The contents of the set *AllCases* (page 842) is the collection of (integer) references to all case files that have been loaded or saved during a specific session of your AIMMS project. The string parameter *CaseFileURL* (page 850) helps you to get the location of each of these cases.

Updatability

The contents of the set *AllCases* (page 842) as well as their corresponding values in *CaseFileURL* (page 850) are maintained by AIMMS itself and cannot be modified directly. They are modified when you load or save cases, or through the function *CaseFileURLtoElement* (page 636).

Note:

- This predeclared identifier is only relevant if the project option `Data_Management_style` is set to `Disk_files_and_folders`.

- The integer case references stored in the set *AllCases* (page 842) are only guaranteed to be unique within a single AIMMS session.

See also:

The set *AllCases* (page 842) and the function *CaseFileURLtoElement* (page 636).

8.6 Date-Time Related Identifiers

The following collection of predefined identifiers contains data used in representing

8.6.1 AllAbbrMonths

The predefined set *AllAbbrMonths* (page 851) contains the abbreviated English names of all months.

```
Set AllAbbrMonths {
    Index      :  IndexAbbrMonths;
    Definition :  {
        data { Jan, Feb, Mar, Apr, May, Jun,
               Jul, Aug, Sep, Oct, Nov, Dec }
    }
}
```

Definition

The set *AllAbbrMonths* (page 851) contains the abbreviated English names of all months.

Updatability

The contents of the set cannot be modified.

Note: The set *AllAbbrMonths* (page 851) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`.

See also:

The sets *AllMonths* (page 852), *LocaleAllAbbrMonths* (page 854), *LocaleAllMonths* (page 855). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.2 AllAbbrWeekdays

The predefined set *AllAbbrWeekdays* (page 851) contains the abbreviated English names of all weekdays.

```
Set AllAbbrWeekdays {
    Index      :  IndexAbbrWeekdays;
    Definition :  data { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
}
```

Definition

The set *AllAbbrWeekdays* (page 851) contains the abbreviated English names of all weekdays.

Updatability

The contents of the set cannot be modified.

Note: The set *AllAbbrWeekdays* (page 851) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`.

See also:

The sets *AllWeekdays* (page 853), *LocaleAllAbbrWeekdays* (page 855), *LocaleAllWeekdays* (page 856). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.3 AllMonths

The predefined set *AllMonths* (page 852) contains the unabbreviated English names of all months.

```
Set AllMonths {
    Index      :  IndexMonths;
    Definition :  {
        data { January, February, March, April,
               May, June, July, August,
               September, October, November, December }
    }
}
```

Definition

The set *AllMonths* (page 852) contains the unabbreviated English names of all months.

Updatability

The contents of the set cannot be modified.

Note: The set *AllMonths* (page 852) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`.

See also:

The sets *AllAbbrMonths* (page 851), *LocaleAllMonths* (page 855), *LocaleAllMonths* (page 855). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.4 AllTimeZones

The predefined set *AllTimeZones* (page 853) contains the set of all available time zones.

```
Set AllTimeZones {
    Index      :  IndexTimeZones;
}
```

Definition

The set *AllTimeZones* (page 853) contains the set of all time zones as defined by the operating system, plus a number of predefined time zones.

Updatability

The contents of the set cannot be modified.

Note: The set *AllTimeZones* (page 853) can be used in the `\%TZ` specifier of a time slot or period format. Such time zone specifications can be used, for instance, in the `TimeslotFormat` attribute of a `Calendar`.

See also:

Calendars are discussed in full detail in Calendars of the Language Reference, the time zone specific part of a date-time format in Support for Time Zones and Daylight Saving Time.

8.6.5 AllWeekdays

The predefined set *AllWeekdays* (page 853) contains the unabbreviated English names of all weekdays.

```
Set AllWeekdays {
    Index      :  IndexWeekdays;
    Definition :  {
        data { Monday, Tuesday, Wednesday, Thursday,
               Friday, Saturday, Sunday }
    }
}
```

Definition

The set *AllWeekdays* (page 853) contains the unabbreviated English names of all weekdays.

Updatability

The contents of the set cannot be modified.

Note: The set *AllWeekdays* (page 853) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`.

See also:

The sets *AllAbbrWeekdays* (page 851), *LocaleAllWeekdays* (page 856), *LocaleAllWeekdays* (page 856). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.6 LocaleAllAbbrMonths

The predefined set *LocaleAllAbbrMonths* (page 854) contains the abbreviated names of all months according the current system locale.

```
Set LocaleAllAbbrMonths {
    Index      :  LocaleIndexAbbrMonths;
}
```

Definition

The set *LocaleAllAbbrMonths* (page 854) contains the abbreviated names of all months according to the current system locale.

Updatability

During system startup, the set *LocaleAllAbbrMonths* (page 854) is filled with the set of abbreviated month names according to the current system locale. The contents of the set cannot be modified.

Note: The set *LocaleAllAbbrMonths* (page 854) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The sets *AllAbbrMonths* (page 851), *AllMonths* (page 852), *LocaleAllMonths* (page 855). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.7 LocaleAllAbbrWeekdays

The predefined set *LocaleAllAbbrWeekdays* (page 855) contains the abbreviated names of all weekdays according the current system locale.

```
Set LocaleAllAbbrWeekdays {
    Index      :  LocaleIndexAbbrWeekdays;
}
```

Definition

The set *LocaleAllAbbrWeekdays* (page 855) contains the abbreviated names of all weekdays according to the current system locale.

Updatability

During system startup, the set *LocaleAllAbbrWeekdays* (page 855) is filled with the set of abbreviated weekday names according to the current system locale. The contents of the set cannot be modified.

Note: The set *LocaleAllAbbrWeekdays* (page 855) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The sets *AllAbbrWeekdays* (page 851), *AllWeekdays* (page 853), *LocaleAllWeekdays* (page 856). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.8 LocaleAllMonths

The predefined set *LocaleAllMonths* (page 855) contains the unabbreviated names of all months according the current system locale.

```
Set LocaleAllMonths {
    Index      :  LocaleIndexMonths;
}
```

Definition

The set *LocaleAllMonths* (page 855) contains the unabbreviated names of all months according to the current system locale.

Updatability

During system startup, the set *LocaleAllMonths* (page 855) is filled with the set of unabbreviated month names according to the current system locale. The contents of the set cannot be modified.

Note: The set *LocaleAllMonths* (page 855) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The sets *AllAbbrMonths* (page 851), *AllMonths* (page 852), *LocaleAllAbbrMonths* (page 854). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.9 LocaleAllWeekdays

The predefined set *LocaleAllWeekdays* (page 856) contains the unabbreviated names of all weekdays according the current system locale.

```
Set LocaleAllWeekdays {
    Index      :  LocaleIndexWeekdays;
}
```

Definition

The set *LocaleAllWeekdays* (page 856) contains the unabbreviated names of all weekdays according to the current system locale.

Updatability

During system startup, the set *LocaleAllWeekdays* (page 856) is filled with the set of unabbreviated weekday names according to the current system locale. The contents of the set cannot be modified.

Note: The set *LocaleAllWeekdays* (page 856) can be used to construct a date-time format specification as specified in Format of Time Slots and Periods Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The sets *AllAbbrWeekdays* (page 851), *AllWeekdays* (page 853), *LocaleAllAbbrWeekdays* (page 855). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.10 LocaleLongDateFormat

The predefined string parameter *LocaleLongDateFormat* (page 857) contains the AIMMS date-time format equivalent with the long date format as specified in the current system locale.

```
StringParameter LocaleLongDateFormat;
```

Definition

The string parameter *LocaleLongDateFormat* (page 857) contains the AIMMS date-time format equivalent with the long date format as specified in the current system locale.

Updatability

During system startup, the string parameter *LocaleLongDateFormat* (page 857) is computed on the basis of the information in the current system locale. The contents of the string parameter cannot be modified.

Note: The string parameter *LocaleLongDateFormat* (page 857) can be used, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The string parameters *LocaleShortDateFormat* (page 857), *LocaleTimeFormat* (page 858). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.11 LocaleShortDateFormat

The predefined string parameter *LocaleShortDateFormat* (page 857) contains the AIMMS date-time format equivalent with the short date format as specified in the current system locale.

```
StringParameter LocaleShortDateFormat;
```

Definition

The string parameter *LocaleShortDateFormat* (page 857) contains the AIMMS date-time format equivalent with the short date format as specified in the current system locale.

Updatability

During system startup, the string parameter *LocaleShortDateFormat* (page 857) is computed on the basis of the information in the current system locale. The contents of the string parameter cannot be modified.

Note: The string parameter *LocaleShortDateFormat* (page 857) can be used, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The string parameters *LocaleLongDateFormat* (page 857), *LocaleTimeFormat* (page 858). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.12 LocaleTimeFormat

The predefined string parameter *LocaleTimeFormat* (page 858) contains the AIMMS date-time format equivalent with the time format specified in the current system locale.

```
StringParameter LocaleTimeFormat;
```

Definition

The string parameter *LocaleTimeFormat* (page 858) contains the AIMMS date-time format equivalent with the time format specified in the current system locale.

Updatability

During system startup, the string parameter *LocaleTimeFormat* (page 858) is computed on the basis of the information in the current system locale. The contents of the string parameter cannot be modified.

Note: The string parameter *LocaleTimeFormat* (page 858) can be used, for instance, in the `TimeslotFormat` attribute of a `Calendar`. The current system locale can be modified through the **Regional Settings** dialog box in the Windows **Control Panel**.

See also:

The string parameters *LocaleLongDateFormat* (page 857), *LocaleShortDateFormat* (page 857). Calendars are discussed in full detail in Calendars of the Language Reference, date-time formats in Format of Time Slots and Periods

8.6.13 LocaleTimeZoneName

The predefined string parameter *LocaleTimeZoneName* (page 859) contains the local name of all standard time zones.

```
StringParameter LocaleTimeZoneName {
     IndexDomain :  IndexTimeZones;
}
```

See also:

The predeclared identifier *LocaleTimeZoneNameDST* (page 859) contains the local name of all daylight saving time zones.

8.6.14 LocaleTimeZoneNameDST

The predefined string parameter *LocaleTimeZoneNameDST* (page 859) contains the local name of all daylight saving time zones.

```
StringParameter LocaleTimeZoneNameDST {
    IndexDomain  :  IndexTimeZones;
}
```

See also:

The predeclared identifier *LocaleTimeZoneName* (page 859) contains the local name of all standard time zones.

8.7 Error Handling Related Identifiers

The following collection of predefined identifiers contains data regarding the Error Handling functions.

8.7.1 errh::ErrorCodes

The predefined set *errh::ErrorCodes* (page 859) contains the error codes of the errors encountered during this AIMMS session.

```
Set ErrorCodes {
    Index       :  IndexErrorCodes;
}
```

Updatability

This set is grown during AIMMS error handling by storing the codes of all errors encountered.

8.7.2 errh::PendingErrors

The predefined set *errh::PendingErrors* (page 860) contains the error numbers of the errors that can be processed by the current error filter.

```
Set PendingErrors {
    SubsetOf    :  Integers;
    Index       :  IndexPendingErrors;
}
```

Updatability

The contents of this set cannot be modified. It is initialized when an error filter becomes active.

8.7.3 errh::AllErrorCategories

The predefined set *errh::AllErrorCategories* (page 860) contains the error categories that can be assigned to an error.

```
Set AllErrorCategories {
    Index       :  IndexErrorCategories;
}
```

The names below are the elements in the set. The elements are shown indented in order to show the structure that is used by the function `errh::InsideCategory`.

Engine Errors from the AIMMS engine.

Internal This is about AIMMS internal logic that fails. These types of errors shouldn't occur, but if they do, they should be handled. Severe internal errors (after generating a dump file) and internal assertions that fail

Authorization Protecting the intellectual property of the developer of the AIMMS application. % Your money

Licensing Protecting the intellectual property of the developers of the AIMMS system. % Our money

Memory Running out of memory.

Limit Reaching an AIMMS limit.

Compiler Errors detected by the AIMMS compiler.

Syntax Errors related to the form of AIMMS model text.

Semantics Errors related to the (allowed) interpretation of AIMMS model text.

Legacy Errors related to GAMS, AIMMS 2 or the conversion from a GAMS or AIMMS 2 model to AIMMS 3.

Execution Errors detected by the AIMMS execution engine.

Math Errors such as division by zero, sqrt or log of a negative number.

InvalidArgument Passing invalid arguments to the intrinsic functions of AIMMS.

Unit Runtime unit consistency checks that fail.

IO Database, File and Case IO errors.

External Passing argument data to / from external functions and procedures and errors generated during the execution of external functions.

Generation Runtime errors that occur during the generation of a mathematical program

MathematicalProgramming Violating the requirements of a particular mathematical programming class, or the selection of a mathematical programming class that is too difficult or too easy for the problem at hand.

NonlinearEvaluation Errors that happen during the evaluation of the (derivatives) of a constraint.

Solver Errors from the solution algorithms as part of the entire AIMMS package.

GUI Errors on pages

User Errors from `RAISE` statements or `ASSERT` statements.

Updatability

The contents of this set cannot be modified.

8.7.4 errh::AllErrorSeverities

The predefined set *errh::AllErrorSeverities* (page 861) contains the error categories that can be assigned to an error.

```
Set AllErrorSeverities {
    Index      :  IndexErrorSeverities;
}
```

The names below are the elements in the set.

severe A severe internal error is an error that has occurred in the AIMMS logic itself.

error A normal error which indicates a situation from which normally execution shouldn't continue.

warning Something that should be looked at, but doesn't necessarily indicate a problem. %

informational Summarizing information that might be of interest. %

detail Detailed information.

Updatability

The contents of this set cannot be modified.

CHAPTER

NINE

SUFFICES

9.1 Common Suffices

The following collection of suffices are common to all identifier types.

9.1.1 .dim

Definition

The `.dim` suffix returns the dimension of the identifier at hand.

Datatype

The value of the `.dim` suffix is numeric.

Dimension

The dimension of the `.dim` suffix itself is scalar.

Note:

- This suffix is deprecated and it is advised to use the intrinsic function *IdentifierDimension* (page 587) instead.
- See also Working with the Set AllIdentifiers of the Language Reference.

9.1.2 .txt

Definition

The `.txt` suffix returns the contents of the text attribute of the identifier at hand. When that attribute is empty it returns the name of the identifier itself.

Datatype

The value of a `.txt` suffix is a string.

Dimension

The `.txt` suffix is scalar.

Note:

- This suffix is typically used with an index into the set *AllIdentifiers* (page 821), as illustrated in the common example in Listing 9.1.
- See also Working with the Set AllIdentifiers of the Language Reference.
- The GAMS equivalent name is `.ts`.
- This suffix is deprecated.

9.1.3 .type

Definition

The `.type` suffix returns the type of the identifier at hand.

Datatype

The value of the `.type` suffix is an element in the Set *AllIdentifierTypes* (page 800).

Dimension

The `.type` suffix is scalar.

Note:

- This suffix is typically used with an index into the set *AllIdentifiers* (page 821), as illustrated in the common example in Listing 9.1.
- See also Working with the Set AllIdentifiers of the Language Reference.
- This suffix is deprecated, see *IdentifierType* (page 591).

9.1.4 .unit

Definition

The `.unit` suffix returns the unit of the identifier at hand.

Datatype

The datatype of the `.unit` suffix is string

Dimension

The `.unit` suffix is scalar.

Note:

- This suffix is typically used with an index into the set *AllIdentifiers* (page 821), as illustrated in the common example in Listing 9.1.
- See also the function *IdentifierUnit* (page 592)
- See also Working with the Set AllIdentifiers of the Language Reference.

9.1.5 Example

These suffixes are typically appended to an index into the set *AllIdentifiers* (page 821) or a subset thereof. Consider the following declaration:

```
Set SelectedIdentifiers {
    SubsetOf   :  AllIdentifiers;
    Index      :  si;
    OrderBy    :  si;
}
```

Then the following loop will make a simple overview of those identifiers:

Listing 9.1: Common suffix example

```
    SelectedIdentifiers := AllParameters ; ! Or some other selection.

    put outf ;

    outf.pagewidth := 255 ; ! Wide
    put "type":20, "  ", "name":32, "  ", "dim  ", "unit":20, "  ", "Text", / ;
    put "-"*20,    "  ", "-"*32,    "  ", "---  ", "-"*20,    "  ", "-"*40, / ;

    for ( si ) do                  ! For each selected identifier
        put si.type:20, "  "       ! Type
            si:32, "  ",           ! name
            "(",si.dim:1:0, ")  ", ! dimension
            si.unit:20, "  ",      ! unit
```

(continues on next page)

(continued from previous page)

```
            si.txt, /              ! Documenting text.
    endfor ;

    putclose ;
```

Note: Note that the suffixes *.dim* (page 863), *.txt* (page 863) and *.type* (page 864) are deprecated. See also Working with the Set AllIdentifiers of the Language Reference.

9.2 Horizon Suffices

The collection of suffices available to a horizon are.

9.2.1 .beyond

Definition

The Horizon suffix `.beyond` is a subset of the horizon. This subset contains those periods that come after the planning periods.

Datatype

The value of the `.beyond` suffix is set.

Note: See also Horizons of the Language Reference.

9.2.2 .past

Definition

The Horizon suffix `.past` is a subset of the horizon. This subset contains those periods that come before the current period.

Datatype

The value of the `.past` suffix is set.

Note: See also Horizons of the Language Reference.

9.2.3 .planning

Definition

The Horizon suffix `.planning` is a subset of the horizon. This subset is an adjacent set of `interval length` attribute periods starting with the `current period` attribute of the horizon at hand.

Datatype

The value of the `.planning` suffix is set.

Note: See also Horizons of the Language Reference.

See also Horizons of the Language Reference.

9.3 Variable and Constraint Suffices

AIMMS variables support the following collection of suffixes. The suffixes supported by AIMMS common to variables and constraints are the following collection of suffixes common to variables and constraint:

9.3.1 .Basic

Definition

When the property `Basic` of a constraint or variable is set or when the option `Always store basics` is set to `on`, the `.Basic` suffix contains basis status of the constraint or variable at the end of a solve.

Datatype

The value of the `.Basic` suffix is an element in the predeclared set *AllBasicValues* (page 794).

Dimension

The `.Basic` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- The default of the option `Always store basics` is `on`.
- In order to access the basic status of the definition of a variable `X` use the notation `X_definition.Basic`.
- See also Variable Declaration and Attributes of the Language Reference

---

9.3.2 .Level

Definition

The `.Level` suffix contains the current value of a variable. When the property `Level` of a constraint is set, the `.Level` suffix contains the current value of the left hand side of the constraint after the last solve.

Datatype

The value of the `.Level` suffix is numeric.

Dimension

The `.Level` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- When a variable without a suffix is used inside an assignment statement or a parameter definition the `.Level` suffix is automatically used.
- See also Constraint Levels, Bounds and Marginals of the Language Reference.
- The GAMS and AIMMS 2 equivalent suffix name is `.l`.

9.3.3 .Lower

Definition

The `.Lower` suffix contains the lower bound of a variable. When the property `Bounds` of a constraint is set, the `.Lower` suffix contains the minimum value the left hand side of the constraint may attain. Note that for a <= constraint this value is `-INF`. This value is set at the end of the generation step by AIMMS.

Datatype

The value of the `.Lower` suffix is numeric.

Dimension

The `.Lower` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- When the `.lower` suffix of a variable is equal to the `.upper` suffix (see *.Upper* (page 869)) of a variable that variable is treated as a frozen variable and subsequently removed from the generated mathematical program independently from the setting of the `.nonvar` suffix (see Variable Declaration and Attributes).
- In order to access the lower bound of the definition of a variable `X` use the notation `X_definition.Lower`.
- See also Sections Variable Declaration and Attributes and Constraint Levels, Bounds and Marginals of the Language Reference.

- The GAMS and AIMMS 2 equivalent suffix name is `.lo`.

9.3.4 .Stochastic

Definition

When the property `Stochastic` of a parameter or variable is set, the `.Stochastic` suffix contains the stochastic data of that parameter or variable. When the definition of a constraint contains a parameter or variable with the `Stochastic` property set the `.Stochastic` suffix of that constraint contains the stochastic rows.

Datatype

The value of the `.Stochastic` suffix is numeric.

Dimension

The dimension of `.Stochastic` suffix is one higher than that of the identifier at hand. The domain of the `.Stochastic` suffix is prefixed with the set *AllStochasticScenarios* (page 834) to the domain of the identifier at hand. The index domain of the `.Stochastic` suffix is prefixed with the index `IndexStochasticScenarios` to the index domain of the identifier at hand.

Note:

- See also Stochastic Programming of the Language Reference.

9.3.5 .Upper

Definition

The `.Upper` suffix contains the upper bound of a variable. When the property `Bounds` of a constraint is set, the `.Upper` suffix contains the maximum value the left hand side of the constraint may attain. Note that for a >= constraint this value is `INF`. This value is set at the end of the generation step by AIMMS.

Datatype

The value of the `.Upper` suffix is numeric.

Dimension

The `.Upper` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- When the `.Lower` suffix (see *.Lower* (page 868)) of a variable is equal to the `.Upper` suffix of that variable this variable is treated as a frozen variable and subsequently removed from the generated mathematical program independently from the setting of the `.nonvar` suffix (see Variable Declaration and Attributes).
- In order to access the upper bound of the definition of a variable `X` use the notation `X_definition.Upper`.
- See also Sections Variable Declaration and Attributes and Constraint Levels, Bounds and Marginals of the Language Reference.
- The `GAMS` and `AIMMS 2` equivalent suffix name is `.up`.

9.3.6 .Violation

Definition

The `.Violation` suffix of a variable contains the amount by which one of the bounds of that variable is violated. The `.Violation` suffix of a constraint contains the amount by which the definition of that constraint is violated.

Datatype

The value of the `.Violation` suffix is numeric.

Dimension

The `.Violation` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- When a variable `X` has a definition the suffix `.DefinitionViolation` can be used to obtain the violation of the defining constraint of `X`. An alternative is to use `X_definition.Violation`.
- See also Inspecting Your Model for Infeasibilities of the Language Reference and *.DefinitionViolation* (page 874).

9.3.7 .ExtendedConstraint

Definition

The `.ExtendedConstraint` suffix is the extended constraint associated with a constraint, variable or mathematical program. It is an identifier in itself and typically used wih AOA.

Dimension

The dimension of the suffix `.ExtendedConstraint` is one higher than the dimension of the identifier at hand. The domain of the suffix `.ExtendedConstraint` is the set AllGMPExtensions followed by the domain of the identifier at hand.

Note:

- See also Modifying an Extended Math Program Instance of the Language Reference.

9.3.8 .ExtendedVariable

Definition

The `.ExtendedVariable` suffix is the extended variable associated with a constraint, variable or mathematical program. It is an identifier in itself and typically used with the AOA solver.

Dimension

The dimension of the suffix `.ExtendedVariable` is one higher than the dimension of the identifier at hand. The domain of the suffix `.ExtendedVariable` is the set *AllGMPExtensions* (page 798) followed by the domain of the identifier at hand.

Note:

- See also Modifying an Extended Math Program Instance of the Language Reference.

9.4 Variable Suffices

AIMMS variables support the following collection of suffixes.

9.4.1 .ReducedCost

Definition

When the property `ReducedCost` of a variable is set or when the option `Always_store_marginals` is set to on, the `.ReducedCost` suffix contains the reduced cost of that variable.

Datatype

The value of the `.ReducedCost` suffix is numeric.

Dimension

The `.ReducedCost` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- The GAMS equivalent suffix name is `.m`.
- The default of the option `Always_store_marginals` is `off`.
- See also Variable Declaration and Attributes of the Language Reference.

9.4.2 .Nonvar

Definition

The `.Nonvar` suffix controls whether individual variables are frozen or not. This suffix can take on three values:

0 This variable is not frozen and a value for the variable should be found in the next solve statement.

1 This variable is frozen and it will retain its value during the SOLVE statement. The corresponding column will be removed from the generated mathematical program for the sake of efficiency.

-1 This variable is frozen and it will retain its value during the SOLVE statement. The corresponding column will *not* be removed from the generated mathematical program but can be manipulated during subsequent calls of the GMP function library.

Datatype

The value of the `.Nonvar` suffix is an integer in the range $\{-1, 0, 1\}$ and the default is 0.

Dimension

The `.Nonvar` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- When the `.lower` suffix of a variable is equal to the `.upper` suffix of the same variable that variable is treated as a frozen variable and subsequently removed from the generated mathematical program independently from the setting of the `.Nonvar` suffix.
- See also Variable Declaration and Attributes of the Language Reference.
- The option `Bound_tolerance` can affect variables that are frozen using the `.Nonvar` suffix, using a value of 1, but only if the level value of the variable is outside its bounds. If the bound violation is greater than the bound tolerance then the level value will be rounded to the nearest bound, and otherwise not.
- The `.Nonvar` suffix should not be confused with the GAMS suffix `.fx`. This latter suffix is a shorthand for the GAMS suffixes `.l`, `.lo` and `.up`.

9.4.3 .Relax

Definition

The variable suffix `.Relax` controls whether the integer variable at hand is relaxed to a continuous range or not. This suffix can take on two values:

0 This variable is not relaxed and its restriction to take on only integral values is passed on to the solver.

1 This variable is relaxed to the continuous range directly encompassing its original integral range.

Datatype

The value of the `.Relax` suffix is an integer in the range $\{0, 1\}$ and the default is 0.

Dimension

The `.Relax` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- See also Variable Declaration and Attributes of the Language Reference.

9.4.4 .Complement

Definition

The variable suffix `.Complement` contains the level value of the complementarity constraint after solving a complementarity problem.

Datatype

The value of the `.Complement` suffix is numeric.

Dimension

The `.Complement` suffix has the same dimension and domain as that of the variable at hand.

Note:

- The `Complement` suffix is only applicable for complementarity variables.
- See also Complementarity Problems of the Language Reference.

9.4.5 .DefinitionViolation

Definition

The variable suffix `.DefinitionViolation` contains the amount by which the defining constraint of that variable is violated when conducting an infeasibility analysis.

Datatype

The value of the `.DefinitionViolation` suffix is numeric.

Dimension

The `.DefinitionViolation` suffix has the same dimension and domain as that of the variable at hand.

Note:

- See also Infeasibility Analysis of the Language Reference.

9.4.6 .Derivative

Definition

The variable suffix `.Derivative` contains the derivative values of a variable used in an external function which is again used inside a constraint. The `.Derivative` suffix is only applicable inside the `derivative call` attribute of external functions.

Datatype

The value of the `.Derivative` suffix is numeric.

Dimension

The dimension of the suffix `.Derivative` is the dimension of the external function plus the dimension of the variable. The domain of the suffix `.Derivative` is the domain of the external function followed by the domain of the variable.

Note:

- See also Derivative Computation of the Language Reference.

9.4.7 .Priority

Definition

The variable suffix `.Priority` controls branching priority in the branch and bound solution process.

Datatype

The value of the `.Priority` suffix is numeric.

Dimension

The `.Priority` suffix has the same dimension and domain as that of the constraint or variable at hand.

Note:

- See also Variable Declaration and Attributes of the Language Reference.
- The GAMS equivalent suffix name is `.prior`.

9.4.8 .SmallestCoefficient

Definition

When the property `CoefficientRange` of a variable is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` a coefficient range sensitivity analysis is conducted such that the optimal basis remains constant. As a result of this analysis the variable suffix `.SmallestCoefficient` contains the smallest objective coefficient value.

Datatype

The value of the `.SmallestCoefficient` suffix is numeric.

Dimension

The `.SmallestCoefficient` suffix has the same dimension and domain as that of the variable at hand.

Note:

- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Variable Declaration and Attributes of the Language Reference.

9.4.9 .LargestCoefficient

Definition

When the property `CoefficientRange` of a variable is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` a coefficient range sensitivity analysis is conducted such that the optimal basis remains constant. As a result of this analysis the variable suffix `.LargestCoefficient` contains the largest objective coefficient value.

Datatype

The value of the `.LargestCoefficient` suffix is numeric.

Dimension

The `.LargestCoefficient` suffix has the same dimension and domain as that of the variable at hand.

Note:

- The default of the option `Calculate_Sensitivity_Ranges` is on.
- See also Variable Declaration and Attributes of the Language Reference.

9.4.10 .NominalCoefficient

Definition

When the property `CoefficientRange` of a variable is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` a coefficient range sensitivity analysis is conducted such that the optimal basis remains constant. As a result of this analysis the variable suffix `.NominalCoefficient` contains the nominal objective coefficient value.

Datatype

The value of the `.NominalCoefficient` suffix is numeric.

Dimension

The `.NominalCoefficient` suffix has the same dimension and domain as that of the variable at hand.

Note:

- The default of the option `Calculate_Sensitivity_Ranges` is on.
- See also Variable Declaration and Attributes of the Language Reference.

9.4.11 .SmallestValue

Definition

When the property `ValueRange` of a variable is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` a value range sensitivity analysis is conducted such that the objective value remains constant. As a result of this analysis the variable suffix `.SmallestValue` contains the smallest possible value of that variable.

Datatype

The value of the `.SmallestValue` suffix is numeric.

Dimension

The `.SmallestValue` suffix has the same dimension and domain as that of the variable at hand.

Note:

- The default of the option `Calculate_Sensitivity_Ranges` is on.
- See also Variable Declaration and Attributes of the Language Reference.

9.4.12 .LargestValue

Definition

When the property `ValueRange` of a variable is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` a value range sensitivity analysis is conducted such that the objective value remains constant. As a result of this analysis the variable suffix `.LargestValue` contains the largest possible value of that variable.

Datatype

The value of the `.LargestValue` suffix is numeric.

Dimension

The `.LargestValue` suffix has the same dimension and domain as that of the variable at hand.

Note:

- The default of the option `Calculate_Sensitivity_Ranges` is on.
- See also Variable Declaration and Attributes of the Language Reference.

9.5 Constraint Suffices

AIMMS constraints support the following collection of suffices.

9.5.1 .Convex

Definition

The constraint suffix `.Convex` is an indicator to the solver `Baron` that this constraint is convex.

Datatype

The value of the `.Convex` suffix is an integer in the range {0,1} and the default is 0.

Dimension

The `.Convex` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- See also Constraint Suffices for Global Optimization of the Language Reference.

9.5.2 .ShadowPrice

Definition

When the property `ShadowPrice` of a contraint is set or when the option `Always_store_marginals` is set to `on`, the `.ShadowPrice` suffix contains the shadow price of the constraint as computed by the solver. The shadow price of a constraint is the marginal change in the objective value with respect to a change in the right-hand side of the constraint.

Datatype

The value of the `.ShadowPrice` suffix is numeric.

Dimension

The `.ShadowPrice` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- When a variable `X` has a definition the suffix can also be applied to `X` but this is not encouraged by the syntax highlighting. The preferred notation is `X_definition.ShadowPrice`.
- The GAMS equivalent suffix name is `.m`.
- The default of the option `Always_store_basics` is `off`.
- See also Constraint Declaration and Attributes of the Language Reference.

9.5.3 .RelaxationOnly

Definition

The constraint suffix `.RelaxationOnly` is an indicator to the solver `Baron` that this constraint should be included as a relaxation to the branch-and-bound algorithm, while it should be excluded from the local search.

Datatype

The value of the `.RelaxationOnly` suffix is an integer in the range {0,1} and the default is 0.

Dimension

The `.RelaxationOnly` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- See also Constraint Suffices for Global Optimization of the Language Reference.

9.5.4 .SmallestShadowPrice

Definition

When the property `SmallestShadowPrice` of a contraint is set and when the option `Calculate_Sensitivity_Ranges` is set to `on`, the `.SmallestShadowPrice` suffix contains the smallest shadow price of the constraint while holding the objective value constant.

Datatype

The value of the `.SmallestShadowPrice` suffix is numeric.

Dimension

The `.SmallestShadowPrice` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- When a variable `X` has a definition the suffix can also be applied to `X` but this is not encouraged by the syntax highlighting. The preferred usage is `X_definition.SmallestShadowPrice`.
- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Constraint Declaration and Attributes of the Language Reference.

9.5.5 .LargestShadowPrice

Definition

When the property `LargestShadowPrice` of a contraint is set and when the option `Calculate_Sensitivity_Ranges` is set to `on`, the `.LargestShadowPrice` suffix contains the largest shadow price of the constraint while holding the objective value constant.

Datatype

The value of the `.LargestShadowPrice` suffix is numeric.

Dimension

The `.LargestShadowPrice` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- When a variable `X` has a definition the suffix can also be applied to `X` but this is not encouraged by the syntax highlighting. The preferred usage is `X_definition.LargestShadowPrice`.
- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Constraint Declaration and Attributes of the Language Reference.

9.5.6 .SmallestRightHandSide

Definition

When the property `RightHandSideRange` of a contraint is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` the `.SmallestRightHandSide` suffix contains the smallest right hand side such that the basis remains constant.

Datatype

The value of the `.SmallestRightHandSide` suffix is numeric.

Dimension

The `.SmallestRightHandSide` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- When a variable `X` has a definition the suffix can also be applied to `X` but this is not encouraged by the syntax highlighting. The preferred usage is `X_definition.SmallestRightHandSide`.
- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Constraint Declaration and Attributes of the Language Reference.

9.5.7 .LargestRightHandSide

Definition

When the property `RightHandSideRange` of a contraint is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` the `.LargestRightHandSide` suffix contains the largest right hand side such that the basis remains constant.

Datatype

The value of the `.LargestRightHandSide` suffix is numeric.

Dimension

The `.LargestRightHandSide` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- When a variable `X` has a definition the suffix can also be applied to `X` but this is not encouraged by the syntax highlighting. The preferred usage is `X_definition.LargestRightHandSide`.
- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Constraint Declaration and Attributes of the Language Reference.

9.5.8 .NominalRightHandSide

Definition

When the property `RightHandSideRange` of a contraint is set and the option `Calculate_Sensitivity_Ranges` is not set to `off` the `.NominalRightHandSide` suffix contains the right hand side value of the constraint. In case of a ranged constraint it contains the largest of the two constraint bounds.

Datatype

The value of the `.NominalRightHandSide` suffix is numeric.

Dimension

The `.NominalRightHandSide` suffix has the same dimension and domain as that of the constraint at hand.

Note:

- When a variable `X` has a definition the suffix can also be applied to `X` but this is not encouraged by the syntax highlighting. The preferred usage is `X_definition.NominalRightHandSide`.
- The default of the option `Calculate_Sensitivity_Ranges` is `on`.

- See also Constraint Declaration and Attributes of the Language Reference.

See also Constraint Declaration and Attributes of the Language Reference.

9.6 Mathematical Program Suffices

AIMMS mathematical programs support the following four collections of suffices. The first group of suffices steers the solution process. These suffices are specified in the model before the solve statement and are used during the solution process.

- *.bratio* (page 884)
- *.cutoff* (page 884)
- *.domlim* (page 884)
- *.iterlim* (page 885)
- *.limrow* (page 885)
- *.nodlim* (page 886)
- *.optca* (page 886)
- *.optcr* (page 886)
- *.reslim* (page 887)
- *.tolinfrep* (page 887)
- *.workspace* (page 888)

The second group of suffixes contain information obtained during and at the end of the solution process. these suffixes can be accessed after the solve statement.

- *.SolverStatus* (page 889)
- *.ProgramStatus* (page 888)
- *.SolverCalls* (page 888)
- *.Objective* (page 890)
- *.Incumbent* (page 889)
- *.BestBound* (page 890)
- *.GenTime* (page 891)
- *.SolutionTime* (page 892)
- *.Iterations* (page 891)
- *.Nodes* (page 891)
- *.NumberOfBranches* (page 892)
- *.NumberOfConstraints* (page 893)
- *.NumberOfFails* (page 893)
- *.NumberOfNonzeros* (page 894)
- *.NumberOfVariables* (page 895)
- *.NumberOfInfeasibilities* (page 893)

- *.SumOfInfeasibilities* (page 894)

The third group of suffixes control which AIMMS procedure should be called during the solution process and whether this calling should take place.

- *.CallbackProcedure* (page 895)
- *.CallbackIterations* (page 895)
- *.CallbackTime* (page 896)
- *.CallbackStatusChange* (page 896)
- *.CallbackIncumbent* (page 897)
- *.CallbackReturnStatus* (page 897)
- *.CallbackAddCut* (page 897)

The fourth group of suffixes are obsolete ones. They are only retained in order not to invalidate converted AIMMS 2 and GAMS models.

- .solveopt
- .prioropt
- .scaleopt
- .optfile
- .solprint
- .sysout
- .numnlins
- .numnlnz
- .domusd
- .nodusd
- .integer1
- .integer2
- .integer3
- .integer4
- .integer5
- .real1
- .real2
- .real3
- .real4
- .real5
- .line
- .limcol

9.6.1 .bratio

Definition

The `.bratio` suffix controls the basis acceptance test. When specified it overrides the option `accept_basis`.

Datatype

The value of the `.bratio` suffix is numeric.

Note:

- The suffix `.bratio` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

9.6.2 .cutoff

Definition

The `.cutoff` suffix can be specified when solving mixed integer programs. When specified it overrides the option `cutoff`.

Datatype

The value of the `.cutoff` suffix is numeric.

Note:

- The suffix `.cutoff` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

9.6.3 .domlim

Definition

When the number of domain violations during the optimization of a nonlinear program exceeds the value of the suffix `.domlim` the solution process is stopped. When specified this suffix overrides the option `maximal_number_of_domain_errors`.

Datatype

The value of the .domlim suffix is numeric.

Note:

- The suffix .domlim is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.4 .iterlim

Definition

The .iterlim suffix limits the number of iterations that can be used to solve the mathematical program. When specified this suffix overrides the option iteration_limit.

Datatype

The value of the .iterlim suffix is numeric.

Note:

- The suffix .iterlim is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.5 .limrow

Definition

The .limrow suffix limits the number of rows printed in the constraint listing per symbolic constraint. When specified it overrides the option Number_of_rows_per_constraint_in_listing.

Datatype

The value of the .limrow suffix is numeric.

Note:

- The suffix .limrow is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.6 .nodlim

Definition

The .nodlim controls the maximum number of nodes created during the Branch and Bound process. When specified it overrides the option maximal_number_of_nodes.

Datatype

The value of the .nodlim suffix is numeric.

Note:

- The suffix .nodlim is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.7 .optca

Definition

When specified, the solution process stops if the solver can guarantee that the current best solution is within the value of suffix optca of the global optimum. This is only valid for mixed integer programming models including mixed integer quadratic problems. When specified the suffix .optca overrides the option MIP_Absolute_Optimality_Tolerance.

Datatype

The value of the .optca suffix is numeric.

Note:

- The suffix .optca is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.8 .optcr

Definition

When specified the solution procedure stops if the solver can guarantee that the current best solution is within suffix .optcr of the global optimum. This is only valid for mixed integer programming models including mixed integer quadratic problems. The .optcr suffix controls the append mode of the file. When specified the suffix .optcr overwrites the option MIP_Relative_Optimality_Tolerance

Datatype

The value of the .optcr suffix is numeric.

Note:

- The suffix .optcr is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.9 .reslim

Definition

When specified, the solution process stops after .reslim seconds. When specified it overrides the option time_limit.

Datatype

The value of the .reslim suffix is numeric.

Note:

- The suffix .optcr is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.10 .tolinfrep

Definition

When specified, the suffix .tolinfrep is the tolerance on row feasibility when computing the values of the suffixes .NumberOfInfeasibilities and .SumOfInfeasibilities. When specified the option .tolinfrep overrides the option bound_tolerance.

Datatype

The value of the .tolinfrep suffix is numeric.

Note:

- The suffix .tolinfrep is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

9.6.11 .workspace

Definition

The `.workspace` suffix controls the amount of workspace to be used by the solver in Mb. When specified it overrides the option `workspace`.

Datatype

The value of the `.workspace` suffix is numeric.

Note:

- The suffix `.workspace` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

9.6.12 .ProgramStatus

Definition

The mathematical program suffix `.ProgramStatus` contains the status of the mathematical program at the end of the solve.

Datatype

The value of the `.ProgramStatus` suffix is an element in the set *AllSolutionStates* (page 807).

Note:

- The related GAMS and AIMMS 2 name is `.modelstat` but that value is a numeric code.
- The `.ProgramStatus` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.13 .SolverCalls

Definition

The mathematical program suffix `.SolverCalls` contains the number of times the mathematical program has been solved.

Datatype

The value of the `.SolverCalls` suffix is an integer.

Note:

- The GAMS and AIMMS 2 equivalent name is `.number`.
- The `.SolverCalls` suffix is also mentioned in Table Suffices of a mathematical program statistics from AIMMS of the Language Reference.

9.6.14 .SolverStatus

Definition

The mathematical program suffix `.SolverStatus` suffix contains the solver status at the end of the solve statement.

Datatype

The value of the `.SolverStatus` suffix is element and its range is *AllSolutionStates* (page 807).

Note:

- The related GAMS and AIMMS 2 name is `.SolveStat` but that value is a numeric code.
- The `.SolverStatus` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.15 .Incumbent

Definition

The `.Incumbent` suffix contains the current best solution during the solution process of `MIP`, `MIQP` and `MIQCP` problems.

Datatype

The value of the `.Incumbent` suffix is numeric.

Note:

- The `.Incumbent` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.16 .Objective

Definition

The mathematical program suffix `.Objective` suffix contains the value of the objective at the end of the solve.

Datatype

The value of the `.Objective` suffix is numeric. When the solve is not successful or infeasible the value of the `.Objective` is `NA`.

Note:

- For multi-objective models, the `.Objective` suffix refers to the (blended) objective with the highest priority.
- The equivalent GAMS and AIMMS 2 name is `.objval`.
- The `.Objective` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.17 .BestBound

Definition

The `.BestBound` suffix contains the current best bound during the branch-and-bound solution process of `MIP`, `MIQP` and `MIQCP` problems.

Datatype

The value of the `.BestBound` suffix is numeric.

Note:

- The `.BestBound` suffix also contains the current best bound during a solve with BARON.
- The `.BestBound` suffix also contains the current best bound during the solve of a nonconvex QP problem with CPLEX, if the CPLEX option *Solution Target* is set to 'Search for global optimum'.
- The `.BestBound` suffix suffix also contains the current best bound during the solve of a QP, QCP, NLP or MINLP problem with GUROBI.
- For multi-objective models, the `.BestBound` suffix refers to the (blended) objective with the highest priority.
- The `.BestBound` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.18 .Nodes

Definition

The mathematical program suffix `.Nodes` contains the number of nodes visited during the Branch and Bound search.

Datatype

The value of the `.Nodes` suffix is an integer.

Note:

- The equivalent `GAMS` and AIMMS 2 name is `.nodusd`.
- The `.Nodes` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.19 .GenTime

Definition

The mathematical program suffix `.GenTime` contains the time required to generate the mathematical program.

Datatype

The value of the `.GenTime` suffix is numeric and in wallclock seconds.

Note:

- The suffix `.GenTime` has unit [second] iff (1) this unit has been declared, and (2) the option `solution_time_has_unit_seconds` is set to on. In all other cases the suffix has no unit.
- The equivalent GAMS and AIMMS 2 name is `.resgen`.
- The `.GenTime` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.20 .Iterations

Definition

The mathematical program suffix `.Iterations` contains the number of iterations executed by the solver.

Datatype

The value of the `.Iterations` suffix is an integer.

Note:

- The GAMS and AIMMS 2 equivalent name is `.itrusd`.
- The `.Iterations` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.21 .SolutionTime

Definition

The mathematical program suffix `.SolutionTime` contains the time required to solve the mathematical program.

Datatype

The value of the `.SolutionTime` suffix is numeric.

Note:

- The suffix `.SolutionTime` has unit [second] iff (1) this unit has been declared, and (2) the option `solution_time_has_unit_seconds` is kept to its default of `on`. In all other cases the suffix has no unit.
- The GAMS and AIMMS 2 equivalent name is `.resusd`.
- The `.SolutionTime` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.22 .NumberOfBranches

Definition

The mathematical program suffix `.NumberOfBranches` contains the number of nodes visited by a CP solver.

Datatype

The value of the `.NumberOfBranches` suffix is an integer.

Note:

- The `.NumberOfBranches` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.23 .NumberOfConstraints

Definition

The mathematical program suffix `.NumberOfConstraints` contains the number of individual constraints in the generated mathematical program.

Datatype

The value of the `.NumberOfConstraints` suffix is an integer.

Note:

- The GAMS and AIMMS 2 equivalent name is `.numequ`.
- The `.NumberOfConstraints` suffix is also mentioned in Table Suffices of a mathematical program statistics from AIMMS of the Language Reference.

9.6.24 .NumberOfFails

Definition

The mathematical program suffix `.NumberOfFails` contains the number of leaf nodes searched by a CP solver for which it has been proved that no solution exists.

Datatype

The value of the `.NumberOfFails` suffix is an integer.

Note:

- The `.NumberOfFails` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.25 .NumberOfInfeasibilities

Definition

The mathematical program suffix `.NumberOfInfeasibilities` contains the number of individual constraints that are infeasible at the end of the solve.

Datatype

The value of the `.NumberOfInfeasibilities` suffix is an integer.

Note:

- The GAMS and AIMMS 2 equivalent name is `.numinfes`.
- The `.NumberOfInfeasibilities` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.26 .SumOfInfeasibilities

Definition

The `.SumOfInfeasibilities` contains the sum of the infeasibilities at the end of a solve.

Datatype

The value of the `.SumOfInfeasibilities` suffix is numeric.

Note:

- The GAMS and AIMMS 2 equivalent name is `.suminfes`.
- The `.SumOfInfeasibilities` suffix is also mentioned in Table Suffices of a mathematical program filled by the solver of the Language Reference.

9.6.27 .NumberOfNonzeros

Definition

The mathematical program suffix `.NumberOfNonzeros` contains the number of nonzeros in the generated mathematical program.

Datatype

The value of the `.NumberOfNonzeros` suffix is an integer.

Note:

- The GAMS and AIMMS 2 equivalent name is `.numnz`.
- The `.NumberOfNonzeros` suffix is also mentioned in Table Suffices of a mathematical program statistics from AIMMS of the Language Reference.

9.6.28 .NumberOfVariables

Definition

The mathematical program suffix `.NumberOfVariables` contains the number of individual variables in the generated mathematical program.

Datatype

The value of the `.NumberOfVariables` suffix is an integer.

Note:

- The GAMS and AIMMS 2 equivalent name is `.numvar`.
- The `.NumberOfVariables` suffix is also mentioned in Table Suffices of a mathematical program statistics from AIMMS of the Language Reference.

9.6.29 .CallbackIterations

Definition

The suffix `.CallbackIterations` states after how many iterations the AIMMS procedure in the suffix `.CallbackProcedure` should be called.

Datatype

The value of the `.CallbackIterations` suffix is numeric and the default is 0. When the value of this suffix is 0, the callback procedure in the suffix `.CallbackProcedure` is not called.

Note:

- See also Suffices and Callbacks of the Language Reference.

9.6.30 .CallbackProcedure

Definition

The suffix `.CallbackProcedure` contains the name of the AIMMS procedure to be called for every suffix `.CallbackIterations` iterations executed.

Datatype

The value of the `.CallbackProcedure` suffix is an element in the set of *AllProcedures* (page 825) and the default is the empty element `''`.

Note:

- See also Suffices and Callbacks of the Language Reference.

9.6.31 .CallbackStatusChange

Definition

The mathematical program suffix `.CallbackStatusChange` contains the name of the AIMMS procedure to be called upon a status change of the generated mathematical program during the solution process.

Datatype

The value of the `.CallbackStatusChange` suffix is an element in the set of *AllProcedures* (page 825) and the default is the empty element `''`.

Note:

- See also Suffices and Callbacks of the Language Reference.

9.6.32 .CallbackTime

Definition

The mathematical program suffix `.CallbackTime` contains the name of the AIMMS procedure to be called after a certain number of seconds have elapsed.

Datatype

The value of the `.CallbackTime` suffix is an element in the set of *AllProcedures* (page 825) and the default is the empty element `''`.

Note:

- See also Suffices and Callbacks of the Language Reference.
- The `CallbackTime` callback procedure is supported by CPLEX, GUROBI, CBC, XA, CP OPTIMIZER, CONOPT, KNITRO, SNOPT and IPOPT.
- The number of (elapsed) seconds is determined by the general solvers option `Progress Time Interval`. This option also specifies the interval for updating the Progress Window during a solve. As a consequence, the information passed to this callback procedure will be the same as the information displayed in the Progress Window (except for small differences for the solving time).

- The time callback will be called less often if CPLEX uses dynamic search as the MIP Search Strategy instead of branch-and-cut. In that case the interval between two successive calls might sometimes be larger than the interval as specified by the option `Progress Time Interval`.

9.6.33 .CallbackIncumbent

Definition

The mathematical program suffix `.CallbackIncumbent` contains the name of the AIMMS procedure to be called when a new incumbent is found during the solution process.

Datatype

The value of the `.CallbackIncumbent` suffix is an element in the set of *AllProcedures* (page 825) and the default is the empty element `''`.

Note:

- See also Suffices and Callbacks of the Language Reference.

9.6.34 .CallbackReturnStatus

Definition

The mathematical program suffix `.CallbackReturnStatus` controls the continuation of the solution process. It can be set from within one of the callback procedures.

Datatype

The value of the `.CallbackReturnStatus` suffix is an element in the set `ContinueAbort`.

Note:

- See also Suffices and Callbacks of the Language Reference.

9.6.35 .CallbackAddCut

Definition

The mathematical program suffix `.CallbackAddCut` contains the name of the AIMMS procedure to be called to add additional cuts.

Datatype

The value of the `.CallbackAddCut` suffix is an element in the set of *AllProcedures* (page 825) and the default is the empty element `''`.

Note:

- See also Suffices and Callbacks of the Language Reference.

9.7 File Suffices

AIMMS files support the following three collections of suffices. File suffix group 1: the suffixes that apply to the entire file.

- *.Ap* (page 899)
- *.blank_zeros* (page 899)
- *.case* (page 900)
- *.PageNumber* (page 900)
- *.PageMode* (page 900)
- *.PageSize* (page 901)
- *.PageWidth* (page 901)

File suffix group 2: the suffixes that control page layout.

- *.TopMargin* (page 902)
- *.LeftMargin* (page 902)
- *.BottomMargin* (page 902)
- *.BodyCurrentColumn* (page 903)
- *.BodyCurrentRow* (page 903)
- *.BodySize* (page 904)
- *.FooterCurrentColumn* (page 904)
- *.FooterCurrentRow* (page 904)
- *.FooterSize* (page 905)
- *.HeaderCurrentColumn* (page 905)
- *.HeaderCurrentRow* (page 906)
- *.HeaderSize* (page 906)

File suffix group 3: the suffixes that control the formatting of individual elements.

- *.lj* (page 906)
- *.lw* (page 907)
- *.nd* (page 907)
- *.nj* (page 908)

- *.nr* (page 908)
- *.nw* (page 909)
- *.nz* (page 909)
- *.sj* (page 909)
- *.sw* (page 910)
- *.tf* (page 910)
- *.tj* (page 911)
- *.tw* (page 911)

9.7.1 .Ap

Definition

The `.Ap` suffix controls the append mode of the file.

Datatype

The value of the `.Ap` suffix is an integer in the range {0,1} and the default is 0. The interpretation of the possible values is:

0 Overwrite

1 Append

Note:

- The file attribute *mode* should be used instead.

9.7.2 .blank_zeros

Definition

The `.blank_zeros` suffix controls whether or not numbers (almost) equal to 0.0 should be printed as blanks or as 0.0's according to the current format.

Datatype

The value of the `.blank_zeros` suffix is an integer in the range {0..2} and the default is 0. The possible values are:

0 Do not print numbers equal or within AIMMS tolerances equal to 0.0 as blanks.

1 Print numbers equal or within AIMMS tolerances equal to 0.0 as blanks.

2 Print numbers after formatting equal to 0.0 as blanks.

9.7.3 .case

Definition

The `.case` suffix controls whether or not the output is translated to upper case.

Datatype

The value of the `.case` suffix is an integer in the range {1,2} and the default is 0. The interpretation of the possible values is:

0 Leave the output in mixed case.

1 Translate the output to upper case.

9.7.4 .PageMode

Definition

The file suffix `.PageMode` controls the formatting style of the page.

Datatype

The value of the `.PageMode` suffix is an element in the predeclared set `OnOff` and the default is `Off`. The interpretation of the possible values is:

`On` Structure output in pages

`Off` Do not structure output in pages.

Note:

- The equivalent GAMS and AIMMS 2 name is *.pc* but this value is numeric.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.5 .PageNumber

Definition

The file suffix `.PageNumber` contains the number of the current page.

Datatype

The value of the `.PageNumber` suffix is numeric.

Note:

- The equivalent GAMS and AIMMS 2 name is `.lp`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.6 .PageSize

Definition

The file suffix `.PageSize` controls the maximum number of lines on a page including header, body and footer.

Datatype

The value of the `.PageSize` suffix is an integer in the range {3..200}.

Note:

- The equivalent GAMS and AIMMS 2 name is `.ps`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.7 .PageWidth

Definition

The file suffix `.PageWidth` controls the maximum number of characters per line. When specified it overrides the option `listing_page_width`.

Datatype

The value of the `.PageWidth` suffix is an integer in the range {30..32767}.

Note:

- The suffix `.PageWidth` is initialized to `-1`. AIMMS considers it specified when its value is not equal to `-1`.
- The equivalent GAMS and AIMMS 2 name is `.pw`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.8 .TopMargin

Definition

The file suffix `.TopMargin` controls the top margin in number of lines.

Datatype

The value of the `.TopMargin` suffix is an integer in the range {0..``option listing_size``} and the default is 0.

Note:

- The equivalent GAMS and AIMMS 2 name up to AIMMS 3.3 is *.tm*.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.9 .BottomMargin

Definition

The `.BottomMargin` is the bottom margin in number of lines.

Datatype

The value of the `.BottomMargin` suffix is an integer in the range {0..``option listing_size``} and the default is 0.

Note:

- The equivalent GAMS and AIMMS 2 name up to AIMMS 3.3 is `.bm`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.10 .LeftMargin

Definition

The `.LeftMargin` is the left margin in number of characters.

Datatype

The value of the `.LeftMargin` suffix is an integer in the range {0..``option listing_page_width``} and the default is 0.

Note:

- The equivalent GAMS and AIMMS 2 name up to AIMMS 3.3 is `.lm`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.11 .BodyCurrentColumn

Definition

The `.BodyCurrentColumn` contains the current column position in the file.

Datatype

The value of the `.BodyCurrentColumn` suffix is an integer in the range {0..``option listing_page_width``} and the default is 0.

Note:

- The equivalent GAMS and AIMMS 2 name is `.cc`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.12 .BodyCurrentRow

Definition

The `.BodyCurrentRow` contains the current line number of the current page.

Datatype

The value of the `.BodyCurrentRow` suffix is an integer in the range {0..``option listing_size``} and the default is 1.

Note:

- The equivalent GAMS and AIMMS 2 name is `.cr`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.13 .BodySize

Definition

The `.BodySize` contains the number of lines on the current page.

Datatype

The value of the `.BodySize` suffix is an integer in the range {0..``option listing_size``} and the default is 1.

Note:

- The equivalent GAMS and AIMMS 2 name is `.ll`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.14 .FooterCurrentColumn

Definition

The `.FooterCurrentColumn` contains the current column position in the page footer.

Datatype

The value of the `.FooterCurrentColumn` suffix is an integer in the range {0..``option listing_page_width``} and the default is 0.

Note:

- The equivalent GAMS and AIMMS 2 name is `.ftcc`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.15 .FooterCurrentRow

Definition

The `.FooterCurrentRow` contains the current line number of the footer of the current page.

Datatype

The value of the `.FooterCurrentRow` suffix is an integer in the range {0..``option listing_size``} and the default is 1.

Note:

- The equivalent GAMS and AIMMS 2 name is `.ftcr`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.16 .FooterSize

Definition

The `.FooterSize` contains the number of lines in the footer of the page.

Datatype

The value of the `.FooterSize` suffix is an integer in the range {0..``option listing_size``} and the default is 1.

Note:

- The equivalent GAMS and AIMMS 2 name is `.ftll`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.17 .HeaderCurrentColumn

Definition

The `.HeaderCurrentColumn` contains the current column position in the header of the page.

Datatype

The value of the `.HeaderCurrentColumn` suffix is an integer in the range {0..``option listing_page_width``} and the default is 0.

Note:

- The equivalent GAMS and AIMMS 2 name is `.hdcc`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.18 .HeaderCurrentRow

Definition

The `.HeaderCurrentRow` contains the current row number in the header of the page.

Datatype

The value of the `.HeaderCurrentRow` suffix is an integer in the range {0..``option listing_size``} and the default is 1.

Note:

- The equivalent GAMS and AIMMS 2 name is `.hdcr`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.19 .HeaderSize

Definition

The `.HeaderSize` contains the number of lines in the header of the page.

Datatype

The value of the `.HeaderSize` suffix is an integer in the range {0..``option listing_size``} and the default is 1.

Note:

- The equivalent GAMS and AIMMS 2 name is `.hdll`.
- See also Structuring a Page in Page Mode of the Language Reference.

9.7.20 .lj

Definition

The `.lj` suffix controls the element justification. When specified it overrides the option `put_element_justification`.

Datatype

The value of the .lj suffix is integer in the range {1..3} and the default is -1. The possible values are:

1 Right

2 Left

3 Center

Note:

- The suffix .lj is initialized to -1. AIMMS considers it specified when its value is not equal to -1.
- The suffix .lj is a legacy from GAMS and AIMMS 2.

9.7.21 .lw

Definition

The .lw suffix controls the element field width. When specified it overrides the option put_element_width.

Datatype

The value of the .lw suffix is an integer in the range {0..``option listing_page_width``} and the default is -1.

Note:

- The suffix .lw is initialized to -1. AIMMS considers it specified when its value is not equal to -1.
- The suffix .lw is a legacy from GAMS and AIMMS 2.

9.7.22 .nd

Definition

The .nd suffix controls the number of decimals displayed. When specified it overrides the option put_number_decimals.

Datatype

The value of the .nd suffix is an integer in the range {0..``option listing_page_width``} and the default is -1.

Note:

- The suffix .nd is initialized to -1. AIMMS considers it specified when its value is not equal to -1.
- The suffix .nd is a legacy from GAMS and AIMMS 2.

9.7.23 .nj

Definition

The .nj suffix controls numeric justification. When specified it overrides the option put_number_justification.

Datatype

The value of the .nj suffix is integer in the range {1..3} and the default is -1. The possible values are:

1 Right

2 Left

3 Center

Note:

- The suffix .nj is initialized to -1. AIMMS considers it specified when its value is not equal to -1.
- The suffix .nj is a legacy from GAMS and AIMMS 2.

9.7.24 .nr

Definition

The .nr suffix controls the numeric formatting method. When specified it overrides the option put_number_style.

Datatype

The value of the .nr suffix is an integer in the range {0..3} and the default is -1. The possible values are:

0 Fit field or e format

1 Fit field width

2 Always e format

3 Fit field or e format or 0

Note:

- The suffix .nr is initialized to -1. AIMMS considers it specified when its value is not equal to -1.
- The suffix .nr is a legacy from GAMS and AIMMS 2.

9.7.25 .nw

Definition

The `.nw` suffix controls numeric field width. When specified it overrides the option `put_number_width`.

Datatype

The value of the `.nw` suffix is an integer in the range {0..``option listing_page_width``} and the default is -1.

Note:

- The suffix `.nw` is initialized to `-1`. AIMMS considers it specified when its value is not equal to `-1`.
- The suffix `.nw` is a legacy from GAMS and AIMMS 2.

9.7.26 .nz

Definition

The `.nz` suffix controls the nonzero tolerance. When specified it overrides the option `put_number_tolerance`.

Datatype

The value of the `.nz` suffix is a floating point number in the range [0,1] and the default is -1.0.

Note:

- The suffix `.nz` is initialized to `-1.0`. AIMMS considers it specified when its value is not equal to `-1.0`.
- The suffix `.nz` is a legacy from GAMS and AIMMS 2.

9.7.27 .sj

Definition

The `.sj` suffix controls the justification of the texts associated with elements in a *GAMS* model. In an AIMMS model a string parameter is used instead of associating texts with elements. When specified it overrides the option `put_string_justification`.

Datatype

The value of the `.sj` suffix is integer in the range {1..3} and the default is `-1`. The possible values are:

1 Right

2 Left

3 Center

Note:

- The suffix `.sj` is initialized to `-1`. AIMMS considers it specified when its value is not equal to `-1`.
- The suffix `.sj` is a legacy from GAMS and AIMMS 2.

9.7.28 .sw

Definition

The `.sw` suffix controls the field width of the texts associated with elements in a *GAMS* model. In an AIMMS model a string parameter is used instead of associating texts with elements. A value of 0 implies variable length. When specified it overrides the option `put_string_width`.

Datatype

The value of the `.sw` suffix is an integer in the range {0..``option listing_page_width``} and the default is -1.

Note:

- The suffix `.sw` is initialized to `-1`. AIMMS considers it specified when its value is not equal to `-1`.
- The suffix `.sw` is a legacy from GAMS and AIMMS 2.

9.7.29 .tf

Definition

The `.tf` suffix controls the text fill mode when putting the text associated with identifiers. There is no option associated with this suffix.

Datatype

The value of the `.tf` suffix is an integer in the range {0..2} and the default is `2`. The possible values are:

0 No fill.

1 Fill existing only.

2 Fill always.

Note:

- The suffix `.tf` is a legacy from GAMS and AIMMS 2.

9.7.30 .tj

Definition

The `.tj` suffix controls the justification when putting the text associated with identifiers. When specified it overrides the option `put_string_justification`.

Datatype

The value of the `.tj` suffix is integer in the range {1..3} and the default is `-1`. The possible values are:

1 Right

2 Left

3 Center

Note:

- The suffix `.tj` is initialized to `-1`. AIMMS considers it specified when its value is not equal to `-1`.
- The suffix `.tj` is a legacy from GAMS and AIMMS 2.

9.7.31 .tw

Definition

The `.tw` suffix controls field width when putting the text associated with identifiers. When specified it overrides the option `put_string_width`.

Datatype

The value of the `.tw` suffix is an integer in the range {0..``option listing_page_width``} and the default is -1.

Note:

- The suffix `.tw` is initialized to `-1`. When its value is not equal to `-1` AIMMS considers it specified.
- The suffix `.tw` is a legacy from GAMS and AIMMS 2.

CHAPTER

TEN

DEPRECATED

10.1 Deprecated Language Elements

The current implementation of AIMMS supports the following deprecated features, but it may cease to do so in a future implementation. The current implementation does so to support converted GAMS and AIMMS 2 applications.

10.1.1 Deprecated Keywords

The keywords for which direct replacements are available are documented in Table 10.1.

Table 10.1: AIMMS deprecated keywords and their modern equivalents

Deprecated	Modern equivalent
clean	CleanDependents
CumulativeDistribution	DistributionCumulative
eps	zero
evaluate	update
FailureCount	FailCount
InverseCumulativeDistribution	DistributionInverseCumulative
maximise	maximize
maximising	maximize
maximizing	maximize
minimise	minimize
minimising	minimize
minimizing	minimize
net_inflow	netinflow
net_outflow	netoutflow
puttl	puthd

The Deprecated Keyword abort

The keyword abort is a GAMS keyword that can be followed by a condition and a list of identifiers to be displayed. The execution run is interrupted after executing this statement. Suggested rewrite: use a display statement followed by a halt statement or a raise error statement. See also

- display (see The DISPLAY Statement of the Language Reference),
- halt (see The HALT Statement of the Language Reference) and
- raise error (see Raising Errors and Warnings of the Language Reference).

The Deprecated Keywords yes and no

The keywords yes and no are GAMS keywords that can be used in assignments to sets in order to add or remove elements. Suggested rewrite: use the AIMMS set syntax. For instance, replace

```
s1(i) $ cond1(i) := yes ;
s2(i) $ cond2(i) := no  ;
```

by the following code:

```
s1 += { i | cond1(i) } ;
s2 -= { i | cond2(i) } ;
```

The Deprecated Keyword system

The GAMS keyword system is followed by a suffix. The AIMMS language supports the following equivalent code for selected system suffixes as documented in Table Table 10.2.

Table 10.2: The keyword system and selected suffixes with their modern counterparts

Deprecated	Modern equivalent
.date	CurrentToString("%Am|AllAbbrMonths| %d, %c%y")
.time	CurrentToString("%H:%M:%S")
.version	AimmsRevisionString(string parameter, 4);
.page	currentOutputFile.PageNumber

The system suffixes .ifile, .ofile, .rdate, .rfile, .rtime, .sfile, and .title are pointless within the AIMMS environment.

10.1.2 Deprecated Intrinsic Procedures and Functions

The mapping of the matrix manipulation procedures to GMP procedures and functions is documented in Matrix Manipulation Procedures of the Language Reference. The following intrinsic functions are deprecated, but can be replaced by an equivalent call to an existing intrinsic procedure or function:

- FindRString(SearchString, Key, CaseSensitive, WordOnly, IgnoreWhite) can be replaced by a call to FindNthString(SearchString, Key, -1, CaseSensitive, WordOnly, IgnoreWhite) where -1 indicates that searching should be done right to left, see also *FindNthString* (page 70).
- One may replace SQLDirect with DirectSQL
- One may replace StringToLabel with StringToElement

The deprecated iterative operators are documented in Table 10.3.

Table 10.3: AIMMS deprecated iterative operators and their modern equivalents

Deprecated	Modern equivalent
smax	max
smin	min
arg	nth

10.1.3 Deprecated Suffixes

Table 10.4: AIMMS deprecated suffixes and their modern equivalents

Deprecated	Modern equivalent
Variables	
.l	.level
.lo	.lower
.up	.upper
.freeze	.nonvar
.prior	.priority
Files	
.bm	.BottomMargin
.cc	.BodyCurrrentColumn
.cr	.BodyCurrrentRow
.ftcc	.FooterCurrrentColumn
.ftcr	.FooterCurrrentRow
.ftll	.HeaderSize
.hdcc	.HeaderCurrrentColumn
.hdcr	.HeaderCurrrentRow
.hdll	.FooterSize
.lm	.LeftMargin
.lp .pn	.PageNumber
.pc	.PageMode
.ps	.PageSize
.pw	.PageWidth
.tm	.TopMargin
Mathematical programs	
.bestest .objest	.BestBound
.CallbackNewIncumbent	.CallbackIncumbent
.iterusd	.iterations
.nodusd	.nodes
.number	.SolverCalls
.numequ	.NumberOfConstraints
.numinfes	.NumberOfInfeasibilities
.numintvar	.NumberOfIntegerVariables
.numnlequ	.NumberOfNonlinearConstraints
.numnlins	.NumberOfNonlinearInstructions
.numnlnz .numnlz	.NumberOfNonlinearNonzeros
.numnlvar	.NumberOfNonlinearVariables
.numnz	.NumberOfNonzeros
.numSOS1	.NumberOfSOS1Constraints
.numSOS2	.NumberOfSOS2Constraints
.numvar	.NumberOfVariables
.objval	.Objective
.resgen	.GenTime
.resusd	.SolutionTime
.suminfes	.SumOfInfeasibilities

Most deprecated suffixes can be directly translated into their modern equivalents, as documented in Table 10.4. The following suffixes deserve some more consideration:

- .ap The append mode of a file, 0: replace contents when opening the file, 1: append to file. This functionality

is now covered by the mode attribute of that file, see The File Declaration of the Language Reference.

- .m The marginal value of a variable or constraint. For a constraint the suffix .m should be replaced by the suffix .ShadowPrice. For a variable the suffix .m should be replaced by the suffix .ReducedCost.
- .modelstat This suffix of a mathematical program is numeric, it should be replaced by the element valued suffix .ProgramStatus. Note that Element(AllSolutionStates, mp.solvestat+1) = mp.ProgramStatus. See also Mathematical program and solver status of the Language Reference and *AllSolutionStates* (page 807).
- .solvestat or .solverstat These suffixes of a mathematical program are numeric, they should be replaced by the element valued suffix .SolverStatus. Note that Element(AllSolutionStates, mp.solvestat+15) = mp.SolverStatus. See also Mathematical program and solver status of the Language Reference and *AllSolutionStates* (page 807).
- .dim This should be replaced by a call to *IdentifierDimension* (page 587).
- .txt This should be replaced by a call to *IdentifierText* (page 590).
- .type This should be replaced by a call to *IdentifierType* (page 591).

10.2 Matrix Manipulation Procedures

The matrix manipulation procedures have been deprecated since AIMMS version 3.5, and have been removed in AIMMS version 4.96. New projects should use the GMP library instead. Please refer to Table 10.5 for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

Table 10.5: Removed matrix manipulation procedures

Deprecated Procedure	Counterpart
MatrixModifyCoefficient	*GMP::Coefficient::Set* (page 316)
MatrixModifyQuadraticCoefficient	*GMP::Coefficient::SetQuadratic* (page 318)
MatrixModifyRightHandSide	*GMP::Row::SetRightHandSide* (page 472)
MatrixModifyLeftHandSide	*GMP::Row::SetLeftHandSide* (page 468)
MatrixModifyRowType	*GMP::Row::SetType* (page 476)
MatrixAddRow	*GMP::Row::Add* (page 448)
MatrixRegenerateRow	*GMP::Row::Generate* (page 455)
MatrixDeactivateRow	*GMP::Row::Deactivate* (page 449)
MatrixActivateRow	*GMP::Row::Activate* (page 445)
MatrixModifyLowerBound	*GMP::Column::SetLowerBound* (page 343)
MatrixModifyUpperBound	*GMP::Column::SetUpperBound* (page 350)
MatrixModifyColumnType	*GMP::Column::SetType* (page 347)
MatrixAddColumn	*GMP::Column::Add* (page 322)
MatrixFreezeColumn	*GMP::Column::Freeze* (page 326)
MatrixUnfreezeColumn	*GMP::Column::Unfreeze* (page 354)
MatrixModifyType	*GMP::Instance::SetMathematicalProgrammingType* (page 420)
MatrixModifyDirection	*GMP::Instance::SetDirection* (page 419)
MatrixGenerate	*GMP::Instance::Generate* (page 384)
MatrixSolve	*GMP::Instance::Solve* (page 424)
MatrixSaveState	*GMP::Instance::SaveState* (page 409)
MatrixRestoreState	*GMP::Instance::RestoreState* (page 409)

10.3 Data Management via a Single Data Manager File

AIMMS supports the following functions for accessing the cases in the **Data Manager**; the chosen `Data_Management_style` is `single_data_manager_file`:

10.3.1 Cases

CaseCreate

The procedure *CaseCreate* (page 917) creates a new case node in the Data Management tree. The name of the case and the folder in which it is created is given as an argument to the function.

```
CaseCreate(
          case_path,     ! (input) scalar string expression
          case           ! (output) element parameter into AllCases
          )
```

Arguments

case_path A string expression holding the path and name of the new case. The path is specified relative to the root of the case tree.

case An element parameter into *AllCases* (page 842). On successful return this parameter will refer to the newly created element in *AllCases* (page 842).

Return Value

The procedure returns 1 if the case is created successfully. It returns 0 if the case could not be created or if the case already exists.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the specified path contains folders that do not exist, then these folders are created automatically. To check whether a specific case path already exists you can use the function `CaseFind`.
- If the option `Data_Management_style` is set to `disk_files_and_folders` there is no valid replacement.

See also:

The procedures *CaseFind* (page 918), *CaseDelete* (page 917).

CaseDelete

The procedure *CaseDelete* (page 917) deletes a specific case node from the Data Management tree.

```
CaseDelete(
          case     ! (input) element parameter into AllCases
          )
```

Arguments

case An element parameter into *AllCases* (page 842), representing the case that you want to delete.

Return Value

The procedure returns 1 if the case is deleted successfully, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *FileDelete* (page 770) instead.

See also:

The procedure *CaseFind* (page 918).

CaseFind

The procedure *CaseFind* (page 918) searches the Data Management tree for a case with a specific name.

```
CaseFind(
        case_path,     ! (input) scalar string expression
        case           ! (output) element parameter into AllCases
        )
```

Arguments

case_path A string expression holding the path and name of a case. The path is specified relative to the root of the case tree.

case An element parameter into *AllCases* (page 842). On successfull return this parameter will refer to the case found.

Return Value

The procedure returns 1 if the case is found, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders` there is no valid replacement.

See also:

The procedures *CaseCreate* (page 917), *CaseDelete* (page 917).

CaseGetChangedStatus

The function *CaseGetChangedStatus* (page 919) returns whether the data of the currently active case has changed and thus needs to be saved.

```
CaseGetChangedStatus
```

Arguments

None

Return Value

The function returns 1 if the data has changed, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *DataChangeMonitorHasChanged* (page 645) instead.

See also:

The functions *CaseSetChangedStatus* (page 930), *CaseSave* (page 925).

CaseGetDatasetReference

With the function *CaseGetDatasetReference* (page 919) you can, for every data category, obtain a reference to the dataset that is included in a specific case.

```
CaseGetDatasetReference(
        case,              ! (input) element from the set AllCases
        data_category,     ! (input) element from the set AllDataCategories
        dataset            ! (output) element parameter into AllDataSets
        )
```

Arguments

case An element in the set *AllCases* (page 842), refering to the case for which you want to retrieve the dataset reference.

data-category An element in the set *AllDataCategories* (page 844), refering to the specific data category for which you want to obtain the dataset reference.

dataset An element parameter into *AllDataSets* (page 845), on return this argument will contain the included dataset. It is set to the empty element if no dataset is included or if the dataset no longer exists.

Return Value

If any of the first two arguments does not refer to a valid case or data category, or if the data category is not part of the case type, then the function returns -1 and *CurrentErrorMessage* (page 835) will contain a proper error message. If a dataset is included, and this dataset still exists, then the function returns 1 and the argument *dataset* will refer to that dataset. If there is no dataset included, then the function returns 1 and *dataset* is set to the empty element. If a dataset is included, but this dataset has been deleted, then the function returns 0 and *dataset* is set to the empty element.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- You can use the functions `CaseGetType` and `CaseTypeCategories` to check whether a specific data category is part of a case.
- If the option `Data_Management_style` is set to `disk_files_and_folders` there is no valid replacement.

See also:

The functions *CaseGetType* (page 920), *CaseTypeCategories* (page 946).

CaseGetType

The procedure *CaseGetType* (page 920) retrieves the case type for a specific case.

```
CaseGetType(
        case,           ! (input) element of the set AllCases
        case_type       ! (output) element parameter into AllCaseTypes
        )
```

Arguments

case An element of the set *AllCases* (page 842), refering to the case for which you want to retrieve its case type.

case_type An element parameter into *AllCaseTypes* (page 843), on successfull return this argument will contain the case type for the given case.

Return Value

The procedure returns 1 on success, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseFileGetContentType* (page 628) instead.

CaseLoadCurrent

The procedure *CaseLoadCurrent* (page 921) loads an existing case as the new current case. You can use it to load either a case that is passed as argument to the procedure, or a case that the user can select via a dialog box. If the data of the currently loaded case has changed, then the user is asked to save this data first.

```
CaseLoadCurrent(
    case,    ! (input/output) An element parameter into AllCases
    [dialog],                    ! (optional) 0 or 1
    [keepUnreferencedRuntimeLibs  ! (optional) 0 or 1
)
```

Arguments

case An element parameter into the pre-defined set *AllCases* (page 842). If the argument *dialog* is set to 0, then no dialog box is shown and the case to which the element parameter currently refers is loaded. If the argument *dialog* is set to 1, then the value of the element parameter is used to initialize the dialog box. The case that the user has selected and is loaded successfully is returned through this argument.

dialog (optional) An integer value indicating whether or not the user gets a dialog box in which he can select the case to load. The default value is 1, thus if this argument is omitted the dialog box will be shown.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the case is loaded, but not referenced in the case, should be kept in memory or destroyed during the case load. The default is 0 indicating that the runtime libraries not referenced in the case should be destroyed during the case load.

Return Value

The procedure returns 1 on success. If the user cancelled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If you want to suppress the dialog box for the unsaved data, then you may call `CaseSetChangedStatus(0)` prior to *CaseLoadCurrent* (page 921).
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseCommandLoadAsActive* (page 634) instead.

See also:

The procedures *CaseLoadIntoCurrent* (page 922), *CaseMerge* (page 923), *CaseSave* (page 925), *CaseSetChangedStatus* (page 930).

CaseLoadIntoCurrent

The procedure *CaseLoadIntoCurrent* (page 922) loads the data of an existing case into the current case. You can use it to load either a case that is passed as argument to the procedure, or a case that the user can select via a dialog box. The data that is stored in the case will overwrite any data of the currently active case, and thus this current case is set to have changed data.

```
CaseLoadIntoCurrent(
    case,   ! (input/output) An element parameter into AllCases
    [dialog]                    ! (optional) 0 or 1
    [keepUnreferencedRuntimeLibs  ! (optional) 0 or 1
)
```

Arguments

case An element parameter into the pre-defined set *AllCases* (page 842). If the argument *dialog* is set to 0, then no dialog is shown and the case to which the element parameter currently refers is loaded. If the argument *dialog* is set to 1, then the value of the element parameter is used to initialize the dialog box. The case that the user has selected and is loaded successfully is returned through this argument.

dialog (optional) An integer value indicating whether or not the user gets a dialog box in which he can select the case to load. The default value is 1, thus if this argument is omitted the dialog box will be shown.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the case is loaded, but not referenced in the case, should be kept in memory or destroyed during the case load. The default is 0 indicating that the runtime libraries not referenced in the case should be destroyed during the case load.

Return Value

The procedure returns 1 on success. If the user cancelled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseCommandLoadIntoActive* (page 637) instead.

See also:

The procedures *CaseLoadCurrent* (page 921), *CaseMerge* (page 923), *CaseSave* (page 925), *CaseSetChangedStatus* (page 930).

CaseMerge

The procedure *CaseMerge* (page 923) merges the data of an existing case with the current data. You can use it to merge either a case that is passed as argument to the procedure, or a case that the user can select via a dialog box. Only the non-default data that is stored in the case will be merged with the data of the currently active case. This current case is set to have changed data.

```
CaseMerge(
    case,   ! (input/output) An element parameter into AllCases
    [dialog],                    ! (optional) 0 or 1
    [keepUnreferencedRuntimeLibs  ! (optional) 0 or 1
)
```

Arguments

case An element parameter into the pre-defined set *AllCases* (page 842). If the argument *dialog* is set to 0, then no dialog box is shown and the case to which the element parameter currently refers is merged. If the argument *dialog* is set to 1, then the value of the element parameter is used to initialize the dialog box. The case that the user has selected and is merged successfully is returned through this argument.

dialog (optional) An integer value indicating whether or not the user gets a dialog box in which he can select the case to merge. The default value is 1, thus if this argument is omitted the dialog box will be shown.

keepUnreferencedRuntimeLibs (optional) An integer value indicating whether or not any runtime libraries in existence before the case is merged, but not referenced in the case, should be kept in memory or destroyed during the case merge. The default is 1 indicating that the runtime libraries not referenced in the case will be retained during the case merge.

Return Value

The procedure returns 1 on success. If the user cancelled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseCommandMergeIntoActive* (page 637) instead.

See also:

The procedures *CaseLoadCurrent* (page 921), *CaseLoadIntoCurrent* (page 922), *CaseSave* (page 925), *CaseGetChangedStatus* (page 919).

CaseNew

The procedure *CaseNew* (page 924) starts a new case. The procedure is similar to the command **New Case** from the **Data** menu. The procedure does not change any of the current data, it only assures that there is no longer a current case. If you did have a current case and the data of this case has been changed, then AIMMS will ask whether or not this case should be saved first.

```
CaseNew
```

Arguments

None

Return Value

The procedure returns 1 on success. If the user cancelled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseCommandNew* (page 638) instead.
- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If you use *CaseNew* (page 924), then the name of this new case is not specified until you save the case. If you want to start a new named case, then you can use the following piece of code:

```
if ( CaseGetChangedStatus ) then
    if ( CaseSave = 0 ) then
        return ;
    endif ;
endif ;
if ( CaseSelectNew( a_case ) ) then
    CaseSetCurrent( a_case );
    CaseSetChangedStatus( a_case, 1 );
endif ;
```

See also:

The procedures *CaseLoadCurrent* (page 921), *CaseSave* (page 925), *CaseSelectNew* (page 929), *CaseSetCurrent* (page 931).

CaseSave

The procedure *CaseSave* (page 925) saves the data to the current case. If there is no current case, then the procedure behaves exactly as the `CaseSaveAs` procedure. If the case has active references to datasets that contain changed data, then these datasets are saved as well.

```
CaseSave(
        [confirm]       ! (optional) 0, 1 or 2
        )
```

Arguments

confirm (optional) An integer to indicate whether or not the procedure should ask for confirmation before overwriting the existing case. If 0, then no confirmation dialog box is shown. If 1 (default), then whether the confirmation dialog box is shown depends on the case type properties. If 2, then always a confirmation dialog box is shown.

Return Value

The procedure returns 1 if the case is saved successfully. It returns 0 if the user canceled the save operation. If any other error occurs, then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain an error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseCommandSave* (page 638) or *CaseFileSave* (page 625) instead.

See also:

The procedures *CaseSaveAs* (page 927), *CaseSaveAll* (page 926), *CaseLoadCurrent* (page 921), *CaseGetChangedStatus* (page 919).

CaseSaveAll

With the procedure *CaseSaveAll* (page 926) you can save (via a single call) the current case and all active datasets that need saving.

```
CaseSaveAll(
        [confirm]       ! (optional) integer value (0, 1 or 2)
        )
```

Arguments

confirm (optional) If 0, then cases and datasets are saved without confirmation. If 2, then AIMMS will display a dialog box for the cases and datasets that are about to be saved and ask for confirmation. If 1 (default), then AIMMS will use the properties of the case type and data categories to determine whether a confirmation dialog box should be displayed.

Return Value

The procedure returns 1 if the user chooses not to save the data or if the user chooses to save the data and the save was executed successfully. It returns 0 if the user cancelled any of the dialog boxes. If any other error occurs then the procedure returns −1 and `CurrentErrorMessage` will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- This function always returns 1 if the IDE is not loaded, for example when running the component version of AIMMS or when running with the command line option `--as-server`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseDialogConfirmAndSave* (page 640) and *CaseCommandSave* (page 638) instead.

See also:

The procedures *CaseSave* (page 925), *DatasetSave* (page 940).

CaseSaveAs

The procedure *CaseSaveAs* (page 927) shows a dialog box in which the user can specify a (new) case to which the data is saved. If the case has active references to datasets that contain changed data, then these datasets are saved as well. When saving these datasets the procedure will simply overwrite the current datasets, thus with *CaseSaveAs* (page 927) you can only change the current case and not any of the current datasets.

```
CaseSaveAs(
        case        ! (output) element parameter in AllCases
        )
```

Arguments

case An element parameter in *AllCases* (page 842). On return this parameter will refer to the case that the user selected.

Return Value

The procedure returns 1 if the case is saved successfully. It returns 0 if the user canceled the save operation. If any other error occurs, then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain an error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseCommandSaveAs* (page 639) instead.

See also:

The procedures *CaseSave* (page 925), *CaseSaveAll* (page 926), *CaseLoadCurrent* (page 921), *CaseGetChangedStatus* (page 919).

CaseSelect

The procedure *CaseSelect* (page 928) shows a dialog box in which the user can select an existing case.

```
CaseSelect(
        case,       ! (output) element parameter in AllCases
        [title]     ! (optional) string expression
        )
```

Arguments

case An element parameter in *AllCases* (page 842). On return the case will refer to the selected case.

title (optional) A string expression that is used as the title for the dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user did select a case. If the user presses **Cancel**, then the procedure returns 0.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseDialogSelectForLoad* (page 640) or *CaseDialogSelectForSave* (page 641) instead.

See also:

The procedure *CaseSelectNew* (page 929).

CaseSelectMultiple

The procedure *CaseSelectMultiple* (page 928) shows a dialog box in which the user can select a number of cases (and datasets). The selected subset of cases and datasets is stored in the pre-defined set `CurrentCaseSelection`, which is used in the page objects for which the property **Multiple Cases** is set.

```
CaseSelectMultiple(
        [cases_only]    ! (optional) 0 or 1
        )
```

Arguments

cases_only (optional) This argument controls whether the user can only select cases or can select both datasets and cases. If this argument is omitted, then the default value is 0, which means that both cases and datasets can be selected.

Return Value

The procedure returns 1 if the user pressed the **OK** button, and 0 if the user pressed **Cancel**.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseDialogSelectMultiple* (page 642) instead.

CaseSelectNew

The procedure *CaseSelectNew* (page 929) shows a dialog box in which the user can select a new case.

```
CaseSelect(
        case,       ! (output) element parameter in AllCases
        [title]     ! (optional) string expression
        )
```

Arguments

case An element parameter in *AllCases* (page 842). On return the case will refer to the selected case.

title (optional) A string expression that is used as the title for the dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user did select a case. If the user pressed **Cancel**, then the procedure returns 0.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If via this procedure the user creates a new case (i.e. a new case node in the data management tree), then this case does not yet contain any data. The case will only contain data after you explicitly save data to the case.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseDialogSelectForLoad* (page 640) or *CaseDialogSelectForSave* (page 641) instead.

See also:

The procedures *CaseSelect* (page 928), *CaseSetCurrent* (page 931), *CaseSave* (page 925).

CaseReadFromSingleFile

The procedure *CaseReadFromSingleFile* (page 930) reads the data from a single case file on disk.

```
CaseReadFromSingleFile(
        inputFileName               ! (input) scalar string expression
        )
```

Arguments

inputFileName A string expression holding the path and name of the input file.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The procedures *CaseWriteToSingleFile* (page 932), *CaseSave* (page 925).

CaseSetChangedStatus

The procedure *CaseSetChangedStatus* (page 930) can set the status of the current case to either changed or unchanged.

```
CaseSetChangedStatus(
        status,                 ! (input) 0 or 1
        [include_datasets]      ! (optional) 0 or 1
        )
```

Arguments

status An integer value holding the new case status: 0 for unchanged, 1 for changed.

include_datasets (optional) An integer to indicate whether or not the the status of the included and active datasets should be set as well. If you omit this argument, then the default value is 0 (status of datasets is not set).

Return Value

The procedure returns 1.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *DataChangeMonitorCreate* (page 643) or :aimms:func:` DataChangeMonitorReset` instead.

See also:

The procedures *CaseGetChangedStatus* (page 919), *DatasetSetChangedStatus* (page 944).

CaseSetCurrent

The procedure *CaseSetCurrent* (page 931) sets the case that is regarded as the current case. It does not load or save any data or checks whether data needs to be saved. You can, for example, use it to make a newly created case the current case, so that during a `CaseSave` the data is written to this case.

```
CaseSetCurrent(
        case                ! (input) element of the set AllCases
        )
```

Arguments

case An element of the set *AllCases* (page 842), refering to the case that should become the current case.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the option `Data_Management_style` is set to `disk_files_and_folders`, please use the function *CaseFileSetCurrent* (page 635) instead.

See also:

The procedures *CaseNew* (page 924), *CaseCreate* (page 917), *CaseSelectNew* (page 929), *CaseSave* (page 925).

CaseWriteToSingleFile

The procedure *CaseWriteToSingleFile* (page 932) writes the current data to a case file on disk.

```
CaseWriteToSingleFile(
        outputFileName                ! (input) scalar string expression
        )
```

Arguments

outputFileName A string expression holding the path and name of the output file.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- The procedure *CaseWriteToSingleFile* (page 932) uses the current case type to determine which data should be written. This is usually the case type of the last loaded case. If you want to make sure that a specific case type is used, you can preset the case type via the predefined element parameter `CurrentDefaultCaseType`.

The files written by *CaseWriteToSingleFile* (page 932) can only be read by `CaseReadFromSingleFile`.

See also:

The procedures *CaseReadFromSingleFile* (page 930), *CaseSave* (page 925).

10.3.2 Datasets

AIMMS supports the following functions for accessing the datasets in the **Data Manager**:

DatasetCreate

The procedure *DatasetCreate* (page 933) creates a new dataset node in the Data Management tree. The data category, the name of the dataset and the folder in which it is created is given as an argument to the procedure.

```
DatasetCreate(
        data_category,      ! (input) element in AllDataCategories
        dataset_path,       ! (input) scalar string expression
        dataset             ! (output) element parameter into AllDataSets
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which a dataset must be created.

dataset_path A string expression holding the path and name of the new dataset. The path is specified relative to the corresponding data category root node in the Data Management tree.

dataset An element parameter into *AllDataSets* (page 845). On successful return this parameter will refer to the newly created element in *AllDataSets* (page 845).

Return Value

The procedure returns 1 if the dataset is created successfully. It returns 0 if the dataset could not be created or if the dataset already exists.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If the specified path contains folders that do not exist, then these folders are created automatically. To check whether a specific dataset path already exists you can use the procedure `DatasetFind`.

See also:

The procedures *DatasetFind* (page 934), *DatasetDelete* (page 933).

DatasetDelete

The procedure *DatasetDelete* (page 933) deletes a specific dataset node from the Data Management tree.

```
DatasetDelete(
        data_category,    ! (input) element in AllDataCategories
        dataset           ! (input) element parameter into AllDataSets
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which a dataset is be deleted.

dataset An element parameter into *AllDataSets* (page 845), representing the dataset that you want to delete.

Return Value

The procedure returns 1 if the dataset is deleted successfully, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedure *DatasetFind* (page 934).

DatasetFind

The procedure *DatasetFind* (page 934) searches the Data Management tree for a dataset with a specific name and belonging to a specific data category.

```
DatasetFind(
        data_category,     ! (input) element in AllDataCategories
        dataset_path,      ! (input) scalar string expression
        dataset            ! (output) element parameter into AllDataSets
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which the datasets must be searched.

dataset_path A string expression holding the path and name of a dataset. The path is specified relative to the corresponding data category root node in the Data Management tree.

dataset An element parameter into *AllDataSets* (page 845). On successful return this parameter will refer to the dataset found.

Return Value

The procedure returns 1 if the dataset is found, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetCreate* (page 933), *DatasetDelete* (page 933).

DatasetGetCategory

The procedure *DatasetGetCategory* (page 935) retrieves the data category of a specific dataset.

```
DatasetGetCategory(
        dataset,        ! (input) element of the set AllDataSets
        data_category   ! (output) element parameter into AllDataCategories
        )
```

Arguments

dataset An element of the set *AllDataSets* (page 845), refering to the dataset for which you want to retrieve its data category.

data_category An element parameter into *AllDataCategories* (page 844), on successfull return this argument will contain the data category of the given dataset.

Return Value

The procedure returns 1 on success, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

DatasetGetChangedStatus

The function *DatasetGetChangedStatus* (page 936) returns whether the data associated with a specific data category has changed and thus needs to be saved.

```
DatasetGetChangedStatus(
        data_category          ! (input) element in AllDataCategories
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which the changed status must be retrieved.

Return Value

The function returns 1 if the data has changed, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The functions *DatasetSetChangedStatus* (page 944), *DatasetSave* (page 940).

DatasetLoadCurrent

The procedure *DatasetLoadCurrent* (page 936) loads an existing dataset as the new current dataset for a specific data category. You can use it to load either a dataset that is passed as argument to the procedure, or a dataset that the user can select via a dialog box. If the data of the corresponding data category has changed, then the user is asked to save this data first.

```
DatasetLoadCurrent(
       data_category, ! (input) element in AllDataCategories
       dataset,       ! (input/output) an element parameter into AllDataSets
       [dialog]       ! (optional) 0 or 1
       )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which a dataset is loaded.

dataset An element parameter in the set *AllDataSets* (page 845). If the argument *dialog* is set to 0, then no dialog box is shown and the dataset to which the element parameter currently refers is loaded. If the argument *dialog* is set to 1, then the value of the element parameter is used to initialize the dialog box. The dataset that the user has selected and is loaded successfully is returned through this argument.

dialog (optional) An integer value indicating whether or not the user gets a dialog box in which he can select the dataset to load. The default value is 1, thus if this argument is omitted the dialog box will be shown.

Return Value

The procedure returns 1 on success. If the user canceled the operation, then the procedure returns 0. If any other error occurs then the procedure returns -1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If you want to suppress the dialog box for the unsaved data, then you may call `DatasetSetChangedStatus(category,0)` prior to *DatasetLoadCurrent* (page 936).

See also:

The procedures *DatasetLoadIntoCurrent* (page 937), *DatasetMerge* (page 938), *DatasetSave* (page 940), *DatasetSetChangedStatus* (page 944).

DatasetLoadIntoCurrent

The procedure *DatasetLoadIntoCurrent* (page 937) loads the data of an existing dataset as the new current dataset for a specific data category. You can use it to load either a dataset that is passed as argument to the procedure, or a dataset that the user can select via a dialog box. The data that is stored in the dataset will overwrite any data of the currently active dataset, and thus this current dataset is set to have changed data.

```
DatasetLoadIntoCurrent(
       data_category,    ! (input) element in AllDataCategories
       dataset,          ! (input/output) an element parameter
                         ! into AllDataSets
       [dialog]          ! (optional) 0 or 1
       )
```

Arguments

category An element in *AllDataCategories* (page 844), specifying the data category for which a dataset is loaded.

dataset An element parameter in the set *AllDataSets* (page 845). If the argument *dialog* is set to 0, then no dialog box is shown and the dataset to which the element parameter currently refers is loaded. If the argument *dialog* is set to 1, then the value of the element parameter is used to initialize the dialog box. The dataset that the user has selected and is loaded successfully is returned through this argument.

dialog (optional) An integer value indicating whether or not the user gets a dialog box in which he can select the dataset to load. The default value is 1, thus if this argument is omitted the dialog box will be shown.

Return Value

The procedure returns 1 on success. If the user canceled the operation, then the procedure returns 0. If any other error occurs then the procedure returns -1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetLoadCurrent* (page 936), *DatasetMerge* (page 938), *DatasetSave* (page 940), *DatasetSetChangedStatus* (page 944).

DatasetMerge

The procedure *DatasetMerge* (page 938) merges the data of an existing dataset with the current data. You can use it to merge either a dataset that is passed as argument to the procedure, or a dataset that the user can select via a dialog box. Only the non-default data that is stored in the dataset will be merged with the current data.

```
DatasetMerge(
       data_category, ! (input) element in AllDataCategories
       dataset,       ! (input/output) an element parameter into AllDataSets
       [dialog]       ! (optional) 0 or 1
       )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which a dataset is loaded.

dataset An element parameter in the set *AllDataSets* (page 845). If the argument *dialog* is set to 0, then no dialog box is shown and the dataset to which the element parameter currently refers is loaded. If the argument *dialog* is set to 1, then the value of the element parameter is used to initialize the dialog box. The dataset that the user has selected and is loaded successfully is returned through this argument.

dialog (optional) An integer value indicating whether or not the user gets a dialog box in which he can select the dataset to load. The default value is 1, thus if this argument is omitted the dialog box will be shown.

Return Value

The procedure returns 1 on success. If the user cancelled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetLoadCurrent* (page 936), *DatasetLoadIntoCurrent* (page 937), *DatasetSave* (page 940), *DatasetGetChangedStatus* (page 936).

DatasetNew

The procedure *DatasetNew* (page 939) starts a new unnamed dataset for a specific data category. The procedure is similar to the command **Dataset New** from the **Data** menu. The procedure does not change any of the current data, it only sets the current dataset to unnamed. If you did have a currently named dataset and the data of this dataset has been changed, then AIMMS will ask whether or not this dataset should be saved first.

```
DatasetNew(
        data_category       ! (input) an element of AllDataCategories
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to start a new unnamed dataset.

Return Value

The procedure returns 1 on success. If the user cancelled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If you use CaseNew, then the name of this new case is not specified until you save the case. If you want to start a new named case, then you can use the following piece of code:

```
if ( CaseGetChangedStatus ) then
    if ( CaseSave = 0 ) then
        return ;
    endif ;
endif ;
if ( CaseSelectNew( a_case ) ) then
    CaseSetCurrent( a_case );
    CaseSetChangedStatus( a_case, 1 );
endif ;
```

See also:

The procedures *DatasetLoadCurrent* (page 936), *DatasetSave* (page 940), *DatasetSelectNew* (page 943), *DatasetSetCurrent* (page 945).

DatasetSave

The procedure *DatasetSave* (page 940) saves the data of a data category to the active dataset. If there is no named active dataset, then the procedure behaves exactly as the `DatasetSaveAs` procedure.

```
DatasetSave(
        data_category,    ! (input) element in AllDataCategories
        [confirm]         ! (optional) 0, 1 or 2
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to save the data.

confirm (optional) An integer to indicate whether or not the procedure should ask for confirmation before overwriting the existing dataset. If 0, then no confirmation dialog box is shown. If 1 (default), then whether or not the confirmation dialog box is shown depends on the case type properties. If 2, then always a confirmation dialog box is shown.

Return Value

The procedure returns 1 if the dataset is saved successfully. It returns 0 if the user canceled the save operation. If any other error occurs, then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain an error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetSaveAs* (page 942), *DatasetSaveAll* (page 941), *DatasetLoadCurrent* (page 936) and function *DatasetGetChangedStatus* (page 936).

DatasetSaveAll

The procedure *DatasetSaveAll* (page 941) saves the data of all data category to the active datasets. If there are no named active datasets, then the procedure behaves according to the `DatasetSaveAs` procedure.

```
DatasetSaveAll(
        [confirm]       ! (optional) 0, 1 or 2
        )
```

Arguments

confirm (optional) An integer to indicate whether or not the procedure should ask for confirmation before overwriting the existing datasets. If 0, then no confirmation dialog box is shown. If 1 (default), then whether or not the confirmation dialog box is shown depends on the case type properties. If 2, then always a confirmation dialog box is shown.

Return Value

The procedure returns 1 if the datasets are saved successfully. It returns 0 if the user canceled the save operation. If any other error occurs, then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain an error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetSaveAs* (page 942), *DatasetSave* (page 940), *DatasetLoadCurrent* (page 936), *DatasetGetChangedStatus* (page 936).

DatasetSaveAs

The procedure *DatasetSaveAs* (page 942) shows a dialog box in which the user can specify a (new) dataset to which the data is saved.

```
DatasetSaveAs(
       data_category,    ! (input) element in AllDataCategories
       dataset           ! (output) element parameter in AllDataSets
       )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to save the data.

dataset An element parameter in *AllDataSets* (page 845). On return this parameter will refer to the dataset that the user selected.

Return Value

The procedure returns 1 if the dataset is saved successfully. It returns 0 if the user cancelled the save operation. If any other error occurs, then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain an error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetSave* (page 940), *DatasetSaveAll* (page 941), *DatasetLoadCurrent* (page 936), *DatasetGetChangedStatus* (page 936).

DatasetSelect

The procedure *DatasetSelect* (page 942) shows a dialog box in which the user can select an existing dataset for a given data category.

```
DatasetSelect(
        data_category,    ! (input) element in AllDataCategories
        dataset,          ! (output) element parameter in AllDataSets
        [title]           ! (optional) string expression
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to the user to select a dataset.

dataset An element parameter in *AllDataSets* (page 845). On return the dataset will refer to the selected dataset.

title (optional) A string expression that is used as the title for the dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user did select a dataset. If the user pressed **Cancel**, then the procedure returns 0.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedure *DatasetSelectNew* (page 943).

DatasetSelectNew

The procedure *DatasetSelectNew* (page 943) shows a dialog box in which the user can select a new dataset for a given data category.

```
DatasetSelectNew(
        data_category,    ! (input) element in AllDataCategories
        dataset,          ! (output) element parameter in AllDataSets
        [title]           ! (optional) string expression
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to the user to select a new dataset.

dataset An element parameter in *AllDataSets* (page 845). On return the dataset will refer to the selected dataset.

title (optional) A string expression that is used as the title for the dialog box. If this argument is omitted, then a default title is used.

Return Value

The procedure returns 1 if the user did select a dataset. If the user pressed **Cancel**, then the procedure returns 0.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If via this procedure the user creates a new dataset (i.e. a new dataset node in the data management tree), then this case dataset does not yet contain any data. The dataset will only contain data after you explicitly save data to it.

See also:

The procedures *DatasetSelect* (page 942), *DatasetSetCurrent* (page 945), *DatasetSave* (page 940).

DatasetSetChangedStatus

The procedure *DatasetSetChangedStatus* (page 944) can set the status of a data category to either changed or unchanged.

```
DatasetSetChangedStatus(
        data_category,    ! (input) element in AllDataCategories
        status            ! (input) 0 or 1
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to set the changed status.

status An integer value holding the new dataset status: 0 for unchanged, 1 for changed.

Return Value

The procedure returns 1.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The function *DatasetGetChangedStatus* (page 936).

DatasetSetCurrent

The procedure *DatasetSetCurrent* (page 945) sets the dataset that is regarded as the current dataset for a given data category. It does not load or save any data or checks whether data needs to be saved. You can, for example, use it to make a newly created dataset the current dataset, so that during a DatasetSave the data is written to this dataset.

```
DatasetSetCurrent(
        data_category,     ! (input) element in AllDataCategories
        dataset            ! (input) element of the set AllDataSets
        )
```

Arguments

data_category An element in *AllDataCategories* (page 844), specifying the data category for which you want to set the current dataset.

dataset An element of the set *AllDataSets* (page 845), refering to the dataset that should become the current dataset.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DatasetNew* (page 939), *DatasetCreate* (page 933), *DatasetSelectNew* (page 943), *DatasetSave* (page 940).

10.3.3 Data Manager Files

AIMMS supports the following Data Manager functions, that are not specific for cases or datasets only:

CaseTypeCategories

The procedure *CaseTypeCategories* (page 946) retrieves the sub-collection of data categories that is included in a specific case type.

```
CaseTypeCategories(
        case_type,          ! (input) element of the set AllCaseTypes
        category_set        ! (output) subset of AllDataCategories
        )
```

Arguments

case_type An element of the set *AllCaseTypes* (page 843), refering to the case type for which you want to retrieve the included data categories.

category_set A subset of the set *AllDataCategories* (page 844), on successfull return this subset is filled with the data categories included in the case type.

Return Value

The procedure returns 1 on success, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *CaseGetType* (page 920), *CaseTypeContents* (page 946), *DataCategoryContents* (page 947).

CaseTypeContents

The procedure *CaseTypeContents* (page 946) retrieves the sub-collection of identifiers that is contained in a specific case type.

```
CaseTypeContents(
        case_type,          ! (input) element of the set AllCaseTypes
        identifier_set      ! (output) subset of AllIdentifiers
        )
```

Arguments

case_type An element of the set *AllCaseTypes* (page 843), refering to the case type for which you want to retrieve the contents.

identifier_set A subset of the set *AllIdentifiers* (page 821), on successful return this subset is filled with all identifiers contained in the case type.

Return Value

The procedure returns 1 on success, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- The procedure returns the contents of the case type itself, as well as the contents of all data categories that are included in the case type.

See also:

The procedures *CaseGetType* (page 920), *CaseTypeCategories* (page 946), *DataCategoryContents* (page 947).

DataCategoryContents

The procedure *DataCategoryContents* (page 947) retrieves the sub-collection of identifiers that is contained in a specific data category.

```
DataCategoryContents(
        data_category,       ! (input) element of the set AllDataCategories
        identifier_set       ! (output) subset of AllIdentifiers
        )
```

Arguments

data_category An element of the set *AllDataCategories* (page 844), refering to the data category for which you want to retrieve the contents.

identifier_set A subset of the set *AllIdentifiers* (page 821), on successful return this subset is filled with all identifiers contained in the data category.

Return Value

The procedure returns 1 on success, 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *CaseTypeCategories* (page 946), *CaseTypeContents* (page 946).

DataFileCopy

With the procedure *DataFileCopy* (page 948) you can copy a data file stored within a data manager file, to another data file within the same data manager file.

```
DataFileCopy(
        datafile,      ! (input) element in the set AllDataFiles
        acronym,       ! (input) string
        copiedDatafile ! (output) element parameter into AllDataFiles
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845), *AllCases* (page 842) or *AllDataSets* (page 845).

acronym The name of the new data file to be created

copiedDatafile On success, contains the element in *AllDataFiles* (page 845) associated with the datafile into which the original data file was copied.

Return Value

The procedure returns 1 if the data file has been copied, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- If a datafile with the given acronym already exists in the data manager file, the call to *DataFileCopy* (page 948) will fail.

DataFileExists

With the procedure *DataFileExists* (page 948) you can check whether a specific element from the set *AllDataFiles* (page 845) still refers to a valid case or dataset. Especially when multiple users have access to the same data file, an element may become invalid.

```
DataFileExists(
        datafile      ! (input) element in the set AllDataFiles
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845), *AllCases* (page 842) or *AllDataSets* (page 845).

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- Note that *AllCases* (page 842) and *AllDataSets* (page 845) are subsets of *AllDataFiles* (page 845).

See also:

The procedure *DataFileGetName* (page 952).

DataFileGetAcronym

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileGetAcronym* (page 949) you can obtain the acronym that is specified in the data manager for any element of the set *AllDataFiles* (page 845) (cases or datasets).

```
DataFileGetAcronym(
        datafile,     ! (input) element in the set AllDataFiles
        acronym       ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

acronym A scalar string valued parameter. On return this parameter will contain the acronym of the datafile. If no acronym is specified, then an empty string is returned.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetName* (page 952).

DataFileGetComment

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileGetComment* (page 950) you can obtain the comment that is specified in the data manager for any element of the set *AllDataFiles* (page 845) (cases or datasets).

```
DataFileGetComment(
        datafile,      ! (input) element in the set AllDataFiles
        comment        ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

comment A scalar string valued parameter. On return this parameter will contain the comment of the datafile. If no comment is specified, then an empty string is returned.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetName* (page 952).

DataFileGetDescription

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileGetDescription* (page 951) you can obtain the description that the user entered via the properties of a case or dataset.

```
DataFileGetDescription(
        datafile,       ! (input) element in the set AllDataFiles
        description     ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

name A scalar string valued parameter. On return this parameter will contain the description of the datafile. If no description has been specified, then this string is empty.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetName* (page 952), *DataFileGetAcronym* (page 949).

DataFileGetGroup

With the procedure *DataFileGetGroup* (page 951) you can obtain the group name associated with the user that currently owns a specific case or dataset.

```
DataFileGetGroup(
        datafile,       ! (input) element in the set AllDataFiles
        group           ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

group A scalar string valued parameter. On return this parameter will contain the group name associated with the user that owns the datafile. If there is no current owner, or if the project does not have a user database associated with it, then an empty string is returned.

Return Value

The procedure returns 1 if the given datafile exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetOwner* (page 953).

DataFileGetName

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileGetName* (page 952) you can obtain the name in the data manager for any element of the set *AllDataFiles* (page 845) (cases or datasets).

```
DataFileGetName(
        datafile,     ! (input) element in the set AllDataFiles
        name          ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

name A scalar string valued parameter. On return this parameter will contain the name of the datafile. This name does not include the name of the folder(s) in which it is located.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetPath* (page 953), *DataFileGetAcronym* (page 949).

DataFileGetOwner

With the procedure *DataFileGetOwner* (page 953) you can obtain the name of the user that currently owns a specific case or dataset.

```
DataFileGetOwner(
        datafile,       ! (input) element in the set AllDataFiles
        owner           ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

owner A scalar string valued parameter. On return this parameter will contain the name of the user that owns the datafile. If there is no current owner, or if the project does not have a user database associated with it, then an empty string is returned.

Return Value

The procedure returns 1 if the given datafile exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetGroup* (page 951).

DataFileGetPath

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileGetPath* (page 953) you can obtain the path in the data manager for any element of the set *AllDataFiles* (page 845) (cases or datasets). The path of a datafile consists of a sequence folder names and the name of the datafile itself, separated by backslash characters.

```
DataFileGetPath(
        datafile,      ! (input) element in the set AllDataFiles
        path           ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

path A scalar string valued parameter. On return this parameter will contain the path of the datafile.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetName* (page 952), *DataFileGetAcronym* (page 949).

DataFileGetTime

With the procedure *DataFileGetTime* (page 954) you can obtain the time on which the data of a specific case or dataset was last modified (saved).

```
DataFileGetTime(
        datafile,      ! (input) element in the set AllDataFiles
        time           ! (output) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

time A scalar string valued parameter. On return this parameter will contain a string representation of the modification time, using AIMMS' standard date and time format: "YYYY-MM-DD hh:mm:ss".

Return Value

The procedure returns 1 if the given datafile exists and contains saved data. If the datafile does not exist, or if no data has yet been saved in the datafile, then the procedure returns 0.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *FileTime* (page 776).

DataFileReadPermitted

With the procedure *DataFileReadPermitted* (page 955) you can check whether the current user has read permission for the specified case or dataset. For example, you can use this procedure to issue your own error message if the permission is not granted. If the current user does not have read permission, then any call to a data manager procedure that involves a read operation will result in an error message, and fails.

```
DataFileReadPermitted(
        datafile      ! (input) element in the set AllDataFiles
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

Return Value

The procedure returns 1 if the current user does have read permission, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedure *DataFileWritePermitted* (page 957).

DataFileSetAcronym

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileSetAcronym* (page 955) you can set the acronym for the data file corresponding to any element of the set *AllDataFiles* (page 845) (cases or datasets).

```
DataFileSetAcronym(
        datafile,      ! (input) element in the set AllDataFiles
        acronym        ! (input) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

acronym A scalar string valued parameter. This parameter contains the acronym to be associated with the datafile. If an empty string is specified, any existing acronym will be deleted.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetAcronym* (page 949).

DataFileSetComment

The predefined set *AllDataFiles* (page 845) (and its subsets *AllCases* (page 842) and *AllDataSets* (page 845)), is an integer set. The mapping of these integers onto the cases and datasets in the project is maintained by the data manager, and is not editable. With the procedure *DataFileSetComment* (page 956) you can set the comment for the data file corresponding to any element of the set *AllDataFiles* (page 845) (cases or datasets).

```
DataFileSetComment(
        datafile,      ! (input) element in the set AllDataFiles
        comment        ! (input) scalar string parameter
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

comment A scalar string valued parameter. This parameter contains the comment to be associated with the datafile. If an empty string is specified, any existing comment will be deleted.

Return Value

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataFileExists* (page 948), *DataFileGetComment* (page 950).

DataFileWritePermitted

With the procedure *DataFileWritePermitted* (page 957) you can check whether the current user has write permission for the specified case or dataset. For example, you can use this procedure to issue your own error message if the permission is not granted. If the current user does not have write permission, then any call to a data manager procedure that involves a write (save) operation will result in an error message, and fails.

```
DataFileWritePermitted(
        datafile      ! (input) element in the set AllDataFiles
        )
```

Arguments

datafile An element in the set *AllDataFiles* (page 845).

Return Value

The procedure returns 1 if the current user does have write permission, and 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedure *DataFileReadPermitted* (page 955).

DataImport220

With the procedure *DataImport220* (page 957) you can load a separate AIMMS case file, such as the case files that were created with AIMMS 2.20. After importing a case file using this procedure you can save the data as a new case node in the Data Management tree.

```
DataImport220(
        filename        ! (input/output) a string parameter
        )
```

Arguments

filename A string parameter, that on return will contain the name of the file that the user selected for importing.

Return Value

The procedure returns 1 on success. If the user canceled the operation, then the procedure returns 0. If any other error occurs then the procedure returns −1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This procedure is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.
- This procedure is especially useful for converting old cases to the new AIMMS.

See also:

The procedure *CaseSaveAs* (page 927).

DataManagerFileNew

With the procedure *DataManagerFileNew* (page 958) you can create a new, empty data file. On success, the new data file will be used as the current data file for the project.

```
DataManagerFileNew(
        filename,       ! (input) a scalar string expression
        [UseAsDefault]  ! (optional, default 1) a scalar binary expression
        )
```

Arguments

filename A string containing the name of the new data file (relative to the project directory)

UseAsDefault A binary value to indicate whether the new data file should be used as the default data file the next time the project is opened (value 1) or not (value 0).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataManagerFileOpen* (page 959), *DataManagerFileGetCurrent* (page 960).

DataManagerFileOpen

With the procedure *DataManagerFileOpen* (page 959) you can open an existing data file. On success, the data file will be used as the current data file for the project.

```
DataManagerFileOpen(
        filename,          ! (input) a scalar string expression
        [UseAsDefault]     ! (optional, default 1) a scalar binary expression
        )
```

Arguments

filename A string containing the name of the existing data file (relative to the project directory).

UseAsDefault A binary value to indicate whether the data file should be used as the default data file the next time the project is opened (value 1) or not (value 0).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataManagerFileNew* (page 958), *DataManagerFileGetCurrent* (page 960).

DataManagerExport

With the procedure *DataManagerExport* (page 959) you can export a collection of cases and datasets from the data management tree to a new data file.

```
DataManagerExport(
        filename,       ! (input) a scalar string expression
        datafiles       ! (input/output) a subset of AllDataFiles
        )
```

Arguments

filename A string containing the name of the data file to which the cases and datasets must be exported.

datafiles A subset of *AllDataFiles* (page 845), containing the cases and datasets that you want to export. Any dataset that is referred to by a case in this set is automatically added to the set.

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedure *DataManagerImport* (page 961).

DataManagerFileGetCurrent

With the procedure *DataManagerFileGetCurrent* (page 960) you can obtain the name of the current data file.

```
DataManagerFileGetCurrent(
        filename          ! (output) a scalar string
        )
```

Arguments

filename A string to contain the name of the current data file (relative to the project directory).

Return Value

The procedure returns 1 on success, or 0 otherwise.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures *DataManagerFileNew* (page 958), *DataManagerFileOpen* (page 959).

DataManagerImport

With the procedure *DataManagerImport* (page 961) you can import the entire data management tree that is stored in another data file into your current data management tree. If the imported tree contains cases (or datasets) that already exist in the current tree, then you can choose whether these cases (or datasets) should overwrite the current nodes or should be imported as new nodes.

```
DataManagerImport(
        filename,       ! (input) a scalar string expression
        [overwrite]     ! (optional) 0, 1 or 2
        )
```

Arguments

filename A string containing the name of the data file that must be imported.

overwrite (optional) This integer indicates whether or not existing cases (or datasets) are overwritten by cases (or datasets) from the imported file. If 0 (the default), then a dialog box is displayed in which the user can decide to overwrite the existing node or to create a new node. If 1, then existing nodes are always overwritten. If 2, then all imported cases and datasets will result in new nodes in the tree.

Return Value

The procedure returns 1 on success. If the user canceled the operation, then the procedure returns 0. If any other error occurs then the procedure returns -1 and *CurrentErrorMessage* (page 835) will contain a proper error message.

Note:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedure *DataManagerExport* (page 959).

10.4 Deprecated AIMMS 220 Functions

AIMMS supports the following deprecated functions originating from AIMMS 220.

10.4.1 ListingFileCopy

With the procedure *ListingFileCopy* (page 962) you can copy the current contents of the listing file to a given file.

```
ListingFileCopy(
     toFileName,          ! (input) string expression
     overwrite            ! (optional) default 1.
     )
```

Arguments

toFileName The file name of the file to which the contents of the listing file must be copied.

overwrite if equal to 0 then do not overwrite an existing file, otherwise overwrite an existing file when needed.

Return Value

The procedure returns 1 on success, or 0 otherwise.

See also:

The procedure *ListingFileDelete* (page 962).

10.4.2 ListingFileDelete

The function *ListingFileDelete* (page 962) deletes the current contents of the listing file associated with an AIMMS project.

```
ListingFileDelete()
```

Return Value

The function returns 1 on success, or 0 otherwise.

See also:

The function *ListingFileCopy* (page 962).

INDEX

A

Abs() (*built-in function*), 1
ActiveCard() (*built-in function*), 45
Aggregate() (*procedure*), 91
AggregationTypes() (*set*), 792
AimmsRevisionString() (*procedure*), 752
AimmsStringConstants() (*set*), 791
AllAbbrMonths() (*set*), 851
AllAbbrWeekdays() (*set*), 851
AllAimmsStringConstantElements() (*set*), 791
AllAssertions() (*set*), 816
AllAttributeNames() (*set*), 793
AllAuthorizationLevels() (*set*), 779
AllAvailableCharacterEncodings() (*set*), 779
AllBasicValues() (*set*), 794
AllCaseComparisonModes() (*set*), 794
AllCaseFileContentTypes() (*set*), 848
AllCases() (*set*), 842
AllCaseTypes() (*set*), 843
AllCharacterEncodings() (*set*), 782
AllColors() (*set*), 784
AllColumnTypes() (*set*), 795
AllConstraintProgrammingRowTypes() (*set*), 804
AllConstraints() (*set*), 817
AllConventions() (*set*), 817
AllDatabaseTables() (*set*), 818
AllDataCategories() (*set*), 844
AllDataColumnCharacteristics() (*set*), 795
AllDataFiles() (*set*), 845
AllDataSets() (*set*), 845
AllDataSourceProperties() (*set*), 796
AllDefinedParameters() (*set*), 819
AllDefinedSets() (*set*), 819
AllDifferencingModes() (*set*), 797
AllExecutionStatuses() (*set*), 798
AllFileAttributes() (*set*), 799
AllFiles() (*set*), 820
AllFunctions() (*set*), 820
AllGeneratedMathematicalPrograms() (*set*), 833
AllGMPEvents() (*set*), 821
AllGMPExtensions() (*set*), 798
AllIdentifiers() (*set*), 821
AllIdentifierTypes() (*set*), 800
AllIndices() (*set*), 822
AllIntegerVariables() (*set*), 823
AllIntrinsics() (*set*), 785
AllIsolationLevels() (*set*), 801
AllKeywords() (*set*), 785
AllMacros() (*set*), 823
AllMathematicalProgrammingRowTypes() (*set*), 806
AllMathematicalProgrammingTypes() (*set*), 802
AllMathematicalPrograms() (*set*), 824
AllMatrixManipulationDirections() (*set*), 802
AllMatrixManipulationProgrammingTypes() (*set*), 803
AllMonths() (*set*), 852
AllNonLinearConstraints() (*set*), 824
AllOptions() (*set*), 786
AllParameters() (*set*), 825
AllPredeclaredIdentifiers() (*set*), 786
AllProcedures() (*set*), 825
AllProfilerTypes() (*set*), 804
AllProgressCategories() (*set*), 834
AllQuantities() (*set*), 826
AllRowColumnStatuses() (*set*), 805
AllRowTypes() (*set*), 805
AllSections() (*set*), 827
AllSets() (*set*), 827
AllSolutionStates() (*set*), 807
AllSolverInterrupts() (*set*), 807
AllSolvers() (*set*), 787
AllSolverSessionCompletionObjects() (*set*), 828
AllSolverSessions() (*set*), 828
AllStochasticConstraints() (*set*), 829
AllStochasticGenerationModes() (*set*), 808
AllStochasticParameters() (*set*), 829
AllStochasticScenarios() (*set*), 834
AllStochasticVariables() (*set*), 830
AllSuffixNames() (*set*), 809
AllSymbols() (*set*), 787
AllTimeZones() (*set*), 853
AllUpdatableIdentifiers() (*set*), 831
AllValueKeywords() (*set*), 809
AllVariables() (*set*), 831

AllVariablesConstraints() (*set*), 832
AllViolationTypes() (*set*), 810
AllWeekdays() (*set*), 853
ArcCos() (*built-in function*), 1
ArcCosh() (*built-in function*), 3
ArcSin() (*built-in function*), 4
ArcSinh() (*built-in function*), 6
ArcTan() (*built-in function*), 7
ArcTanh() (*built-in function*), 9
ASCIICharacterEncodings() (*set*), 780
ASCIIUnicodeCharacterEncodings() (*set*), 781
AtomicUnit() (*built-in function*), 84
AttributeContainsString() (*built-in function*), 575
AttributeLength() (*built-in function*), 574
AttributeToString() (*built-in function*), 573

B

Beta() (*built-in function*), 225
Binomial() (*built-in function*), 215

C

CallerAttribute() (*built-in function*), 577
CallerLine() (*built-in function*), 578
CallerNode() (*built-in function*), 579
CallerNumberOfLocations() (*built-in function*), 580
Card() (*built-in function*), 46
CaseCommandLoadAsActive() (*procedure*), 634
CaseCommandLoadIntoActive() (*procedure*), 637
CaseCommandMergeIntoActive() (*procedure*), 637
CaseCommandNew() (*procedure*), 638
CaseCommandSave() (*procedure*), 638
CaseCommandSaveAs() (*procedure*), 639
CaseCompareIdentifier() (*built-in function*), 623
CaseCreate() (*procedure*), 917
CaseCreateDifferenceFile() (*built-in function*), 626
CaseDelete() (*procedure*), 917
CaseDialogConfirmAndSave() (*procedure*), 640
CaseDialogSelectForLoad() (*procedure*), 640
CaseDialogSelectForSave() (*procedure*), 641
CaseDialogSelectMultiple() (*procedure*), 642
CaseFileGetContentType() (*procedure*), 628
CaseFileLoad() (*procedure*), 622
CaseFileMerge() (*procedure*), 624
CaseFileSave() (*procedure*), 625
CaseFileSectionExists() (*procedure*), 629
CaseFileSectionGetContentType() (*procedure*), 630
CaseFileSectionLoad() (*procedure*), 631
CaseFileSectionMerge() (*procedure*), 632
CaseFileSectionRemove() (*built-in function*), 633
CaseFileSectionSave() (*procedure*), 633
CaseFileSetCurrent() (*procedure*), 635
CaseFileURL() (*set*), 850
CaseFileURLtoElement() (*procedure*), 636
CaseFind() (*procedure*), 918
CaseGetChangedStatus() (*built-in function*), 919
CaseGetDatasetReference() (*built-in function*), 919
CaseGetType() (*procedure*), 920
CaseLoadCurrent() (*procedure*), 921
CaseLoadIntoCurrent() (*procedure*), 922
CaseMerge() (*procedure*), 923
CaseNew() (*procedure*), 924
CaseReadFromSingleFile() (*procedure*), 930
CaseSave() (*procedure*), 925
CaseSaveAll() (*procedure*), 926
CaseSaveAs() (*procedure*), 927
CaseSelect() (*procedure*), 928
CaseSelectMultiple() (*procedure*), 928
CaseSelectNew() (*procedure*), 929
CaseSetChangedStatus() (*procedure*), 930
CaseSetCurrent() (*procedure*), 931
CaseTypeCategories() (*procedure*), 946
CaseTypeContents() (*procedure*), 946
CaseWriteToSingleFile() (*procedure*), 932
Ceil() (*built-in function*), 10
Character() (*built-in function*), 69
CharacterNumber() (*built-in function*), 69
CloneElement() (*procedure*), 48
CloseDataSource() (*built-in function*), 646
Combination() (*built-in function*), 252
CommitTransaction() (*procedure*), 647
ConstraintVariables() (*built-in function*), 582
ContinueAbort() (*set*), 811
ConvertReferenceDate() (*built-in function*), 92
ConvertUnit() (*built-in function*), 86
Cos() (*built-in function*), 11
Cosh() (*built-in function*), 12
cp::ActivityBegin() (*cp method*), 283
cp::ActivityEnd() (*cp method*), 284
cp::ActivityLength() (*cp method*), 285
cp::ActivitySize() (*cp method*), 286
cp::AllDifferent() (*cp method*), 263
cp::Alternative() (*cp method*), 287
cp::BeginAtBegin() (*cp method*), 288
cp::BeginAtEnd() (*cp method*), 289
cp::BeginBeforeBegin() (*cp method*), 290
cp::BeginBeforeEnd() (*cp method*), 291
cp::BeginOfNext() (*cp method*), 291
cp::BeginOfPrevious() (*cp method*), 292
cp::BinPacking() (*cp method*), 265
cp::Cardinality() (*cp method*), 268
cp::Channel() (*cp method*), 270
cp::Count() (*cp method*), 272
cp::EndAtBegin() (*cp method*), 293
cp::EndAtEnd() (*cp method*), 294
cp::EndBeforeBegin() (*cp method*), 295
cp::EndBeforeEnd() (*cp method*), 295
cp::EndOfNext() (*cp method*), 296

cp::EndOfPrevious() (*cp method*), 297
cp::GroupOfNext() (*cp method*), 298
cp::GroupOfPrevious() (*cp method*), 298
cp::LengthOfNext() (*cp method*), 299
cp::LengthOfPrevious() (*cp method*), 300
cp::Lexicographic() (*cp method*), 274
cp::ParallelSchedule() (*cp method*), 276
cp::Sequence() (*cp method*), 278
cp::SequentialSchedule() (*cp method*), 280
cp::SizeOfNext() (*cp method*), 301
cp::SizeOfPrevious() (*cp method*), 302
cp::Span() (*cp method*), 302
cp::Synchronize() (*cp method*), 303
CreateTimeTable() (*procedure*), 94
Cube() (*built-in function*), 14
CurrentAuthorizationLevel() (*set*), 788
CurrentAutoUpdatedDefinitions() (*set*), 835
CurrentCase() (*set*), 846
CurrentCaseFileContentType() (*set*), 849
CurrentCaseSelection() (*set*), 847
CurrentDataSet() (*set*), 847
CurrentDefaultCaseType() (*set*), 849
CurrentErrorMessage() (*set*), 835
CurrentFile() (*set*), 836
CurrentFileName() (*set*), 837
CurrentGroup() (*set*), 789
CurrentInputs() (*set*), 837
CurrentMatrixBlockSizes() (*set*), 838
CurrentMatrixColumnCount() (*set*), 839
CurrentMatrixRowCount() (*set*), 840
CurrentPageNumber() (*set*), 840
CurrentSolver() (*set*), 790
CurrentToMoment() (*built-in function*), 96
CurrentToString() (*built-in function*), 97
CurrentToTimeSlot() (*built-in function*), 97
CurrentUser() (*set*), 792

D

DataCategoryContents() (*procedure*), 947
DataChangeMonitorCreate() (*built-in function*), 643
DataChangeMonitorDelete() (*built-in function*), 644
DataChangeMonitorHasChanged() (*built-in function*), 645
DataChangeMonitorReset() (*built-in function*), 646
DataFileCopy() (*procedure*), 948
DataFileExists() (*procedure*), 948
DataFileGetAcronym() (*procedure*), 949
DataFileGetComment() (*procedure*), 950
DataFileGetDescription() (*procedure*), 951
DataFileGetGroup() (*procedure*), 951
DataFileGetName() (*procedure*), 952
DataFileGetOwner() (*procedure*), 953
DataFileGetPath() (*procedure*), 953
DataFileGetTime() (*procedure*), 954
DataFileReadPermitted() (*procedure*), 955
DataFileSetAcronym() (*procedure*), 955
DataFileSetComment() (*procedure*), 956
DataFileWritePermitted() (*procedure*), 957
DataImport220() (*procedure*), 957
DataManagementExit() (*procedure*), 643
DataManagerExport() (*procedure*), 959
DataManagerFileGetCurrent() (*procedure*), 960
DataManagerFileNew() (*procedure*), 958
DataManagerFileOpen() (*procedure*), 959
DataManagerImport() (*procedure*), 961
DatasetCreate() (*procedure*), 933
DatasetDelete() (*procedure*), 933
DatasetFind() (*procedure*), 934
DatasetGetCategory() (*procedure*), 935
DatasetGetChangedStatus() (*built-in function*), 936
DatasetLoadCurrent() (*procedure*), 936
DatasetLoadIntoCurrent() (*procedure*), 937
DatasetMerge() (*procedure*), 938
DatasetNew() (*procedure*), 939
DatasetSave() (*procedure*), 940
DatasetSaveAll() (*procedure*), 941
DatasetSaveAs() (*procedure*), 942
DatasetSelect() (*procedure*), 942
DatasetSelectNew() (*procedure*), 943
DatasetSetChangedStatus() (*procedure*), 944
DatasetSetCurrent() (*procedure*), 945
DateDifferenceDays() (*built-in function*), 122
DateDifferenceYearFraction() (*built-in function*), 124
DaylightSavingEndDate() (*built-in function*), 99
DaylightSavingStartDate() (*built-in function*), 100
DebuggerBreakPoint() (*built-in function*), 719
DeclaredSubset() (*built-in function*), 583
Degrees() (*built-in function*), 15
Delay() (*built-in function*), 756
DepreciationLinearLife() (*built-in function*), 127
DepreciationLinearRate() (*built-in function*), 129
DepreciationNonLinearFactor() (*built-in function*), 133
DepreciationNonLinearLife() (*built-in function*), 135
DepreciationNonLinearRate() (*built-in function*), 137
DepreciationNonLinearSumOfYear() (*built-in function*), 131
DepreciationSum() (*built-in function*), 140
DialogAsk() (*procedure*), 687
DialogError() (*built-in function*), 688
DialogGetColor() (*procedure*), 688
DialogGetDate() (*procedure*), 689
DialogGetElement() (*procedure*), 689
DialogGetElementByData() (*procedure*), 690
DialogGetElementByText() (*procedure*), 691

DialogGetNumber() (*procedure*), 691
DialogGetPassword() (*procedure*), 692
DialogGetString() (*procedure*), 693
DialogMessage() (*built-in function*), 693
DialogProgress() (*built-in function*), 694
DirectoryCopy() (*procedure*), 760
DirectoryCreate() (*procedure*), 761
DirectoryDelete() (*procedure*), 761
DirectoryExists() (*procedure*), 762
DirectoryGetCurrent() (*procedure*), 763
DirectoryGetFiles() (*procedure*), 763
DirectoryGetSubdirectories() (*procedure*), 765
DirectoryMove() (*procedure*), 766
DirectoryOfLibraryProject() (*built-in function*), 767
DirectorySelect() (*procedure*), 767
DirectSQL() (*procedure*), 647
DisAggregate() (*procedure*), 101
DiskWindowVoid() (*set*), 811
DistributionCumulative() (*built-in function*), 247
DistributionDensity() (*built-in function*), 248
DistributionDeviation() (*built-in function*), 248
DistributionInverseCumulative() (*built-in function*), 249
DistributionInverseDensity() (*built-in function*), 250
DistributionKurtosis() (*built-in function*), 250
DistributionMean() (*built-in function*), 251
DistributionSkewness() (*built-in function*), 251
DistributionVariance() (*built-in function*), 252
Div() (*built-in function*), 16
DomainIndex() (*built-in function*), 585

E

Element() (*built-in function*), 50
ElementCast() (*built-in function*), 51
ElementRange() (*built-in function*), 52
EnvironmentGetString() (*procedure*), 752
EnvironmentSetString() (*built-in function*), 753
errh::Adapt() (*errh method*), 727
errh::AllErrorCategories() (*set*), 860
errh::AllErrorSeverities() (*set*), 861
errh::Attribute() (*errh method*), 728
errh::Category() (*errh method*), 729
errh::Code() (*errh method*), 730
errh::Column() (*errh method*), 731
errh::CreationTime() (*errh method*), 731
errh::ErrorCodes() (*set*), 859
errh::Filename() (*errh method*), 732
errh::InsideCategory() (*errh method*), 733
errh::IsMarkedAsHandled() (*errh method*), 734
errh::Line() (*errh method*), 735
errh::MarkAsHandled() (*errh method*), 736
errh::Message() (*errh method*), 737
errh::Multiplicity() (*errh method*), 738
errh::Node() (*errh method*), 739
errh::NumberOfLocations() (*errh method*), 739
errh::PendingErrors() (*set*), 860
errh::Severity() (*errh method*), 740
ErrorF() (*built-in function*), 17
EvaluateUnit() (*built-in function*), 87
Execute() (*built-in function*), 756
ExitAimms() (*built-in function*), 757
Exp() (*built-in function*), 18
Exponential() (*built-in function*), 227
ExtremeValue() (*built-in function*), 229

F

Factorial() (*built-in function*), 253
FileAppend() (*procedure*), 768
FileCopy() (*procedure*), 769
FileDelete() (*procedure*), 770
FileEdit() (*procedure*), 770
FileExists() (*procedure*), 771
FileGetSize() (*procedure*), 771
FileMove() (*procedure*), 772
FilePrint() (*procedure*), 773
FileRead() (*procedure*), 773
FileSelect() (*procedure*), 774
FileSelectNew() (*procedure*), 775
FileTime() (*procedure*), 776
FileTouch() (*procedure*), 776
FileView() (*procedure*), 777
FindNthString() (*built-in function*), 70
FindReplaceNthString() (*built-in function*), 72
FindReplaceStrings() (*built-in function*), 73
FindString() (*built-in function*), 74
FindUsedElements() (*procedure*), 53
First() (*built-in function*), 55
Floor() (*built-in function*), 20
FormatString() (*built-in function*), 76

G

Gamma() (*built-in function*), 231
GarbageCollectStrings() (*built-in function*), 77
GenerateCut() (*procedure*), 570
GenerateXML() (*procedure*), 684
GeoFindCoordinates() (*procedure*), 754
Geometric() (*built-in function*), 217
GetAnnotationValues() (*built-in function*), 586
GetDataSourceProperty() (*built-in function*), 652
GMP::Benders::AddFeasibilityCut() (*procedure*), 304
GMP::Benders::AddOptimalityCut() (*procedure*), 307
GMP::Benders::CreateMasterProblem() (*GMP::Benders method*), 308

GMP::Benders::CreateSubProblem() (*GMP::Benders method*), 309
GMP::Benders::UpdateSubProblem() (*procedure*), 311
GMP::Coefficient::Get() (*GMP::Coefficient method*), 312
GMP::Coefficient::GetMinAndMax() (*procedure*), 313
GMP::Coefficient::GetQuadratic() (*GMP::Coefficient method*), 315
GMP::Coefficient::GetRaw() (*procedure*), 314
GMP::Coefficient::Set() (*procedure*), 316
GMP::Coefficient::SetMulti() (*procedure*), 317
GMP::Coefficient::SetQuadratic() (*procedure*), 318
GMP::Coefficient::SetRaw() (*procedure*), 319
GMP::Column::Add() (*procedure*), 322
GMP::Column::AddMulti() (*procedure*), 322
GMP::Column::Delete() (*procedure*), 323
GMP::Column::DeleteMulti() (*procedure*), 324
GMP::Column::DeleteRaw() (*procedure*), 325
GMP::Column::Freeze() (*procedure*), 326
GMP::Column::FreezeMulti() (*procedure*), 327
GMP::Column::FreezeRaw() (*procedure*), 328
GMP::Column::GetLowerBound() (*GMP::Column method*), 330
GMP::Column::GetLowerBoundRaw() (*procedure*), 331
GMP::Column::GetName() (*GMP::Column method*), 332
GMP::Column::GetScale() (*GMP::Column method*), 332
GMP::Column::GetStatus() (*GMP::Column method*), 333
GMP::Column::GetType() (*GMP::Column method*), 334
GMP::Column::GetUpperBound() (*GMP::Column method*), 335
GMP::Column::GetUpperBoundRaw() (*procedure*), 336
GMP::Column::SetAsMultiObjective() (*procedure*), 337
GMP::Column::SetAsObjective() (*procedure*), 339
GMP::Column::SetDecomposition() (*procedure*), 339
GMP::Column::SetDecompositionMulti() (*procedure*), 341
GMP::Column::SetLowerBound() (*procedure*), 343
GMP::Column::SetLowerBoundMulti() (*procedure*), 345
GMP::Column::SetLowerBoundRaw() (*procedure*), 346
GMP::Column::SetType() (*procedure*), 347
GMP::Column::SetTypeMulti() (*procedure*), 348
GMP::Column::SetTypeRaw() (*procedure*), 349
GMP::Column::SetUpperBound() (*procedure*), 350
GMP::Column::SetUpperBoundMulti() (*procedure*), 351
GMP::Column::SetUpperBoundRaw() (*procedure*), 352
GMP::Column::Unfreeze() (*procedure*), 354
GMP::Column::UnfreezeMulti() (*procedure*), 355
GMP::Column::UnfreezeRaw() (*procedure*), 356
GMP::Event::Create() (*GMP::Event method*), 357
GMP::Event::Delete() (*procedure*), 357
GMP::Event::Reset() (*procedure*), 358
GMP::Event::Set() (*procedure*), 358
GMP::Instance::AddIntegerEliminationRows() (*procedure*), 359
GMP::Instance::AddLimitBinaryDeviationRow() (*procedure*), 361
GMP::Instance::CalculateSubGradient() (*procedure*), 363
GMP::Instance::Copy() (*GMP::Instance method*), 364
GMP::Instance::CreateBlockMatrices() (*GMP::Instance method*), 365
GMP::Instance::CreateDual() (*GMP::Instance method*), 370
GMP::Instance::CreateFeasibility() (*GMP::Instance method*), 372
GMP::Instance::CreateMasterMIP() (*GMP::Instance method*), 374
GMP::Instance::CreatePresolved() (*GMP::Instance method*), 375
GMP::Instance::CreateProgressCategory() (*GMP::Instance method*), 377
GMP::Instance::CreateSolverSession() (*GMP::Instance method*), 378
GMP::Instance::Delete() (*procedure*), 378
GMP::Instance::DeleteIntegerEliminationRows() (*procedure*), 379
GMP::Instance::DeleteLimitBinaryDeviationRow() (*procedure*), 380
GMP::Instance::DeleteMultiObjectives() (*procedure*), 380
GMP::Instance::DeleteSolverSession() (*procedure*), 381
GMP::Instance::FindApproximatelyFeasibleSolution() (*procedure*), 382
GMP::Instance::FixColumns() (*procedure*), 383
GMP::Instance::Generate() (*GMP::Instance method*), 384
GMP::Instance::GenerateRobustCounterpart() (*GMP::Instance method*), 386
GMP::Instance::GenerateStochasticProgram() (*GMP::Instance method*), 387
GMP::Instance::GetAttributeValue()

(*GMP::Instance method*), 389
GMP::Instance::GetBestBound() (*GMP::Instance method*), 391
GMP::Instance::GetColumnNumbers() (*GMP::Instance method*), 392
GMP::Instance::GetDirection() (*GMP::Instance method*), 393
GMP::Instance::GetInfeasibleData() (*procedure*), 393
GMP::Instance::GetMathematicalProgrammingType() (*GMP::Instance method*), 395
GMP::Instance::GetMemoryUsed() (*GMP::Instance method*), 396
GMP::Instance::GetNumberOfColumns() (*GMP::Instance method*), 396
GMP::Instance::GetNumberOfIndicatorRows() (*GMP::Instance method*), 397
GMP::Instance::GetNumberOfIntegerColumns() (*GMP::Instance method*), 397
GMP::Instance::GetNumberOfNonlinearColumns() (*GMP::Instance method*), 398
GMP::Instance::GetNumberOfNonlinearNonzeros() (*GMP::Instance method*), 398
GMP::Instance::GetNumberOfNonlinearRows() (*GMP::Instance method*), 399
GMP::Instance::GetNumberOfNonzeros() (*GMP::Instance method*), 399
GMP::Instance::GetNumberOfRows() (*GMP::Instance method*), 400
GMP::Instance::GetNumberOfSOS1Rows() (*GMP::Instance method*), 400
GMP::Instance::GetNumberOfSOS2Rows() (*GMP::Instance method*), 401
GMP::Instance::GetObjective() (*GMP::Instance method*), 401
GMP::Instance::GetObjectiveColumnNumber() (*GMP::Instance method*), 402
GMP::Instance::GetObjectiveRowNumber() (*GMP::Instance method*), 403
GMP::Instance::GetOptionValue() (*GMP::Instance method*), 404
GMP::Instance::GetRowNumbers() (*GMP::Instance method*), 405
GMP::Instance::GetSolver() (*GMP::Instance method*), 406
GMP::Instance::GetSymbolicMathematicalProgram() (*GMP::Instance method*), 406
GMP::Instance::MemoryStatistics() (*procedure*), 407
GMP::Instance::Rename() (*GMP::Instance method*), 408
GMP::Instance::RestoreState() (*procedure*), 409
GMP::Instance::SaveState() (*procedure*), 409
GMP::Instance::SetCallbackAddCut() (*procedure*), 410
GMP::Instance::SetCallbackAddLazyConstraint() (*procedure*), 411
GMP::Instance::SetCallbackBranch() (*procedure*), 412
GMP::Instance::SetCallbackCandidate() (*procedure*), 413
GMP::Instance::SetCallbackHeuristic() (*procedure*), 414
GMP::Instance::SetCallbackIncumbent() (*procedure*), 415
GMP::Instance::SetCallbackIterations() (*procedure*), 416
GMP::Instance::SetCallbackStatusChange() (*procedure*), 417
GMP::Instance::SetCallbackTime() (*procedure*), 417
GMP::Instance::SetCutoff() (*procedure*), 418
GMP::Instance::SetDirection() (*procedure*), 419
GMP::Instance::SetIterationLimit() (*procedure*), 420
GMP::Instance::SetMathematicalProgrammingType() (*procedure*), 420
GMP::Instance::SetMemoryLimit() (*procedure*), 421
GMP::Instance::SetOptionValue() (*procedure*), 421
GMP::Instance::SetSolver() (*procedure*), 423
GMP::Instance::SetStartingPointSelection() (*procedure*), 423
GMP::Instance::SetTimeLimit() (*procedure*), 424
GMP::Instance::Solve() (*procedure*), 424
GMP::Linearization::Add() (*procedure*), 425
GMP::Linearization::AddSingle() (*procedure*), 427
GMP::Linearization::Delete() (*procedure*), 429
GMP::Linearization::GetDeviation() (*GMP::Linearization method*), 430
GMP::Linearization::GetDeviationBound() (*GMP::Linearization method*), 430
GMP::Linearization::GetLagrangeMultiplier() (*GMP::Linearization method*), 431
GMP::Linearization::GetType() (*GMP::Linearization method*), 432
GMP::Linearization::GetWeight() (*GMP::Linearization method*), 432
GMP::Linearization::RemoveDeviation() (*procedure*), 433
GMP::Linearization::SetDeviationBound() (*procedure*), 434
GMP::Linearization::SetType() (*procedure*), 434
GMP::Linearization::SetWeight() (*procedure*), 435
GMP::ProgressWindow::DeleteCategory() (*proce-*

dure), 436
GMP::ProgressWindow::DisplayLine() (*procedure*), 436
GMP::ProgressWindow::DisplayProgramStatus() (*procedure*), 437
GMP::ProgressWindow::DisplaySolver() (*procedure*), 438
GMP::ProgressWindow::DisplaySolverStatus() (*procedure*), 439
GMP::ProgressWindow::FreezeLine() (*procedure*), 440
GMP::ProgressWindow::Transfer() (*procedure*), 440
GMP::ProgressWindow::UnfreezeLine() (*procedure*), 441
GMP::QuadraticCoefficient::Get() (*GMP::QuadraticCoefficient method*), 442
GMP::QuadraticCoefficient::Set() (*procedure*), 443
GMP::Robust::EvaluateAdjustableVariables() (*procedure*), 444
GMP::Row::Activate() (*procedure*), 445
GMP::Row::ActivateMulti() (*procedure*), 446
GMP::Row::ActivateRaw() (*procedure*), 447
GMP::Row::Add() (*procedure*), 448
GMP::Row::AddMulti() (*procedure*), 448
GMP::Row::Deactivate() (*procedure*), 449
GMP::Row::DeactivateMulti() (*procedure*), 450
GMP::Row::DeactivateRaw() (*procedure*), 451
GMP::Row::Delete() (*procedure*), 452
GMP::Row::DeleteIndicatorCondition() (*procedure*), 453
GMP::Row::DeleteMulti() (*procedure*), 453
GMP::Row::DeleteRaw() (*procedure*), 454
GMP::Row::Generate() (*procedure*), 455
GMP::Row::GenerateMulti() (*procedure*), 457
GMP::Row::GetConvex() (*GMP::Row method*), 458
GMP::Row::GetIndicatorColumn() (*GMP::Row method*), 459
GMP::Row::GetIndicatorCondition() (*GMP::Row method*), 459
GMP::Row::GetLeftHandSide() (*GMP::Row method*), 460
GMP::Row::GetName() (*GMP::Row method*), 462
GMP::Row::GetRelaxationOnly() (*GMP::Row method*), 462
GMP::Row::GetRightHandSide() (*GMP::Row method*), 463
GMP::Row::GetRightHandSideRaw() (*procedure*), 464
GMP::Row::GetScale() (*GMP::Row method*), 465
GMP::Row::GetStatus() (*GMP::Row method*), 465
GMP::Row::GetType() (*GMP::Row method*), 466
GMP::Row::SetConvex() (*procedure*), 467
GMP::Row::SetIndicatorCondition() (*procedure*), 467
GMP::Row::SetLeftHandSide() (*procedure*), 468
GMP::Row::SetPoolType() (*procedure*), 470
GMP::Row::SetPoolTypeMulti() (*procedure*), 471
GMP::Row::SetRelaxationOnly() (*procedure*), 472
GMP::Row::SetRightHandSide() (*procedure*), 472
GMP::Row::SetRightHandSideMulti() (*procedure*), 474
GMP::Row::SetRightHandSideRaw() (*procedure*), 475
GMP::Row::SetType() (*procedure*), 476
GMP::Row::SetTypeMulti() (*procedure*), 477
GMP::Row::SetTypeRaw() (*procedure*), 478
GMP::Solution::Check() (*procedure*), 479
GMP::Solution::ConstraintListing() (*procedure*), 480
GMP::Solution::ConstructMean() (*procedure*), 484
GMP::Solution::Copy() (*procedure*), 485
GMP::Solution::Count() (*GMP::Solution method*), 486
GMP::Solution::Delete() (*procedure*), 486
GMP::Solution::DeleteAll() (*procedure*), 487
GMP::Solution::GetBestBound() (*GMP::Solution method*), 487
GMP::Solution::GetColumnValue() (*GMP::Solution method*), 488
GMP::Solution::GetDistance() (*GMP::Solution method*), 489
GMP::Solution::GetFirstOrderDerivative() (*GMP::Solution method*), 490
GMP::Solution::GetIterationsUsed() (*GMP::Solution method*), 490
GMP::Solution::GetMemoryUsed() (*GMP::Solution method*), 491
GMP::Solution::GetNodesUsed() (*GMP::Solution method*), 492
GMP::Solution::GetObjective() (*GMP::Solution method*), 492
GMP::Solution::GetPenalizedObjective() (*GMP::Solution method*), 493
GMP::Solution::GetProgramStatus() (*GMP::Solution method*), 494
GMP::Solution::GetRowValue() (*GMP::Solution method*), 495
GMP::Solution::GetSolutionsSet() (*GMP::Solution method*), 496
GMP::Solution::GetSolverStatus() (*GMP::Solution method*), 496
GMP::Solution::GetTimeUsed() (*GMP::Solution method*), 497
GMP::Solution::IsDualDegenerated() (*GMP::Solution method*), 497
GMP::Solution::IsInteger() (*GMP::Solution*

method), 498
GMP::Solution::IsPrimalDegenerated() (*GMP::Solution method*), 499
GMP::Solution::Move() (*procedure*), 500
GMP::Solution::RandomlyGenerate() (*procedure*), 500
GMP::Solution::RetrieveFromModel() (*procedure*), 501
GMP::Solution::RetrieveFromSolverSession() (*procedure*), 502
GMP::Solution::SendToModel() (*procedure*), 503
GMP::Solution::SendToModelSelection() (*procedure*), 504
GMP::Solution::SendToSolverSession() (*procedure*), 505
GMP::Solution::SetColumnValue() (*procedure*), 506
GMP::Solution::SetIterationCount() (*procedure*), 507
GMP::Solution::SetMIPStartFlag() (*procedure*), 508
GMP::Solution::SetObjective() (*procedure*), 509
GMP::Solution::SetProgramStatus() (*procedure*), 510
GMP::Solution::SetRowValue() (*procedure*), 510
GMP::Solution::SetSolverStatus() (*procedure*), 512
GMP::Solution::UpdatePenaltyWeights() (*procedure*), 512
GMP::Solver::FreeEnvironment() (*procedure*), 513
GMP::Solver::GetAsynchronousSessionsLimit() (*GMP::Solver method*), 514
GMP::Solver::InitializeEnvironment() (*procedure*), 516
GMP::Solver::SetEnvironmentDoubleParameter() (*procedure*), 518
GMP::Solver::SetEnvironmentIntegerParameter() (*procedure*), 519
GMP::Solver::SetEnvironmentStringParameter() (*procedure*), 521
GMP::SolverSession::AddBendersFeasibilityCut() (*procedure*), 522
GMP::SolverSession::AddBendersOptimalityCut() (*procedure*), 524
GMP::SolverSession::AddLinearization() (*procedure*), 526
GMP::SolverSession::AsynchronousExecute() (*procedure*), 528
GMP::SolverSession::CreateProgressCategory() (*GMP::SolverSession method*), 529
GMP::SolverSession::Execute() (*procedure*), 530
GMP::SolverSession::ExecutionStatus() (*GMP::SolverSession method*), 531
GMP::SolverSession::GenerateBinaryEliminationRow() (*procedure*), 531
GMP::SolverSession::GenerateBranchLowerBound() (*procedure*), 533
GMP::SolverSession::GenerateBranchRow() (*procedure*), 534
GMP::SolverSession::GenerateBranchUpperBound() (*procedure*), 534
GMP::SolverSession::GenerateCut() (*procedure*), 535
GMP::SolverSession::GetAttributeValue() (*GMP::SolverSession method*), 537
GMP::SolverSession::GetBestBound() (*GMP::SolverSession method*), 540
GMP::SolverSession::GetCallbackInterruptStatus() (*GMP::SolverSession method*), 541
GMP::SolverSession::GetCandidateObjective() (*GMP::SolverSession method*), 542
GMP::SolverSession::GetIIS() (*procedure*), 542
GMP::SolverSession::GetInstance() (*GMP::SolverSession method*), 544
GMP::SolverSession::GetIterationsUsed() (*GMP::SolverSession method*), 545
GMP::SolverSession::GetMemoryUsed() (*GMP::SolverSession method*), 545
GMP::SolverSession::GetNodeNumber() (*GMP::SolverSession method*), 546
GMP::SolverSession::GetNodeObjective() (*GMP::SolverSession method*), 547
GMP::SolverSession::GetNodesLeft() (*GMP::SolverSession method*), 547
GMP::SolverSession::GetNodesUsed() (*GMP::SolverSession method*), 548
GMP::SolverSession::GetNumberOfBranchNodes() (*GMP::SolverSession method*), 549
GMP::SolverSession::GetObjective() (*GMP::SolverSession method*), 549
GMP::SolverSession::GetOptionValue() (*GMP::SolverSession method*), 550
GMP::SolverSession::GetProgramStatus() (*GMP::SolverSession method*), 551
GMP::SolverSession::GetSolver() (*GMP::SolverSession method*), 551
GMP::SolverSession::GetSolverStatus() (*GMP::SolverSession method*), 552
GMP::SolverSession::GetTimeUsed() (*GMP::SolverSession method*), 552
GMP::SolverSession::Interrupt() (*procedure*), 553
GMP::SolverSession::RejectIncumbent() (*procedure*), 554
GMP::SolverSession::SetObjective() (*procedure*), 554
GMP::SolverSession::SetOptionValue() (*procedure*), 555

GMP::SolverSession::Transfer() (*procedure*), 556
GMP::SolverSession::WaitForCompletion() (*procedure*), 556
GMP::SolverSession::WaitForSingleCompletion() (*GMP::SolverSession method*), 557
GMP::Stochastic::AddBendersFeasibilityCut() (*procedure*), 558
GMP::Stochastic::AddBendersOptimalityCut() (*procedure*), 559
GMP::Stochastic::BendersFindFeasibilityReference() (*GMP::Stochastic method*), 560
GMP::Stochastic::BendersFindReference() (*GMP::Stochastic method*), 560
GMP::Stochastic::CreateBendersRootproblem() (*GMP::Stochastic method*), 561
GMP::Stochastic::GetObjectiveBound() (*GMP::Stochastic method*), 562
GMP::Stochastic::GetRelativeWeight() (*GMP::Stochastic method*), 563
GMP::Stochastic::GetRepresentativeScenario() (*GMP::Stochastic method*), 564
GMP::Stochastic::MergeSolution() (*procedure*), 564
GMP::Stochastic::UpdateBendersSubproblem() (*procedure*), 565
GMP::Tuning::SolveSingleMPS() (*procedure*), 566
GMP::Tuning::TuneMultipleMPS() (*procedure*), 567
GMP::Tuning::TuneSingleGMP() (*procedure*), 569

H

HistogramAddObservation() (*procedure*), 254
HistogramAddObservations() (*procedure*), 254
HistogramCreate() (*built-in function*), 255
HistogramDelete() (*procedure*), 256
HistogramGetAverage() (*built-in function*), 256
HistogramGetBounds() (*built-in function*), 257
HistogramGetDeviation() (*built-in function*), 257
HistogramGetFrequencies() (*procedure*), 258
HistogramGetKurtosis() (*built-in function*), 259
HistogramGetObservationCount() (*built-in function*), 259
HistogramGetSkewness() (*built-in function*), 260
HistogramSetDomain() (*procedure*), 260
HyperGeometric() (*built-in function*), 219

I

IdentifierAttributes() (*built-in function*), 587
IdentifierDimension() (*built-in function*), 587
IdentifierElementRange() (*built-in function*), 588
IdentifierGetUsedInformation() (*procedure*), 722
IdentifierMemory() (*built-in function*), 723
IdentifierMemoryStatistics() (*procedure*), 723
IdentifierShowAttributes() (*built-in function*), 594
IdentifierShowTreeLocation() (*built-in function*), 594
IdentifierText() (*built-in function*), 590
IdentifierType() (*built-in function*), 591
IdentifierUnit() (*built-in function*), 592
IndexRange() (*built-in function*), 593
Integers() (*set*), 812
InvestmentConstantCumulativeInterestPayment() (*built-in function*), 152
InvestmentConstantCumulativePrincipalPayment() (*built-in function*), 154
InvestmentConstantFutureValue() (*built-in function*), 143
InvestmentConstantInterestPayment() (*built-in function*), 148
InvestmentConstantNumberPeriods() (*built-in function*), 155
InvestmentConstantPeriodicPayment() (*built-in function*), 145
InvestmentConstantPresentValue() (*built-in function*), 147
InvestmentConstantPrincipalPayment() (*built-in function*), 150
InvestmentConstantRate() (*built-in function*), 157
InvestmentConstantRateAll() (*built-in function*), 159
InvestmentSingleFutureValue() (*built-in function*), 165
InvestmentVariableInternalRateReturn() (*built-in function*), 166
InvestmentVariableInternalRateReturnAll() (*built-in function*), 168
InvestmentVariableInternalRateReturnInPeriodic() (*built-in function*), 170
InvestmentVariableInternalRateReturnInPeriodicAll() (*built-in function*), 172
InvestmentVariableInternalRateReturnModified() (*built-in function*), 174
InvestmentVariablePresentValue() (*built-in function*), 161
InvestmentVariablePresentValueInPeriodic() (*built-in function*), 162
IsRuntimeIdentifier() (*built-in function*), 594

L

Last() (*built-in function*), 57
LicenseExpirationDate() (*procedure*), 746
LicenseMaintenanceExpirationDate() (*procedure*), 746
LicenseNumber() (*procedure*), 747
LicenseStartDate() (*procedure*), 747
LicenseType() (*procedure*), 748
ListingFileCopy() (*procedure*), 962
ListingFileDelete() (*built-in function*), 962

LoadDatabaseStructure() (*procedure*), 648
LocaleAllAbbrMonths() (*set*), 854
LocaleAllAbbrWeekdays() (*set*), 855
LocaleAllMonths() (*set*), 855
LocaleAllWeekdays() (*set*), 856
LocaleLongDateFormat() (*set*), 857
LocaleShortDateFormat() (*set*), 857
LocaleTimeFormat() (*set*), 858
LocaleTimeZoneName() (*set*), 859
LocaleTimeZoneNameDST() (*set*), 859
Log() (*built-in function*), 21
Log10() (*built-in function*), 22
Logistic() (*built-in function*), 233
LogNormal() (*built-in function*), 235

M

MapVal() (*built-in function*), 23
Max() (*built-in function*), 24
MaximizingMinimizing() (*set*), 813
me::AllowedAttribute() (*me method*), 599
me::ChangeType() (*me method*), 600
me::ChangeTypeAllowed() (*me method*), 601
me::Children() (*me method*), 602
me::ChildTypeAllowed() (*me method*), 603
me::Compile() (*me method*), 605
me::Create() (*me method*), 606
me::CreateLibrary() (*me method*), 607
me::Delete() (*me method*), 608
me::ExportNode() (*procedure*), 609
me::GetAttribute() (*me method*), 610
me::ImportLibrary() (*me method*), 611
me::ImportNode() (*procedure*), 612
me::IsRunnable() (*me method*), 612
me::Move() (*me method*), 613
me::Parent() (*me method*), 614
me::Rename() (*me method*), 615
me::SetAnnotation() (*me method*), 617
me::SetAttribute() (*me method*), 618
MemoryInUse() (*built-in function*), 724
MemoryStatistics() (*procedure*), 725
MergeReplace() (*set*), 813
Min() (*built-in function*), 25
Mod() (*built-in function*), 26
MomentToString() (*built-in function*), 102
MomentToTimeSlot() (*built-in function*), 103

N

NegativeBinomial() (*built-in function*), 221
Normal() (*built-in function*), 237

O

ODBCDateTimeFormat() (*string parameter*), 841
OnOff() (*set*), 814
OpenDocument() (*procedure*), 758
OptionGetDefaultString() (*procedure*), 741
OptionGetKeywords() (*procedure*), 742
OptionGetString() (*procedure*), 742
OptionGetValue() (*procedure*), 743
OptionSetString() (*procedure*), 744
OptionSetValue() (*procedure*), 745
Ord() (*built-in function*), 58

P

PageClose() (*procedure*), 695
PageCopyTableToClipboard() (*procedure*), 696
PageCopyTableToExcel() (*procedure*), 697
PageGetActive() (*procedure*), 698
PageGetAll() (*procedure*), 699
PageGetChild() (*procedure*), 699
PageGetFocus() (*procedure*), 700
PageGetNext() (*procedure*), 701
PageGetNextInTreeWalk() (*procedure*), 702
PageGetParent() (*procedure*), 702
PageGetPrevious() (*procedure*), 703
PageGetTitle() (*procedure*), 704
PageGetUsedIdentifiers() (*procedure*), 704
PageOpen() (*procedure*), 705
PageOpenSingle() (*procedure*), 705
PageRefreshAll() (*built-in function*), 706
PageSetCursor() (*procedure*), 706
PageSetFocus() (*procedure*), 707
Pareto() (*built-in function*), 239
PeriodToString() (*built-in function*), 104
Permutation() (*built-in function*), 253
PivotTableDeleteState() (*procedure*), 708
PivotTableReloadState() (*procedure*), 708
PivotTableSaveState() (*procedure*), 709
Poisson() (*built-in function*), 223
Power() (*built-in function*), 27
Precision() (*built-in function*), 28
PriceDecimal() (*built-in function*), 117
PriceFractional() (*built-in function*), 118
PrintEndReport() (*procedure*), 710
PrinterGetCurrentName() (*procedure*), 712
PrinterSetupDialog() (*built-in function*), 714
PrintPage() (*procedure*), 711
PrintPageCount() (*procedure*), 712
PrintStartReport() (*procedure*), 713
ProfilerCollectAllData() (*built-in function*), 721
ProfilerContinue() (*built-in function*), 720
ProfilerData() (*set*), 789
ProfilerPause() (*built-in function*), 719
ProfilerRestart() (*built-in function*), 720
ProfilerStart() (*built-in function*), 720
ProjectDeveloperMode() (*built-in function*), 749

R

Radians() (*built-in function*), 29
RateEffective() (*built-in function*), 119
RateNominal() (*built-in function*), 120
ReadGeneratedXML() (*procedure*), 684
ReadXML() (*procedure*), 685
ReferencedIdentifiers() (*built-in function*), 595
RegexReplace() (*built-in function*), 78
RegexSearch() (*built-in function*), 77
RestoreInactiveElements() (*procedure*), 60
RetrieveCurrentVariableValues() (*procedure*), 571
RollbackTransaction() (*procedure*), 649
Round() (*built-in function*), 30

S

SaveDatabaseStructure() (*procedure*), 649
ScalarValue() (*built-in function*), 32
ScheduleAt() (*procedure*), 759
SectionIdentifiers() (*built-in function*), 597
SecurityCouponDays() (*built-in function*), 197
SecurityCouponDaysPostSettlement() (*built-in function*), 199
SecurityCouponDaysPreSettlement() (*built-in function*), 198
SecurityCouponNextDate() (*built-in function*), 196
SecurityCouponNumber() (*built-in function*), 193
SecurityCouponPreviousDate() (*built-in function*), 195
SecurityDiscountedPrice() (*built-in function*), 179
SecurityDiscountedRate() (*built-in function*), 183
SecurityDiscountedRedemption() (*built-in function*), 180
SecurityDiscountedYield() (*built-in function*), 182
SecurityGetGroups() (*procedure*), 749
SecurityGetUsers() (*procedure*), 750
SecurityMaturityAccruedInterest() (*built-in function*), 192
SecurityMaturityCouponRate() (*built-in function*), 189
SecurityMaturityPrice() (*built-in function*), 188
SecurityMaturityYield() (*built-in function*), 191
SecurityPeriodicAccruedInterest() (*built-in function*), 209
SecurityPeriodicCouponRate() (*built-in function*), 204
SecurityPeriodicDuration() (*built-in function*), 210
SecurityPeriodicDurationModified() (*built-in function*), 212
SecurityPeriodicPrice() (*built-in function*), 200
SecurityPeriodicRedemption() (*built-in function*), 202
SecurityPeriodicYield() (*built-in function*), 207
SecurityPeriodicYieldAll() (*built-in function*), 205
SessionArgument() (*procedure*), 759
SetAddRecursive() (*procedure*), 61
SetAsString() (*built-in function*), 63
SetElementAdd() (*procedure*), 64
SetElementRename() (*procedure*), 65
ShowHelpTopic() (*built-in function*), 726
ShowMessageWindow() (*built-in function*), 715
ShowProgressWindow() (*built-in function*), 715
Sign() (*built-in function*), 34
Sin() (*built-in function*), 34
Sinh() (*built-in function*), 36
SolverGetControl() (*procedure*), 750
SolverReleaseControl() (*procedure*), 751
Spreadsheet::AddNewSheet() (*procedure*), 675
Spreadsheet::AssignParameter() (*procedure*), 669
Spreadsheet::AssignSet() (*procedure*), 666
Spreadsheet::AssignTable() (*procedure*), 671
Spreadsheet::AssignValue() (*procedure*), 663
Spreadsheet::ClearRange() (*procedure*), 673
Spreadsheet::CloseWorkbook() (*procedure*), 680
Spreadsheet::ColumnName() (*Spreadsheet method*), 660
Spreadsheet::ColumnNumber() (*Spreadsheet method*), 661
Spreadsheet::CopyRange() (*procedure*), 676
Spreadsheet::CreateWorkbook() (*procedure*), 681
Spreadsheet::DeleteSheet() (*procedure*), 677
Spreadsheet::GetAllSheets() (*procedure*), 678
Spreadsheet::Print() (*procedure*), 682
Spreadsheet::RetrieveParameter() (*procedure*), 670
Spreadsheet::RetrieveSet() (*procedure*), 667
Spreadsheet::RetrieveTable() (*procedure*), 674
Spreadsheet::RetrieveValue() (*procedure*), 668
Spreadsheet::RunMacro() (*procedure*), 679
Spreadsheet::SaveWorkbook() (*procedure*), 682
Spreadsheet::SetActiveSheet() (*procedure*), 662
Spreadsheet::SetOption() (*procedure*), 664
Spreadsheet::SetUpdateLinksBehavior() (*procedure*), 665
Spreadsheet::SetVisibility() (*procedure*), 663
SQLColumnData() (*built-in function*), 656
SQLCreateConnectionString() (*built-in function*), 658
SQLDriverName() (*built-in function*), 657
SQLNumberOfColumns() (*built-in function*), 652
SQLNumberOfDrivers() (*built-in function*), 654
SQLNumberOfTables() (*built-in function*), 654
SQLNumberOfViews() (*built-in function*), 655
SQLTableName() (*built-in function*), 659
SQLViewName() (*built-in function*), 659
Sqr() (*built-in function*), 37
Sqrt() (*built-in function*), 39
StartTransaction() (*procedure*), 650

StatusMessage() (*built-in function*), 695
StringCapitalize() (*built-in function*), 80
StringLength() (*built-in function*), 80
StringOccurrences() (*built-in function*), 81
StringToElement() (*built-in function*), 66
StringToLower() (*built-in function*), 82
StringToMoment() (*built-in function*), 106
StringToTimeSlot() (*built-in function*), 108
StringToUnit() (*built-in function*), 88
StringToUpper() (*built-in function*), 82
SubRange() (*built-in function*), 67
SubString() (*built-in function*), 83

T

Tan() (*built-in function*), 40
Tanh() (*built-in function*), 42
TestDatabaseColumn() (*procedure*), 653
TestDatabaseTable() (*procedure*), 651
TestDataSource() (*procedure*), 650
TestDate() (*built-in function*), 109
TestInternetConnection() (*procedure*), 755
TimeSlotCharacteristic() (*built-in function*), 110
TimeSlotCharacteristics() (*set*), 815
TimeSlotToMoment() (*built-in function*), 111
TimeSlotToString() (*built-in function*), 113
TimeZoneOffSet() (*built-in function*), 114
TreasuryBillBondEquivalent() (*built-in function*), 187
TreasuryBillPrice() (*built-in function*), 184
TreasuryBillYield() (*built-in function*), 185
Triangular() (*built-in function*), 241
Trunc() (*built-in function*), 43

U

UnicodeCharacterEncodings() (*set*), 781
Uniform() (*built-in function*), 243
Unit() (*built-in function*), 89
UserColorAdd() (*procedure*), 716
UserColorDelete() (*procedure*), 716
UserColorGetRGB() (*procedure*), 717
UserColorModify() (*procedure*), 717

V

Val() (*built-in function*), 44
VariableConstraints() (*built-in function*), 598

W

Weibull() (*built-in function*), 245
WriteXML() (*procedure*), 686

Y

YesNo() (*set*), 815