AIMMS

The Function Reference

AIMMS

AIMMS 4

September 16, 2019

Contents

Part I

Elementary Computational Operations

Chapter 1

Arithmetic Functions

AIMMS supports the following arithmetic functions:

- Abs
- ArcCosh
- ArcCos
- ArcSin
- ArcSinh
- ArcTanh
- ArcTan
- Ceil
- Cos
- Cosh
- Cube
- Degrees
- Div
- ErrorF
- Exp
- Floor
- Log
- Log10
- MapVal
- Max
- Min
- Mod
- Power
- Precision
- Radians
- Round
- ScalarValue
- Sign
- Sin
- Sinh
- Sqr
- Sqrt
- Tan
- Tanh

- `Trunc`
- `Val`

Abs

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function Abs returns the absolute value of x.

Remarks:

The function Abs 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 Section 6.1.4 of the Language Reference.

ArcCos

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

Arguments:

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

Return value:

The ArcCos function returns the arccosine of x in the range 0 to π radians.

Remarks:

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

See also:

The functions ArcSin, ArcTan, Cos. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ArcCosh

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

Arguments:

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

Return value:

The `ArcCosh` function returns the inverse hyperbolic cosine of x in the range from 0 to ∞.

Remarks:

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

See also:

The functions `ArcSinh`, `ArcTanh`, `Cosh`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ArcSin

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

Arguments:

x

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

Return value:

The `ArcSin` function returns the arcsine of x in the range $-\pi/2$ to $\pi/2$ radians.

Remarks:

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

See also:

The functions `ArcCos`, `ArcTan`, `Sin`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ArcSinh

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

Arguments:

x

A scalar numerical expression.

Return value:

The `ArcSinh` function returns the inverse hyperbolic sine of x in the range from $-\infty$ to ∞.

Remarks:

The function `ArcSinh` can be used in constraints of nonlinear mathematical programs.

See also:

The functions `ArcCosh`, `ArcTanh`, `Sinh`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ArcTan

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

Arguments:

x
: A scalar numerical expression.

Return value:

The ArcTan function returns the arctangent of x in the range $-\pi/2$ to $\pi/2$ radians.

Remarks:

The function ArcTan can be used in constraints of nonlinear mathematical programs.

See also:

The functions ArcSin, ArcCos, Tan. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ArcTanh

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

Arguments:

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

Return value:

The ArcTanh function returns the inverse hyperbolic tangent of *x*.

Remarks:

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

See also:

The functions ArcCosh, ArcSinh, Tanh. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Ceil

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

Arguments:

x
A scalar numerical expression.

Return value:

The function `Ceil` returns the smallest integer value $\geq x$.

Remarks:

- The function `Ceil` will round to the nearest integer, if it lies within the equality tolerances `equality_absolute_tolerance` and `equality_relative_tolerance`.
- The function `Ceil` 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` will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

The functions `Floor`, `Round`, `Precision`, `Trunc`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference. Numeric tolerances are discussed in Section 6.2.2 of the Language Reference.

Cos

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

Arguments:

x
: A scalar numerical expression in radians.

Return value:

The Cos function returns the cosine of x in the range -1 to 1.

Remarks:

The function Cos can be used in constraints of nonlinear mathematical programs.

See also:

The functions Sin, Tan, ArcCos. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Cosh

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

Arguments:

x
: A scalar numerical expression.

Return value:

The `Cosh` function returns the hyperbolic cosine of x in the range 1 to ∞.

Remarks:

The function `Cosh` can be used in constraints of nonlinear mathematical programs.

See also:

The functions `Sinh`, `Tanh`, `ArcCosh`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Cube

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function Cube returns x^3.

Remarks:

The function Cube can be used in constraints of nonlinear mathematical programs.

See also:

The functions Power, Sqr, and Sqrt. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Degrees

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function `Degrees` returns the value of *x* converted from radians to degrees.

Remarks:

The function `Degrees` can be used in constraints of linear and nonlinear mathematical programs.

See also:

The function `Radians`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

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` returns x divided by y rounded down to an integer.

Remarks:

A run-time error results if y equals 0.

See also:

Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ErrorF

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

Arguments:

x
: A scalar numerical expression.

Return value:

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

Remarks:

The function ErrorF can be used in constraints of nonlinear mathematical programs.

See also:

Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Exp

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function Exp returns the exponential value e^x.

Remarks:

The function Exp can be used in constraints of nonlinear mathematical programs.

See also:

The functions Log, Log10. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Floor

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function `Floor` returns the largest integer value $\leq x$.

Remarks:

- The function `Floor` will round to the nearest integer, if it lies within the equality tolerances `equality_absolute_tolerance` and `equality_relative_tolerance`.
- The function `Floor` 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` will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

The functions `Ceil`, `Round`, `Precision`, `Trunc`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference. Numeric tolerances are discussed in Section 6.2.2 of the Language Reference.

Log

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

Arguments:

x

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

Return value:

The function `Log` returns the natural logarithm $\ln(x)$.

Remarks:

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

See also:

The functions `Exp`, `Log10`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Log10

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

Arguments:

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

Return value:

The function `Log10` returns the base-10 logarithm of x.

Remarks:

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

See also:

The functions `Exp`, `Log`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

MapVal

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

Arguments:

x

A scalar numerical expression.

Return value:

The function `MapVal` 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

See also:

Special numbers in AIMMS and the `MapVal` function are discussed in full detail in Section 6.1.1 of the Language Reference.

Max

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

Arguments:

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

Return value:

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

Remarks:

The function `Max` 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` function switches are discontinous.

See also:

The function `Min`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Min

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

Arguments:

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

Return value:

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

Remarks:

The function `Min` 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` function switches are discontinous.

See also:

The function `Max`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

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` 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.

Remarks:

- A run-time error results if y equals 0.
- The function `Mod` 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 Section 6.1.4 of the Language Reference.

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` returns x raised to the power y.

Remarks:

- 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`, `Sqr`, and `Sqrt`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

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` returns *x* rounded to *y* significant digits.

Remarks:

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

See also:

The functions `Round`, `Ceil`, `Floor`, `Trunc`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Radians

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function `Radians` returns the value of *x* converted from degrees to radians.

Remarks:

The function `Radians` can be used in constraints of linear and nonlinear mathematical programs.

See also:

The function `Degrees`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

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

Remarks:

- The function `Round` can be used in constraints of nonlinear mathematical programs. However, nonlinear solvers may experience convergence problems around the discontinuities of the `Round` function.
- When the numerical expression contains a unit, the function `Round` 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`.

See also:

The functions `Precision`, `Ceil`, `Floor`, `Trunc`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

ScalarValue

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

Arguments:

identifier
A scalar element expression into `AllIdentifiers`

suffix
A scalar element expression into `AllSuffixNames`

Return value:

The function `ScalarValue` returns the value contained in the scalar identifier *identifier* or scalar reference *identifier.suffix*.

Remarks:

When *identifier* or *identifier.suffix* is not a scalar numerical valued reference, the function `ScalarValue` returns 0.0.

See also:

The function `Val`.
The `ScalarValue` function is a function that operates on subsets of `AllIdentifiers`. Other functions that operate on subsets of `AllIdentifiers` are referenced in Section 25.4 of the Language Reference.

Sign

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function `Sign` returns $+1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$.

Remarks:

The function `Sign` can be used in constraints of nonlinear mathematical programs. However, nonlinear solver may experience convergence problems round 0.

See also:

The function `Abs`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Sin

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

Arguments:

x
: A scalar numerical expression in radians.

Return value:

The `Sin` function returns the sine of x in the range -1 to 1.

Remarks:

The function `Sin` can be used in constraints of nonlinear mathematical programs.

See also:

The functions `Cos`, `Tan`, `ArcSin`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Sinh

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

Arguments:

x
: A scalar numerical expression.

Return value:

The `Sinh` function returns the hyperbolic sine of x in the range $-\infty$ to ∞.

Remarks:

The function `Sinh` can be used in the constraints of nonlinear mathematical programs.

See also:

The functions `Cosh`, `Tanh`, `ArcSinh`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Sqr

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

Arguments:

x

A scalar numerical expression.

Return value:

The function Sqr returns x^2.

Remarks:

The function Sqr can be used in constraints of nonlinear mathematical programs.

See also:

The functions Power, Cube, and Sqrt. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Sqrt

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

Arguments:

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

Return value:

The function `Sqrt` returns the $\sqrt{x}$.

Remarks:

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

See also:

The functions `Power`, `Cube`, and `Sqr`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Tan

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

Arguments:

x

A scalar numerical expression in radians.

Return value:

The Tan function returns the tangent of x in the range $-\infty$ to ∞.

Remarks:

The function Tan can be used in constraints of nonlinear mathematical programs.

See also:

The functions Cos, Sin, ArcTan. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Tanh

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

Arguments:

x
: A scalar numerical expression.

Return value:

The Tanh function returns the hyperbolic tangent of x in the range -1 to 1.

Remarks:

The function Tanh can be used in constraints of nonlinear mathematical programs.

See also:

The functions Cosh, Sinh, ArcTanh. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference.

Trunc

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

Arguments:

x
: A scalar numerical expression.

Return value:

The function `Trunc` returns the truncated value of x: `Sign`(x)·`Floor`(`Abs`(x)).

Remarks:

- The function `Trunc` will round to the nearest integer, if it lies within the equality tolerances `equality_absolute_tolerance` and `equality_relative_tolerance`.
- The function `Trunc` 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` will first convert the expression to the corresponding base unit, before evaluating the function itself.

See also:

The functions `Ceil`, `Floor`, `Round`, `Precision`. Arithmetic functions are discussed in full detail in Section 6.1.4 of the Language Reference. Numeric tolerances are discussed in Section 6.2.2 of the Language Reference.

Val

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

Arguments:

str
: A scalar string or element expression.

Return value:

The function `Val` returns the numerical value represented by the string or element *str*.

Remarks:

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` function is discussed in full detail in Section 5.2.1 of the Language Reference.

Chapter 2

Set Related Functions

AIMMS supports the following set related functions:

- ActiveCard
- Card
- CloneElement
- Element
- ElementCast
- ElementRange
- FindUsedElements
- First
- Last
- Ord
- RestoreInactiveElements
- RetrieveCurrentVariableValues
- SetAddRecursive
- SetElementAdd
- SetElementRename
- StringToElement
- SubRange

ActiveCard

The function `ActiveCard` returns the cardinality of active elements in its identifier argument, or the cardinality of active elements of a suffix of that identifier.

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

Arguments:

Identifier
: A reference to a set or an indexed identifier.

Suffix
: An element in the predefined set `AllSuffixNames`.

Return value:

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

Remarks:

The `ActiveCard` 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.

See also:

The function `Card` and `Count` operator (see also Section 6.1.6 of the Language Reference).

Card

The function Card returns the cardinality of its identifier argument, or of the cardinality of a suffix of that identifier.

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

Arguments:

Identifier
A reference to a set or an indexed identifier.

Suffix
An element in the predefined set of AllSuffixNames.

Return value:

If *Identifier* is a set, the function Card returns the number of elements in *Identifier*. If *Identifier* is an indexed identifier, the function Card returns the number of nondefault values stored for *Identifier*. If *Suffix* is given, the number of nondefault values stored for the given suffix of *Identifier*.

Remarks:

- The Card 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 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 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 Section 7.3 of the Language Reference.

See also:

The function `ActiveCard` and the `Count` operator (see also Section 6.1.6 of the Language Reference).

CloneElement

The procedure CloneElement 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 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 Section 25.3 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.

Remarks:

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`, the procedure `FindUsedElements` and the procedure `RestoreInactiveElements`.

Element

With the function `Element` 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*.

Remarks:

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

ElementCast

With the function `ElementCast` 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.

See also:

The procedure `SetElementAdd`.

ElementRange

With the function `ElementRange` 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.

FindUsedElements

The procedure `FindUsedElements` 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`, 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*.

First

With the function `First` 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*.

Remarks:

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

Last

With the function `Last` 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*.

Remarks:

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

Ord

The function Ord 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 returns the ordinal number of *index* in set *set*.

Remarks:

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

RestoreInactiveElements

The procedure `RestoreInactiveElements` 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`, 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*.

Remarks:

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

RetrieveCurrentVariableValues

With the procedure `RetrieveCurrentVariableValues` 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`, holding all the variables for which you want to retrieve the current values.

See also:

Solver callback procedures are discussed in full detail in Section 15.2 of the Language Reference

SetAddRecursive

With the procedure `SetAddRecursive` 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*.

Remarks:

- 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*

SetElementAdd

With the procedure `SetElementAdd` you can add new elements to a set. When you apply `SetElementAdd` 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 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.

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.

Remarks:

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.

See also:

- The function `ElementCast` and the procedures `SetElementRename` and `StringToElement`.
- The lexical conventions for set elements in Section 2.3 of the Language Reference.

SetElementRename

With the procedure `SetElementRename` 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.

Remarks:

- 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.

See also:

- The procedure `SetElementAdd`, and the function `StringToElement`.
- The lexical conventions for set elements in Section 2.3 of the Language Reference.

StringToElement

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

```
StringToElement(
     Set,            ! (input) a set expression
     Name,           ! (input) a scalar string
     [create]        ! (optional) 0 or 1, default 0
     )
```

Arguments:

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

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.

See also:

- The function `ElementCast` and the procedure `SetElementAdd`.
- The lexical conventions for set elements in Section 2.3 of the Language Reference.

SubRange

The function SubRange 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.

Chapter 3

String Manipulation Functions

AIMMS supports the following functions for manipulating strings:

- `Character`
- `CharacterNumber`
- `FindNthString`
- `FindReplaceNthString`
- `FindReplaceStrings`
- `FindString`
- `FormatString`
- `GarbageCollectStrings`
- `RegexSearch`
- `StringCapitalize`
- `StringLength`
- `StringOccurrences`
- `StringToLower`
- `StringToUpper`
- `SubString`

Character

The function `Character` 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 $\{0..55295\} \cup \{57344..65535\}$.

Return value:

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

See also:

The function `CharacterNumber`.

CharacterNumber

The function `CharacterNumber` 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 .. 65535 }.

See also:

The function `Character`.

FindNthString

The function `FindNthString` 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 expession
     [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.

Remarks:

As with all string comparisons within AIMMS, the function `FindNthString` 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.

See also:

The functions `FindString`, `StringOccurrences`, `RegexSearch`.

FindReplaceNthString

The function `FindReplaceNthString` 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 expession
     [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.

Remarks:

As with all string comparisons within AIMMS, the function `FindReplaceNthString` 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.

See also:

The functions `FindNthString`, `StringOccurrences` and `FindReplaceStrings`.

FindReplaceStrings

The function FindReplaceStrings 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.

Remarks:

As with all string comparisons within AIMMS, the function FindReplaceStrings 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.

See also:

The functions FindString, StringOccurrences and FindReplaceNthString.

FindString

The function FindString 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.

Remarks:

As with all string comparisons within AIMMS, the function FindString 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.

See also:

The functions FindNthString, RegexSearch.

FormatString

With the `FormatString` function you can compose a string that is built up from combinations of numbers, strings and set elements. The `FormatString` 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.

See also:

For a detailed description of the conversion specifiers in AIMMS see Section 5.3.2 of the Language Reference.

GarbageCollectStrings

The procedure `GarbageCollectStrings` 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()
```

Remarks:

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.

RegexSearch

The function `RegexSearch` 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
: 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'.

Remarks:

- 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. For further references go to this link `https://en.cppreference.com/w/cpp/regex/ecmascript` You can find more information on ESMA Script regular expressions via this link: ECMA Regular expressions. You can find more information on POSIX regular expressions via this link: POSIX Basic Regular Expressions.
- To include a special character in a string, it should be escaped by the backslash character (for more information on special characters see also Section 5.3.2 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 line *(regular expression "`\\\\|$`")*.

See also:

The functions `FindString`, `FindNthString`.

StringCapitalize

The function `StringCapitalize` 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.

See also:

The functions `StringToLower`, `StringToUpper`.

StringLength

The function `StringLength` 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.

StringOccurrences

The function `StringOccurrences` 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*.

See also:

The functions `FindString`, `FindNthString`.

StringToLower

The function `StringToLower` 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.

See also:

The functions `StringToUpper`, `StringCapitalize`.

StringToUpper

The function `StringToUpper` 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.

See also:

The functions `StringToLower`, `StringCapitalize`.

SubString

The function `SubString` 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.

Remarks:

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.

Chapter 4

Unit Functions

AIMMS supports the following functions for unit related functions:

- AtomicUnit
- ConvertUnit
- EvaluateUnit
- StringToUnit
- Unit

AtomicUnit

With the function `AtomicUnit` 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*.

Remarks:

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.

See also:

Unit expressions are discussed in full detail in Chapter 32 of the Language Reference.

ConvertUnit

With the function ConvertUnit 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, 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*.

See also:

Unit expressions and conventions are discussed in full detail in Chapter 32 of the Language Reference.

EvaluateUnit

With the function `EvaluateUnit` 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*.

Remarks:

The function `EvaluateUnit` is an extension of AIMMS' local unit override capabilities which allows computed unit expressions.

See also:

Unit expressions are discussed in full detail in Chapter 32 of the Language Reference.

StringToUnit

With the function `StringToUnit` 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.

See also:

Unit expressions discussed in full detail in Chapter 32 of the Language Reference.

Unit

The function `Unit` 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*.

Remarks:

The function `Unit` 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 Chapter 32 of the Language Reference.

Chapter 5

Time Functions

AIMMS supports the following time-related functions:

- Aggregate
- ConvertReferenceDate
- CreateTimeTable
- CurrentToMoment
- CurrentToString
- CurrentToTimeSlot
- DaylightSavingEndDate
- DaylightSavingStartDate
- DisAggregate
- MomentToString
- MomentToTimeSlot
- PeriodToString
- StringToMoment
- StringToTimeSlot
- TestDate
- TimeSlotCharacteristic
- TimeSlotToMoment
- TimeSlotToString
- TimeZoneOffSet

Aggregate

With the procedure Aggregate 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 (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.

See also:

The procedure DisAggregate. Time-dependent aggregation and disaggregation is discussed in full detail in Section 33.5 of the Language Reference.

ConvertReferenceDate

The function ConvertReferenceDate 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 with respect to which *ReferenceDate* is expressed.

ToTimezone
An element of AllTimeZones 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 is a reference date in *ToTimezone* corresponding to the reference date *ReferenceDate* in *FromTimezone*.

See also:

AIMMS support for time zones is discussed in full detail in Sections 33.7.4 and 33.10 of the Language Reference.

CreateTimeTable

With the procedure `CreateTimeTable` 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.

See also:

The procedures `Aggregate`, `DisAggregate`. For a more detailed description of the creation of timetables, see Section 33.4 of the Language Reference.

CurrentToMoment

The function `CurrentToMoment` 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* on page 544 of the Language Reference.

Return value:

The result of `CurrentToMoment` is the elapsed time in *Unit* since *ReferenceDate*.

See also:

- The function `StringToMoment`.
- The AIMMS blog post: Creating StopWatch in AIMMS to time execution illustrates the use of some time functions. The purpose of `CurrentToMoment` in that post is to compute the time since a starting point.

CurrentToString

The function `CurrentToString` 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 Section 33.7.

Return value:

The result of `CurrentToString` is a description of the current time according to *Format*.

Remarks:

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`, `CurrentToMoment`.
- The AIMMS blog post: Creating StopWatch in AIMMS to time execution illustrates the use of some time functions. The purpose of `CurrentToString` in that post is to mark the starting point.

CurrentToTimeSlot

The function `CurrentToTimeSlot` 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` returns the time slot in the calendar that contains the current moment.

Remarks:

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 `StringToTimeSlot`, `MomentToTimeSlot`.

DaylightSavingEndDate

The function `DaylightSavingEndDate` 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`.

Return value:

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

See also:

AIMMS support for time zones is discussed in full detail in Sections 33.7.4 and 33.10 of the Language Reference.

DaylightSavingStartDate

The function `DaylightSavingStartDate` 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`.

Return value:

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

See also:

AIMMS support for time zones is discussed in full detail in Sections 33.7.4 and 33.10 of the Language Reference.

DisAggregate

With the procedure `DisAggregate` 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` (`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.

See also:

The procedure `Aggregate`. Time-dependent aggregation and disaggregation is discussed in full detail in Section 33.5 of the Language Reference.

MomentToString

The function `MomentToString` 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 Section 33.7.

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* on page 544 of the Language Reference.

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

Return value:

The result of `MomentToString` is a string describing the corresponding moment according to *Format*.

See also:

The function `StringToMoment`.

MomentToTimeSlot

The function `MomentToTimeSlot` 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` 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.

See also:

The functions `TimeSlotToMoment`, `CurrentToTimeSlot`, `StringToTimeSlot`.

PeriodToString

With the function `PeriodToString` 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 Section 33.7

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` is a string describing the corresponding moment according to *Format*.

See also:

The procedure `CreateTimeTable`.

StringToMoment

The function `StringToMoment` 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 Section 33.7.

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* on page 544 of the Language Reference.

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` is the elapsed time in *unit* between *reference-date* and *date*.

See also:

The functions `MomentToString`, `CurrentToMoment`.

StringToTimeSlot

The function `StringToTimeSlot` 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 Section 33.7.

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` returns the time slot in the calendar that contains the given moment.

See also:

The functions `CurrentToTimeSlot`, `MomentToTimeSlot`.

TestDate

The function `TestDate` 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 Section 33.7.

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` 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` will contain a proper error message.

Examples:

```
ok := TestDate( "%c%y-%m-%d", "2015-xx-xx" ); ! ok becomes 0; Not numeric.
ok := TestDate( "%c%y-%m-%d", "2015-02-29" ); ! ok becomes 0; Feb 2015 has only 28 days.
ok := TestDate( "%c%y-%m-%d", "2016-02-29" ); ! ok becomes 1; Feb 29, 2016 exists.
ok := TestDate( "%c%y-%m-%d", "2015-04-31" ); ! ok becomes 0; April 31 does not exist.
ok := TestDate( "%c%y-%m-%d", "2015-04-01" ); ! ok becomes 1; April 01 does exist (-;
ok := TestDate( "%m-%d", "03-03", requireUnique:1 ); ! Not unique, ok becomes 0.
ok := TestDate( "%m-%d", "03-03", requireUnique:0 ); ! Uniqueness not required; ok becomes 1.
```

See also:

The function `CurrentToString`.

TimeSlotCharacteristic

The function `TimeSlotCharacteristic` 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`, 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`.

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

Return value:

The function `TimeSlotCharacteristic` returns a numerical value for the requested time slot characteristic.

See also:

The function `TimeSlotCharacteristic` is discussed in full detail in Section 33.4 of the Language Reference.

TimeSlotToMoment

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

See also:

The functions `MomentToTimeSlot`, `CurrentToTimeSlot`, `StringToTimeSlot`.

TimeSlotToString

The function TimeSlotToString 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 Section 33.7.

Calendar
An identifier of type calendar.

Timeslot
A specific time-slot in the calendar.

Return value:

The function TimeSlotToString returns a string representation of the time slot.

See also:

The functions MomentToString, CurrentToTimeSlot, StringToTimeSlot.

TimeZoneOffSet

The function `TimeZoneOffSet` 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`.

ToTZ
: An element from the set `AllTimeZones`.

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` is the offset, in minutes, between *FromTZ* and *ToTZ*.

Remarks:

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).

See also:

AIMMS support for time zones is discussed in full detail in Sections 33.7.4 and 33.10 of the Language Reference.

Chapter 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' financial functions always expects 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:

- General conversion functions
- Day count bases and dates
- Depreciation of assets
- Investments and loans
- Securities

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`
- `PriceFractional`

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`
- `RateNominal`

PriceDecimal

The function `PriceDecimal` 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` 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.

Remarks:

- 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` is similar to the Excel function `DOLLARDE`.

See also:

The function `PriceFractional`.

PriceFractional

The function `PriceFractional` 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` returns the *DecimalPrice* expressed as a fractional number.

Remarks:

- The system of equations on which the conversion between decimal and fractional prices is based, is explained for the function `PriceDecimal` (the inverse of `PriceFractional`).
- The function `FractionalDecimal` is similar to the Excel function `DOLLARFR`.

See also:

The function `PriceDecimal`.

RateEffective

The function `RateEffective` 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` 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.

Remarks:

- 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` is similar to the Excel function `EFFECT`.

See also:

The function `RateNominal`.

RateNominal

The function `RateNominal` 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` returns the nominal annual interest rate expressed as a fraction.

Remarks:

- The equation on which the conversion between nominal and effective rates is based, is explained for the function `RateEffective` (the inverse of `RateNominal`).
- 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` is similar to the Excel function `NOMINAL`.

See also:

The function `RateEffective`.

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' 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' 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`.

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`
- `DateDifferenceYearFraction`

DateDifferenceDays

The function `DateDifferenceDays` 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` returns the number of days between the two dates.

Remarks:

The function `DateDifferenceDays` is similar to the Excel function `DAYS300`.

See also:

Day count basis methods.

DateDifferenceYearFraction

The function `DateDifferenceYearFraction` 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` returns the difference between *FirstDate* and *SecondDate* in fractions of a year.

Remarks:

The function `DateDifferenceYearFraction` is similar to the Excel function `YEARFRAC`.

See also:

Day count basis methods.

6.3 Depreciations

Depreciation functions

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 method.

AIMMS supports the following linear depreciation by constant amounts functions:

- `DepreciationLinearLife`
- `DepreciationLinearRate`

AIMMS supports the following non-linear depreciation by linear declining amounts functions:

- DepreciationNonLinearSumOfYear

Aimms supports the following non-linear depreciation by non-linear declining amounts functions:

- DepreciationNonLinearLife
- DepreciationNonLinearFactor
- DepreciationNonLinearRate
- DepreciationSum

DepreciationLinearLife

The function `DepreciationLinearLife` 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 $[0, \mathit{Cost})$.

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 $\{1, \mathit{Life} + 1\}$. 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` returns the depreciation of an asset for the specified period.

Equation:

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equation

$$\begin{aligned} \tilde{d}_1 &= f_{PN}\frac{c-s}{L} \\ \tilde{d}_i &= \frac{c-s}{L} \qquad (i \neq 1). \end{aligned}$$

Remarks:

The function `DepreciationLinearLife` is similar to the Excel function `SLN`.

See also:

Day count basis methods. General equations for computing depreciations.

DepreciationLinearRate

The function `DepreciationLinearRate` 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` returns the depreciation of an asset for the specified period.

Equation:

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equation

$$\begin{aligned} \tilde{d}_1 &= f_{PN} r c \\ \tilde{d}_i &= r c \qquad (i \neq 1) \end{aligned}$$

where r is the depreciation rate.

Remarks:

- 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` is similar to the Excel function `AMORLINC`.

See also:

Day count basis methods. General equations for computing depreciations.

DepreciationNonLinearSumOfYear

The function `DepreciationNonLinearSumOfYear` 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 $[0, \mathit{Cost})$.

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 $\{1, \mathit{Life} + 1\}$. 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` returns the depreciation of an asset for the specified period.

Equation:

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equation

$$\begin{aligned}
\tilde{d}_1 &= \frac{c - s}{\frac{1}{2}L(L+1)} L f_{PN} \\
\tilde{d}_i &= \frac{c - s}{\frac{1}{2}L(L+1)} (L + 2 - i - f_{PN}) \qquad (i \neq 1).
\end{aligned}$$

Remarks:

The function `DepreciationNonLinearSumOfYear` is similar to the Excel function `SYD`.

See also:

Day count basis methods. General equations for computing depreciations.

DepreciationNonLinearLife

The function DepreciationNonLinearLife 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
    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` returns the depreciation of an asset for the specified period.

Equation:

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equations

$$\tilde{d}_1 = \begin{cases} f_{PN} r v_1 & \text{for } \mathit{Mode} = 0 \\ \left(1 - (1-r)^{f_{PN}}\right) v_1 & \text{for } \mathit{Mode} = 1 \end{cases}$$

$$\tilde{d}_i = r v_i \qquad (i \neq 1)$$

where the depreciation rate r equals

$$r = 1 - \left(\frac{s}{c}\right)^{1/L}$$

Remarks:

The function `DepreciationLinearNonLife` is similar to the Excel function `DB`.

See also:

Day count basis methods. General equations for computing depreciations.

DepreciationNonLinearFactor

The function `DepreciationNonLinearFactor` 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` returns the depreciation of an asset for the specified period.

Equation:

The method-dependent depreciation $\tilde{d}_i$ is expressed by the equations

$$\begin{aligned} \tilde{d}_1 &= \begin{cases} f_{PN} r c & \text{for } \mathit{Mode} = 0 \\ \left(1 - (1-r)^{f_{PN}}\right) c & \text{for } \mathit{Mode} = 1 \end{cases} \\ \tilde{d}_i &= (c - d_1) r (1-r)^{i-2} \qquad\qquad (i \neq 1) \end{aligned}$$

where the depreciation rate r equals

$$r = \frac{f}{L}$$

with f the *Factor* argument.

Remarks:

- 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`.

See also:

Day count basis methods. General equations for computing depreciations.

DepreciationNonLinearRate

The function `DepreciationNonLinearRate` 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` 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

$$f = \begin{cases} 1.5 & \text{for } \frac{1}{4} \le r < \frac{1}{2} \\ 2.0 & \text{for } \frac{1}{6} \le r < \frac{1}{4} \\ 2.5 & \text{for } r < \frac{1}{6} \end{cases}$$

Remarks:

The function `DepreciationLinearNonRate` is similar to the Excel function `AMORDEGRC`.

See also:

Day count basis methods. General equations for computing depreciations.

DepreciationSum

The function `DepreciationSum` 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 $\{1, \mathit{Life}\}$. *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 $\{\mathit{StartPeriod}, \mathit{Life}\}$. *EndPeriod* must have the same unit as *Life*.

Factor
: The rate by which the depreciation declines is $\frac{\mathit{Factor}}{\mathit{Life}}$. *Factor* must be a numerical expression in the range $[1, \infty)$. In case $\mathtt{Factor} = 2$ 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. $\mathit{Mode} = 0$: 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. $\mathit{Mode} = 1$: 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 $\mathit{Switch} = 0$: do not switch, if $\mathit{Switch} = 1$: switch. The default is 1.

Return value:

The function `DepreciationSum` returns the depreciation of an asset for the specified period.

Remarks:

The function `DepreciationSum` is similar to the Excel function `VDB`.

See also:

The functions `DepreciationNonLinearFactor`, `DepreciationLinearLife`. Day count basis methods. General equations for computing depreciations.

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
- InvestmentConstantFutureValue
- InvestmentConstantPeriodicPayment
- InvestmentConstantInterestPayment
- InvestmentConstantPrincipalPayment
- InvestmentConstantCumulativeInterestPayment
- InvestmentConstantCumulativePrincipalPayment
- InvestmentConstantNumberPeriods
- InvestmentConstantRateAll
- InvestmentConstantRate

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
- InvestmentVariablePresentValueInPeriodic
- InvestmentSingleFutureValue
- InvestmentVariableInternalRateReturnAll
- InvestmentVariableInternalRateReturn
- InvestmentVariableInternalRateReturnInPeriodicAll
- InvestmentVariableInternalRateReturnInPeriodic
- InvestmentVariableInternalRateReturnModified

InvestmentConstantPresentValue

The function `InvestmentConstantPresentValue` 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` returns the total amount that a series of future payments is worth at this moment.

Remarks:

- 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` is similar to the Excel function `PV`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantFutureValue

The function `InvestmentConstantFutureValue` 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` returns the cash balance you want to attain after the last payment is made.

Remarks:

- 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` is similar to the Excel function `FV`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantPeriodicPayment

The function `InvestmentConstantPeriodicPayment` 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` returns the periodic payment for the investment.

Remarks:

- 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` is similar to the Excel function `PMT`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantInterestPayment

The function `InvestmentConstantInterestPayment` 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` returns the interest payment for the specified period.

Equation:

The interest payment i_i in period i is computed through the equation

$$i_i = -v_p r(1+r)^{i-1-T} - p\left(\left((1+r)^{i-1-T} - 1\right)(1+r)^T + rT\right)$$

Remarks:

- 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` is similar to the Excel function `IPMT`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantPrincipalPayment

The function `InvestmentConstantPrincipalPayment` 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` 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.

Remarks:

- 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` is similar to the Excel function `PPMT`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantCumulativeInterestPayment

The function `InvestmentConstantCumulativeInterestPayment` 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` returns the sum of the interest payments for the periods in the specified interval.

Remarks:

- 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` is similar to the Excel function `CUMIPMT`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantCumulativePrincipalPayment

The function `InvestmentConstantCumulativePrincipalPayment` 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 $\{1, \mathit{NumberPeriods}\}$.

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 $\{\mathit{StartPeriod}, \mathit{NumberPeriods}\}$.

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. $\mathit{Type} = 0$: Payments are due at the end of each period. $\mathit{Type} = 1$: Payments are due at the beginning of each period.

Return value:

The function `InvestmentConstantCumulativePrincipalPayment` returns the sum of the principal payments for the periods in the specified interval.

Remarks:

- 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` is similar to the Excel function `CUMPRINC`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantNumberPeriods

The function `InvestmentConstantNumberPeriods` 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
    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` returns the number of periods for an investment based on periodic, constant payments and a constant interest rate.

Remarks:

The function `InvestmentConstantNumberPeriods` is similar to the Excel function `NPER`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantRateAll

The procedure `InvestmentConstantRateAll` 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.

Remarks:

- When you want to use this procedure in an objective function or constraint you have to use `InvestmentConstantRate`.
- The function `InvestmentConstantRateAll` is similar to the Excel function `RATE`.

See also:

General equations for investments with constant, periodic payments.

InvestmentConstantRate

The function `InvestmentConstantRate` 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` returns the interest rate for an investment based on periodic, constant payments and a constant interest rate.

Remarks:

- The function `InvestmentConstantRate` can be used in an objective function or constraint. The input parameters *PresentValue*, *FutureValue* and *Payment* can be used as variables.
- The function `InvestmentConstantRate` is similar to the Excel function `RATE`.

See also:

General equations for investments with constant, periodic payments.

InvestmentVariablePresentValue

The function `InvestmentVariablePresentValue` 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` 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.

Remarks:

- 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` 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` is similar to the Excel function `NPV`.

See also:

The function `InvestmentConstantPresentValue`.

InvestmentVariablePresentValueInPeriodic

The function `InvestmentVariablePresentValueInPeriodic` 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` 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.

Remarks:

- 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` is similar to the Excel function `XNPV`.

See also:

Day count basis methods.

InvestmentSingleFutureValue

The function `InvestmentSingleFutureValue` 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` 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.

Remarks:

- 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` is similar to the Excel function `FVSCHEDULE`.

InvestmentVariableInternalRateReturnAll

The procedure `InvestmentVariableInternalRateReturnAll` 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.

Remarks:

- 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` is similar to the Excel function `IRR`.

See also:

The functions `InvestmentVariableInternalRateReturn`, `InvestmentVariableInternalRateReturnInPeriodic`.

InvestmentVariableInternalRateReturn

The function `InvestmentVariableInternalRateReturn` 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` 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.

Remarks:

- The function InvestmentVariableInternalRateReturn can be used in an objective function or constraint. The input parameter *Value* can be used as a variable.
- The function InvestmentVariableInternalRateReturn is similar to the Excel function IRR.

See also:

The functions InvestmentVariableInternalRateReturnAll, InvestmentVariableInternalRateReturnInPeriodic.

InvestmentVariableInternalRateReturnInPeriodicAll

The procedure `InvestmentVariableInternalRateReturnInPeriodicAll` 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
    NumberSolutions,          ! (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.

Remarks:

- When you want to use the procedure in an objective function or constraint you have to use `InvestmentVariableInternalRateReturnInPeriodic`.
- The procedure `InvestmentVariableInternalRateReturnInPeriodicAll` is similar to the Excel function `XIRR`.

See also:

The functions `InvestmentVariableInternalRateReturn`, `InvestmentVariableInternalRateReturnInPeriodic`. Day count basis methods.

InvestmentVariableInternalRateReturnInPeriodic

The function `InvestmentVariableInternalRateReturnInPeriodic` 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` 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.

Remarks:

- The function `InvestmentVariableInternalRateReturnInPeriodic` can be used in an objective function or constraint. The input parameter *Value* can be used as a variable.
- The function `InvestmentVariableInternalRateReturnInPeriodic` is similar to the Excel function `XIRR`.

See also:

The functions `InvestmentVariableInternalRateReturn`, `InvestmentVariableInternalRateReturnInPeriodicAll`. Day count basis methods.

InvestmentVariableInternalRateReturnModified

The function `InvestmentVariableInternalRateReturnModified` 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` returns the modified internal rate of return for the investment.

Equation:

The internal rate of return r is the solution of the equation

$$(1+r)^{n-1} = -\frac{\mathrm{NPV}(v^+, r_r)(1+r_r)^n}{\mathrm{NPV}(v^-, r_f)(1+r_f)}$$

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`.

Remarks:

- 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 negative *Value*.
- The function `InvestmentVariableInternalRateReturnModified` is similar to the Excel function `MIRR`.

See also:

The function `InvestmentVariableInternalRateReturn`.

6.5 Securities

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 securitys 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.

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`
- `SecurityDiscountedRedemption`
- `SecurityDiscountedYield`
- `SecurityDiscountedRate`
- `TreasuryBillPrice`
- `TreasuryBillYield`
- `TreasuryBillBondEquivalent`

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, 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, 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.

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
- SecurityMaturityCouponRate
- SecurityMaturityYield
- SecurityMaturityAccruedInterest

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,
- 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,
- 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
- SecurityCouponPreviousDate
- SecurityCouponNextDate
- SecurityCouponDays
- SecurityCouponDaysPreSettlement
- SecurityCouponDaysPostSettlement
- SecurityPeriodicPrice
- SecurityPeriodicRedemption
- SecurityPeriodicCouponRate
- SecurityPeriodicYieldAll
- SecurityPeriodicYield
- SecurityPeriodicAccruedInterest
- SecurityPeriodicDuration
- SecurityPeriodicDurationModified

SecurityDiscountedPrice

The function SecurityDiscountedPrice 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 percentage 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 returns the price of the security at settlement date.

Remarks:

- 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 is similar to the Excel function PRICEDISC.

See also:

Day count basis methods. General equations for discounted securities.

SecurityDiscountedRedemption

The function SecurityDiscountedRedemption 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 percentage 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 returns the amount paid at maturity date.

Remarks:

- 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 is similar to the Excel function RECEIVED.

See also:

Day count basis methods. General equations for discounted securities.

SecurityDiscountedYield

The function `SecurityDiscountedYield` 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` returns the annual rate the security's value increases as a percentage of the price.

Remarks:

- 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` is similar to the Excel function `YIELDDISC`.

See also:

Day count basis methods. General equations for discounted securities.

SecurityDiscountedRate

The function `SecurityDiscountedRate` 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` returns the annual rate the security's value increases as a percentage of the redemption value.

Remarks:

- 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` is similar to the Excel function `DISC`.

See also:

Day count basis methods. General equations for discounted securities.

TreasuryBillPrice

The function `TreasuryBillPrice` 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 percentage of the redemption. *DiscountRate* must be a positive real number.

Return value:

The function `TreasuryBillPrice` returns the price of a Treasury bill at settlement date.

Remarks:

- 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` is similar to the Excel function `TBILLPRICE`.

See also:

General equations for discounted securities.

TreasuryBillYield

The function `TreasuryBillYield` 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` returns the annual rate the Treasury bill's value increases as a percentage of the price.

Remarks:

- 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` is similar to the Excel function `TBILLYIELD`.

See also:

General equations for discounted securities.

TreasuryBillBondEquivalent

The function `TreasuryBillBondEquivalent` 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` returns the bond equivalent yield of a Treasury bill.

Remarks:

- 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` is similar to the Excel function `TBILLEQ`.

See also:

General equations for discounted securities.

SecurityMaturityPrice

The function SecurityMaturityPrice 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 returns the price of the security at settlement date.

Remarks:

- 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` is similar to the Excel function `PRICEMAT`.

See also:

Day count basis methods. General equations for securities with one coupon.

SecurityMaturityCouponRate

The function `SecurityMaturityCouponRate` 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
    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` returns the annual interest rate of the security as a percentage of the par value.

Remarks:

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.

See also:

Day count basis methods. General equations for securities with one coupon.

SecurityMaturityYield

The function `SecurityMaturityYield` 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` returns the annual rate the security's value increases as a percentage of the price.

Remarks:

- 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` is similar to the Excel function `YIELDMAT`.

See also:

Day count basis methods. General equations for securities with one coupon.

SecurityMaturityAccruedInterest

The function SecurityMaturityAccruedInterest 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 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 SecurityMaturityAccruedInterest returns the interest accrued from issue date until settlement date.

Remarks:

- 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 is similar to the Excel function ACCRINTM.

See also:

Day count basis methods. General equations for securities with one coupon.

SecurityCouponNumber

The function SecurityCouponNumber 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 returns the number of coupon payments from the settlement date until the maturity date.

Remarks:

The function SecurityCouponNumber is similar to the Excel function COUPNUM.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityCouponPreviousDate

The function `SecurityCouponPreviousDate` 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.

Remarks:

The function `SecurityCouponPreviousDate` is similar to the Excel function `COUPPCD`.

See also:

General equations for securities with multiple coupons.

SecurityCouponNextDate

The function `SecurityCouponNextDate` 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.

Remarks:

The function `SecurityCouponNextDate` is similar to the Excel function `COUPNCD`.

See also:

General equations for securities with multiple coupons.

SecurityCouponDays

The function `SecurityCouponDays` 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` returns the number of days of the coupon period in which the settlement date falls.

Remarks:

The function `SecurityCouponDays` is similar to the Excel function `COUPDAYS`.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityCouponDaysPreSettlement

The function SecurityCouponDaysPreSettlement 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 returns the number of days from the previous coupon-date until the settlement date, using the specified day-count basis.

Remarks:

The function SecurityCouponDaysPreSettlement is similar to the Excel function COUPDAYBS.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityCouponDaysPostSettlement

The function SecurityCouponDaysPostSettlement 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 returns the number of days from the first coupon-date next to settlement date until settlement date.

Remarks:

The function SecurityCouponDaysPostSettlement is similar to the Excel function COUPDAYSNC.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicPrice

The function `SecurityPeriodicPrice` 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` returns the price of the security at settlement date.

Remarks:

- 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` is similar to the Excel function `PRICE`.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicRedemption

The function SecurityPeriodicRedemption 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 returns the amount repaid for the security at the maturity date.

Remarks:

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.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicCouponRate

The function SecurityPeriodicCouponRate 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 returns the interest rate per year of the security as a percentage of the par value.

Remarks:

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.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicYieldAll

The procedure SecurityPeriodicYieldAll 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.

Remarks:

- When you want to use this procedure in an objective function or constraint you have to use `SecurityPeriodicYield`.
- The function `SecurityPeriodicYieldAll` is similar to the Excel function `YIELD`.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicYield

The function SecurityPeriodicYield 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` returns the yield of a security that pays interest at the end of each coupon period.

Remarks:

- 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` is similar to the Excel function `YIELD`.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicAccruedInterest

The function `SecurityPeriodicAccruedInterest` 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` returns the interest accrued from the begin of the coupon period until settlement date.

Remarks:

This function can be used in an objective function or constraint and the input parameters *ParValue* and *CouponRate* can be used as a variable.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicDuration

The function `SecurityPeriodicDuration` 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` 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 for securities with multiple coupons.

Remarks:

- 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` is similar to the Excel function `DURATION`.

See also:

Day count basis methods. General equations for securities with multiple coupons.

SecurityPeriodicDurationModified

The function SecurityPeriodicDurationModified 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 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.

Remarks:

- 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` is similar to the Excel function `MDURATION`.

See also:

The function `SecurityPeriodicDuration`. Day count basis methods. General equations for securities with multiple coupons.

Chapter 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
- Geometric
- HyperGeometric
- NegativeBinomial
- Poisson

The functions for continuous distributions are:

- Beta
- Exponential
- ExtremeValue
- Gamma
- Logistic
- LogNormal
- Normal
- Pareto
- Triangular
- Uniform
- Weibull

The following functions that operate on distributions are available:

- DistributionCumulative
- DistributionInverseCumulative
- DistributionDensity
- DistributionInverseDensity
- DistributionMean
- DistributionDeviation
- DistributionVariance
- DistributionSkewness
- DistributionKurtosis

The combinatoric functions are:

- Combination

- `Factorial`
- `Permutation`

Binomial

The function `Binomial` 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` returns a random value drawn from a binomial distribution with a probability of success *ProbabilityOfSuccess* and number of tries *NumberOfTries*

See also:

The `Binomial` distribution is discussed in full detail in Appendix A of the Language Reference.

Geometric

The function `Geometric` draws a random value from a geometric distribution.

```
Geometric(
   ProbabilityOfSuccess    ! (input) numerical expression
         )
```

Arguments:

ProbabilityOfSuccess
: A scalar numerical expression in the range $(0, 1)$.

Return value:

The function `Geometric` returns a random value drawn from a geometric distribution with a probability of success *ProbabilityOfSuccess*.

See also:

The `Geometric` distribution is discussed in full detail in Appendix A of the Language Reference.

HyperGeometric

The function `HyperGeometric` 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 $1, \ldots, \mathit{PopulationSize}$.

PopulationSize
: A integer numerical expression > 0.

Return value:

The function `HyperGeometric` returns a random value drawn from a hypergeometric distribution with a probability of success *ProbabilityOfSuccess*, number of tries *NumberOfTries* and population size *PopulationSize*.

Remarks:

The probability of success *ProbabilityOfSuccess* must assume one of the values i/size, where i is in the range $1, \ldots, \mathit{PopulationSize} - 1$.

See also:

The `HyperGeometric` distribution is discussed in full detail in Appendix A of the Language Reference.

NegativeBinomial

The function `NegativeBinomial` draws a random value from a negative binomial distribution.

```
NegativeBinomial(
    ProbabilityOfSuccess,   ! (input) numerical expression
    NumberOfSuccesses       ! (input) integer expression
    )
```

Arguments:

ProbabilityOfSuccess
A scalar numerical expression in the range $(0, 1)$.

NumberOfSuccesses
A integer numerical expression > 0.

Return value:

The function `NegativeBinomial` returns a random value drawn from a negative binomial distribution with probability *ProbabilityOfSuccess* and number of successes *NumberOfSuccesses*.

See also:

The `NegativeBinomial` distribution is discussed in full detail in Appendix A of the Language Reference.

Poisson

The function `Poisson` 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` returns a random value drawn from a Poisson distribution with average number of occurrences *AverageNumberOfSuccesses*.

See also:

The `Poisson` distribution is discussed in full detail in Appendix A of the Language Reference.

Beta

The function Beta 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 returns a random value drawn from a beta distribution with shapes *ShapeAlpha*, *ShapeBeta*, lower bound *Minimum* and upper bound *Maximum*.

Remarks:

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(*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.

See also:

The Beta distribution is discussed in full detail in Appendix A of the Language Reference.

Exponential

The function `Exponential` 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` returns a random value drawn from a exponential distribution with lower bound *lowerbound* and scale *scale*.

Remarks:

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`(*lambda*) returns a random value drawn from a exponential distribution with rate $lambda = 1/scale$ and lower bound 0.

See also:

The `Exponential` distribution is discussed in full detail in Appendix A of the Language Reference.

ExtremeValue

The function `ExtremeValue` draws a random value from an extreme value distribution.

```
ExtremeValue(
       location,    ! (optional) numerical expression
       scale        ! (optional) numerical expression
       )
```

Arguments:

location
: A scalar numerical expression.

scale
: A scalar numerical expression > 0.

Return value:

The function `ExtremeValue` returns a random value drawn from an extreme value distribution with location *location* and scale *scale*.

See also:

The `ExtremeValue` distribution is discussed in full detail in Appendix A of the Language Reference.

Gamma

The function Gamma 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 returns a random value drawn from a gamma distribution with shape *Shape*, lower bound *Lowerbound* and scale *Scale*.

Remarks:

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(*alpha*, *Shape*) returns a random value drawn from a gamma distribution with rate $alpha = 1/Scale$, shape *Shape* and lower bound 0.

See also:

The Gamma distribution is discussed in full detail in Appendix A of the Language Reference.

Logistic

The function `Logistic` 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` returns a random value drawn from a logistic distribution with mean *Location* and scale *Scale*.

See also:

The `Logistic` distribution is discussed in full detail in Appendix A of the Language Reference.

LogNormal

The function LogNormal 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 returns a random value drawn from a lognormal distribution with shape *Shape*, lower bound *Lowerbound* and scale *Scale*.

Remarks:

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(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 $Scale = m$.

See also:

The LogNormal distribution is discussed in full detail in Appendix A of the Language Reference.

Normal

The function `Normal` 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` returns a random value drawn from a normal distribution with mean *Mean* and standard deviation *Deviation*.

See also:

The `Normal` distribution is discussed in full detail in Appendix A of the Language Reference.

Pareto

The function `Pareto` 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` returns a random value drawn from a Pareto distribution with shape *Shape*, location *Location* and scale *Scale*.

Remarks:

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`(*s*, *beta*) returns a random value drawn from a Pareto distribution with shape *beta*, location 0 and scale *s*.

See also:

The `Pareto` distribution is discussed in full detail in Appendix A of the Language Reference.

Triangular

The function `Triangular` 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` 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 $0 < Shape < 1$.

Remarks:

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`(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 $Shape = (b - a)/(c - a)$.

See also:

The `Triangular` distribution is discussed in full detail in Appendix A of the Language Reference.

Uniform

The function `Uniform` 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` returns a random value drawn from a uniform distribution with lower bound *Minimum* and upper bound *Maximum*.

Remarks:

The arguments must satisfy the relation $Minimum < Maximum$.

See also:

The `Uniform` distribution is discussed in full detail in Appendix A of the Language Reference.

Weibull

The function `Weibull` 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` returns a random value drawn from a Weibull distribution with shape *Shape* lower bound *Lowerbound*, and scale *Scale*.

Remarks:

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`(*Lowerbound*, *Shape*, *Scale*), the arguments were ordered differently.

See also:

The `Weibull` distribution is discussed in full detail in Appendix A of the Language Reference.

DistributionCumulative

The function `DistributionCumulative` 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*.

Remarks:

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`. The function `DistributionCumulative` is discussed in full detail in Appendix A of the Language Reference.

DistributionInverseCumulative

The function `DistributionInverseCumulative` 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`(*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*.

Remarks:

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`. The function `DistributionInverseCumulative` is discussed in full detail in Appendix A of the Language Reference.

DistributionDensity

The function `DistributionDensity` 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`(*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`, `DistributionInverseDensity`. The function `DistributionDensity` is discussed in full detail in Appendix A of the Language Reference.

DistributionInverseDensity

The function `DistributionInverseDensity` 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`(*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`. The function `DistributionInverseDensity` is discussed in full detail in Appendix A of the Language Reference.

DistributionMean

The function `DistributionMean` 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`(*distribution*) returns the mean of the given *distribution*.

See also:

You can find more information about the mean of a distribution in Appendix A of the Language Reference.

DistributionDeviation

The function `DistributionDeviation` 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`(*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 Appendix A of the Language Reference.

DistributionVariance

The function `DistributionVariance` 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`(*distribution*) returns the variance of the given *distribution*.

See also:

You can find more information about the variance of a distribution in Appendix A of the Language Reference.

DistributionSkewness

The function `DistributionSkewness` 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`(*distribution*) returns the skewness of the given *distribution*.

See also:

You can find more information about the skewness of a distribution in Appendix A of the Language Reference.

DistributionKurtosis

The function `DistributionKurtosis` 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`(*distribution*) returns the kurtosis of the given *distribution*.

See also:

You can find more information about the kurtosis of a distribution in Appendix A of the Language Reference.

Combination

The function `Combination` 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` 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 Section 6.1.7.

Factorial

The function `Factorial` returns the factorial of an integer number.

```
Factorial(
         n             ! (input) integer expression
         )
```

Arguments:

n

An integer numerical expression ≥ 0.

Return value:

The function `Factorial` returns the factorial value $n!$.

See also:

Combinatoric functions are discussed in full detail in Section 6.1.7.

Permutation

The function `Permutation` 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` 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 Section 6.1.7.

Chapter 8

Histogram Functions

AIMMS supports the following functions for creating and managing histograms:

- HistogramAddObservation
- HistogramAddObservations
- HistogramCreate
- HistogramDelete
- HistogramGetAverage
- HistogramGetBounds
- HistogramGetDeviation
- HistogramGetFrequencies
- HistogramGetKurtosis
- HistogramGetObservationCount
- HistogramGetSkewness
- HistogramSetDomain

HistogramAddObservation

The procedure `HistogramAddObservation` 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`, `HistogramCreate`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramAddObservations

The procedure `HistogramAddObservations` 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`, `HistogramCreate`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramCreate

The function `HistogramCreate` 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`, `HistogramAddObservation`, `HistogramAddObservations`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the User's Guide.

HistogramDelete

The procedure HistogramDelete 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. Histogram support in AIMMS is discussed in full detail in Section A.6 of the User's Guide.

HistogramGetAverage

The function `HistogramGetAverage` 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`, `HistogramGetObservationCount`, `HistogramGetDeviation`, `HistogramGetSkewness`, `HistogramGetKurtosis`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the User's Guide.

HistogramGetBounds

Through the function HistogramGetBounds 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, HistogramSetDomain. Histogram support in Aimms is discussed in full detail in Section A.6 of the Language Reference.

HistogramGetDeviation

The function `HistogramGetDeviation` 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`, `HistogramGetObservationCount`, `HistogramGetAverage`, `HistogramGetSkewness`, `HistogramGetKurtosis`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramGetFrequencies

Through the procedure `HistogramGetFrequencies` 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`, `HistogramAddObservation`, `HistogramAddObservations`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramGetKurtosis

The function `HistogramGetKurtosis` 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`, `HistogramGetObservationCount`, `HistogramGetAverage`, `HistogramGetDeviation`, `HistogramGetSkewness`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramGetObservationCount

The function `HistogramGetObservationCount` 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`, `HistogramGetAverage`, `HistogramGetDeviation`, `HistogramGetSkewness`, `HistogramGetKurtosis`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramGetSkewness

The function `HistogramGetSkewness` 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`, `HistogramGetObservationCount`, `HistogramGetAverage`, `HistogramGetDeviation`, `HistogramGetKurtosis`. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

HistogramSetDomain

With the procedure `HistogramSetDomain` 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
     [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, HistogramGetBounds. Histogram support in AIMMS is discussed in full detail in Section A.6 of the Language Reference.

Chapter 9

Forecasting Functions

Aimms supports the following functions for making forecasts:

- `forecasting::MovingAverage`
- `forecasting::WeightedMovingAverage`
- `forecasting::ExponentialSmoothing`
- `forecasting::ExponentialSmoothingTrend`
- `forecasting::ExponentialSmoothingTrendSeasonality`
- `forecasting::ExponentialSmoothingTune`
- `forecasting::ExponentialSmoothingTrendTune`
- `forecasting::ExponentialSmoothingTrendSeasonalityTune`
- `forecasting::SimpleLinearRegression`

9.1 Introduction

Aimms is a development tool for decision support application. Important to decision support are good forecasts. The **AimmsForecasting** library provides tools to compute forecasts from historical data. The usage of this library is discussed in this chapter.

installation

Before the functions in this section can be used in your model, you will need to add the library

prefix

The prefix of the **AIMMSForecasting** library is **forecasting**.

Restriction

This library does not support the special values `NA`, `ZERO`, `-INF`, `INF`, and `UNDF`.

9.2 Time series forecasting

9.2.1 Notational conventions time series forecasting

For time series forecasting, such as **Moving Average** and **Exponential Smoothing**, we follow the conventions below.

Observations and Estimates

The `AIMMSForecasting` library uses as input observations made in the history. It provides estimates over both the history and the horizon. A single set and index is used to index both the history and the estimates, this set is called the **time set**. In addition, you will need to specify the number of elements that belong to the history. The corresponding mathematical notation is:

T	number of observations
H	length of horizon
$\{1 \ldots T + H\}$	time set
t	index in time set
$y_t, t \in \{1 \ldots T\}$	observation
$e_t, t \in$ time set	estimate

Table 9.1: Time series forecasting notation

The forecasts are provided in e_t, $t \in \{T + 1 \ldots T + H\}$.

residuals

The residual, r_t where $t \in \{1 \ldots T\}$, is the difference between the corresponding observation y_t and estimate e_t. To obtain the residuals, you will need to provide a parameter declared over the time set.

error measures

From the residuals, error measures such as Mean Absolute Deviation (MAD), Mean Absolute Percentage Error (MAPE), and Mean Squared Deviation (MSD) can be computed.

predeclared index **ems**

Whenever one of the time series forecasting functions communicates the error measures, it uses identifiers declared over the index `forecasting::ems`, declared as follows:

```
Set ErrorMeasureSet {
    Index: ems;
    Definition: {
        data {
            MAD,  ! Mean Absolute Deviation
            MAPE, ! Mean Absolute Percentage Error (provided as fraction)
            MSE   ! Mean Squared Error
        }
```

```
    }
}
```

To obtain the error measures, you will need to provide a parameter indexed over `forecasting::ems` to the time series forecasting functions. Note that the MAPE is a fraction, in order to use it as a percentage, you can use the predeclared quantity `SI_unitless`. For instance, given the declarations:

```
Quantity SI_Unitless {
    BaseUnit: -;
    Conversions: % -> - : # -> # / 100;
    Comment: "Expresses a dimensionless value.";
}
Parameter myMAPE {
    Unit: %;
}
Parameter myErrorMeasures {
    IndexDomain: forecasting::ems;
}
```

The following statements:

```
myMAPE := myErrorMeasures('MAPE') ;
display myErrorMeasures, myMAPE ;
```

The output may look as follows:

```
myErrorMeasures := data { MAPE : 0.417092,  MAD : 1.785714,  MSE : 3.982143 } ;
myMAPE := 41.709184 [%] ;
```

forecasting::MovingAverage

one The moving average procedure is a time series forecasting procedure. Essentially, this procedure forecasts by taking the average over the last N observations.

Mathematical Formulation:

Using the notation for observations and estimates given in Table 9.1, the estimates are defined as:

$$e_t = \sum_{\tau=t-1-N}^{t-1} \tilde{y}_\tau / N \quad \text{where } \tilde{y}_\tau = \begin{cases} y_1 & \text{if } \tau < 1 \\ y_\tau & \text{if } \tau \in \{1..T\} \\ e_\tau & \text{if } \tau > T \end{cases} \tag{9.1}$$

Function Prototype:

To provide the error measures and residuals only when you need them, there are three flavors of the `MovingAverage` procedure provided:

```
forecasting::MovingAverage(    ! Provides the estimates, but not the
                               ! error measures nor the residuals
      dataValues,              ! Input, parameter indexed over time set
      estimates,               ! Output, parameter indexed over time set
      noObservations,          ! Scalar input, length history
      noAveragingPeriods)      ! Scalar input, averaging length


forecasting::MovingAverageEM(  ! Provides estimates and error measures,
                               ! but not the residuals
      dataValues,              ! Input, parameter indexed over time set
      estimates,               ! Output, parameter indexed over time set
      noObservations,          ! Scalar input, length history
      noAveragingPeriods,      ! Scalar input, averaging length
      ErrorMeasures)           ! Output, indexed over forecasting::ems


forecasting::MovingAverageEMR( ! Provides estimates, error measures,
                               ! and residuals
      dataValues,              ! Input, parameter indexed over time set
      estimates,               ! Output, parameter indexed over time set
      noObservations,          ! Scalar input, length history
      noAveragingPeriods,      ! Scalar input, averaging length
      ErrorMeasures,           ! Output, indexed over forecasting::ems
      Residuals)               ! Output, parameter indexed over time set
```

Here, the time set is a set that encompasses both the history and the horizon.

Arguments:

dataValues
: A one dimensional parameter containing the observations for the first T elements of the time set.

estimates
: A one dimensional parameter containing the estimates for all elements in the time set.

noObservations
: Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

noAveragingPeriods
: Specifies the number of values used to compute a single average. This parameter corresponds to N in the mathematical notation above.

ErrorMeasures
: The error measures as presented in Section 9.2.

Residuals
: The residuals as presented in Section 9.2.

Example:

With declarations and data as specified in Table 9.2 the call:

```
forecasting::MovingAverage(
    dataValues         :  sampDat,
    estimates          :  sampEst1,
    noObservations     :  31,
    noAveragingPeriods :  5);
```

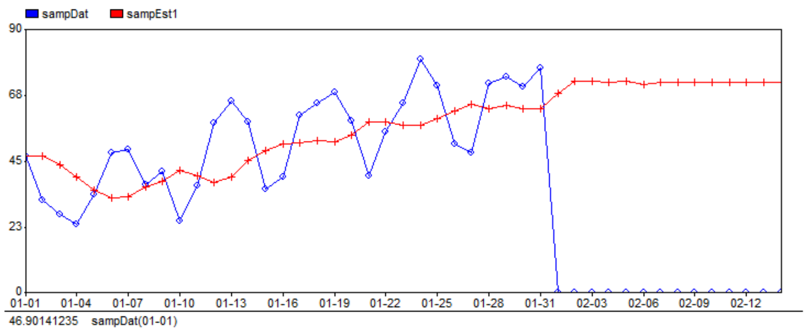
Will result in the following output:

```
sampEst1 := data
{ 01-01 : 46.90141235,  01-02 : 46.90141235,  01-03 : 43.90055356,
  01-04 : 39.91352947,  01-05 : 35.21374997,  01-06 : 32.58034743,
  01-07 : 32.80406692,  01-08 : 36.23403532,  01-09 : 38.29416296,
  01-10 : 41.90033337,  01-11 : 40.11936207,  01-12 : 37.82654624,
  01-13 : 39.63855369,  01-14 : 45.29956164,  01-15 : 48.72714940,
  01-16 : 50.90593288,  01-17 : 51.53221241,  01-18 : 52.07507740,
  01-19 : 51.93466798,  01-20 : 53.96574913,  01-21 : 58.57734065,
  01-22 : 58.68254227,  01-23 : 57.55058365,  01-24 : 57.53652213,
  01-25 : 59.80228910,  01-26 : 62.23264531,  01-27 : 64.41936052,
  01-28 : 62.97427964,  01-29 : 64.37015056,  01-30 : 63.12741111,
  01-31 : 63.07679348,  02-01 : 68.24492039,  02-02 : 72.30667944,
  02-03 : 72.41222140,  02-04 : 72.12629586,  02-05 : 72.41553283,
  02-06 : 71.50112998,  02-07 : 72.15237190,  02-08 : 72.12151039,
  02-09 : 72.06336819,  02-10 : 72.05078266,  02-11 : 71.97783263,
  02-12 : 72.07317316,  02-13 : 72.05733341,  02-14 : 72.04449801 } ;
```

This can be graphically displayed as:

```
Parameter sampDat {
    IndexDomain: d;
}
Parameter sampEst1 {
    IndexDomain: d;
}
Calendar dayCalendar {
    Index: d;
    Parameter: e_d;
    Unit: day;
    BeginDate: "2014-01-01";
    EndDate: "2014-02-14";
    TimeslotFormat: "%m-%d";
}

sampDat := data
{ 01-01 : 46.90141235,  01-02 : 31.89711841,  01-03 : 26.96629187,
  01-04 : 23.40251489,  01-05 : 33.73439963,  01-06 : 48.02000981,
  01-07 : 49.04696039,  01-08 : 37.26693007,  01-09 : 41.43336694,
  01-10 : 24.82954314,  01-11 : 36.55593066,  01-12 : 58.10699762,
  01-13 : 65.57196981,  01-14 : 58.57130575,  01-15 : 35.72346055,
  01-16 : 39.68732832,  01-17 : 60.82132259,  01-18 : 64.86992271,
  01-19 : 68.72671146,  01-20 : 58.78141816,  01-21 : 40.21333644,
  01-22 : 55.16152950,  01-23 : 64.79961509,  01-24 : 80.05554631,
  01-25 : 70.93319924,  01-26 : 51.14691246,  01-27 : 47.93612512,
  01-28 : 71.77896968,  01-29 : 73.84184908,  01-30 : 70.68011104,
  01-31 : 76.98754704 } ;
```

Table 9.2: Sample declarations and input data for the time series calculation



Here the history is from 01-01 till 01-31 and the horizon is from 02-01 till 02-14.

forecasting::WeightedMovingAverage

The weighted moving average procedure is a time series forecasting procedure. Essentially, this procedure forecasts by taking the weighted average over the last N observations.

Mathematical Formulation:

Using the notation for observations and estimates given in Table 9.1, the estimates are defined as:

$$e_t = \sum_{j=1,\tau=t-(N+1)+j}^{N} w_j \tilde{y}_\tau \quad \text{where } \tilde{y}_\tau = \begin{cases} y_1 & \text{if } \tau < 1 \\ y_\tau & \text{if } \tau \in \{1..T\} \\ e_\tau & \text{if } \tau > T \end{cases} \tag{9.2}$$

Function Prototype:

To provide the error measures and residuals only when you need them, there are three flavors of the `WeightedMovingAverage` procedure provided:

```
forecasting::WeightedMovingAverage(
      ! Provides the estimates,
      ! but not the error measures nor the residuals
      dataValues,              ! Input, parameter indexed over time set
      estimates,               ! Output, parameter indexed over time set
      noObservations,          ! Scalar input, length history
      weights,                 ! Input, parameter
      noAveragingPeriods)      ! Scalar input, averaging length
```

```
forecasting::WeightedMovingAverageEM(
      ! Provides estimates and error measures, but not the residuals
      dataValues,              ! Input, parameter indexed over time set
      estimates,               ! Output, parameter indexed over time set
      noObservations,          ! Scalar input, length history
      weights,                 ! Input, parameter
      noAveragingPeriods,      ! Scalar input, averaging length
      ErrorMeasures)           ! Output, indexed over forecasting::ems
```

```
forecasting::WeightedMovingAverageEMR(
      ! Provides estimates, error measures, and residuals
      dataValues,              ! Input, parameter indexed over time set
      estimates,               ! Output, parameter indexed over time set
      noObservations,          ! Scalar input, length history
      weights,                 ! Input, parameter
      noAveragingPeriods,      ! Scalar input, averaging length
      ErrorMeasures,           ! Output, indexed over forecasting::ems
      Residuals)               ! Output, parameter indexed over time set
```

Here, the time set is a set that encompasses both the history and the horizon.

Arguments:

dataValues
A one dimensional parameter containing the observations for the first T elements of the time set.

estimates
A one dimensional parameter containing the estimates for all elements in the time set.

noObservations
Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

weights
Specifies the weights. The weights should be indexed over a subset of `Integers`: $\{1..N\}$, in the range $[0, 1]$ and sum to 1.

noAveragingPeriods
Specifies the number of values used to compute a single average. This parameter corresponds to N in the mathematical notation above.

ErrorMeasures
The error measures as presented in Section 9.2.

Residuals
The residuals as presented in Section 9.2.

Example:

With declarations and data as specified in Table 9.2 the call:

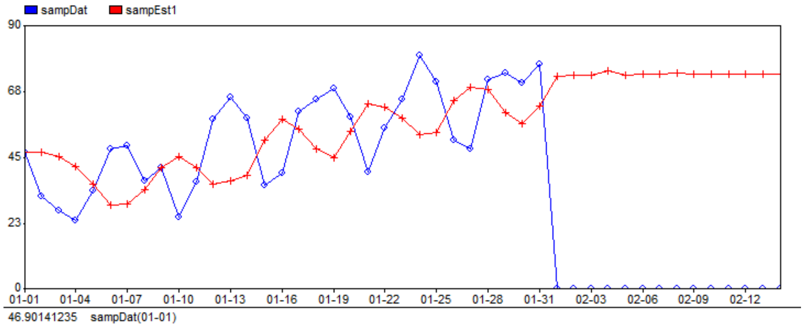
```
weightSet := ElementRange(1,4);
locWeights := data { 1 : 0.1, 2 : 0.2, 3: 0.3, 4: 0.4 } ;
forecasting::WeightedMovingAverage(
    dataValues         :  sampDat,
    estimates          :  sampEst1,
    noObservations     :  31,
    weights            :  locWeights,
    noAveragingPeriods :  4);
```

Will result in the following output:

```
sampEst1 := data
{ 01-01 : 46.901412,  01-02 : 46.901412,  01-03 : 45.400983,
  01-04 : 41.907042,  01-05 : 36.063210,  01-06 : 28.902678,
  01-07 : 29.356152,  01-08 : 33.990024,  01-09 : 41.435848,
  01-10 : 45.518815,  01-11 : 41.568491,  01-12 : 35.958284,
  01-13 : 37.144096,  01-14 : 39.077193,  01-15 : 51.025996,
  01-16 : 58.200997,  01-17 : 54.913605,  01-18 : 48.165158,
  01-19 : 44.846840,  01-20 : 53.967984,  01-21 : 63.412990,
  01-22 : 62.343600,  01-23 : 58.683930,  01-24 : 53.088836,
  01-25 : 53.599271,  01-26 : 64.608926,  01-27 : 69.237841,
  01-28 : 68.325173,  01-29 : 60.482475,  01-30 : 56.579581,
  01-31 : 62.544522,  02-01 : 72.698920,  02-02 : 73.408174,
  02-03 : 73.248910,  02-04 : 74.611221,  02-05 : 73.212924,
```

```
02-06 : 73.581479,  02-07 : 73.683663,  02-08 : 73.893028,
02-09 : 73.485649,  02-10 : 73.664861,  02-11 : 73.704989,
02-12 : 73.706377,  02-13 : 73.605353,  02-14 : 73.679252 } ;
```

This can be graphically displayed as:



Here the history is from 01-01 till 01-31 and the horizon is from 02-01 till 02-14.

forecasting::ExponentialSmoothing

The exponential smoothing procedure is a time series forecasting procedure. This procedure forecasts by weighted average of an observation and a previous forecast.

Mathematical Formulation:

Using the notation in Table 9.1, the estimates are defined as:

$$e_t = \alpha y_{t-1} + (1-\alpha)e_{t-1} \tag{9.3}$$

To initialize this sequence, we take

$$\begin{aligned} e_0 &= y_1 \\ y_0 &= y_1 \end{aligned} \tag{9.4}$$

To calculate the forecasts for $t \geq T+2$, we take y_t for all $t \in \{T+1 \ldots T+H\}$ to be equal to e_t. This results in $y_t = y_{t-1}$ for all $t \in \{T+2 \ldots T+H\}$; which is graphically depicted as a horizontal line.

The weighting factor α is a parameter in the range $(0,1)$; high values of α give more weight to recent observations.

Function Prototype:

To provide the error measures and residuals only when you need them, there are three flavors of the `ExponentialSmoothing` procedure provided:

```
forecasting::ExponentialSmoothing(
! Provides the estimates, but not the error measures nor the residuals
      dataValues,      ! Input, parameter indexed over time set
      estimates,       ! Output, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha)           ! Scalar input, weight of observation

forecasting::ExponentialSmoothingEM(
! Provides estimates and error measures, but not the residuals
      dataValues,      ! Input, parameter indexed over time set
      estimates,       ! Output, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar input, weight of observation
      ErrorMeasures)   ! Output, indexed over forecasting::ems

forecasting::ExponentialSmoothingEMR(
! Provides estimates, error measures, and residuals
      dataValues,      ! Input, parameter indexed over time set
      estimates,       ! Output, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar input, weight of observation
      ErrorMeasures,   ! Output, indexed over forecasting::ems
      Residuals)       ! Output, parameter indexed over time set
```

Arguments:

dataValues
A one dimensional parameter containing the observations for the first T elements of the time set.

estimates
A one dimensional parameter containing the estimates for all elements in the time set.

noObservations
Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

alpha
Specifies the weighting factor for the observation. This parameter corresponds to α in the mathematical notation above.

ErrorMeasures
The error measures as presented in Section 9.2.

Residuals
The residuals as presented in Section 9.2.

Remarks:

In order to use this function, the AIMMSForecasting system library needs to be added to the application.

Example:

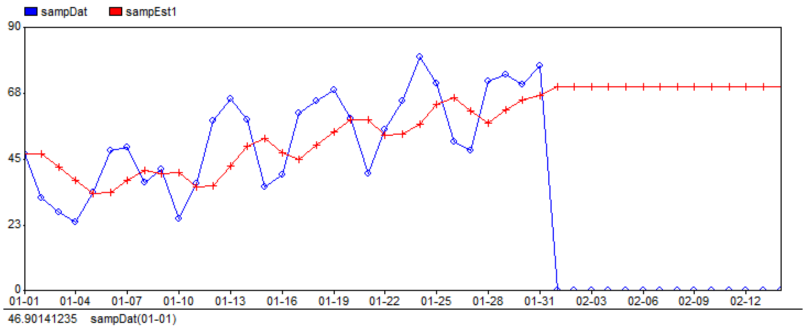
With declarations and data as specified in Table 9.2 the call:

```
forecasting::ExponentialSmoothing(
    dataValues         :  sampDat,
    estimates          :  sampEst1,
    noObservations     :  31,
    alpha              :  0.3);
```

Will result in the following output:

```
sampEst1 := data
{ 01-01 : 46.90141235,  01-02 : 46.90141235,  01-03 : 42.40012417,
  01-04 : 37.76997448,  01-05 : 33.45973660,  01-06 : 33.54213551,
  01-07 : 37.88549780,  01-08 : 41.23393658,  01-09 : 40.04383462,
  01-10 : 40.46069432,  01-11 : 35.77134897,  01-12 : 36.00672348,
  01-13 : 42.63680572,  01-14 : 49.51735495,  01-15 : 52.23354019,
  01-16 : 47.28051629,  01-17 : 45.00255990,  01-18 : 49.74818871,
  01-19 : 54.28470891,  01-20 : 58.61730967,  01-21 : 58.66654222,
  01-22 : 53.13058049,  01-23 : 53.73986519,  01-24 : 57.05779016,
  01-25 : 63.95711700,  01-26 : 66.04994167,  01-27 : 61.57903291,
  01-28 : 57.48616057,  01-29 : 61.77400331,  01-30 : 65.39435704,
  01-31 : 66.98008324,  02-01 : 69.98232238,  02-02 : 69.98232238,
  02-03 : 69.98232238,  02-04 : 69.98232238,  02-05 : 69.98232238,
  02-06 : 69.98232238,  02-07 : 69.98232238,  02-08 : 69.98232238,
  02-09 : 69.98232238,  02-10 : 69.98232238,  02-11 : 69.98232238,
  02-12 : 69.98232238,  02-13 : 69.98232238,  02-14 : 69.98232238 } ;
```

This can be graphically displayed as:



forecasting::ExponentialSmoothingTrend

The exponential smoothing with trend procedure is a time series forecasting procedure. This procedure is an extension from the exponential smoothing whereby the forecast also captures a trend. The reader interested in the mathematical background is referred to

- https://www.otexts.org/book/fpp
- http://en.wikipedia.org/wiki/Exponential_smoothing

Function Prototype:

To provide the error measures and residuals only when you need them, there are three flavors of the `ExponentialSmoothingTrend` procedure provided:

```
forecasting::ExponentialSmoothingTrend(
! Provides the estimates, but not the error measures nor the residuals
      dataValues,       ! Input, parameter indexed over time set
      estimates,        ! Output, parameter indexed over time set
      noObservations,   ! Scalar input, length history
      alpha,            ! Scalar input, weight of observation
      beta)             ! Scalar input, weight of change in observation

forecasting::ExponentialSmoothingTrendEM(
! Provides estimates and error measures, but not the residuals
      dataValues,       ! Input, parameter indexed over time set
      estimates,        ! Output, parameter indexed over time set
      noObservations,   ! Scalar input, length history
      alpha,            ! Scalar input, weight of observation
      beta,             ! Scalar input, weight of change in observation
      ErrorMeasures)    ! Output, indexed over forecasting::ems

forecasting::ExponentialSmoothingTrendEMR(
! Provides estimates, error measures, and residuals
      dataValues,       ! Input, parameter indexed over time set
      estimates,        ! Output, parameter indexed over time set
      noObservations,   ! Scalar input, length history
      alpha,            ! Scalar input, weight of observation
      beta,             ! Scalar input, weight of change in observation
      ErrorMeasures,    ! Output, indexed over forecasting::ems
      Residuals)        ! Output, parameter indexed over time set
```

Arguments:

dataValues
: A one dimensional parameter containing the observations for the first T elements of the time set.

estimates
: A one dimensional parameter containing the estimates for all elements in the time set.

noObservations
Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

alpha
Specifies the weighting factor for the observation. This parameter corresponds to α in the mathematical notation above.

beta
Specifies the weighting factor for the change in observation.

ErrorMeasures
The error measures as presented in Section 9.2.

Residuals
The residuals as presented in Section 9.2.

Example:

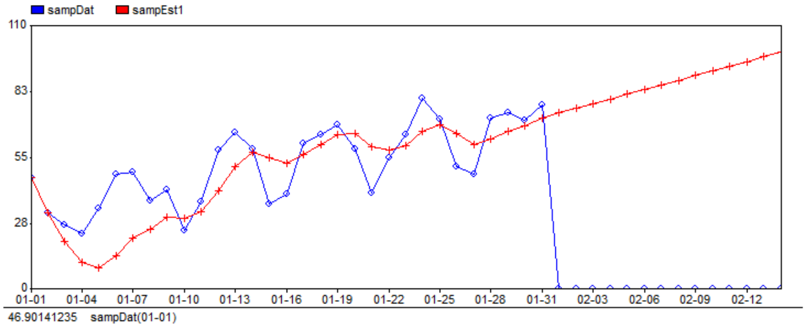
With declarations and data as specified in Table 9.2 the call:

```
forecasting::ExponentialSmoothingTrend(
    dataValues          :  sampDat,
    estimates           :  sampEst1,
    noObservations      :  31,
    alpha               :  0.3,
    beta                :  0.3);
```

Will result in the following output:

```
sampEst1 := data
{ 01-01 : 46.90141235,  01-02 : 31.89711841,  01-03 : 19.91486469,
  01-04 : 11.09278244,  01-05 :  9.12476621,  01-06 : 14.24770491,
  01-07 : 21.18135461,  01-08 : 25.00880483,  01-09 : 30.04118231,
  01-10 : 29.60799603,  01-11 : 32.39262113,  01-12 : 41.18187664,
  01-13 : 51.09710805,  01-14 : 57.24030837,  01-15 : 54.80598480,
  01-16 : 52.57369145,  01-17 : 56.19151171,  01-18 : 60.35524890,
  01-19 : 64.83322220,  01-20 : 65.33462956,  01-21 : 59.52540116,
  01-22 : 58.20531338,  01-23 : 59.89873706,  01-24 : 66.10199203,
  01-25 : 68.96338627,  01-26 : 65.20775937,  01-27 : 60.35010811,
  01-28 : 62.98534714,  01-29 : 66.24030430,  01-30 : 68.25439193,
  01-31 : 71.77479879,  02-01 : 73.73138118,  02-02 : 75.68796357,
  02-03 : 77.64454596,  02-04 : 79.60112835,  02-05 : 81.55771074,
  02-06 : 83.51429313,  02-07 : 85.47087552,  02-08 : 87.42745791,
  02-09 : 89.38404030,  02-10 : 91.34062269,  02-11 : 93.29720508,
  02-12 : 95.25378747,  02-13 : 97.21036985,  02-14 : 99.16695224 } ;
```

This can be graphically displayed as:


sampDat
sampEst1
110
83
55
28
0
01-01
01-04
01-07
01-10
01-13
01-16
01-19
01-22
01-25
01-28
01-31
02-03
02-06
02-09
02-12
46.90141235 sampDat(01-01)

forecasting::ExponentialSmoothingTrendSeasonality

The exponential smoothing with trend and seasonality procedure is a time series forecasting procedure. This procedure is an extension from the exponential smoothing whereby the forecast also captures both a trend and a seasonality. The reader interested in the mathematical background is referred to

- https://www.otexts.org/book/fpp
- http://en.wikipedia.org/wiki/Exponential_smoothing

Function Prototype:

To provide the error measures and residuals only when you need them, there are three flavors of the `ExponentialSmoothingTrendSeasonality` procedure provided:

```
forecasting::ExponentialSmoothingTrendSeasonality(
! Provides the estimates, but not the error measures nor the residuals
      dataValues,      ! Input, parameter indexed over time set
      estimates,       ! Output, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar input, weight of observation
      beta,            ! Scalar input, weight of change in observation
      gamma,           ! Scalar input, weight of seasonality
      periodLength)    ! Scalar input, length of season


forecasting::ExponentialSmoothingTrendSeasonalityEM(
! Provides estimates and error measures, but not the residuals
      dataValues,      ! Input, parameter indexed over time set
      estimates,       ! Output, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar input, weight of observation
      beta,            ! Scalar input, weight of change in observation
      gamma,           ! Scalar input, weight of seasonality
      periodLength,    ! Scalar input, length of season
      ErrorMeasures)   ! Output, indexed over forecasting::ems


forecasting::ExponentialSmoothingTrendSeasonalityEMR(
! Provides estimates, error measures, and residuals
      dataValues,      ! Input, parameter indexed over time set
      estimates,       ! Output, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar input, weight of observation
      beta,            ! Scalar input, weight of change in observation
      gamma,           ! Scalar input, weight of seasonality
      periodLength,    ! Scalar input, length of season
      ErrorMeasures,   ! Output, indexed over forecasting::ems
      Residuals)       ! Output, parameter indexed over time set
```

Arguments:

dataValues
: A one dimensional parameter containing the observations for the first T elements of the time set.

estimates
: A one dimensional parameter containing the estimates for all elements in the time set.

noObservations
: Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

noAveragingPeriods
: Specifies the number of values used to compute a single average. This parameter corresponds to N in the mathematical notation above.

alpha
: Specifies the weighting factor for the observation. This parameter corresponds to α in the mathematical notation above.

beta
: Specifies the weighting factor for the change in observation.

gamma
: Specifies the weighting factor for the seasonality.

periodLength
: Specifies the period length.

ErrorMeasures
: The error measures as presented in Section 9.2.

Residuals
: The residuals as presented in Section 9.2.

Example:

With declarations and data as specified in Table 9.2 the call:

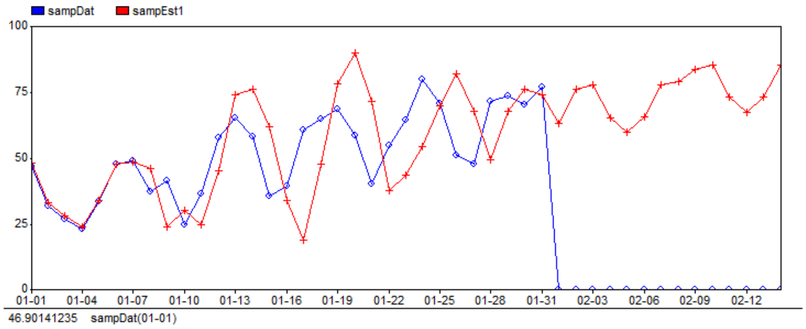
```
forecasting::ExponentialSmoothingTrendSeasonality(
    dataValues        :  sampDat,
    estimates         :  sampEst1,
    noObservations    :  31,
    alpha             :  0.5,
    beta              :  0.3,
    gamma             :  0.3,
    periodLength      :  7);
```

Will result in the following output:

```
sampEst1 := data
{ 01-01 : 48.17421514,  01-02 : 33.42448176,  01-03 : 28.16272649,
  01-04 : 24.07455476,  01-05 : 33.94263017,  01-06 : 47.93386652,
  01-07 : 48.83947317,  01-08 : 46.31365850,  01-09 : 23.89344424,
```

```
01-10 : 30.27764654,  01-11 : 24.95849413,  01-12 : 45.51882876,
01-13 : 74.25387499,  01-14 : 76.43874408,  01-15 : 62.30360776,
01-16 : 34.03705964,  01-17 : 18.95751109,  01-18 : 47.97903657,
01-19 : 78.64240904,  01-20 : 90.15243324,  01-21 : 71.83828787,
01-22 : 37.68452884,  01-23 : 43.80677029,  01-24 : 54.55643634,
01-25 : 70.28818669,  01-26 : 82.29733841,  01-27 : 67.89367583,
01-28 : 49.77439370,  01-29 : 67.81915419,  01-30 : 76.48587445,
01-31 : 74.36541195,  02-01 : 63.51664916,  02-02 : 76.26956592,
02-03 : 77.83862565,  02-04 : 65.67879532,  02-05 : 59.94750898,
02-06 : 65.94274949,  02-07 : 77.84397349,  02-08 : 79.13679316,
02-09 : 83.83707749,  02-10 : 85.40613721,  02-11 : 73.24630688,
02-12 : 67.51502054,  02-13 : 73.51026105,  02-14 : 85.41148505 } ;
```

This can be graphically displayed as:



forecasting::ExponentialSmoothingTune

The `forecasting::ExponentialSmoothingTune` procedure is a time series forecasting helper procedure of `forecasting::ExponentialSmoothing` by computing the α for which the mean squared error is minimized.

Function Prototype:

```
forecasting::ExponentialSmoothingTune(
! Provides the alpha for which the mean squared error is minimized.
      dataValues,      ! Input, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar output, weight of observation
                       ! that minimizes mean squared error
      alphaLow,        ! Optional input, default 0.01
      alphaUpp)        ! Optional input, default 0.99
```

Arguments:

dataValues
: A one dimensional parameter containing the observations for the first T elements of the time set.

noObservations
: Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

alpha
: Upon return it provides the weighting factor α for which the mean squared error is minimized when using `forecasting::ExponentialSmoothing` on the same `dataValues`.

alphaLow
: Lowerbound on α, default 0.01.

alphaUpp
: Upperbound on α, default 0.99.

Remarks:

- In order to use this function, the AIMMSForecasting system library needs to be added to the application.
- Please note that this function performs an optimization step; a nonlinear programming solver should be available and, in an AIMMS PRO environment, it should be run server side.

forecasting::ExponentialSmoothingTrendTune

The `forecasting::ExponentialSmoothingTrendTune` procedure is a time series forecasting helper procedure of `forecasting::ExponentialSmoothingTrend` by computing the α and β for which the mean squared error is minimized.

Function Prototype:

```
forecasting::ExponentialSmoothingTrendTune(
! Provides the alpha for which the mean squared error is minimized.
      dataValues,      ! Input, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar output,
      beta,            ! Scalar output,
      alphaLow,        ! Optional input, default 0.01
      alphaUpp,        ! Optional input, default 0.99
      betaLow,         ! Optional input, default 0.01
      betaUpp)         ! Optional input, default 0.99
```

Arguments:

dataValues
: A one dimensional parameter containing the observations for the first T elements of the time set.

noObservations
: Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

alpha, beta
: α and β are scalar output parameters of this procedure. The values for α and β are such that the mean squared error of the estimates returned by `forecasting::ExponentialSmoothingTrend` are minimized.

alphaLow
: Lowerbound on α, default 0.01.

alphaUpp
: Upperbound on α, default 0.99.

betaLow
: Lowerbound on β, default 0.01.

betaUpp
: Upperbound on β, default 0.99.

Remarks:

- In order to use this function, the AIMMSForecasting system library needs to be added to the application.

- Please note that this function performs an optimization step; a nonlinear programming solver should be available and, in an AIMMS PRO environment, it should be run server side.

forecasting::ExponentialSmoothingTrendSeasonalityTune

The `forecasting::ExponentialSmoothingTrendSeasonalityTune` procedure is a time series forecasting helper procedure of `forecasting::ExponentialSmoothingTrendSeasonality` by computing the α, β, and γ for which the mean squared error is minimized.

Function Prototype:

```
forecasting::ExponentialSmoothingTrendSeasonalityTune(
! Provides the alpha for which the mean squared error is minimized.
      dataValues,      ! Input, parameter indexed over time set
      noObservations,  ! Scalar input, length history
      alpha,           ! Scalar output,
      beta,            ! Scalar output,
      gamma,           ! Scalar output,
      periodLength,    ! Scalar input, length of season
      alphaLow,        ! Optional input, default 0.01
      alphaUpp,        ! Optional input, default 0.99
      betaLow,         ! Optional input, default 0.01
      betaUpp,         ! Optional input, default 0.99
      gammaLow,        ! Optional input, default 0.01
      gammaUpp)        ! Optional input, default 0.99
```

Arguments:

dataValues
A one dimensional parameter containing the observations for the first T elements of the time set.

noObservations
Specifies the number of elements that belong to the history of the time set. This parameter corresponds to T in the notation presented in Table 9.1.

alpha, beta, gamma
α, β, and γ are scalar output parameters of this procedure. The values for α, β, and γ are such that the mean squared error of the estimates returned by `forecasting::ExponentialSmoothingTrendSeasonality` are minimized.

periodLength
Specifies the period length.

alphaLow
Lowerbound on α, default 0.01.

alphaUpp
Upperbound on α, default 0.99.

betaLow
Lowerbound on β, default 0.01.

betaUpp
Upperbound on β, default 0.99.

gammaLow
Lowerbound on γ, default 0.01.

gammaUpp
Upperbound on γ, default 0.99.

Remarks:

- In order to use this function, the AIMMSForecasting system library needs to be added to the application.
- Please note that this function performs an optimization step; a nonlinear programming solver should be available and, in an AIMMS PRO environment, it should be run server side.

9.3 Simple Linear Regression

9.3.1 Notational conventions for simple linear regression

For simple linear regression we follow the conventions below.

Observations and Estimates

The `AimmsForecasting` library uses as input data observations for the independent variable and the dependent variable. It provides estimates for the coefficients of the simple linear regression line.

N	number of observations
$x_i, i \in \{1 \ldots N\}$	observations of the independent variable
$y_i, i \in \{1 \ldots N\}$	observations of the dependent variable
$\bar{x} = (1/N) \sum_{i=1}^{N} x_i$	average of the independent observations
$\bar{y} = (1/N) \sum_{i=1}^{N} y_i$	average of the dependent observations
$\hat{y}_i, i \in \{1 \ldots N\}$	predictions of the dependent variable
β_0, β_1	coefficients of the linear relationship (random)
$\hat{\beta}_0, \hat{\beta}_1$	coefficients of the linear regression line (estimates)
$e_i, i \in \{1 \ldots N\}$	error (residual) for observation data points

Table 9.3: Simple Linear Regression notation

Linear Relationship

The linear relationship between x_i and y_i is modeled by the equation:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i \tag{9.5}$$

where ϵ_i is an error term which averages out to 0 for every i.

Linear Regression

The random β_0 and β_1 are estimated by $\hat{\beta}_0$ and $\hat{\beta}_1$, such that the prediction for y_i is given by the equation:

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i \tag{9.6}$$

So, the predictions based on simple linear regression corresponding to the observation data points (x_i, y_i) are provided in $\hat{y}_i, i \in \{1 \ldots N\}$.

Residuals

The error (residual) e_i for the data point i is the difference between the observed y_i and the predicted $\hat{y}_i$, so $e_i = y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i$. In order to obtain the residuals, the user will need to provide a one-dimensional parameter declared over the set of observations.

Variation Components

Given the values of the observations, the estimates, and the residuals, several components of variation can be computed, such as **sum of squares total** = SST, **sum of squares error** = SSE, and **sum of squares regression** = SSR, which are defined as follows:

$$SST = \sum_{i=1}^{N} (y_i - \bar{y})^2 \tag{9.7}$$

$$SSE = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} e_i^2 \tag{9.8}$$

$$SSR = \sum_{i=1}^{N} (\hat{y}_i - \bar{y})^2 \tag{9.9}$$

These components of variation satisfy the relation $SST = SSE + SSR$.

Furthermore, it is also possible to compute the **coefficient of determination** = R^2, the **sample linear correlation** = r_{xy}, and the **standard error of the estimate** = s_e, which are defined as follows:

$$R^2 = \frac{SSR}{SST} \tag{9.10}$$

$$r_{xy} = \begin{cases} +\sqrt{R^2} & \text{if } \hat{\beta}_1 \geq 0 \\ -\sqrt{R^2} & \text{if } \hat{\beta}_1 \leq 0 \end{cases} \tag{9.11}$$

$$s_e = \sqrt{\frac{SSE}{N-2}} \tag{9.12}$$

Predeclared index **vcs**

The linear regression functions return the values of the line coefficients in a parameter declared over the index `forecasting::co` declared as follows:

```
Set LRcoeffSet{
    Index: co;
    Definition: {
      data {
        0,      ! Intercept Coefficient of Regression Line
        1       ! Slope Coefficient of Regression Line
      }
    }
  }
```

Whenever one of the linear regression functions communicates back components of variations, it uses identifiers declared over the index `forecasting::vcs` declared as follows:

```
Set VariationCompSet {
    Index: vcs;
    Definition: {
       data {
           SST,       ! Sum of Squares Total
```

```
          SSE,       ! Sum of Squares Error
          SSR,       ! Sum of Squares Regression
          Rsquare,   ! Coefficient of Determination
          MultipleR, ! Sample Linear Correlation Rxy
          Se         ! Standard Error
         }
     }
   }
```

In order to obtain the variation components, the user will need to provide a parameter indexed over `forecasting::vcs` to the linear regression functions.

forecasting::SimpleLinearRegression

The simple linear regression procedure computes the regression line coefficients based on the values of the observations for the independent and the dependent variables. If desired, the values for variation components and the residuals can be returned as well.

Mathematical Formulation:

Using the notation for observations and estimates given in Table 9.3, the estimates of the coefficients of the linear regression line are determined as follows:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(x_i - \hat{x})^2} \tag{9.13}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x} \tag{9.14}$$

These values provide the minimum in $\hat{\beta}_0$, $\hat{\beta}_1$ of the function

$$F(\hat{\beta}_0, \hat{\beta}_1) = \sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i)^2) \tag{9.15}$$

Therefore, the values $\hat{\beta}_0$ and $\hat{\beta}_1$ given above are called the **least squares estimates** of β_0 and β_1. With these coefficients, the regression line 9.6 is called the **least squares regression line**. Every least squares regression line has the following two properties:

- It passes through the point $(\bar{x}, \bar{y})$
- $\sum_{i=1}^{N} e_i = 0$

Function Prototype:

In order to provide the variation components and residuals only when needed, there are three flavors of the `SimpleLinearRegression` procedure provided:

```
forecasting::SimpleLinearRegression( ! Provides the estimates of the line
                                     ! coefficients, but not the variation
                                     ! components nor the residuals
        xIndepVarValue,            ! Input, parameter for independent
        yDepVarValue,              ! Input, parameter for dependent
        LRcoeff)                   ! Output,parameter for line coefficients

forecasting::SimpleLinearRegressionVC( ! Provides the estimates of the line
                                       ! coefficients and the variation
                                       ! components
xIndepVarValue,              ! Input, parameter for independent
yDepVarValue,                ! Input, parameter for dependent
LRcoeff,                     ! Output,parameter for line coefficients
VariationComp)               ! Output,parameter variation components
```

```
forecasting::SimpleLinearRegressionVCR( ! Provides the estimates of the line
                                        ! coefficients, the variation
                                        ! components and the residuals
      xIndepVarValue,              ! Input, parameter for independent
      yDepVarValue,                ! Input, parameter for dependent
      LRcoeff,                     ! Output,parameter for line coefficients
      VariationComp,               ! Output,parameter variation components
      yEstimates,                  ! Output,parameter for estimates
      eResiduals)                  ! Output,parameter for residuals
```

Arguments:

xIndepVarValue
A one dimensional parameter containing the observations for the independent variable

yDepVarValue
A one dimensional parameter containing the observations for the dependent variable

LRcoeff
A one dimensional parameter for storing the coefficients of the regression line

VariationComp
A one dimensional parameter for storing the values of the variation components

yEstimates
A one dimensional parameter for storing the values of the estimates

eResiduals
A one dimensional parameter for storing the values of the residuals

Example:

Suppose that we are looking at cost data for producing one type of machine. The number of units produced is an independent variable and the total production costs is a dependent variable. For this situation, consider the following observations data:

```
Set sObservationsSet {
SubsetOf: Integers;
Index: i_ob;
Definition: data{1..10};}

Parameter MachinesProd {
IndexDomain: i_ob;
Definition: {
data{
1 : 10,
2 : 20,
3 : 30,
4 : 40,
5 : 45,
```

```
6 : 50,
7 : 60,
8 : 55,
9 : 70,
10 : 40
}}}

Parameter CostOfMachinesProd {
IndexDomain: i_ob;
Definition: {
data{
1 :  257.40,
2 :  601.60,
3 :  782.00,
4 :  765.40,
5 :  895.50,
6 : 1133.00,
7 : 1152.80,
8 : 1132.70,
9 : 1459.20,
10 :  970.10}}}
```

With the declarations and the data as specified, the following function call:

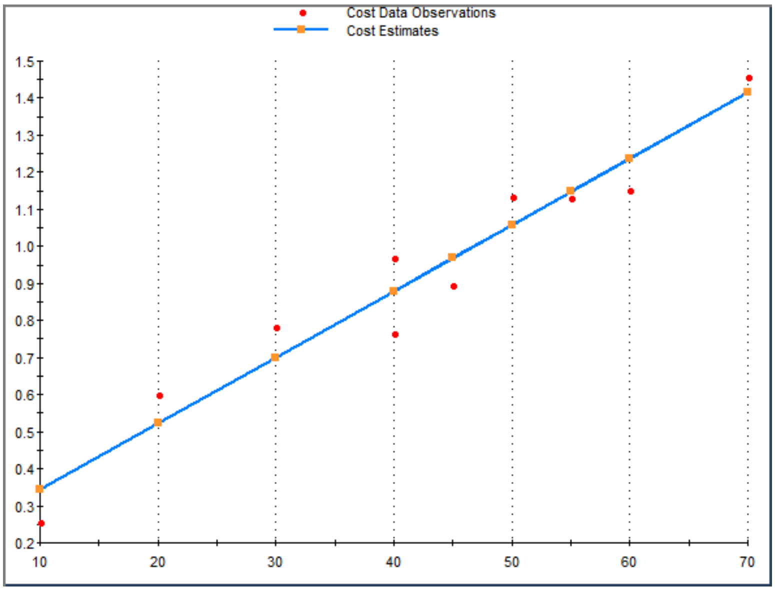
```
forecasting::SimpleLinearRegressionVCR(
                 xIndepVarValue       :  MachinesProd,
                 yDepVarValue         :  CostOfMachinesProd,
                 LRcoeff              :  Coeff,
                 VariationComp        :  VariationMeasure,
                 yEstimates           :  CostEstimate,
                 eResiduals           :  CostError);
```

will result in the following output data:

```
Coeff := data
{
0  :  164.87790700,     ! Intercept Coefficient of Regression Line
1  :   17.85933555      ! Slope Coefficient of Regression Line
}

VariationMeasure := data
{
SST        :  1021762.50100,          ! Sum of Squares Total
SSE        :    61705.34367,          ! Sum of Squares Error
SSR,       :   960057.15730,          ! Sum of Squares Regression
Rsquare,   :        0.9396089173,     ! Coefficient of Determination
MultipleR, :        0.9693342650,     ! Sample Linear Correlation
Se         :       87.8246432300,     ! Standard Error
}


CostEstimate  := data
{
   1  :   343.4712625,
   2  :   522.0646179,
   3  :   700.6579734,
   4  :   879.2513289,
   5  :   968.5480066,
   6  :  1057.8446840,
```

```
   7  :  1236.4380400,
   8  :  1147.1413620,
   9  :  1415.0313950,
  10  :   879.2513289
}

CostError  := data
{
  1  :   -86.07126246,
  2  :    79.53538206,
  3  :    81.34202658,
  4  :  -113.85132890,
  5  :   -73.04800664,
  6  :    75.15531561,
  7  :   -83.63803987,
  8  :   -14.44136213,
  9  :    44.16860465,
 10  :   90.84867110
}
```

The cost data observations, the cost estimates and the resulting simple linear regression line can be graphically displayed as shown in the following figure (where the cost figures on the y-axis are scaled by a factor 1000):



Part II

Algorithmic Capabilities

Chapter 10

Constraint Programming Functions

AIMMS supports the following functions for constraint programming:

- cp::AllDifferent
- cp::BinPacking
- cp::Cardinality
- cp::Channel
- cp::Count
- cp::Lexicographic
- cp::ParallelSchedule
- cp::Sequence
- cp::SequentialSchedule

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.

Remarks:

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);
}
```

Examples:

```
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 : C }
    }
}
```

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 {
    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:

- Chapter 22 on Constraint Programming in the Language Reference.
- Further information on index binding can be found in the Chapter on Index Binding 9 in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Calldifferent.html` which references this function as `alldifferent`.

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
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.

Examples:

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` 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 }
   }
}
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 that illustrate how the index binding and indexed arguments can be used. Further information on index binding can be found in the Chapter on Index Binding 9 in the Language Reference.
- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog www.emn.fr/z-info/sdemasse/gccat/Cbin_packing.html which references this function as bin_packing.

cp::Cardinality

The function `cp::Cardinality` 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` 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.

Examples:

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` and `cp::Sequence`.
- The Chapter on Constraint Programming 22 in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Cglobal_cardinality.html` which references this function as `global_cardinality`.

cp::Channel

The function `cp::Channel` links two arrays of variables such that they are uniquely matched to each other. For instance, see Figure 10.1. This function is often used to model different perspectives of the same problem.

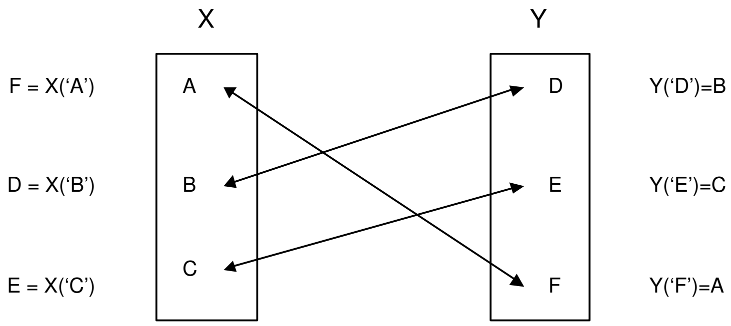


Figure 10.1: A situation accepted by `cp::Channel`

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`.

Remarks:

- The `cp::Channel` constraint is also referred to in the Constraint Programming literature as `Inverse`.
- The `cp::Channel` 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))`.

Examples:

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:

- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Cinverse.html` which references this function as `inverse`.

cp::Count

The function `cp::Count` 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`. 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.

Examples:

```
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;
    Range        :  integer;
}
Constraint CountUsedWarehouses {
    IndexDomain  :  w;
    Definition   : {
        cp::count( s, supplyingWarehouse(s), w,
                   '=', warehouseUsage(w) )
    }
}
```

See also:

- The functions `cp::Cardinality` and `cp::Sequence`.
- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Ccount.html` which references this function as `count`, or, depending on a particular choice of ⊗, as `atleast`, `atmost` or `exactly`.

cp::Lexicographic

The function `cp::Lexicographic` 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.

Remarks:

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.

Examples:

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;
}
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`.
- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Clex_less.html` which references this function as `lex_less` and `lex_lesseq`.

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

$$\begin{array}{l} \forall t : \sum_{j|s_j \le t < e_j} w_j \le c \\ \forall j : s_j + d_j = e_j. \end{array}$$

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.

Remarks:

- 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` illustrate how the index binding and vector arguments are used.
- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Ccumulative.html` which references this function as `cumulative`.

cp::Sequence

The function `cp::Sequence` 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

$$\begin{array}{llll} \forall i = 1..n-q+1: & l \leq \sum_{j=0}^{q-1}(x_{i+j} \in S) & \leq u & c = 0 \\ \forall i = 1..n: & l \leq \sum_{j=0}^{q-1}(x_{(i+j-1)\%n+1} \in S) & \leq u & c \neq 0 \end{array}$$

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..card(range(inspectedValues))}.

lowerBound
: The lower bound on the number of occurences. This expression does not involve variables. This argument should be in the range {0..upperBound}.

upperBound
: The upper bound on the number of occurences. This expression does not involve variables. This argument should be in the range {lowerBound..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.

Examples:

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);
    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 and cp::Cardinality.
- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog www.emn.fr/z-info/sdemasse/gccat/Camong_seq.html which references this function as among_seq.

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.

Remarks:

- 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.

Examples:

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);
    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`.

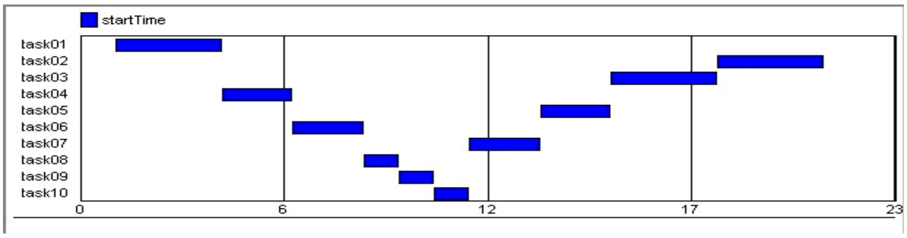


Figure 10.2: Gantt chart for solution of `cp::SequentialSchedule`

See also:

- The examples at the function `cp::AllDifferent` illustrate how the index binding and vector arguments are used.
- Chapter 22 on Constraint Programming in the Language Reference.
- The global constraint catalog `www.emn.fr/z-info/sdemasse/gccat/Cdisjunctive.html` which references this function as `disjunctive`.

Chapter 11

Scheduling Functions

AIMMS supports the following functions for scheduling:

- cp::ActivityBegin
- cp::ActivityEnd
- cp::ActivityLength
- cp::ActivitySize
- cp::Alternative
- cp::BeginAtBegin
- cp::BeginAtEnd
- cp::BeginBeforeBegin
- cp::BeginBeforeEnd
- cp::BeginOfNext
- cp::BeginOfPrevious
- cp::EndAtBegin
- cp::EndAtEnd
- cp::EndBeforeBegin
- cp::EndBeforeEnd
- cp::EndOfNext
- cp::EndOfPrevious
- cp::GroupOfNext
- cp::GroupOfPrevious
- cp::LengthOfNext
- cp::LengthOfPrevious
- cp::SizeOfNext
- cp::SizeOfPrevious
- cp::Span
- cp::Synchronize

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.

Examples:

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` and `cp::ActivityEnd`.

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

$$\begin{cases} a.\mathtt{end} & \text{if } a.\mathtt{present} \\ d & \text{otherwise} \end{cases}$$

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.

Examples:

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` and `cp::ActivityBegin`.

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.

Examples:

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` and `cp::ActivityBegin`.

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

$$\begin{cases} a.\mathtt{size} & \text{if } a.\mathtt{present} \\ d & \text{otherwise} \end{cases}$$

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.

Examples:

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` and `cp::ActivityBegin`.

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.

Examples:

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` and `cp::Synchronize`.

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` and `cp::BeginBeforeEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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` and `cp::BeginBeforeEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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

$$\begin{array}{ll} a.\texttt{Present} = 0 & \vee \\ b.\texttt{Present} = 0 & \vee \\ a.\texttt{Begin} + d \le b.\texttt{Begin} & \end{array}$$

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` and `cp::BeginBeforeEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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` and `cp::BeginAtEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::BeginOfNext

The function `cp::BeginOfNext` 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` and `cp::EndOfNext`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::BeginOfPrevious

The function `cp::BeginOfPrevious` 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
        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` and `cp::EndOfPrevious`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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` and `cp::BeginBeforeEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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` and `cp::BeginBeforeEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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` and `cp::BeginBeforeEnd`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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

$$\begin{array}{ll} a.\texttt{Present} = 0 & \vee \\ b.\texttt{Present} = 0 & \vee \\ a.\texttt{End} + d \le b.\texttt{End} & \end{array}$$

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` and `cp::EndBeforeBegin`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::EndOfNext

The function `cp::EndOfNext` 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` and `cp::EndOfPrevious`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::EndOfPrevious

The function `cp::EndOfPrevious` 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` and `cp::EndOfNext`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::GroupOfNext

The function `cp::GroupOfNext` 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` and `cp::EndOfPrevious`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::GroupOfPrevious

The function `cp::GroupOfPrevious` 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` and `cp::EndOfNext`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::LengthOfNext

The function `cp::LengthOfNext` refers to the length of the next activity in a sequence of activities. A length is an integer in the range $\{0..card(\texttt{problemscheduledomain}) - 1\}$.

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` and `cp::EndOfPrevious`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::LengthOfPrevious

The function `cp::LengthOfPrevious` refers to the length of the previous activity in a sequence of activities. A size is an integer in the range $\{0..card(\texttt{problemscheduledomain}) - 1\}$.

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` and `cp::EndOfNext`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::SizeOfNext

The function `cp::SizeOfNext` refers to the size of the next activity in a sequence of activities. A size is an integer in the range $\{0..card(\texttt{problemscheduledomain}) - 1\}$.

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` and `cp::EndOfPrevious`, and
- Chapter 22 on Constraint Programming in the Language Reference.

cp::SizeOfPrevious

The function `cp::SizeOfPrevious` refers to the size of the previous activity in a sequence of activities. A size is an integer in the range $\{0..card(\texttt{problemscheduledomain}) - 1\}$.

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` and `cp::EndOfNext`, and
- Chapter 22 on Constraint Programming in the Language Reference.

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` and `cp::Synchronize`.

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` and `cp::Span`.

Chapter 12

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:

- Procedures and functions in the GMP::Benders namespace
- Procedures and functions in the GMP::Coefficient namespace
- Procedures and functions in the GMP::Column namespace
- Procedures and functions in the GMP::Event namespace
- Procedures and functions in the GMP::Instance namespace
- Procedures and functions in the GMP::Linearization namespace
- Procedures and functions in the GMP::ProgressWindow namespace
- Procedures and functions in the GMP::QuadraticCoefficient namespace
- Procedures and functions in the GMP::Robust namespace
- Procedures and functions in the GMP::Row namespace
- Procedures and functions in the GMP::Solution namespace
- Procedures and functions in the GMP::Solver namespace
- Procedures and functions in the GMP::SolverSession namespace
- Procedures and functions in the GMP::Stochastic namespace
- Procedures and functions in the GMP::Tuning namespace

12.1 GMP::Benders Procedures and Functions

AIMMS supports the following procedures and functions for implementing an automatic Benders' decomposition algorithm:

- GMP::Benders::AddFeasibilityCut
- GMP::Benders::AddOptimalityCut
- GMP::Benders::CreateMasterProblem
- GMP::Benders::CreateSubProblem
- GMP::Benders::UpdateSubProblem

GMP::Benders::AddFeasibilityCut

The procedure `GMP::Benders::AddFeasibilityCut` 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
     [tighten]         ! (optional, default 0) a scalar binary expression
)
```

Arguments:

GMP1
An element in the set `AllGeneratedMathematicalPrograms` representing a Benders' master problem.

GMP2
An element in the set `AllGeneratedMathematicalPrograms` 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.

Remarks:

- The *GMP1* should have been created using the function `GMP::Benders::CreateMasterProblem`.
- The *GMP2* should have been created using the function `GMP::Benders::CreateSubProblem` or the function `GMP::Instance::CreateFeasibility`.
- If the GMP that was created by `GMP::Benders::CreateSubProblem` 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.

Examples:

In the examples below we assume that the Benders' subproblem is infeasible. The way `GMP::Benders::AddFeasibilityCut` 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 Section 21.3 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 : 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 Section 21.5.1 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 );

    GMP::Instance::Solve( gmpF );

    GMP::Benders::AddFeasibilityCut( gmpM, gmpF, 1, NumberOfFeasibilityCuts );
    NumberOfFeasibilityCuts += 1;
endif;
```

See also:

The routines GMP::Benders::CreateMasterProblem, GMP::Benders::CreateSubProblem, GMP::Benders::AddOptimalityCut, GMP::Instance::CreateFeasibility, GMP::SolverSession::AddBendersFeasibilityCut and GMP::SolverSession::AddBendersOptimalityCut.

GMP::Benders::AddOptimalityCut

The procedure `GMP::Benders::AddOptimalityCut` 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` representing a Benders' master problem.

GMP2
An element in the set `AllGeneratedMathematicalPrograms` 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.

Remarks:

- The *GMP1* should have been created using the function `GMP::Benders::CreateMasterProblem`.
- The *GMP2* should have been created using the function `GMP::Benders::CreateSubProblem`.
- 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.

Examples:

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. 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, GMP::Benders::CreateSubProblem, GMP::Benders::AddFeasibilityCut, GMP::SolverSession::AddBendersFeasibilityCut and GMP::SolverSession::AddBendersOptimalityCut.

GMP::Benders::CreateMasterProblem

The function GMP::Benders::CreateMasterProblem 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.

Variables

Variables
: A subset of AllVariables.

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 with the name as specified by the *name* argument.

Remarks:

- A call to GMP::Benders::CreateMasterProblem is typically followed by a call to the function GMP::Benders::CreateSubProblem.
- 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.
- 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 Section 21.5.2 of the Language Reference.
- The *addConstraints* argument is discussed in more detail in Section 21.5.5 of the Language Reference.

Examples:

If the math program has type MIP then often the set of master problem variables equals the set `AllIntegerVariables`.

```
myGMP := GMP::Instance::Generated( MP );

gmpM := GMP::Benders::CreateMasterProblem( myGMP, AllIntegerVariables,
                                           'BendersMasterProblem', 0, 0 );
```

See also:

The routines `GMP::Benders::CreateSubProblem`, `GMP::Benders::AddFeasibilityCut` and `GMP::Benders::AddOptimalityCut`.

GMP::Benders::CreateSubProblem

The function `GMP::Benders::CreateSubProblem` 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`.

GMP2
: An element in the set `AllGeneratedMathematicalPrograms` 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` with the name as specified by the *name* argument.

Remarks:

- The *GMP1* must have type LP, MIP or RMIP.
- The *GMP2* should have been created using the function `GMP::Benders::CreateMasterProblem`. 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 Section 21.5.1 of the Language Reference.

- The *normalizationType* argument is discussed in more detail in Section 21.5.3 of the Language Reference.

Examples:

If the math program has type MIP then often the set of master problem variables equals the set `AllIntegerVariables`. 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`, `GMP::Benders::AddFeasibilityCut`, `GMP::Benders::AddOptimalityCut`, `GMP::Benders::UpdateSubProblem` and `GMP::Instance::CreateFeasibility`.

GMP::Benders::UpdateSubProblem

The procedure `GMP::Benders::UpdateSubProblem` 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` representing a Benders' subproblem.

GMP2
: An element in the set `AllGeneratedMathematicalPrograms` 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.

Remarks:

- The *GMP1* should have been created using the function `GMP::Benders::CreateSubProblem` or the function `GMP::Instance::CreateFeasibility`.
- The *GMP2* should have been created using the function `GMP::Benders::CreateMasterProblem`.
- 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.

Examples:

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`, `GMP::Benders::CreateSubProblem` and `GMP::Instance::CreateFeasibility`.

12.2 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
- GMP::Coefficient::GetQuadratic
- GMP::Coefficient::Set
- GMP::Coefficient::SetMulti
- GMP::Coefficient::SetQuadratic

GMP::Coefficient::Get

The function `GMP::Coefficient::Get` 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`.

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.

Remarks:

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.

Examples:

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` and `GMP::QuadraticCoefficient::Get`.

GMP::Coefficient::GetQuadratic

The function `GMP::Coefficient::GetQuadratic` 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`.

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.

Remarks:

- 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` should be used.

See also:

The routines `GMP::Coefficient::SetQuadratic` and `GMP::QuadraticCoefficient::Get`.

GMP::Coefficient::Set

The procedure `GMP::Coefficient::Set` 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`.

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.

Remarks:

- Use `GMP::Coefficient::SetMulti` 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.

Examples:

Assume that we have the following variable and constraint declarations (in aim format):.

```
Variable y;
Variable z;
Variable x1;
Constraint e1 {
    Definition :  x1 - 2*y - 3*z = 0;
}
Variable x2 {
    Definition :  2*y + 3*z;
}
```

To change the coefficient of variable `y` in constraint `e1` to 4 we use:

```
GMP::Coefficient::Set( myGMP, e1, 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`, `GMP::Coefficient::SetMulti` and `GMP::QuadraticCoefficient::Set`.

GMP::Coefficient::SetMulti

The procedure `GMP::Coefficient::SetMulti` 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`.

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 `binding`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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 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`.

Examples:

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`, `GMP::Coefficient::Set` and `GMP::QuadraticCoefficient::Set`.

GMP::Coefficient::SetQuadratic

The procedure `GMP::Coefficient::SetQuadratic` 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`.

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.

Remarks:

- 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` should be used.

See also:

The routines `GMP::Coefficient::GetQuadratic` and `GMP::QuadraticCoefficient::Set`.

12.3 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
- GMP::Column::Delete
- GMP::Column::Freeze
- GMP::Column::FreezeMulti
- GMP::Column::GetLowerBound
- GMP::Column::GetName
- GMP::Column::GetScale
- GMP::Column::GetStatus
- GMP::Column::GetType
- GMP::Column::GetUpperBound
- GMP::Column::SetAsMultiObjective
- GMP::Column::SetAsObjective
- GMP::Column::SetDecomposition
- GMP::Column::SetDecompositionMulti
- GMP::Column::SetLowerBound
- GMP::Column::SetLowerBoundMulti
- GMP::Column::SetType
- GMP::Column::SetUpperBound
- GMP::Column::SetUpperBoundMulti
- GMP::Column::Unfreeze
- GMP::Column::UnfreezeMulti

GMP::Column::Add

The procedure `GMP::Column::Add` adds a column to the matrix of a generated mathematical program.

```
GMP::Column::Add(
     GMP,            ! (input) a generated mathematical program
     column          ! (input) a scalar reference
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

column
: A scalar reference to a column.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

Coefficients for this column can be added to the matrix by using the procedure `GMP::Coefficient::Set`. After calling `GMP::Column::Add` 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`, `GMP::Column::SetLowerBound` and `GMP::Column::SetUpperBound` the column type and the lower and upper bound can be changed.

See also:

The routines `GMP::Instance::Generate`, `GMP::Coefficient::Set`, `GMP::Column::Delete`, `GMP::Column::SetType`, `GMP::Column::SetLowerBound` and `GMP::Column::SetUpperBound`.

GMP::Column::Delete

The procedure `GMP::Column::Delete` marks a column in the matrix of 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`.

column
: A scalar reference to an existing column in the matrix 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 procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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` to undo this action.

See also:

The routines `GMP::Instance::Generate` and `GMP::Column::Add`.

GMP::Column::Freeze

The procedure GMP::Column::Freeze freezes a column in the matrix of 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.

column
A scalar reference to an existing column in the matrix or the number of that column 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.

Remarks:

- Use GMP::Column::FreezeMulti if many columns corresponding to some variable 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 to undo the freezing.
- During a call to function GMP::Column::Freeze AIMMS stores the upper and lower bound of the column before the function was called. This information is used when function GMP::Column::Unfreeze is called thereafter. This information is not copied by the function GMP::Instance::Copy.

See also:

The routines GMP::Instance::Generate, GMP::Column::FreezeMulti, GMP::Column::Unfreeze and GMP::Instance::Copy.

GMP::Column::FreezeMulti

The procedure `GMP::Column::FreezeMulti` freezes a group of columns, belonging to a variable, in the matrix of 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`.

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 `binding`, that should be used to freeze the column value.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

- 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` to undo the freezing.
- During a call to function `GMP::Column::FreezeMulti` AIMMS stores the upper and lower bound of the column before the function was called. This information is used when function `GMP::Column::UnfreezeMulti` is called thereafter. This information is not copied by the function `GMP::Instance::Copy`.

Examples:

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`, `GMP::Column::Freeze`, `GMP::Column::UnfreezeMulti` and `GMP::Instance::Copy`.

GMP::Column::GetLowerBound

The function `GMP::Column::GetLowerBound` 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`.

column
: A scalar reference to an existing column in the matrix 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 lower bound value for the specified column.

Remarks:

- 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`, even if the column was deleted.

Examples:

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;
}
```

If we want to multiply the lower bound by 1.5 and assign it as the new value by using function `GMP::Column::SetLowerBound` 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`, `GMP::Column::SetLowerBound`, `GMP::Column::GetUpperBound`, `GMP::Column::GetScale` and `GMP::Instance::CreatePresolved`.

GMP::Column::GetName

The function GMP::Column::GetName 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.

column
: A scalar reference to an existing column in the matrix 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 function returns a string.

See also:

The routines GMP::Instance::Generate and GMP::row::GetName.

GMP::Column::GetScale

The function `GMP::Column::GetScale` 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`.

column

A scalar reference to an existing column in the matrix 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 scaling factor for the specified column.

See also:

The routines `GMP::Instance::Generate` and `GMP::Row::GetScale`.

GMP::Column::GetStatus

The function `GMP::Column::GetStatus` returns the status of a column in the matrix of 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`.

column

A scalar reference to an existing column in the matrix or the number of that column 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`. The set `AllRowColumnStatuses` contains the following elements:

- Active,
- Deactivated,
- Deleted,
- NotGenerated,
- PresolveDeleted.

Remarks:

- This function will return '`PresolveDeleted`' only if the generated mathematical program has been created with `GMP::Instance::CreatePresolved`. 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` and `GMP::Instance::CreatePresolved`.

GMP::Column::GetType

The function GMP::Column::GetType 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.

column
: A scalar reference to an existing column in the matrix or the number of that column 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.

See also:

The routines GMP::Instance::Generate and GMP::Column::SetType.

GMP::Column::GetUpperBound

The function GMP::Column::GetUpperBound 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.

column

A scalar reference to an existing column in the matrix 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 upper bound value for the specified column.

Remarks:

- 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, even if the column was deleted.

Examples:

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;
}
```

If we want to multiply the upper bound by 1.5 and assign it as the new value by using function GMP::Column::SetUpperBound 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, GMP::Column::SetUpperBound, GMP::Column::GetLowerBound, GMP::Column::GetScale and GMP::Instance::CreatePresolved.

GMP::Column::SetAsMultiObjective

The procedure GMP::Column::SetAsMultiObjective sets a column as one of the multi-objectives of a generated mathematical program, thereby creating a multi-objective optimization problem.

```
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.

column
: A scalar reference to an existing column in the matrix or the number of that column 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 by which a solution may deviate from the optimal value of the objective of the previous optimization problem. The default value is 0.0.

reltol
: A scalar value specifying the relative tolerance by which a solution may deviate from the optimal value of the objective of the previous optimization problem. The default value is 0.0.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

- 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` 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` to delete all multi-objectives.
- Multi-objective optimization is only supported by CPLEX 12.9 or higher, and GUROBI 8.0 or higher.
- 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.

Examples:

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`.

GMP::Column::SetAsObjective

The procedure GMP::Column::SetAsObjective 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.

column
: A scalar reference to an existing column in the matrix 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 procedure returns 1 on success, and 0 otherwise.

Remarks:

- 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 the old objective column will be treated as a normal column.

See also:

The routines GMP::Column::Add and GMP::Instance::CreateDual.

GMP::Column::SetDecomposition

The procedure `GMP::Column::SetDecomposition` 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`.

column
: A scalar reference to an existing column in the matrix or the number of that column 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.

Remarks:

- Use `Column::SetDecompositionMulti` 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 12.7 or higher. 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 8.0 or higher. 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.

Examples:

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`, `GMP::Instance::Solve` and `GMP::Column::SetDecompositionMulti`.

GMP::Column::SetDecompositionMulti

The procedure `GMP::Column::SetDecompositionMulti` 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`.

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.

Remarks:

- This procedure can be used to specify the decomposition in the Benders algorithm of CPLEX 12.7 or higher. 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 8.0 or higher. 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.

Examples:

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`, `GMP::Instance::Solve` and `GMP::Column::SetDecomposition`.

GMP::Column::SetLowerBound

The procedure GMP::Column::SetLowerBound changes the lower bound of a column in the 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.

column
A scalar reference to an existing column in the matrix or the number of that column 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.

Remarks:

- Use GMP::Column::SetLowerBoundMulti if the lower bound of many columns corresponding to some variable 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.

Examples:

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, GMP::Column::SetLowerBoundMulti, GMP::Column::SetUpperBound, GMP::Column::GetLowerBound and GMP::Column::GetScale.

GMP::Column::SetLowerBoundMulti

The procedure `GMP::Column::SetLowerBoundMulti` changes the lower bound of a group of columns, belonging to a variable, in the 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`.

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 `binding`.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

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`. See `GMP::Column::SetLowerBound` for an example with units.

Examples:

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`, `GMP::Column::SetLowerBound`, `GMP::Column::SetUpperBound`, `GMP::Column::GetLowerBound` and `GMP::Column::GetScale`.

GMP::Column::SetType

The procedure `GMP::Column::SetType` changes the type of a column in the matrix of a generated mathematical program.

```
GMP::Column::SetType(
     GMP,            ! (input) a generated mathematical program
     column,         ! (input) a scalar reference or column number
     type            ! (input) a element in AllColumnTypes
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

column
: A scalar reference to an existing column in the matrix or the number of that column in the range $\{0..n-1\}$ where n is the number of columns in the matrix.

type
: An element in `AllColumnTypes`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions `GMP::Instance::Generate` and `GMP::Column::GetType`.

GMP::Column::SetUpperBound

The procedure GMP::Column::SetUpperBound 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.

column
A scalar reference to an existing column in the matrix or the number of that column 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.

Remarks:

- Use GMP::Column::SetUpperBoundMulti if the upper bound of many columns corresponding to some variable 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.

Examples:

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`, `GMP::Column::SetUpperBoundMulti`, `GMP::Column::SetLowerBound`, `GMP::Column::GetUpperBound` and `GMP::Column::GetScale`.

GMP::Column::SetUpperBoundMulti

The procedure `GMP::Column::SetUpperBoundMulti` changes the upper bound 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`.

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 `binding`.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

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`. See `GMP::Column::SetUpperBound` for an example with units.

Examples:

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`, `GMP::Column::SetUpperBound`, `GMP::Column::SetLowerBound`, `GMP::Column::GetUpperBound` and `GMP::Column::GetScale`.

GMP::Column::Unfreeze

The procedure `GMP::Column::Unfreeze` removes the frozen status of a column in the matrix of 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`.

column
A scalar reference to an existing column in the matrix 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 procedure returns 1 on success, and 0 otherwise.

Remarks:

- Use `GMP::Column::UnfreezeMulti` if many columns corresponding to some variable have to be unfrozen, because that will be more efficient.
- During a call to function `GMP::Column::Freeze` AIMMS stores the upper and lower bound of the column before the function was called. This information is used when function `GMP::Column::Unfreeze` is called thereafter. This information is not copied by the function `GMP::Instance::Copy`. Therefore the call to `GMP::Column::Unfreeze` 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`, `GMP::Column::UnfreezeMulti`, `GMP::Column::Freeze` and `GMP::Instance::Copy`.

GMP::Column::UnfreezeMulti

The procedure GMP::Column::UnfreezeMulti removes the frozen status of a group of columns, belonging to a variable, in the matrix of 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.

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.

Remarks:

During a call to function GMP::Column::FreezeMulti AIMMS stores the upper and lower bound of the column before the function was called. This information is used when function GMP::Column::UnfreezeMulti is called thereafter. This information is not copied by the function GMP::Instance::Copy.

Examples:

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`, `GMP::Column::Unfreeze`, `GMP::Column::FreezeMulti` and `GMP::Instance::Copy`.

12.4 GMP::Event Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing events:

- `GMP::Event::Create`
- `GMP::Event::Delete`
- `GMP::Event::Reset`
- `GMP::Event::Set`

GMP::Event::Create

The function `GMP::Event::Create` 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`.

See also:

The routines `GMP::Event::Delete`, `GMP::Event::Reset` and `GMP::Event::Set`, and Section 16.6 of the Language Reference.

GMP::Event::Delete

The procedure `GMP::Event::Delete` deletes an event.

```
GMP::Event::Delete(
     Event            ! (input) an event
     )
```

Arguments:

Event
An element in the set `AllGMPEvents`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Event::Create`, `GMP::Event::Reset` and `GMP::Event::Set`, and Section 16.6 of the Language Reference.

GMP::Event::Reset

The procedure `GMP::Event::Reset` resets an event.

```
GMP::Event::Reset(
     Event              ! (input) an event
     )
```

Arguments:

Event
An element in the set `AllGMPEvents`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

The state of the event will be reset to the state just after creating the event.

See also:

The routines `GMP::Event::Create`, `GMP::Event::Delete` and `GMP::Event::Reset`, and Section 16.6 of the Language Reference.

GMP::Event::Set

The procedure `GMP::Event::Set` activates an event.

```
GMP::Event::Set(
     Event            ! (input) an event
     )
```

Arguments:

Event
An element in the set `AllGMPEvents`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Event::Create`, `GMP::Event::Delete` and `GMP::Event::Reset`, and Section 16.6 of the Language Reference.

12.5 GMP::Instance Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing generated mathematical program instances:

- GMP::Instance::AddIntegerEliminationRows
- GMP::Instance::CalculateSubGradient
- GMP::Instance::Copy
- GMP::Instance::CreateDual
- GMP::Instance::CreateFeasibility
- GMP::Instance::CreateMasterMIP
- GMP::Instance::CreatePresolved
- GMP::Instance::CreateProgressCategory
- GMP::Instance::CreateSolverSession
- GMP::Instance::Delete
- GMP::Instance::DeleteIntegerEliminationRows
- GMP::Instance::DeleteMultiObjectives
- GMP::Instance::DeleteSolverSession
- GMP::Instance::FindApproximatelyFeasibleSolution
- GMP::Instance::FixColumns
- GMP::Instance::Generate
- GMP::Instance::GenerateRobustCounterpart
- GMP::Instance::GenerateStochasticProgram
- GMP::Instance::GetBestBound
- GMP::Instance::GetColumnNumbers
- GMP::Instance::GetDirection
- GMP::Instance::GetMathematicalProgrammingType
- GMP::Instance::GetMemoryUsed
- GMP::Instance::GetNumberOfColumns
- GMP::Instance::GetNumberOfIndicatorRows
- GMP::Instance::GetNumberOfIntegerColumns
- GMP::Instance::GetNumberOfNonlinearColumns
- GMP::Instance::GetNumberOfNonlinearNonzeros
- GMP::Instance::GetNumberOfNonlinearRows
- GMP::Instance::GetNumberOfNonzeros
- GMP::Instance::GetNumberOfRows
- GMP::Instance::GetNumberOfSOS1Rows
- GMP::Instance::GetNumberOfSOS2Rows
- GMP::Instance::GetObjective
- GMP::Instance::GetObjectiveColumnNumber
- GMP::Instance::GetObjectiveRowNumber
- GMP::Instance::GetOptionValue
- GMP::Instance::GetRowNumbers
- GMP::Instance::GetSolver
- GMP::Instance::GetSymbolicMathematicalProgram

- GMP::Instance::MemoryStatistics
- GMP::Instance::Rename
- GMP::Instance::SetCallbackAddCut
- GMP::Instance::SetCallbackAddLazyConstraint
- GMP::Instance::SetCallbackBranch
- GMP::Instance::SetCallbackCandidate
- GMP::Instance::SetCallbackHeuristic
- GMP::Instance::SetCallbackIncumbent
- GMP::Instance::SetCallbackIterations
- GMP::Instance::SetCallbackStatusChange
- GMP::Instance::SetCallbackTime
- GMP::Instance::SetCutoff
- GMP::Instance::SetDirection
- GMP::Instance::SetIterationLimit
- GMP::Instance::SetMathematicalProgrammingType
- GMP::Instance::SetMemoryLimit
- GMP::Instance::SetOptionValue
- GMP::Instance::SetSolver
- GMP::Instance::SetStartingPointSelection
- GMP::Instance::SetTimeLimit
- GMP::Instance::Solve

GMP::Instance::AddIntegerEliminationRows

The procedure `GMP::Instance::AddIntegerEliminationRows` 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
     elimNo         ! (input) an elimination number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

solution
: An integer scalar reference to a solution.

elimNo
: An integer scalar reference to an elimination number.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- If the GMP is not integer then this procedure will fail.
- Rows and columns added before for *elimNo* 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:

$$l_i + (lev_i - lb_i)z_i \leq (lev_i - lb_i) \tag{12.1}$$

$$u_i + (lev_i - ub_i)z_i \leq 0 \tag{12.2}$$

$$x_i - u_i + l_i = lev_i \tag{12.3}$$

  Every call to `GMP::Instance::AddIntegerEliminationRows` also adds the following constraint:

$$\sum_{i \in S_{lo}} x_i - \sum_{i \in S_{up}} x_i + \sum_{i \in S_{in}} (l_i + u_i) \geq 1 + \sum_{i \in S_{lo}} lev_i - \sum_{i \in S_{up}} lev_i \qquad (12.4)$$

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`:
  - 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 (12.1), (12.2) and (12.3) respectively.)
  - Constraint `mp.ExtendedConstraint('Elimination`*k*`')`, where `mp` denotes the symbolic mathematical program, is added for every call to `GMP::Instance::AddIntegerEliminationRows`. (This constraint corresponds to (12.4).)

  Here k denotes the value of the argument *elimNo*.

Examples:

The procedure `GMP::Instance::AddIntegerEliminationRows` 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.
    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` and `GMP::Solution::IsInteger`. See Section 16.3.6 of the Language Reference for more details on extended suffixes.

GMP::Instance::CalculateSubGradient

The procedure `GMP::Instance::CalculateSubGradient` 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`. The mathematical program should have model type LP or RMIP.

variableSet
A subset of `AllVariables`.

constraintSet
A subset of `AllConstraints`.

session
An element in the set `AllSolverSessions`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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`.
- This procedure can also be called after a successful call to the procedure `GMP::SolverSession::Execute` or the procedure `GMP::SolverSession::AsynchronousExecute`. 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 7.0 or higher.
- This procedure cannot be used if the *GMP* is created by `GMP::Instance::CreateDual`.

Examples:

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`, `GMP::Instance::Solve`, `GMP::SolverSession::Execute` and `GMP::SolverSession::AsynchronousExecute`. See Section 16.3.6 of the Language Reference for more details on extended suffixes.

GMP::Instance::Copy

The function `GMP::Instance::Copy` creates a copy of a generated mathematical program and an associated new element in the set `AllGeneratedMathematicalPrograms`.

```
GMP::Instance::Copy(
     GMP,            ! (input) a generated mathematical program
     name            ! (input) a string expression
     )
```

Arguments:

GMP
: An element in the set `AllGeneratedMathematicalPrograms`.

name
: A string that contains the name for the copy of the generated mathematical program.

Return value:

A new element in the set `AllGeneratedMathematicalPrograms` with the name as specified by the *name* argument.

Remarks:

- 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` 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` (if any) will not be copied.

See also:

The routines `GMP::Instance::Generate`, `GMP::Instance::Rename` and `GMP::Instance::SetSolver`.

GMP::Instance::CreateDual

The function `GMP::Instance::CreateDual` 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`.

name
: A string that holds the name for the dual of the generated mathematical program.

Return value:

A new element in the set `AllGeneratedMathematicalPrograms` with the name as specified by the *name* argument.

Remarks:

- 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` the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved).
- 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 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.

Examples:

Assume that 'PrimalModel' is a mathematical program with the following declaration (in aim 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` will create a dual mathematical program with variables

```
name                                           lower  upper
c1                                              -inf      0
obj_definition                                  -inf    inf
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`. See Section 16.3.6 of the Language Reference for more details on extended suffixes.

GMP::Instance::CreateFeasibility

The function GMP::Instance::CreateFeasibility 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.

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.

$$\begin{aligned}
\max \quad & \sum_{j\in J} a_j x_j \\
\text{s.t.} \quad & \sum_{j\in J} a_{ij} x_j \geq b_i \quad i \in I_1 \\
& \sum_{j\in J} a_{ij} x_j \leq b_i \quad i \in I_2 \\
& \sum_{j\in J} a_{ij} x_j = b_i \quad i \in I_3 \\
& x \geq 0
\end{aligned}$$

Then if we minimize the sum of infeasibilities the feasibility problem becomes:

$$
\begin{array}{lllll}
\min & \sum_{i\in I_1} z_i^p + \sum_{i\in I_2} z_i^n + & & & \\
 & \sum_{i\in I_3} (z_i^p + z_i^n) & & & \\
\text{s.t.} & \sum_{j\in J} a_{ij}x_j + z_i^p & \geq b_i & i \in I_1 \\
 & \sum_{j\in J} a_{ij}x_j - z_i^n & \leq b_i & i \in I_2 \\
 & \sum_{j\in J} a_{ij}x_j + z_i^p - z_i^n & = b_i & i \in I_3 \\
 & x, z^p, z^n \geq 0 & &
\end{array}
$$

If we minimize the maximum infeasibility the feasibility problem becomes:

$$
\begin{array}{lllll}
\min & z^m & & & \\
\text{s.t.} & \sum_{j\in J} a_{ij}x_j + z^m & \geq b_i & i \in I_1 \\
 & \sum_{j\in J} a_{ij}x_j - z^m & \leq b_i & i \in I_2 \\
 & \sum_{j\in J} a_{ij}x_j + z_i^p - z_i^n & = b_i & i \in I_3 \\
 & z^m - z_i^p - z_i^n & \geq 0 & i \in I_3 \\
 & x, z^p, z^n \geq 0 & &
\end{array}
$$

Return value:

A new element in the set `AllGeneratedMathematicalPrograms` with the name as specified by the *name* argument.

Remarks:

- 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` 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` and `GMP::Instance::Solve`.

GMP::Instance::CreateMasterMIP

The function `GMP::Instance::CreateMasterMIP` 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`.

name
: A string that holds the name for the Master MIP.

Return value:

A new element in the set `AllGeneratedMathematicalPrograms` with the name as specified by the *name* argument.

Remarks:

- 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` 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` 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

  $$\texttt{objvar} - \texttt{mp.ExtendedVariable(MasterMIPObjective)} = 0$$

  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`. See Section 16.3.6 of the Language Reference for more details on extended suffixes.

GMP::Instance::CreatePresolved

The function `GMP::Instance::CreatePresolved` 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`.

```
GMP::Instance::CreatePresolved(
     GMP,            ! (input) a generated mathematical program
     name            ! (input) a string expression
)
```

Arguments:

GMP
An element in the set `AllGeneratedMathematicalPrograms`.

name
A string that holds the name for the presolved mathematical program.

Return value:

A new element in the set `AllGeneratedMathematicalPrograms`, with the name as specified by the *name* argument, if the presolver did not find an infeasibility. Else, the empty element.

Remarks:

- By using the functions `GMP::Column::GetStatus` and `GMP::Row::GetStatus` 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` and `GMP::Column::GetUpperBound` 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` 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.

Examples:

Assume that 'MP' is a mathematical program and 'gmpMP' and 'gmpPre' are element parameters with range `AllGeneratedMathematicalPrograms`. 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`, `GMP::Instance::Generate`, `GMP::Instance::Solve`, `GMP::Column::GetStatus`, `GMP::Row::GetStatus`, `GMP::Column::GetLowerBound` and `GMP::Column::GetUpperBound`.

GMP::Instance::CreateProgressCategory

The function `GMP::Instance::CreateProgressCategory` 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`.

Name
: A string that holds the name of the progress category.

Return value:

The function returns an element in the set `AllProgressCategories`.

Remarks:

- 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` 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`.
- 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` and `GMP::SolverSession::CreateProgressCategory`.

GMP::Instance::CreateSolverSession

The function `GMP::Instance::CreateSolverSession` 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`.

Name
: A string that holds the name of the solver session.

Solver
: An element in the set `AllSolvers`.

Return value:

The function returns an element in the set `AllSolverSessions`.

Remarks:

- The function `GMP::Instance::CreateSolverSession` 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` 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`.
- If an element with name specified by the *Name* argument is already present in the set `AllSolverSessions` then the corresponding solver session will first be deleted.

See also:

The routines `GMP::Instance::DeleteSolverSession`, `GMP::Instance::SetSolver`, `GMP::SolverSession::GetInstance` and `GMP::SolverSession::GetSolver`.

GMP::Instance::Delete

The procedure `GMP::Instance::Delete` deletes a generated mathematical program from the set `AllGeneratedMathematicalPrograms`.

```
GMP::Instance::Delete(
     GMP            ! (input) a generated mathematical program
     )
```

Arguments:

GMP
: An element in the set `AllGeneratedMathematicalPrograms`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

All memory associated with the generated mathematical program is also freed.

See also:

The function `GMP::Instance::Generate`.

GMP::Instance::DeleteIntegerEliminationRows

The procedure `GMP::Instance::DeleteIntegerEliminationRows` deletes a particular set of integer elimination rows and columns of a generated mathematical program.

```
GMP::Instance::DeleteIntegerEliminationRows(
     GMP,          ! (input) a generated mathematical program
     elimNo        ! (input) an elimination number
     )
```

Arguments:

GMP
An element in `AllGeneratedMathematicalPrograms`.

elimNo
An integer scalar reference to an elimination number.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The procedure `GMP::Instance::AddIntegerEliminationRows`.

GMP::Instance::DeleteMultiObjectives

The procedure `GMP::Instance::DeleteMultiObjectives` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

A column can be specified as a multi-objective by using the procedure `GMP::Column::SetAsMultiObjective`.

Examples:

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` 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`.

GMP::Instance::DeleteSolverSession

The procedure `GMP::Instance::DeleteSolverSession` deletes the specified solver session.

```
GMP::Instance::DeleteSolverSession(
     solverSession     ! (input) a solver session
     )
```

Arguments:

solverSession
An element in the set `AllSolverSessions`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions `GMP::Instance::CreateSolverSession` and `GMP::SolverSession::GetInstance`.

GMP::Instance::FindApproximatelyFeasibleSolution

The procedure `GMP::Instance::FindApproximatelyFeasibleSolution` 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`.

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.

Remarks:

- 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 can be used to get the sum and number of infeasibilies before and after calling the procedure GMP::Instance::FindApproximatelyFeasibleSolution.

See also:

The routines GMP::Instance::Generate, GMP::Instance::Solve and GMP::Solution::Check.

GMP::Instance::FixColumns

The procedure `GMP::Instance::FixColumns` 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`.

GMP2
: An element in `AllGeneratedMathematicalPrograms`.

solution
: An integer scalar reference to a solution in the solution repository of *GMP2*.

variableSet
: A subset of `AllVariables`.

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.

Remarks:

- 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` and `GMP::SolverSession::GetInstance`.

GMP::Instance::Generate

The function `GMP::Instance::Generate` 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`. 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` with the name as specified by the *name* argument.

Remarks:

- 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`.
- If an element with name specified by the *name* argument is already present in the set `AllGeneratedMathematicalPrograms` the corresponding generated mathematical program will be replaced (or updated in case the same symbolic mathematical program is involved). 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 Section 16.13.1 of the Language Reference.
- A callback procedure should be installed using the appropriate GMP procedure, (e.g., `GMP::Instance::SetCallbackIterations`) instead of using a suffix of the mathematical program (e.g., suffix `CallbackIterations`).
- If an error occurs during the execution of `GMP::Instance::Generate`, 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`.

See also:

The routines `GMP::Instance::Delete` and
`GMP::Instance::SetCallbackIterations`.

GMP::Instance::GenerateRobustCounterpart

The function `GMP::Instance::GenerateRobustCounterpart` 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`. 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` with the name as specified by the *name* argument.

Remarks:

- 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`.

- 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`.

GMP::Instance::GenerateStochasticProgram

The function GMP::Instance::GenerateStochasticProgram 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. The mathematical program should have model type LP or MIP.

StochasticParameters
: A subset of AllStochasticParameters.

StochasticVariables
: A subset of AllStochasticVariables.

Scenarios
: A subset of AllStochasticScenarios.

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. 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. 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` with the name as specified by the *name* argument.

Remarks:

- 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`.
- The objective of the symbolic mathematical program must be a defined variable.

See also:

- Stochastic programming is discussed in Chapter 19 of the Language Reference.
- The procedure `GMP::Instance::Solve`.

GMP::Instance::GetBestBound

The function `GMP::Instance::GetBestBound` 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`.

Return value:

In case of success, the function returns the best known bound. Otherwise it returns `UNDF`.

Remarks:

- This function has only meaning for generated mathematical programs with model type MIP, MIQP or MIQCP.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetMathematicalProgrammingType` and `GMP::Instance::GetObjective`.

GMP::Instance::GetColumnNumbers

The function GMP::Instance::GetColumnNumbers 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.

variableSet
A subset of the set AllVariables.

Return value:

The function returns a subset of the column numbers as a subset of the set Integers.

Examples:

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 this can also be coded as follows. Here ColNrs is a subset of Integers with index c, and Vars a subset of AllVariables.

```
myGMP := GMP::Instance::Generated( MP );

Vars := { 'demand', 'supply' };

ColNrs := GMP::Instance::GetColumnNumbers( myGMP, Vars );

for (c) do
    GMP::Column::SetUpperBound( myGMP, c, 100 );
endfor;
```

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfColumns`, `GMP::Instance::GetRowNumbers`, `GMP::Instance::GetObjectiveColumnNumber` and `GMP::Instance::GetObjectiveRowNumber`.

GMP::Instance::GetDirection

The function `GMP::Instance::GetDirection` 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`.

Return value:

The function returns the optimization direction as an element in `AllMatrixManipulationDirections`.

See also:

The routines `GMP::Instance::Generate` and the procedure `GMP::Instance::SetDirection`.

GMP::Instance::GetMathematicalProgrammingType

The function `GMP::Instance::GetMathematicalProgrammingType` 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`.

Return value:

The function returns the model type as an element in `AllMathematicalProgrammingTypes`.

See also:

The function `GMP::Instance::Generate` and the procedure `GMP::Instance::SetMathematicalProgrammingType`.

GMP::Instance::GetMemoryUsed

The function `GMP::Instance::GetMemoryUsed` 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`.

Return value:

The amount of megabytes used to store a generated mathematical program.

GMP::Instance::GetNumberOfColumns

The function `GMP::Instance::GetNumberOfColumns` 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`.

Return value:

The function returns the number of columns.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfRows` and `GMP::Instance::GetNumberOfNonzeros`.

GMP::Instance::GetNumberOfIndicatorRows

The function GMP::Instance::GetNumberOfIndicatorRows 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.

Return value:

The function returns the number of indicator rows.

See also:

The functions GMP::Instance::Generate and GMP::Instance::GetNumberOfRows.

GMP::Instance::GetNumberOfIntegerColumns

The function `GMP::Instance::GetNumberOfIntegerColumns` 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`.

Return value:

The function returns the number of integer columns.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfColumns` and `GMP::Instance::GetNumberOfNonlinearColumns`.

GMP::Instance::GetNumberOfNonlinearColumns

The function `GMP::Instance::GetNumberOfNonlinearColumns` 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`.

Return value:

The function returns the number of nonlinear columns.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfColumns` and `GMP::Instance::GetNumberOfIntegerColumns`.

GMP::Instance::GetNumberOfNonlinearNonzeros

The function `GMP::Instance::GetNumberOfNonlinearNonzeros` 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`.

Return value:

The function returns the number of nonlinear nonzeros.

See also:

The functions `GMP::Instance::Generate` and `GMP::Instance::GetNumberOfNonzeros`.

GMP::Instance::GetNumberOfNonlinearRows

The function `GMP::Instance::GetNumberOfNonlinearRows` 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`.

Return value:

The function returns the number of nonlinear rows.

See also:

The functions `GMP::Instance::Generate` and `GMP::Instance::GetNumberOfRows`.

GMP::Instance::GetNumberOfNonzeros

The function `GMP::Instance::GetNumberOfNonzeros` 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`.

Return value:

The function returns the number of nonzeros.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfColumns` and `GMP::Instance::GetNumberOfRows`.

GMP::Instance::GetNumberOfRows

The function GMP::Instance::GetNumberOfRows 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.

Return value:

The function returns the number of rows.

See also:

The functions GMP::Instance::Generate, GMP::Instance::GetNumberOfColumns and GMP::Instance::GetNumberOfNonzeros.

GMP::Instance::GetNumberOfSOS1Rows

The function `GMP::Instance::GetNumberOfSOS1Rows` 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`.

Return value:

The function returns the number of SOS rows of type 1.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfRows` and `GMP::Instance::GetNumberOfSOS2Rows`.

GMP::Instance::GetNumberOfSOS2Rows

The function `GMP::Instance::GetNumberOfSOS2Rows` 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`.

Return value:

The function returns the number of SOS rows of type 2.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetNumberOfRows` and `GMP::Instance::GetNumberOfSOS1Rows`.

GMP::Instance::GetObjective

The function `GMP::Instance::GetObjective` 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`.

Return value:

In case of success, the function returns the current objective function value. Otherwise it returns `UNDF`.

See also:

The routines `GMP::Instance::Generate`, `GMP::Instance::Solve` and `GMP::Instance::GetBestBound`.

GMP::Instance::GetObjectiveColumnNumber

The function `GMP::Instance::GetObjectiveColumnNumber` 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`.

Return value:

The function returns the column number as an element of the set `Integers`. If the generated mathematical program does not contain an objective then -1 is returned.

Remarks:

You should assign the return value of this function to an element parameter with range `Integers` if you want to use it as (column) argument to call other GMP routines.

Examples:

Let `ColNo` be an element parameter with range `Integers`.

```
ColNo := GMP::Instance::GetObjectiveColumnNumber( myGMP );

value := GMP::Solution::GetColumnValue( myGMP, 1, ColNo );
```

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetColumnNumbers`, `GMP::Instance::GetObjectiveRowNumber` and `GMP::Instance::GetRowNumbers`.

GMP::Instance::GetObjectiveRowNumber

The function GMP::Instance::GetObjectiveRowNumber 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.

Return value:

The function returns the row number as an element of the set Integers. If the generated mathematical program does not contain an objective then -1 is returned.

Remarks:

- You should assign the return value of this function to an element parameter with range Integers 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.

Examples:

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 );

for (c) do
    GMP::Coefficient::Set( myGMP, RowNo, c, 10 );
endfor;
```

Here RowNo is an element parameter with range Integers and ColNrs a subset of Integers with index c.

See also:

The functions GMP::Instance::Generate, GMP::Instance::GetColumnNumbers, GMP::Instance::GetObjectiveColumnNumber and GMP::Instance::GetRowNumbers.

GMP::Instance::GetOptionValue

The function `GMP::Instance::GetOptionValue` returns the value of a solver specific option corresponding to a generated mathematical program as set with the procedure `GMP::Instance::SetOptionValue`.

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`.

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`.

Remarks:

- If the procedure `GMP::Instance::SetOptionValue` 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`, `GMP::SolverSession::GetOptionValue`, `GMP::SolverSession::SetOptionValue`, `OptionGetString` and `OptionGetKeywords`.

GMP::Instance::GetRowNumbers

The function `GMP::Instance::GetRowNumbers` 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`.

constraintSet
: A subset of the set `AllConstraints`.

Return value:

The function returns a subset of the row numbers as a subset of the set `Integers`.

Examples:

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` this can also be coded as follows. Here `RowNrs` is a subset of `Integers` with index `r`, and `Cons` a subset of `AllConstraints`.

```
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`, `GMP::Instance::GetColumnNumbers`, `GMP::Instance::GetNumberOfRows`, `GMP::Instance::GetObjectiveColumnNumber` and `GMP::Instance::GetObjectiveRowNumber`.

GMP::Instance::GetSolver

The function `GMP::Instance::GetSolver` 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`.

Return value:

The function returns the solver as an element of `AllSolvers`.

Remarks:

The solver can be assigned by the procedure `GMP::Instance::SetSolver`, or derived by AIMMS as the default solver for the model class of the generated mathematical program.

See also:

The routines `GMP::Instance::Generate` and `GMP::Instance::SetSolver`.

GMP::Instance::GetSymbolicMathematicalProgram

The function `GMP::Instance::GetSymbolicMathematicalProgram` 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`.

Return value:

The function returns the symbolic mathematical program as an element of `AllMathematicalPrograms`.

See also:

The function `GMP::Instance::Generate`.

GMP::Instance::MemoryStatistics

With the procedure `GMP::Instance::MemoryStatistics` you can obtain a report containing the statistics collected by AIMMS' memory manager for a single or multiple generated mathematical 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` 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`.

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`.

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.

Remarks:

- The procedure prints a report of the statistics collected by AIMMS' memory manager since the last call to `GMP::Instance::MemoryStatistics`.
- AIMMS will only collect memory statistics if the option `memory_statistics` is `on`.

GMP::Instance::Rename

The procedure GMP::Instance::Rename 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.

Name
A string that holds the new name.

Return value:

GMP::Instance::Rename has no return value.

See also:

The functions GMP::Instance::Generate and GMP::Instance::Copy.

GMP::Instance::SetCallbackAddCut

The procedure `GMP::Instance::SetCallbackAddCut` 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`.

callback
: A reference to a procedure in the set `AllIdentifiers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The procedure `GMP::SolverSession::GenerateCut` can be used inside a `CallbackAddCut` callback procedure to add cuts during the MIP branch & cut process.
- The callback procedure should have exactly one argument; a scalar input element parameter into the set `AllSolverSessions`.
- 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`, `GMP::Instance::SetCallbackAddLazyConstraint`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic`, `GMP::Instance::SetCallbackIncumbent`, `GMP::SolverSession::GenerateBinaryEliminationRow` and `GMP::SolverSession::GenerateCut`.

GMP::Instance::SetCallbackAddLazyConstraint

The procedure `GMP::Instance::SetCallbackAddLazyConstraint` 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`.

callback
A reference to a procedure in the set `AllIdentifiers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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` can be used inside a `CallbackAddLazyConstraint` callback procedure to add (globally or locally valid) lazy constraints during the MIP branch & 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`.
- 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`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic`, `GMP::Instance::SetCallbackIncumbent` and `GMP::SolverSession::GenerateCut`.

GMP::Instance::SetCallbackBranch

The procedure GMP::Instance::SetCallbackBranch 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.

callback
: A reference to a procedure in the set AllIdentifiers.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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.
- 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, GMP::Solution::SendToModel, GMP::Solution::RetrieveFromModel, GMP::Solution::SendToSolverSession, GMP::SolverSession::GenerateBranchLowerBound, GMP::SolverSession::GenerateBranchUpperBound, GMP::SolverSession::GenerateBranchRow, GMP::SolverSession::GetNumberOfBranchNodes, GMP::Instance::Generate,

`GMP::Instance::SetCallbackAddCut`,
`GMP::Instance::SetCallbackAddLazyConstraint`,
`GMP::Instance::SetCallbackCandidate`,
`GMP::Instance::SetCallbackHeuristic` and
`GMP::Instance::SetCallbackIncumbent`.

GMP::Instance::SetCallbackCandidate

The procedure `GMP::Instance::SetCallbackCandidate` 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` the incumbent solution can be rejected. If `GMP::SolverSession::RejectIncumbent` 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`.

callback
: A reference to a procedure in the set `AllIdentifiers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set `AllSolverSessions`.
- 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`, 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`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`, GMP::Instance::SetCallbackBranch, GMP::Instance::SetCallbackHeuristic, GMP::Instance::SetCallbackIncumbent and GMP::SolverSession::RejectIncumbent.

GMP::Instance::SetCallbackHeuristic

The procedure `GMP::Instance::SetCallbackHeuristic` 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`.

callback
A reference to a procedure in the set `AllIdentifiers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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`.
- 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`, `GMP::Solution::SendToModel`, `GMP::Solution::RetrieveFromModel`, `GMP::Solution::SendToSolverSession`, `GMP::Instance::Generate`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate` and `GMP::Instance::SetCallbackIncumbent`.

GMP::Instance::SetCallbackIncumbent

The procedure GMP::Instance::SetCallbackIncumbent 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.

callback
: A reference to a procedure in the set AllIdentifiers.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set AllSolverSessions.
- 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.
- The functionality of the procedure GMP::Instance::SetCallbackIncumbent 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.

See also:

The routines GMP::Instance::Generate, GMP::Instance::SetCallbackAddCut, GMP::Instance::SetCallbackAddLazyConstraint, GMP::Instance::SetCallbackBranch, GMP::Instance::SetCallbackCandidate, GMP::Instance::SetCallbackHeuristic, GMP::Instance::SetCallbackIterations, GMP::Instance::SetCallbackStatusChange and GMP::Instance::SetCallbackTime.

GMP::Instance::SetCallbackIterations

The procedure `GMP::Instance::SetCallbackIterations` 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`.

callback
: A reference to a procedure in the set `AllIdentifiers`.

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.

Remarks:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set `AllSolverSessions`.
- 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.

See also:

The routines `GMP::Instance::Generate`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`,

`GMP::Instance::SetCallbackHeuristic`,
`GMP::Instance::SetCallbackIncumbent`,
`GMP::Instance::SetCallbackStatusChange` and
`GMP::Instance::SetCallbackTime`.

GMP::Instance::SetCallbackStatusChange

The procedure GMP::Instance::SetCallbackStatusChange 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.

callback
: A reference to a procedure in the set AllIdentifiers.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set AllSolverSessions.
- 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.

See also:

The routines GMP::Instance::Generate, GMP::Instance::SetCallbackAddCut, GMP::Instance::SetCallbackAddLazyConstraint, GMP::Instance::SetCallbackBranch, GMP::Instance::SetCallbackCandidate, GMP::Instance::SetCallbackHeuristic, GMP::Instance::SetCallbackIncumbent, GMP::Instance::SetCallbackIterations and GMP::Instance::SetCallbackTime.

GMP::Instance::SetCallbackTime

The procedure `GMP::Instance::SetCallbackTime` 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`.

callback
: A reference to a procedure in the set `AllIdentifiers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The callback procedure should have exactly one argument; a scalar input element parameter into the set `AllSolverSessions`.
- 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, 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`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`,

`GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`,
`GMP::Instance::SetCallbackHeuristic`,
`GMP::Instance::SetCallbackIncumbent` and
`GMP::Instance::SetCallbackStatusChange`.

GMP::Instance::SetCutoff

The procedure `GMP::Instance::SetCutoff` 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`.

cutoff
A scalar value.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

This procedure is only used for MIP models.

See also:

The routines `GMP::Instance::Generate`, `GMP::Instance::Solve`, `GMP::Instance::SetIterationLimit`, `GMP::Instance::GMP::Instance::SetMemoryLimit` and `GMP::Instance::SetTimeLimit`.

GMP::Instance::SetDirection

The procedure `GMP::Instance::SetDirection` 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`.

direction
: An element expression in the set `AllMatrixManipulationDirections`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions `GMP::Instance::Generate` and `GMP::Instance::GetDirection`.

GMP::Instance::SetIterationLimit

The procedure `GMP::Instance::SetIterationLimit` 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`.

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`, `GMP::Instance::Solve`, `GMP::Instance::SetCutoff`, `GMP::Instance::SetMemoryLimit` and `GMP::Instance::SetTimeLimit`.

GMP::Instance::SetMathematicalProgrammingType

The procedure `GMP::Instance::SetMathematicalProgrammingType` 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`.

MathematicalProgrammingType
: One of the elements LP, MIP, RMIP, MINLP or RMINLP (in the set `AllMatrixManipulationProgrammingTypes`).

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions `GMP::Instance::Generate` and `GMP::Instance::GetMathematicalProgrammingType`.

GMP::Instance::SetMemoryLimit

The procedure `GMP::Instance::SetMemoryLimit` 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`.

memory
Maximum number of megabytes 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`, `GMP::Instance::Solve`, `GMP::Instance::SetCutoff`, `GMP::Instance::SetIterationLimit` and `GMP::Instance::SetTimeLimit`.

GMP::Instance::SetOptionValue

The procedure `GMP::Instance::SetOptionValue` 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`.

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.

Remarks:

- 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` 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, GMP::SolverSession::GetOptionValue, GMP::SolverSession::SetOptionValue, OptionGetString and OptionGetKeywords.

GMP::Instance::SetSolver

The procedure `GMP::Instance::SetSolver` can be used to select for a generated mathematical program the solver to be called in subsequent calls to `GMP::Instance::Solve`.

```
GMP::Instance::SetSolver(
     GMP,          ! (input) a generated mathematical program
     solver        ! (input) a solver
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

solver
: An element in the set `AllSolvers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

The solver set in this procedure will also be assigned to any solver session created with the function `GMP::Instance::CreateSolverSession` for the *GMP*, unless the *Solver* argument in the procedure `GMP::Instance::CreateSolverSession` is specified. Note that the procedure `GMP::Instance::SetSolver` cannot be used to change the solver assigned to a solver session after `GMP::Instance::CreateSolverSession` has been called.

See also:

The routines `GMP::Instance::CreateSolverSession`, `GMP::Instance::Generate`, `GMP::Instance::GetSolver` and `GMP::Instance::Solve`.

GMP::Instance::SetStartingPointSelection

The procedure `GMP::Instance::SetStartingPointSelection` 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
     selectedColumnNumbers ! (input) a subset of Integers
     )
```

Arguments:

GMP
: An element in the set `AllGeneratedMathematicalPrograms`.

selectedColumnNumbers
: An expression that results in a subset of the set `Integers`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The functions `GMP::Instance::Generate`, `GMP::Instance::GetColumnNumbers` and `GMP::Instance::Solve`.

GMP::Instance::SetTimeLimit

The procedure GMP::Instance::SetTimeLimit 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.

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, GMP::Instance::Solve, GMP::Instance::SetCutoff, GMP::Instance::SetIterationLimit and GMP::Instance::SetMemoryLimit.

GMP::Instance::Solve

The procedure `GMP::Instance::Solve` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

The procedure `GMP::Instance::Solve` automatically creates a solver session with the same name as the generated mathematical program in the set `AllSolverSessions`.

See also:

The routines `GMP::Instance::Generate`, `GMP::Instance::CreateSolverSession`, `GMP::Solution::RetrieveFromModel`, `GMP::Solution::SendToSolverSession`, `GMP::SolverSession::Execute`, `GMP::Solution::RetrieveFromSolverSession` and `GMP::Solution::SendToModel`.

12.6 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
- GMP::Linearization::AddSingle
- GMP::Linearization::Delete
- GMP::Linearization::GetDeviation
- GMP::Linearization::GetDeviationBound
- GMP::Linearization::GetLagrangeMultiplier
- GMP::Linearization::GetType
- GMP::Linearization::GetWeight
- GMP::Linearization::RemoveDeviation
- GMP::Linearization::SetDeviationBound
- GMP::Linearization::SetType
- GMP::Linearization::SetWeight

GMP::Linearization::Add

The procedure `GMP::Linearization::Add` 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
     [jacTol]               ! (optional) the Jacobian tolerance
     )
```

Arguments:

GMP1
: An element in `AllGeneratedMathematicalPrograms`.

GMP2
: An element in `AllGeneratedMathematicalPrograms`.

solution
: An integer scalar reference to a solution in the solution repository of *GMP2*.

constraintSet
: A subset of `AllNonLinearConstraints`.

deviationsPermitted
: A binary parameter over `AllNonLinearConstraints` 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` 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.

Remarks:

- 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.

$$g(x^0,y^0) + \nabla g(x^0,y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} - z_j \leq b_j$$

  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 '≤' 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 '≥' 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`:
  - 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` and `GMP::Linearization::Delete`. See Section 16.3.6 of the Language Reference for more details on extended suffixes.

GMP::Linearization::AddSingle

The procedure `GMP::Linearization::AddSingle` 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
     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`.

GMP2
: An element in `AllGeneratedMathematicalPrograms`.

solution
: An integer scalar reference to a solution in the solution repository of *GMP2*.

row
: A scalar reference to an existing nonlinear row in *GMP2* for which the linearization is added to *GMP1*.

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.

Remarks:

- 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:

  $$g(x^0,y^0) + \nabla g(x^0,y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} - z_j \leq b_j$$

  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 '≤' 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 '≥' 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`:
  - 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*.

Examples:

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 {
    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` and `GMP::Linearization::Delete`. See Section 16.3.6 of the Language Reference for more details on extended suffixes.

GMP::Linearization::Delete

The procedure `GMP::Linearization::Delete` 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`.

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` and `GMP::Linearization::AddSingle`.

GMP::Linearization::GetDeviation

The function `GMP::Linearization::GetDeviation` 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
     linNo   ! (input) a linearization number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row 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` and `GMP::Linearization::GetDeviationBound`.

GMP::Linearization::GetDeviationBound

The function `GMP::Linearization::GetDeviationBound` 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
     linNo   ! (input) a linearization number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in 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` and `GMP::Linearization::GetDeviation`.

GMP::Linearization::GetLagrangeMultiplier

The function `GMP::Linearization::GetLagrangeMultiplier` 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
     linNo   ! (input) a linearization number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in 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`.

GMP::Linearization::GetType

The function `GMP::Linearization::GetType` 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
     linNo   ! (input) a linearization number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in in the matrix.

linNo
: An integer scalar reference to the rows and columns of the linearization.

Return value:

An element in the set `AllRowTypes`.

See also:

The procedure `GMP::Linearization::SetType`.

GMP::Linearization::GetWeight

The function `GMP::Linearization::GetWeight` 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
     linNo   ! (input) a linearization number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in 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.

Remarks:

- 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` or `GMP::Linearization::AddSingle`.

See also:

The procedures `GMP::Linearization::Add`, `GMP::Linearization::AddSingle` and `GMP::Linearization::SetWeight`.

GMP::Linearization::RemoveDeviation

The procedure `GMP::Linearization::RemoveDeviation` 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
     linNo   ! (input) a linearization number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in 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`, `GMP::Linearization::Add` and `GMP::Linearization::AddSingle`.

GMP::Linearization::SetDeviationBound

The procedure `GMP::Linearization::SetDeviationBound` 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
     linNo,  ! (input) a linearization number
     value   ! (input) a scalar value
     )
```

Arguments:

GMP
An element in `AllGeneratedMathematicalPrograms`.

row
A scalar reference to an existing nonlinear row in 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`, `GMP::Linearization::GetDeviation` and `GMP::Linearization::RemoveDeviation`.

GMP::Linearization::SetType

The procedure `GMP::Linearization::SetType` 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
     linNo,  ! (input) a linearization number
     rowtype ! (input) a row type
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The function `GMP::Linearization::GetType`.

GMP::Linearization::SetWeight

The procedure `GMP::Linearization::SetWeight` 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
     linNo,  ! (input) a linearization number
     value   ! (input) a scalar value
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing nonlinear row in 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`.

12.7 GMP::ProgressWindow Procedures and Functions

AIMMS supports the following procedures and functions for displaying progress information in the Progress Window:

- `GMP::ProgressWindow::DeleteCategory`
- `GMP::ProgressWindow::DisplayLine`
- `GMP::ProgressWindow::DisplayProgramStatus`
- `GMP::ProgressWindow::DisplaySolver`
- `GMP::ProgressWindow::DisplaySolverStatus`
- `GMP::ProgressWindow::FreezeLine`
- `GMP::ProgressWindow::Transfer`
- `GMP::ProgressWindow::UnfreezeLine`

GMP::ProgressWindow::DeleteCategory

The procedure `GMP::ProgressWindow::DeleteCategory` deletes a progress category.

```
GMP::ProgressWindow::DeleteCategory(
     Category         ! (input) a progress category
     )
```

Arguments:

Category
An element in the set `AllProgressCategories`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Instance::CreateProgressCategory` and `GMP::SolverSession::CreateProgressCategory`.

GMP::ProgressWindow::DisplayLine

The procedure `GMP::ProgressWindow::DisplayLine` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- If the *Category* argument is used then the element should be created with the function `GMP::SolverSession::CreateProgressCategory`.
- To freeze (lock) a line the procedure `GMP::ProgressWindow::FreezeLine` should be called. To unfreeze it thereafter the procedure `GMP::ProgressWindow::UnfreezeLine` should be called.

See also:

The routines `GMP::ProgressWindow::DisplaySolverStatus`, `GMP::ProgressWindow::DisplayProgramStatus`, `GMP::ProgressWindow::DisplaySolver`, `GMP::ProgressWindow::FreezeLine`, `GMP::ProgressWindow::UnfreezeLine` and `GMP::SolverSession::CreateProgressCategory`.

GMP::ProgressWindow::DisplayProgramStatus

The procedure `GMP::ProgressWindow::DisplayProgramStatus` 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`.

Category
An element in the set `AllProgressCategories`.

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.

Remarks:

- If the *Category* argument is used then the element should be created with the function `GMP::SolverSession::CreateProgressCategory`.
- The program status can also be displayed by using the procedure `GMP::ProgressWindow::DisplayLine` with title 'Program Status'.

See also:

The routines `GMP::Solution::GetProgramStatus`, `GMP::ProgressWindow::DisplayLine`, `GMP::ProgressWindow::DisplaySolverStatus` and `GMP::SolverSession::CreateProgressCategory`.

GMP::ProgressWindow::DisplaySolver

The procedure `GMP::ProgressWindow::DisplaySolver` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

If the *Category* argument is used then the element should be created with the function `GMP::SolverSession::CreateProgressCategory`.

See also:

The routines `GMP::ProgressWindow::DisplaySolverStatus`, `GMP::ProgressWindow::DisplayProgramStatus`, `GMP::ProgressWindow::DisplayLine` and `GMP::SolverSession::CreateProgressCategory`.

GMP::ProgressWindow::DisplaySolverStatus

The procedure `GMP::ProgressWindow::DisplaySolverStatus` 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`.

Category
: An element in the set `AllProgressCategories`.

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.

Remarks:

- If the *Category* argument is used then the element should be created with the function `GMP::SolverSession::CreateProgressCategory`.
- The solver status can also be displayed by using the procedure `GMP::ProgressWindow::DisplayLine` with title 'Solver Status'.

See also:

The routines `GMP::Solution::GetSolverStatus`, `GMP::ProgressWindow::DisplayLine`, `GMP::ProgressWindow::DisplayProgramStatus` and `GMP::SolverSession::CreateProgressCategory`.

GMP::ProgressWindow::FreezeLine

The procedure GMP::ProgressWindow::FreezeLine 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 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.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- If the *Category* argument is used then the element should be created with the function GMP::SolverSession::CreateProgressCategory.
- 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 was called.

See also:

The procedures GMP::Instance::CreateProgressCategory, GMP::ProgressWindow::DisplayLine, GMP::ProgressWindow::DisplayProgramStatus, GMP::ProgressWindow::DisplaySolverStatus, GMP::ProgressWindow::UnfreezeLine and GMP::SolverSession::CreateProgressCategory.

GMP::ProgressWindow::Transfer

The procedure `GMP::ProgressWindow::Transfer` 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`.

solverSession
An element in the set `AllSolverSessions`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The *Category* should have been created with the function `GMP::SolverSession::CreateProgressCategory`.
- The *solverSession* argument specifies the solver session to which the progress category should be transfered.

Examples:

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`.

GMP::ProgressWindow::UnfreezeLine

The procedure `GMP::ProgressWindow::UnfreezeLine` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- If the *Category* argument is used then the element should be created with the function `GMP::SolverSession::CreateProgressCategory`.
- 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` was called.

See also:

The procedures `GMP::Instance::CreateProgressCategory`, `GMP::ProgressWindow::DisplayLine`, `GMP::ProgressWindow::FreezeLine` and `GMP::SolverSession::CreateProgressCategory`.

12.8 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`
- `GMP::QuadraticCoefficient::Set`

GMP::QuadraticCoefficient::Get

The function `GMP::QuadraticCoefficient::Get` 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`.

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.

Remarks:

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`, `GMP::Coefficient::GetQuadratic` and `GMP::Coefficient::SetQuadratic`.

GMP::QuadraticCoefficient::Set

The procedure `GMP::QuadraticCoefficient::Set` 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`.

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.

Remarks:

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`, `GMP::Coefficient::GetQuadratic` and `GMP::Coefficient::SetQuadratic`.

12.9 GMP::Robust Procedures and Functions

Aimms supports the following procedures and functions related to robust optimization:

- GMP::Robust::EvaluateAdjustableVariables

GMP::Robust::EvaluateAdjustableVariables

The procedure `GMP::Robust::EvaluateAdjustableVariables` 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`.

Variables
: A subset of `AllVariables`.

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.

Remarks:

- The *GMP* must have been created using the procedure `GMP::Instance::GenerateRobustCounterpart`.
- 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`.

Examples:

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`.

12.10 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
- GMP::Row::Add
- GMP::Row::Deactivate
- GMP::Row::Delete
- GMP::Row::DeleteIndicatorCondition
- GMP::Row::Generate
- GMP::Row::GetConvex
- GMP::Row::GetIndicatorColumn
- GMP::Row::GetIndicatorCondition
- GMP::Row::GetLeftHandSide
- GMP::Row::GetName
- GMP::Row::GetRelaxationOnly
- GMP::Row::GetRightHandSide
- GMP::Row::GetScale
- GMP::Row::GetStatus
- GMP::Row::GetType
- GMP::Row::SetConvex
- GMP::Row::SetIndicatorCondition
- GMP::Row::SetLeftHandSide
- GMP::Row::SetPoolType
- GMP::Row::SetPoolTypeMulti
- GMP::Row::SetRelaxationOnly
- GMP::Row::SetRightHandSide
- GMP::Row::SetRightHandSideMulti
- GMP::Row::SetType

GMP::Row::Activate

The procedure GMP::Row::Activate 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.

row
: A scalar reference to an existing 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.

Return value:

The procedure returns 1 on success, and 0 otherwise.

See also:

The routines GMP::Instance::Generate and GMP::Row::Deactivate.

GMP::Row::Add

The procedure `GMP::Row::Add` 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`.

row
: A scalar reference to a row.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- Coefficients for this row can be added to the matrix by using the procedure `GMP::Coefficient::Set`.
- After calling `GMP::Row::Add` the type and the left-hand-side and right-hand-side values are set according to the definition of the corresponding symbolic constraint. By using the procedures `GMP::Row::SetType`, `GMP::Row::SetLeftHandSide` and `GMP::Row::SetRightHandSide` the row type and row bounds can be changed.
- Use procedure `GMP::Row::Generate` to generate a (non-empty) row according to the definition of its associated symbolic constraint.

See also:

The routines `GMP::Instance::Generate`, `GMP::Coefficient::Set`, `GMP::Row::Delete`, `GMP::Row::SetType`, `GMP::Row::SetLeftHandSide`, `GMP::Row::SetRightHandSide` and `GMP::Row::Generate`.

GMP::Row::Deactivate

The procedure `GMP::Row::Deactivate` 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`.

row
: A scalar reference to an existing 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.

Return value:

The procedure returns 1 on success, and 0 otherwise.

See also:

The routines `GMP::Instance::Generate` and `GMP::Row::Activate`.

GMP::Row::Delete

The procedure `GMP::Row::Delete` marks a row in the matrix of 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`.

row
: A scalar reference to an existing 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.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- A deleted row remains present in the generated mathematical program but its contents 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` and `GMP::Row::Add`.

GMP::Row::DeleteIndicatorCondition

The procedure `GMP::Row::DeleteIndicatorCondition` 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`.

row
: A scalar reference to an existing 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.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

This procedure transforms an indicator row into a normal row.

See also:

The routines `GMP::Row::GetIndicatorColumn`, `GMP::Row::GetIndicatorCondition` and `GMP::Row::SetIndicatorCondition`.

GMP::Row::Generate

The procedure `GMP::Row::Generate` 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`.

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.

Remarks:

- 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`.
- 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.

Examples:

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, GMP::Column::Add, GMP::Column::Delete, GMP::Row::Add and GMP::Row::Delete.

GMP::Row::GetConvex

The function GMP::Row::GetConvex 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.

row
A scalar reference to an existing 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.

Return value:

The function returns 1 if the row is convex, and 0 otherwise.

Remarks:

AIMMS cannot detect whether a row is convex or not. A row is marked as being convex if the procedure GMP::Row::SetConvex 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. The Convex suffix is explained in full detail in Section 14.2.6 of the Language Reference.

GMP::Row::GetIndicatorColumn

The function GMP::Row::GetIndicatorColumn 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.

row
: A scalar reference to an existing 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.

Return value:

The function returns the column number if the indicator column exists, and -1 otherwise.

See also:

The routines GMP::Row::DeleteIndicatorCondition, GMP::Row::GetIndicatorCondition and GMP::Row::SetIndicatorCondition.

GMP::Row::GetIndicatorCondition

The function `GMP::Row::GetIndicatorCondition` 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`.

row
A scalar reference to an existing row in the matrix.

Return value:

The function returns the indicator condition.

Remarks:

This function fails if the row has no indicator column.

See also:

The routines `GMP::Row::DeleteIndicatorCondition`, `GMP::Row::GetIndicatorColumn` and `GMP::Row::SetIndicatorCondition`.

GMP::Row::GetLeftHandSide

The function `GMP::Row::GetLeftHandSide` 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` instead.

```
GMP::Row::GetLeftHandSide(
     GMP,            ! (input) a generated mathematical program
     row             ! (input) a scalar reference or row number
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing 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.

Return value:

The function returns the left-hand-side value of the specified row.

Remarks:

If the row has a unit then the scaled left-hand-side value is returned (without unit).

Examples:

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` 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`, `GMP::Row::SetLeftHandSide`, `GMP::Row::GetRightHandSide`, `GMP::Row::GetScale` and `GMP::Solution::GetRowValue`.

GMP::Row::GetName

The function GMP::Row::GetName 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.

row
: A scalar reference to an existing 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.

Return value:

The function returns a string.

See also:

The routines GMP::Instance::Generate and GMP::Column::GetName.

GMP::Row::GetRelaxationOnly

The function `GMP::Row::GetRelaxationOnly` 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`.

row
: A scalar reference to an existing 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.

Return value:

The function returns 1 if the row is a relaxation-only row, and 0 otherwise.

Remarks:

A row is marked as being a relaxation-only row if the procedure `GMP::Row::SetRelaxationOnly` 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`. The `RelaxationOnly` suffix is explained in full detail in Section 14.2.6 of the Language Reference.

GMP::Row::GetRightHandSide

The function `GMP::Row::GetRightHandSide` 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`.

row
: A scalar reference to an existing 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.

Return value:

The function returns the right-hand-side value of the specified row.

Remarks:

If the row has a unit then the scaled right-hand-side value is returned (without unit).

Examples:

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;
}
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` 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, GMP::Row::SetRightHandSide, GMP::Row::GetLeftHandSide and GMP::Row::GetScale.

GMP::Row::GetScale

The function `GMP::Row::GetScale` 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`.

row
: A scalar reference to an existing 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.

Return value:

The scaling factor for the specified row.

See also:

The routines `GMP::Instance::Generate` and `GMP::Column::GetScale`.

GMP::Row::GetStatus

The function GMP::Row::GetStatus 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.

row

A scalar reference to an existing 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.

Return value:

An element in the predefined set AllRowColumnStatuses. The set AllRowColumnStatuses contains the following elements:

- Active,
- Deactivated,
- Deleted,
- NotGenerated,
- PresolveDeleted.

Remarks:

This function will return 'PresolveDeleted' only if the generated mathematical program has been created with GMP::Instance::CreatePresolved. 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 and GMP::Instance::CreatePresolved.

GMP::Row::GetType

The function GMP::Row::GetType 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.

row
A scalar reference to an existing 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.

Return value:

The function returns an element in the predefined set AllRowTypes.

See also:

The routines GMP::Instance::Generate and GMP::Row::SetType.

GMP::Row::SetConvex

The procedure `GMP::Row::SetConvex` 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`.

row
: A scalar reference to an existing 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.

value
: A scalar reference to a 0-1 value.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

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`. The `Convex` suffix is explained in full detail in Section 14.2.6 of the Language Reference.

GMP::Row::SetIndicatorCondition

The procedure `GMP::Row::SetIndicatorCondition` 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`.

row
: A scalar reference to an existing 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.

column
: A scalar reference to an existing column in the matrix or the number of that column 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.

Remarks:

- 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`, `GMP::Row::GetIndicatorColumn` and `GMP::Row::GetIndicatorCondition`.

GMP::Row::SetLeftHandSide

The procedure GMP::Row::SetLeftHandSide 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.

row
: A scalar reference to an existing 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.

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.

Remarks:

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.

Examples:

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, GMP::Row::SetRightHandSide, GMP::Row::GetLeftHandSide and GMP::Row::GetScale.

GMP::Row::SetPoolType

The procedure `GMP::Row::SetPoolType` 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`.

row
: A scalar reference to an existing 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.

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.

Remarks:

- 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` 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 7.0 or higher 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`. The lazy constraint pool and the cut pool are explained in full detail in Section 14.2.4 of the Language Reference.

GMP::Row::SetPoolTypeMulti

The procedure `GMP::Row::SetPoolTypeMulti` 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 scalar reference or row number
     value,          ! (input) a scalar reference
     mode            ! (input) a scalar reference
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

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 `binding`. 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 `binding`. 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.

Remarks:

- 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 7.0 or higher 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`. The lazy constraint pool and the cut pool are explained in full detail in Section 14.2.4 of the Language Reference.

GMP::Row::SetRelaxationOnly

The procedure `GMP::Row::SetRelaxationOnly` 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`.

row
: A scalar reference to an existing 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.

value
: A scalar reference to a 0-1 value.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

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`. The `RelaxationOnly` suffix is explained in full detail in Section 14.2.6 of the Language Reference.

GMP::Row::SetRightHandSide

The procedure `GMP::Row::SetRightHandSide` 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`.

row
: A scalar reference to an existing 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.

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.

Remarks:

- Use `GMP::Row::SetRightHandSideMulti` if the right-hand-side of many rows corresponding to some constraint 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`.

Examples:

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;

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`, `GMP::Row::SetRightHandSideMulti`, `GMP::Row::SetLeftHandSide`, `GMP::Row::GetRightHandSide` and `GMP::Row::GetScale`.

GMP::Row::SetRightHandSideMulti

The procedure `GMP::Row::SetRightHandSideMulti` changes the right-hand-side of a group of row, 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`.

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 `binding`.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

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`. See `GMP::Row::SetRightHandSide` for an example with units.

Examples:

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`, `GMP::Row::SetRightHandSide`, `GMP::Row::SetLeftHandSide`, `GMP::Row::GetRightHandSide` and `GMP::Row::GetScale`.

GMP::Row::SetType

The procedure `GMP::Row::SetType` changes the type of a row in the matrix of a generated mathematical program.

```
GMP::Row::SetType(
     GMP,            ! (input) a generated mathematical program
     row,            ! (input) a scalar reference or row number
     type            ! (input) a element in AllRowTypes
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

row
: A scalar reference to an existing 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.

type
: An element in `AllRowTypes`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Instance::Generate` and `GMP::Row::GetType`.

12.11 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
- GMP::Solution::ConstraintListing
- GMP::Solution::ConstructMean
- GMP::Solution::Copy
- GMP::Solution::Count
- GMP::Solution::Delete
- GMP::Solution::DeleteAll
- GMP::Solution::GetBestBound
- GMP::Solution::GetColumnValue
- GMP::Solution::GetDistance
- GMP::Solution::GetFirstOrderDerivative
- GMP::Solution::GetIterationsUsed
- GMP::Solution::GetMemoryUsed
- GMP::Solution::GetNodesUsed
- GMP::Solution::GetObjective
- GMP::Solution::GetPenalizedObjective
- GMP::Solution::GetProgramStatus
- GMP::Solution::GetRowValue
- GMP::Solution::GetSolutionsSet
- GMP::Solution::GetSolverStatus
- GMP::Solution::GetTimeUsed
- GMP::Solution::IsDualDegenerated
- GMP::Solution::IsInteger
- GMP::Solution::IsPrimalDegenerated
- GMP::Solution::Move
- GMP::Solution::RandomlyGenerate
- GMP::Solution::RetrieveFromModel
- GMP::Solution::RetrieveFromSolverSession
- GMP::Solution::SendToModel
- GMP::Solution::SendToModelSelection
- GMP::Solution::SendToSolverSession
- GMP::Solution::SetColumnValue
- GMP::Solution::SetIterationCount
- GMP::Solution::SetMIPStartFlag
- GMP::Solution::SetObjective
- GMP::Solution::SetProgramStatus
- GMP::Solution::SetRowValue
- GMP::Solution::SetSolverStatus
- GMP::Solution::UpdatePenaltyWeights

See also the section on Managing the solution repository, Section 16.4 of the Language Reference.

GMP::Solution::Check

The procedure GMP::Solution::Check 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
     maxInfeas,      ! (output) maximum infeasibility
     [skipObj]       ! (optional, default 0) a scalar value
     )
```

Arguments:

GMP
: An element in the set AllGeneratedMathematicalPrograms.

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).

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

The option Constraint Listing Feasibility Tolerance determines the feasibility tolerance used by this procedure. If a constraint violation is smaller than this tolerance then it will be ignored.

See also:

The routines GMP::Instance::Generate, GMP::Solution::RetrieveFromModel and GMP::Solution::RetrieveFromSolverSession.

GMP::Solution::ConstraintListing

The procedure `GMP::Solution::ConstraintListing` 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`.

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` 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`.
- The actual call to `GMP::Solution::ConstraintListing` 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` 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 Section 22.2.1, 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.

Remarks:

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`.

GMP::Solution::ConstructMean

The procedure `GMP::Solution::ConstructMean` 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`.

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.

Remarks:

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` 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`.

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` and `GMP::Solution::Move`.

GMP::Solution::Count

The function `GMP::Solution::Count` 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`.

Return value:

The number of non-empty solutions stored in the solution repository.

Remarks:

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` and `GMP::Solution::GetSolutionsSet`.

GMP::Solution::Delete

The procedure `GMP::Solution::Delete` 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`.

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` and `GMP::Solution::DeleteAll`.

GMP::Solution::DeleteAll

The procedure `GMP::Solution::DeleteAll` empties the solution repository of a generated mathematical program.

```
GMP::Solution::DeleteAll(
     GMP            ! (input) a generated mathematical program
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Instance::Generate` and `GMP::Solution::Delete`.

GMP::Solution::GetBestBound

The function `GMP::Solution::GetBestBound` 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`.

solution
An integer scalar reference to a solution.

Return value:

In case of success, the best known bound. Otherwise it returns `UNDF`.

Remarks:

- This function has only meaning for a generated mathematical program with model type MIP, MIQP or MIQCP.
-

See also:

The procedure `GMP::Solution::GetObjective`.

GMP::Solution::GetColumnValue

The function `GMP::Solution::GetColumnValue` 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`.

solution
: An integer scalar reference to a solution.

column
: A scalar reference to an existing column in the matrix or the number of that column 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.

Return value:

The level value or reduced cost of the column.

Remarks:

- 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.
- 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`.

See also:

The routines `GMP::Column::GetScale`, `GMP::Instance::Generate`, `GMP::Solution::GetRowValue` and `GMP::Solution::SetColumnValue`.

GMP::Solution::GetDistance

The function GMP::Solution::GetDistance 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.

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.

Remarks:

The level value of the objective column (if any) is not used.

GMP::Solution::GetFirstOrderDerivative

The function `GMP::Solution::GetFirstOrderDerivative` 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`.

solution
: An integer scalar reference to a solution.

row
: A scalar reference to an existing 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.

column
: A scalar reference to an existing column in the matrix 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 first order derivative of the column in the row.

Remarks:

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`.

GMP::Solution::GetIterationsUsed

The function `GMP::Solution::GetIterationsUsed` 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`.

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` and `GMP::Solution::SetIterationCount`.

GMP::Solution::GetMemoryUsed

The function `GMP::Solution::GetMemoryUsed` 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`.

solution
An integer scalar reference to a solution.

Return value:

The amount of megabytes used to create a solution.

See also:

The procedure `GMP::Instance::SetMemoryLimit`.

GMP::Solution::GetNodesUsed

The function `GMP::Solution::GetNodesUsed` 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`.

solution
: An integer scalar reference to a solution.

Return value:

The number of nodes used to create a solution.

Remarks:

- 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`, `GMP::Instance::SetCallbackCandidate` and `GMP::Instance::SetCallbackHeuristic`.

GMP::Solution::GetObjective

The function `GMP::Solution::GetObjective` 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`.

solution
: An integer scalar reference to a solution.

Return value:

The objective function value of the solution.

Remarks:

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.

See also:

The routines `GMP::Instance::Generate`, `GMP::Solution::GetProgramStatus`, `GMP::Solution::GetSolverStatus` and `GMP::Solution::SetObjective`.

GMP::Solution::GetPenalizedObjective

The function `GMP::Solution::GetPenalizedObjective` 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`.

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).

Remarks:

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

$$P(x,w) = \frac{f(x) + dirval * \sum_{i=1}^{m} (w_i * viol(g_i(x)))}{m^2},$$

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`.

GMP::Solution::GetProgramStatus

The function `GMP::Solution::GetProgramStatus` 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`.

solution
An integer scalar reference to a solution.

Return value:

An element in the set `AllSolutionStates`.

Remarks:

The program status is only available if the solution has been retrieved from the solver, or if the procedure `GMP::Solution::SetProgramStatus` has been called before.

See also:

The routines `GMP::Instance::Generate`, `GMP::Solution::GetSolverStatus`, `GMP::Solution::GetObjective` and `GMP::Solution::SetProgramStatus`.

GMP::Solution::GetRowValue

The function GMP::Solution::GetRowValue 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.

solution
An integer scalar reference to a solution.

row
A scalar reference to an existing 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.

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.

Return value:

The level value or shadow price of the row.

Remarks:

- 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 **ShadowPrice** property of the corresponding constraint should be set.
- 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.

See also:

The routines GMP::Instance::Generate, GMP::Row::GetScale, GMP::Solution::GetColumnValue and GMP::Solution::SetRowValue.

GMP::Solution::GetSolutionsSet

The function `GMP::Solution::GetSolutionsSet` 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`.

Return value:

A subset of `Integers`.

See also:

The functions `GMP::Instance::Generate` and `GMP::Solution::Count` and the section on Managing the solution repository Section 16.4 of the Language Reference.

GMP::Solution::GetSolverStatus

The function `GMP::Solution::GetSolverStatus` 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`.

solution
An integer scalar reference to a solution.

Return value:

An element in the set `AllSolutionStates`.

Remarks:

The solver status is only available if the solution has been retrieved from the solver, or if the procedure `GMP::Solution::SetSolverStatus` has been called before.

See also:

The routines `GMP::Instance::Generate`, `GMP::Solution::GetProgramStatus` and `GMP::Solution::GetObjective` and `GMP::Solution::SetSolverStatus`.

GMP::Solution::GetTimeUsed

The function GMP::Solution::GetTimeUsed 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.

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.

GMP::Solution::IsDualDegenerated

The function `GMP::Solution::IsDualDegenerated` 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`.

solution
: An integer scalar reference to a solution.

Return value:

The function returns 1 if the solution is dual degenerated, and 0 otherwise.

Remarks:

- 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`, `GMP::Solution::IsPrimalDegenerated` and `GMP::Solution::RetrieveFromSolverSession`.

GMP::Solution::IsInteger

The function `GMP::Solution::IsInteger` 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`.

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.

Remarks:

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`, `GMP::Solution::RetrieveFromModel` and `GMP::Solution::RetrieveFromSolverSession`.

GMP::Solution::IsPrimalDegenerated

The function `GMP::Solution::IsPrimalDegenerated` 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`.

solution
An integer scalar reference to a solution.

Return value:

The function returns 1 if the solution is primal degenerated, and 0 otherwise.

Remarks:

- 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`, `GMP::Solution::IsDualDegenerated` and `GMP::Solution::RetrieveFromSolverSession`.

GMP::Solution::Move

The procedure `GMP::Solution::Move` 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`.

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.

Remarks:

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`.

See also:

The routines `GMP::Instance::Generate` and `GMP::Solution::Copy`.

GMP::Solution::RandomlyGenerate

The procedure `GMP::Solution::RandomlyGenerate` 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
     [perturbation]  ! (optional) a scalar value
     )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

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.

Remarks:

- This procedure should be called after calling the function `GMP::Instance::CreatePresolved` 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

$$[x - p * (x - lb), x + p * (ub - x)]$$

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`.

GMP::Solution::RetrieveFromModel

The procedure `GMP::Solution::RetrieveFromModel` 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`.

solution
: An integer scalar reference to a solution.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

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`, `GMP::Solution::SendToModel`, `GMP::Solution::RetrieveFromSolverSession` and `GMP::Solution::SendToSolverSession`.

GMP::Solution::RetrieveFromSolverSession

The procedure `GMP::Solution::RetrieveFromSolverSession` 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`.

solution
An integer scalar reference to a solution.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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` 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` the incumbent solution can be rejected inside a **candidate** callback.

See also:

The routines `GMP::Instance::Generate`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic`, `GMP::Solution::SendToSolverSession`, `GMP::Solution::RetrieveFromModel`, `GMP::Solution::SendToModel`, `GMP::SolverSession::GetNodeObjective` and `GMP::SolverSession::RejectIncumbent`.

GMP::Solution::SendToModel

The procedure `GMP::Solution::SendToModel` 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
     )
```

Arguments:

GMP
An element in `AllGeneratedMathematicalPrograms`.

solution
An integer scalar reference to a solution.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

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`, `GMP::Solution::RetrieveFromModel`, `GMP::Solution::RetrieveFromSolverSession` and `GMP::Solution::SendToSolverSession`.

GMP::Solution::SendToModelSelection

The procedure `GMP::Solution::SendToModelSelection` 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
        )
```

Arguments:

GMP
: An element in `AllGeneratedMathematicalPrograms`.

solution
: An integer scalar reference to a solution.

Identifiers
: A subset of the predefined set `AllVariablesConstraints`, 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`, containing the set of suffixes for which the values of *Identifiers* have to be changed into those of *solution*.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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.

See also:

The routines `GMP::Instance::Generate`, `GMP::Solution::RetrieveFromModel`, `GMP::Solution::RetrieveFromSolverSession`, `GMP::Solution::SendToSolverSession` and `GMP::Solution::SendToModel`

GMP::Solution::SendToSolverSession

The procedure `GMP::Solution::SendToSolverSession` 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`.

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`, `GMP::Solution::RetrieveFromSolverSession`, `GMP::Solution::RetrieveFromModel` and `GMP::Solution::SendToModel`.

GMP::Solution::SetColumnValue

The procedure `GMP::Solution::SetColumnValue` 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`.

solution
: An integer scalar reference to a solution.

column
: A scalar reference to an existing column in the matrix or the number of that column 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.

Remarks:

- 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`.
- Hint values and priorities are only supported by GUROBI 6.5 or higher.

Examples:

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` as follows. Here `ColumnNrs` is a subset of `Integers` 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`, `GMP::Instance::Generate`, `GMP::Instance::GetColumnNumbers`, `GMP::Solution::GetColumnValue`, `GMP::Solution::SendToModel` and `GMP::Solution::SetRowValue`.

GMP::Solution::SetIterationCount

The procedure `GMP::Solution::SetIterationCount` 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`.

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`, `GMP::SolverSession::GetIterationsUsed` and `GMP::Solution::SetProgramStatus`.

GMP::Solution::SetMIPStartFlag

The procedure `GMP::Solution::SetMIPStartFlag` 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`.

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.

Remarks:

- 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.

- Level 5 is only supported by CPLEX 12.7 or higher (for other solver versions it is translated to 0).

See also:

The routines `GMP::Instance::Generate`.

GMP::Solution::SetObjective

The procedure `GMP::Solution::SetObjective` 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`.

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`, `GMP::Solution::GetObjective` and `GMP::Solution::SendToModel`.

GMP::Solution::SetProgramStatus

The procedure `GMP::Solution::SetProgramStatus` 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`.

solution
An integer scalar reference to a solution.

status
An element in the set `AllSolutionStates`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Instance::Generate`, `GMP::Solution::GetProgramStatus` and `GMP::Solution::SetSolverStatus`.

GMP::Solution::SetRowValue

The procedure GMP::Solution::SetRowValue 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.

solution
: An integer scalar reference to a solution.

row
: A scalar reference to an existing row in the matrix or the number of that row in the range $\{0..n-1\}$ where n 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.

Remarks:

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.

Examples:

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 as follows. Here RowNrs is a subset of Integers 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;

! 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`, `GMP::Instance::GetRowNumbers`, `GMP::Row::GetScale`, `GMP::Solution::GetRowValue`, `GMP::Solution::SendToModel` and `GMP::Solution::SetColumnValue`.

GMP::Solution::SetSolverStatus

The procedure `GMP::Solution::SetSolverStatus` 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`.

solution
An integer scalar reference to a solution.

status
An element in the set `AllSolutionStates`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

See also:

The routines `GMP::Instance::Generate`, `GMP::Solution::GetSolverStatus` and `GMP::Solution::SetProgramStatus`.

GMP::Solution::UpdatePenaltyWeights

The procedure `GMP::Solution::UpdatePenaltyWeights` 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`.

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.

Remarks:

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`.

12.12 GMP::Solver Procedures and Functions

AIMMS supports the following procedures and functions for retrieving solver related information, and managing solver environments:

- `GMP::Solver::FreeEnvironment`
- `GMP::Solver::GetAsynchronousSessionsLimit`
- `GMP::Solver::InitializeEnvironment`

GMP::Solver::FreeEnvironment

The procedure GMP::Solver::FreeEnvironment can be used to free a solver environment. By using the procedure GMP::Solver::InitializeEnvironment 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 prococedures GMP::Solver::InitializeEnvironment and GMP::Solver::FreeEnvironment 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.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- This procedure can be used in combination with a normal solve statement.
- This procedure is only supported by GUROBI 7.0 or higher.
- This procedure cannot be called inside a solver callback procedure.
- This procedure cannot be called if one of the solver sessions is asynchronous executing.

Examples:

```
GMP::Solver::InitializeEnvironment( 'Gurobi 7.5' );

solve MP1;

GMP::Solver::FreeEnvironment( 'Gurobi 7.5' );

GMP::Solver::InitializeEnvironment( 'Gurobi 7.5' );

mgGMP := GMP::Instance::Generate( MP2 );
GMP::Instance::Solve( myGMP );

GMP::Solver::FreeEnvironment( 'Gurobi 7.5' );
```

See also:

The procedure `GMP::Solver::InitializeEnvironment`.

GMP::Solver::GetAsynchronousSessionsLimit

The function GMP::Solver::GetAsynchronousSessionsLimit 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.

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. 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).

Remarks:

- 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.
- 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 12.9 and another simultaneous with CPLEX 12.8 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).

Examples:

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 12.9' );
```

The value MaxSes is the limit on asynchronous solver sessions that can run at the same time with CPLEX 12.9 plus CPLEX 12.8 plus CPLEX 12.7, etc.

See also:

The routine GMP::SolverSession::AsynchronousExecute.

GMP::Solver::InitializeEnvironment

The procedure `GMP::Solver::InitializeEnvironment` can be used to initialize a solver environment. By using the procedure `GMP::Solver::FreeEnvironment` 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 prococdures `GMP::Solver::InitializeEnvironment` and `GMP::Solver::FreeEnvironment` 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::InitializeEnvironment(
     solver,            ! (input) a solver
     [computeserver],   ! (input, optional) a string expression
     [port],            ! (input, optional) integer, default -1
     [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`.

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.

port
: The port number used to connect to the compute server. Use the default value of -1, which indicates that the default port should be used, unless your server administrator has changed the recommended port settings.

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 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.

Remarks:

- 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 7.0 or higher.
- If the *computeserver* argument is not specified then the compute server must be specified via a Gurobi client license key file.
- The optional arguments *port*, *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.
- Technical note: if the optional argument *computeserver* is specified then the Gurobi function `GRBloadclientenv` is called underneath, otherwise the Gurobi function `GRBloadenv` (if the AIMMS license features the Gurobi Link-only).

Examples:

```
GMP::Solver::InitializeEnvironment( 'Gurobi 7.5' );

solve MP1;

GMP::Solver::FreeEnvironment( 'Gurobi 7.5' );

GMP::Solver::InitializeEnvironment( 'Gurobi 7.5', computeserver: "my.server.com",
                                    priority: 10 );

mgGMP := GMP::Instance::Generate( MP2 );
GMP::Instance::Solve( myGMP );

GMP::Solver::FreeEnvironment( 'Gurobi 7.5' );
```

See also:

The procedure `GMP::Solver::FreeEnvironment`.

12.13 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
- GMP::SolverSession::AddBendersOptimalityCut
- GMP::SolverSession::AddLinearization
- GMP::SolverSession::AsynchronousExecute
- GMP::SolverSession::CreateProgressCategory
- GMP::SolverSession::Execute
- GMP::SolverSession::ExecutionStatus
- GMP::SolverSession::GenerateBinaryEliminationRow
- GMP::SolverSession::GenerateBranchLowerBound
- GMP::SolverSession::GenerateBranchRow
- GMP::SolverSession::GenerateBranchUpperBound
- GMP::SolverSession::GenerateCut
- GMP::SolverSession::GetBestBound
- GMP::SolverSession::GetCallbackInterruptStatus
- GMP::SolverSession::GetCandidateObjective
- GMP::SolverSession::GetInstance
- GMP::SolverSession::GetIterationsUsed
- GMP::SolverSession::GetMemoryUsed
- GMP::SolverSession::GetNodeNumber
- GMP::SolverSession::GetNodeObjective
- GMP::SolverSession::GetNodesLeft
- GMP::SolverSession::GetNodesUsed
- GMP::SolverSession::GetNumberOfBranchNodes
- GMP::SolverSession::GetObjective
- GMP::SolverSession::GetOptionValue
- GMP::SolverSession::GetProgramStatus
- GMP::SolverSession::GetSolver
- GMP::SolverSession::GetSolverStatus
- GMP::SolverSession::GetTimeUsed
- GMP::SolverSession::Interrupt
- GMP::SolverSession::RejectIncumbent
- GMP::SolverSession::SetObjective
- GMP::SolverSession::SetOptionValue
- GMP::SolverSession::Transfer
- GMP::SolverSession::WaitForCompletion
- GMP::SolverSession::WaitForSingleCompletion

GMP::SolverSession::AddBendersFeasibilityCut

The procedure `GMP::SolverSession::AddBendersFeasibilityCut` 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 & 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` representing a solver session for the Benders' master problem.

GMP
: An element in the set `AllGeneratedMathematicalPrograms` 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.

Remarks:

- The generated mathematical program corresponding to the *solverSession* should have been created using the function `GMP::Benders::CreateMasterProblem`.
- The *GMP* should have been created using the function `GMP::Benders::CreateSubProblem` or the function `GMP::Instance::CreateFeasibility`.
- If the function `GMP::Benders::CreateSubProblem` 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 Section 21.3 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.

Examples:

The way `GMP::Benders::AddFeasibilityCut` 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`. 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. 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, GMP::Benders::CreateSubProblem, GMP::Benders::AddFeasibilityCut, GMP::Benders::AddOptimalityCut, GMP::Instance::CreateFeasibility and GMP::SolverSession::AddBendersOptimalityCut.

GMP::SolverSession::AddBendersOptimalityCut

The procedure `GMP::SolverSession::AddBendersOptimalityCut` 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 & 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` representing a solver session for the Benders' master problem.

GMP
: An element in the set `AllGeneratedMathematicalPrograms` 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.

Remarks:

- The generated mathematical program corresponding to the *solverSession* should have been created using the function `GMP::Benders::CreateMasterProblem`.

- The *GMP* should have been created using the function `GMP::Benders::CreateSubProblem`.
- 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 Section 21.3 of the Language Reference for more information about the Benders' decomposition algorithm in which a single master MIP problem is solved.

Examples:

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`. 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`. 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`, `GMP::Benders::CreateSubProblem`, `GMP::Benders::AddFeasibilityCut`, `GMP::Benders::AddOptimalityCut` and `GMP::SolverSession::AddBendersFeasibilityCut`.

GMP::SolverSession::AddLinearization

The procedure `GMP::SolverSession::AddLinearization` 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 value
     [purgeable]        ! (optional, default 0) a scalar binary expression
     )
```

Arguments:

solverSession
: An element in the set `AllSolverSessions`.

GMP
: An element in `AllGeneratedMathematicalPrograms`.

solution
: An integer scalar reference to a solution in the solution repository of *GMP*.

constraintSet
: A subset of `AllNonLinearConstraints`.

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.

Remarks:

- 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.

$$g(x^0, y^0) + \nabla g(x^0, y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} \leq b_j$$

See also:

The routines `GMP::Linearization::Add`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint` and `GMP::SolverSession::GenerateCut`.

GMP::SolverSession::AsynchronousExecute

The procedure `GMP::SolverSession::AsynchronousExecute` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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` cannot be used by the following solvers: AOA, BARON, CBC, ODH-CPLEX, and PATH.
- Calling `GMP::SolverSession::AsynchronousExecute` inside a callback procedure is not allowed.
- The procedure `GMP::SolverSession::AsynchronousExecute` cannot be used if an external function is used in a constraint.
- The procedures `GMP::SolverSession::WaitForCompletion` and `GMP::SolverSession::WaitForSingleCompletion` 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` for that.

See also:

The routines `GMP::Instance::Copy`, `GMP::SolverSession::Execute`, `GMP::SolverSession::ExecutionStatus` `GMP::SolverSession::Interrupt`, `GMP::SolverSession::WaitForCompletion`, `GMP::SolverSession::WaitForSingleCompletion`, `GMP::Solution::ConstraintListing` and `GMP::Solver::GetAsynchronousSessionsLimit`.

GMP::SolverSession::CreateProgressCategory

The function `GMP::SolverSession::CreateProgressCategory` 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`, 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` 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`.

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`.

Remarks:

- 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`.
- The information displayed in the solver session progress window can be controlled by using the procedures `GMP::ProgressWindow::DisplayLine` and `GMP::ProgressWindow::FreezeLine`.
- A progress category created before for the solver session will be deleted.
- The procedure `GMP::ProgressWindow::Transfer` can be used to share a progress category among several solver sessions.

See also:

The routines `GMP::ProgressWindow::CreateProgressCategory`, `GMP::ProgressWindow::DeleteCategory`, `GMP::ProgressWindow::DisplayLine`, `GMP::ProgressWindow::FreezeLine`, `GMP::ProgressWindow::UnfreezeLine` and `GMP::ProgressWindow::Transfer`.

GMP::SolverSession::Execute

The procedure `GMP::SolverSession::Execute` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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` for that.

See also:

The routines `GMP::Instance::CreateSolverSession`, `GMP::Instance::Solve`, `GMP::SolverSession::AsynchronousExecute` and `GMP::Solution::ConstraintListing`.

GMP::SolverSession::ExecutionStatus

The function `GMP::SolverSession::ExecutionStatus` returns the execution status of a solver session.

```
GMP::SolverSession::ExecutionStatus(
        solverSession      ! (input) a solver session
        )
```

Arguments:

solverSession
An element in the set `AllSolverSessions`.

Return value:

An element in the set `AllExecutionStatuses`. This set contains the following elements:

- NotStarted,
- Pending,
- Running,
- Interrupted,
- Finished.

See also:

The routines `GMP::SolverSession::AsynchronousExecute`, `GMP::SolverSession::Interrupt`, `GMP::SolverSession::WaitForCompletion` and `GMP::SolverSession::WaitForSingleCompletion`.

GMP::SolverSession::GenerateBinaryEliminationRow

The procedure `GMP::SolverSession::GenerateBinaryEliminationRow` 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`.

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.

Remarks:

- 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` adds the following row:

$$\sum_{i \in S_{lo}} x_i - \sum_{i \in S_{up}} x_i \geq 1 - \sum_{i \in S_{up}} lev_i \tag{12.5}$$

  where S_{lo} defines the set of binary columns whose level values equals 0 and S_{up} the set of binary columns whose level values equals 1.

Examples:

The procedure `GMP::SolverSession::GenerateBinaryEliminationRow` 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` 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);

endif;
```

Here 'ThisSession' is an input argument of the callback procedure and a scalar element parameter into the set AllSolverSessions.

See also:

The routines GMP::Instance::AddIntegerEliminationRows, GMP::Instance::SetCallbackAddCut, GMP::Instance::SetCallbackBranch, GMP::Instance::SetCallbackAddLazyConstraint and GMP::SolverSession::GetNumberOfBranchNodes.

GMP::SolverSession::GenerateBranchLowerBound

The procedure GMP::SolverSession::GenerateBranchLowerBound specifies the lower bound change of a column in a branch to be taken from the current node during MIP branch & 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.

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.

Remarks:

- A branch can be specified by adding multiple bound changes and rows (with GMP::SolverSession::GenerateBranchRow) 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, GMP::SolverSession::GenerateBranchUpperBound and GMP::SolverSession::GenerateBranchRow.

GMP::SolverSession::GenerateBranchRow

The procedure `GMP::SolverSession::GenerateBranchRow` adds a row to a branch to be taken from the current node during MIP branch & 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`.

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.

Remarks:

- A branch can be specified by adding multiple rows and bound changes (with `GMP::SolverSession::GenerateBranchLowerBound` and `GMP::SolverSession::GenerateBranchUpperBound`) 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`, `GMP::SolverSession::GenerateBranchLowerBound` and `GMP::SolverSession::GenerateBranchUpperBound`.

GMP::SolverSession::GenerateBranchUpperBound

The procedure `GMP::SolverSession::GenerateBranchUpperBound` specifies the upper bound change of a column in a branch to be taken from the current node during MIP branch & 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`.

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.

Remarks:

- A branch can be specified by adding multiple bound changes and rows (with `GMP::SolverSession::GenerateBranchRow`) 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`, `GMP::SolverSession::GenerateBranchLowerBound` and `GMP::SolverSession::GenerateBranchRow`.

GMP::SolverSession::GenerateCut

The procedure `GMP::SolverSession::GenerateCut` adds a cut to the LP subproblem of the current node during MIP branch & 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`.

row
: A scalar reference to an existing 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.

Remarks:

- 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.

See also:

The procedures GMP::Instance::SetCallbackAddCut and GMP::Instance::SetCallbackAddLazyConstraint. See Section 16.2 of the Language Reference for more details on how to install a callback procedure to add cuts.

GMP::SolverSession::GetBestBound

The function `GMP::SolverSession::GetBestBound` 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`.

Return value:

In case of success, the best known bound. Otherwise it returns `UNDF`.

Remarks:

- This function has only meaning for solver sessions with a corresponding generated mathematical program that has model type MIP, MIQP or MIQCP.

See also:

The routines `GMP::SolverSession::Execute`, `GMP::SolverSession::GetObjective`, `GMP::SolverSession::GetIterationsUsed`, `GMP::SolverSession::GetMemoryUsed` and `GMP::SolverSession::GetTimeUsed`.

GMP::SolverSession::GetCallbackInterruptStatus

The function `GMP::SolverSession::GetCallbackInterruptStatus` 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`.

Return value:

An element in the set `AllSolutionStates`.

Remarks:

When the solver session has not yet been executed, the empty element will be returned.

See also:

The procedures `GMP::SolverSession::Execute`, `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackIncumbent`, `GMP::Instance::SetCallbackIterations`, `GMP::Instance::SetCallbackHeuristic`, `GMP::Instance::SetCallbackStatusChange` and `GMP::Instance::SetCallbackTime`.

GMP::SolverSession::GetCandidateObjective

The function `GMP::SolverSession::GetCandidateObjective` 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`.

Return value:

In case of success, the objective value at the current node. Otherwise it returns `UNDF`.

Remarks:

- 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` 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`, `GMP::Instance::SetCallbackCandidate` and `GMP::Solution::RetrieveFromSolverSession`.

GMP::SolverSession::GetInstance

The function `GMP::SolverSession::GetInstance` 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`.

Return value:

An element in the set `AllGeneratedMathematicalPrograms`.

See also:

The routines `GMP::Instance::Generate` and `GMP::Instance::CreateSolverSession`.

GMP::SolverSession::GetIterationsUsed

The function `GMP::SolverSession::GetIterationsUsed` 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`.

Return value:

The number of iterations used by the solver session.

See also:

The routines `GMP::SolverSession::Execute`, `GMP::Instance::SetIterationLimit`, `GMP::SolverSession::GetMemoryUsed` and `GMP::SolverSession::GetTimeUsed`.

GMP::SolverSession::GetMemoryUsed

The function `GMP::SolverSession::GetMemoryUsed` 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`.

Return value:

The amount of megabytes used to execute a solver session.

Remarks:

- 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.

See also:

The routines `GMP::Instance::SetCallbackIterations`, `GMP::Instance::SetCallbackTime`, `GMP::Instance::SetMemoryLimit`, `GMP::SolverSession::Execute`, `GMP::SolverSession::GetIterationsUsed` and `GMP::SolverSession::GetTimeUsed`.

GMP::SolverSession::GetNodeNumber

The function `GMP::SolverSession::GetNodeNumber` 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`.

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.

Remarks:

- 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`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic` and `GMP::SolverSession::GetNodesUsed`.

GMP::SolverSession::GetNodeObjective

The function `GMP::SolverSession::GetNodeObjective` 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`.

Return value:

In case of success, the objective value at the current node. Otherwise it returns `UNDF`.

Remarks:

- 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` 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 in CPLEX 12.8 or higher.

See also:

The routines `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackHeuristic`, `GMP::Solution::RetrieveFromSolverSession` and `GMP::SolverSession::GetNodeNumber`.

GMP::SolverSession::GetNodesLeft

The function `GMP::SolverSession::GetNodesLeft` 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`.

Return value:

The number of unexplored nodes left in the branch-and-bound tree.

Remarks:

- 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`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic`, `GMP::SolverSession::GetNodeNumber` and `GMP::SolverSession::GetNodesUsed`.

GMP::SolverSession::GetNodesUsed

The function `GMP::SolverSession::GetNodesUsed` 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`.

Return value:

The number of nodes that are processed by the solver session.

Remarks:

- 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`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic`, `GMP::SolverSession::GetNodeNumber` and `GMP::SolverSession::GetNodesLeft`.

GMP::SolverSession::GetNumberOfBranchNodes

The function `GMP::SolverSession::GetNumberOfBranchNodes` 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`.

Return value:

The number of nodes that the solver will create from the current branch.

Remarks:

- 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`.

GMP::SolverSession::GetObjective

The function `GMP::SolverSession::GetObjective` 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`.

Return value:

The objective function value associated with a solver session.

See also:

The routines `GMP::SolverSession::Execute`, `GMP::SolverSession::GetBestBound`, `GMP::SolverSession::GetIterationsUsed`, `GMP::SolverSession::GetMemoryUsed`, `GMP::SolverSession::GetTimeUsed` and `GMP::SolverSession::SetObjective`.

GMP::SolverSession::GetOptionValue

The function `GMP::SolverSession::GetOptionValue` 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`.

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`.

Remarks:

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`, `GMP::Instance::SetOptionValue`, `GMP::SolverSession::SetOptionValue`, `OptionGetString` and `OptionGetKeywords`.

GMP::SolverSession::GetProgramStatus

The function `GMP::SolverSession::GetProgramStatus` 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`.

Return value:

An element in the set `AllSolutionStates`.

See also:

The routines `GMP::SolverSession::Execute` and `GMP::SolverSession::GetSolverStatus`.

GMP::SolverSession::GetSolver

The function `GMP::SolverSession::GetSolver` returns the solver belonging to a solver session.

```
GMP::SolverSession::GetSolver(
     solverSession     ! (input) a solver session
     )
```

Arguments:

solverSession
: An element in the set `AllSolverSessions`.

Return value:

The solver belonging to a solver session as an element of `AllSolvers`.

Remarks:

Which solver is assigned to the solver session is determined by the routines `GMP::Instance::CreateSolverSession` and `GMP::Instance::SetSolver`. Note that if the *Solver* argument of `GMP::Instance::CreateSolverSession` is used then it overrules `GMP::Instance::SetSolver`.

See also:

The routines `GMP::Instance::CreateSolverSession` and `GMP::Instance::SetSolver`.

GMP::SolverSession::GetSolverStatus

The function `GMP::SolverSession::GetSolverStatus` 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`.

Return value:

An element in the set `AllSolutionStates`.

See also:

The routines `GMP::SolverSession::Execute` and `GMP::SolverSession::GetProgramStatus`.

GMP::SolverSession::GetTimeUsed

The function `GMP::SolverSession::GetTimeUsed` 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`.

Return value:

The number of 1/100th seconds used to execute a solver session.

See also:

The routines `GMP::Instance::SetTimeLimit`, `GMP::SolverSession::Execute`, `GMP::SolverSession::GetIterationsUsed` and `GMP::SolverSession::GetMemoryUsed`.

GMP::SolverSession::Interrupt

The procedure `GMP::SolverSession::Interrupt` 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`.

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.

Remarks:

- 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`, `GMP::SolverSession::ExecutionStatus`, `GMP::SolverSession::Interrupt`, `GMP::SolverSession::WaitForCompletion` and `GMP::SolverSession::WaitForSingleCompletion`.

GMP::SolverSession::RejectIncumbent

The procedure `GMP::SolverSession::RejectIncumbent` 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`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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`. See Section 16.2 of the Language Reference for more details on how to install a candidate callback procedure.

GMP::SolverSession::SetObjective

The procedure `GMP::SolverSession::SetObjective` 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`.

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` and `GMP::SolverSession::GetObjective`.

GMP::SolverSession::SetOptionValue

The procedure `GMP::SolverSession::SetOptionValue` 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`.

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.

Remarks:

- 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` if it is called for the solver session, else
  - the procedure `GMP::Instance::SetOptionValue` 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 Section 8.5 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, GMP::Instance::SetOptionValue, GMP::SolverSession::GetOptionValue, OptionGetString and OptionGetKeywords.

GMP::SolverSession::Transfer

The procedure `GMP::SolverSession::Transfer` 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`.

GMP
: An element in the set `AllGeneratedMathematicalPrograms`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

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`, `GMP::Instance::GenerateStochasticProgram` and `GMP::Stochastic::BendersFindReference`.

GMP::SolverSession::WaitForCompletion

The procedure GMP::SolverSession::WaitForCompletion 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.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

This procedure ignores solver sessions that are not asynchronous executing but using the procedure GMP::SolverSession::Execute.

See also:

The routines GMP::Event::Create, GMP::Event::Set, GMP::SolverSession::AsynchronousExecute, GMP::SolverSession::Execute, GMP::SolverSession::ExecutionStatus, GMP::SolverSession::Interrupt and GMP::SolverSession::WaitForSingleCompletion.

GMP::SolverSession::WaitForSingleCompletion

The routine `GMP::SolverSession::WaitForSingleCompletion` 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`.

Return value:

An element in the set `AllSolverSessionCompletionObjects`.

Remarks:

- This routine ignores solver sessions that are not asynchronous executing but using the procedure `GMP::SolverSession::Execute`.
- 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`, `GMP::Event::Set`, `GMP::SolverSession::AsynchronousExecute`, `GMP::SolverSession::Execute`, `GMP::SolverSession::ExecutionStatus`, `GMP::SolverSession::Interrupt` and `GMP::SolverSession::WaitForCompletion`.

12.14 GMP::Stochastic Procedures and Functions

AIMMS supports the following procedures and functions for creating and managing generated stochastic mathematical program instances:

- GMP::Stochastic::AddBendersFeasibilityCut
- GMP::Stochastic::AddBendersOptimalityCut
- GMP::Stochastic::BendersFindFeasibilityReference
- GMP::Stochastic::BendersFindReference
- GMP::Stochastic::CreateBendersRootproblem
- GMP::Stochastic::GetObjectiveBound
- GMP::Stochastic::GetRelativeWeight
- GMP::Stochastic::GetRepresentativeScenario
- GMP::Stochastic::MergeSolution
- GMP::Stochastic::UpdateBendersSubproblem

GMP::Stochastic::AddBendersFeasibilityCut

The procedure `GMP::Stochastic::AddBendersFeasibilityCut` 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`.

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..

Remarks:

- 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`. 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` created by this procedure using *cutNo*.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::AddBendersOptimalityCut`, `GMP::Stochastic::CreateBendersFeasibilitySubproblem` and `GMP::Stochastic::BendersFindReference`.

GMP::Stochastic::AddBendersOptimalityCut

The procedure `GMP::Stochastic::AddBendersOptimalityCut` 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`.

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..

Remarks:

- The *GMP* should have been created by the function `GMP::Stochastic::BendersFindReference`.
- 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`. 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` 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`, `GMP::Stochastic::AddBendersFeasibilityCut`, `GMP::Stochastic::BendersFindReference`,

GMP::Stochastic::GetObjectiveBound and
GMP::Stochastic::GetRelativeWeight.

GMP::Stochastic::BendersFindFeasibilityReference

The function `GMP::Stochastic::BendersFindFeasibilityReference` 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`.

stage
: An integer scalar reference to a stage.

scenario
: An element in the set `AllStochasticScenarios`.

Return value:

An element in the set `AllGeneratedMathematicalPrograms`.

Remarks:

- The function `GMP::Stochastic::CreateBendersRootproblem` creates all Benders feasibility problems for all nodes in the scenario tree, and must be called before calling `GMP::Stochastic::BendersFindReference`.
- The *GMP* should correspond to a root node, i.e., be created by using the function `GMP::Stochastic::CreateBendersRootproblem`.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::BendersFindReference` and `GMP::Stochastic::CreateBendersRootproblem`.

GMP::Stochastic::BendersFindReference

The function `GMP::Stochastic::BendersFindReference` 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`.

stage
An integer scalar reference to a stage.

scenario
An element in the set `AllStochasticScenarios`.

Return value:

An element in the set `AllGeneratedMathematicalPrograms`.

Remarks:

- The function `GMP::Stochastic::CreateBendersRootproblem` creates all Benders problems for all nodes in the scenario tree, and must be called before calling `GMP::Stochastic::BendersFindReference`.
- The *GMP* should correspond to a root node, i.e., be created by using the function `GMP::Stochastic::CreateBendersRootproblem`.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::BendersFindFeasibilityReference` and `GMP::Stochastic::CreateBendersRootproblem`.

GMP::Stochastic::CreateBendersRootproblem

The function `GMP::Stochastic::CreateBendersRootproblem` 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`.

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` with the name as specified by the *name* argument.

Remarks:

- The *GMP* should have been created by the function `GMP::Instance::GenerateStochasticProgram`.
- The generated math program belonging to the node of a Benders subproblem can be obtained by using the function `GMP::Stochastic::BendersFindReference`.
- 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`.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::BendersFindReference` and `GMP::Stochastic::UpdateBendersSubproblem`. See Section 19.1 of the Language Reference for more details on scenario tree, scenarios and stages.

GMP::Stochastic::GetObjectiveBound

The function `GMP::Stochastic::GetObjectiveBound` 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`.

solution
: An integer scalar reference to a solution.

Return value:

In case of success, the level value. Otherwise it returns `UNDF`.

Remarks:

- The *GMP* should have been created by the function `GMP::Stochastic::BendersFindReference`.
- Initially, the column `mp.SubproblemObjectiveBound` is not part of the Benders problem but it will be added if the procedure `GMP::Stochastic::AddBendersOptimalityCut` is called.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::AddBendersOptimalityCut` and `GMP::Stochastic::BendersFindReference`.

GMP::Stochastic::GetRelativeWeight

The function GMP::Stochastic::GetRelativeWeight 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.

stage
: An integer scalar reference to a stage.

scenario
: An element in the set AllStochasticScenarios.

Return value:

In case of success, the relative weight. Otherwise it returns UNDF.

Remarks:

The *GMP* should have been created by the function GMP::Instance::GenerateStochasticProgram.

See also:

The routines GMP::Instance::GenerateStochasticProgram and GMP::Stochastic::GetRepresentativeScenario. See Section 19.1 of the Language Reference for more details on scenario tree, scenarios and stages.

GMP::Stochastic::GetRepresentativeScenario

The function `GMP::Stochastic::GetRepresentativeScenario` 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`.

stage
An integer scalar reference to a stage.

scenario
An element in the set `AllStochasticScenarios`.

Return value:

An element in the set `AllStochasticScenarios`.

Remarks:

The *GMP* should have been created by the function `GMP::Instance::GenerateStochasticProgram`.

See also:

The routines `GMP::Instance::GenerateStochasticProgram` and `GMP::Stochastic::GetRelativeWeight`. See Section 19.1 of the Language Reference for more details on scenario tree, scenarios and stages.

GMP::Stochastic::MergeSolution

The procedure `GMP::Stochastic::MergeSolution` 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`.

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.

Remarks:

- The *GMP* should have been created by the function `GMP::Stochastic::CreateBendersRootproblem` or by the function `GMP::Stochastic::BendersFindReference`.
- It is most efficient to only update the objective value during the last call to `GMP::Stochastic::MergeSolution`, i.e., set *updObj* to 1 for the last call and to 0 for all preceding calls.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::CreateBendersRootproblem` and `GMP::Stochastic::BendersFindReference`.

GMP::Stochastic::UpdateBendersSubproblem

The procedure `GMP::Stochastic::UpdateBendersSubproblem` 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`.

solution
: An integer scalar reference to a solution.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The *GMP* should have been created by the function `GMP::Stochastic::CreateBendersRootproblem` or obtained by the function `GMP::Stochastic::BendersFindReference`.
- 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`.

See also:

The routines `GMP::Instance::GenerateStochasticProgram`, `GMP::Stochastic::BendersFindReference` and `GMP::Stochastic::CreateBendersRootproblem`.

12.15 GMP::Tuning Procedures and Functions

AIMMS supports the following procedures and functions for tuning models:

- `GMP::Tuning::SolveSingleMPS`
- `GMP::Tuning::TuneMultipleMPS`
- `GMP::Tuning::TuneSingleGMP`

GMP::Tuning::SolveSingleMPS

The procedure `GMP::Tuning::SolveSingleMPS` 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
     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`.

SolverStatus
The solver status as an element in the set `AllSolutionStates`.

ProgramStatus
The program status as an element in the set `AllSolutionStates`.

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.

Remarks:

- 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.

Examples:

To solve model 'mod1.mps' using CPLEX 12.9 execute:

```
GMP::Tuning::SolveSingleMPS( 'mod1.mps', 'CPLEX 12.9', SolStat, ProStat, obj, iter,
                             nodes, soltime );
```

See also:

The routine `GMP::Tuning::TuneMultipleMPS`.

GMP::Tuning::TuneMultipleMPS

The procedure `GMP::Tuning::TuneMultipleMPS` 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`.

FixedOptions
: A subset of the predefined set `AllOptions`, 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.

Remarks:

- 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.

Examples:

Assume we have a set 'FixedOptions' defined as:

```
Set FixedOptions {
    SubsetOf   :  AllOptions;
    Definition :  data { 'CPLEX 12.9::mip_search_strategy' };
}
```

Using CPLEX 12.9 we tune all '.mps', '.lp' and '.sav' problems inside the directory 'Set1' by executing:

```
GMP::Tuning::TuneMultipleMPS( "Set1", 'CPLEX 12.9', FixedOptions );
```

Note that the opion 'mip search strategy' is fixed and will not be tuned.

See also:

The routines `GMP::Tuning::SolveSingleMPS` and `GMP::Tuning::TuneSingleGMP`.

GMP::Tuning::TuneSingleGMP

The procedure GMP::Tuning::TuneSingleGMP 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.

FixedOptions
: A subset of the predefined set AllOptions, 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.

Remarks:

- 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.

Examples:

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 12.9::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 12.9).

See also:

The routines `GMP::Instance::Generate`, `GMP::Tuning::SolveSingleMPS` and `GMP::Tuning::TuneMultipleMPS`.

Part III

Model Handling

Chapter 13

Model Query Functions

AIMMS supports the following functions to query the structure of the identifiers in the model:

- AttributeToString
- CallerAttribute
- CallerLine
- CallerNode
- CallerNumberOfLocations
- ConstraintVariables
- DeclaredSubset
- DomainIndex
- IdentifierAttributes
- IdentifierDimension
- IdentifierShowAttributes
- IdentifierShowTreeLocation
- IdentifierElementRange
- IdentifierText
- IdentifierType
- IdentifierUnit
- IndexRange
- IsRuntimeIdentifier
- ReferencedIdentifiers
- SectionIdentifiers
- VariableConstraints

```
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 ;
```

AttributeToString

The function `AttributeToString` 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` specifying the identifier for which an attribute should be converted to a string.

AttributeName
: An element expression in the predefined set `AllAttributeNames` 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` contains an appropriate error message.

Remarks:

In order to protect the intellectual property of the model developer, the string `Encrypted` is returned and the predeclared identifier `CurrentErrorMessage` 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.

See also:

The function `me::GetAttribute`.

CallerAttribute

The function `CallerAttribute` 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 $\{1 \ldots$ `CallerNumberOfLocations`$\}$ The value 1, refers to the caller of the currently running procedure.

Return value:

This function returns an element in `AllAttributeNames`.

See also:

- The example at `CallerNumberOfLocations`
- The functions `errh::Attribute`, `CallerLine`, `CallerNode`, and `CallerNumberOfLocations`.

CallerLine

The function `CallerLine` 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 $\{1 \ldots$ `CallerNumberOfLocations`$\}$ The value 1, refers to the caller of the currently running procedure.

Return value:

This function returns a line number.

See also:

- The example at `CallerNumberOfLocations`
- The functions `CallerAttribute`, `errh::Line`, `CallerNode`, and `CallerNumberOfLocations`.

CallerNode

The function CallerNode 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 $\{1 \ldots$ CallerNumberOfLocations$\}$ The value 1, refers to the caller of the currently running procedure.

Return value:

This function returns an element in AllSymbols.

See also:

- The example at CallerNumberOfLocations
- The functions CallerAttribute, CallerLine, errh::Node, and CallerNumberOfLocations.

CallerNumberOfLocations

The function `CallerNumberOfLocations` 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
```

See also:

The functions `CallerAttribute`, `CallerLine`, `CallerNode`, and `errh::NumberOfLocations`.

ConstraintVariables

The function `ConstraintVariables` 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.

Remarks:

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;
        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`, containing the variables found.

See also:

The function `VariableConstraints` and `ReferencedIdentifiers`.

DeclaredSubset

The function `DeclaredSubset` 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`.

supersetName
An element expression in the predefined set `AllIdentifiers`.

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`.

DomainIndex

The function DomainIndex returns the indexPosition-th index of identifierName as an element in AllIdentifiers.

```
DomainIndex(
     identifierName,      ! (input) scalar element parameter
     indexPosition
     )       ! (input) scalar integer parameter
```

Arguments:

identifierName
: An element expression in the predefined set AllIdentifiers 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 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 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, DeclaredSubset and IndexRange.

IdentifierAttributes

The function IdentifierAttributes 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 containing all the attributes for the specified identifier.

IdentifierDimension

The function `IdentifierDimension` returns the data dimension of `identifierName`.

```
IdentifierDimension(
     identifierName)       ! (input) scalar element parameter
```

Arguments:

identifierName
An element expression in the predefined set `AllIdentifiers` 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.

Remarks:

This function replaces the deprecated suffix `.dim`.

See also:

- The functions `DomainIndex` and `IndexRange`.
- Section 25.4 of the Language Reference.
- The common example on page 670.

IdentifierShowAttributes

The function `IdentifierShowAttributes` 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`.

IdentifierShowTreeLocation

The function `IdentifierShowTreeLocation` 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 whcih you want to show the location in the Model Explorer.

See also:

The function `IdentifierShowAttributes`.

IdentifierElementRange

The function IdentifierElementRange returns the range as a set.

```
IdentifierElementRange(
     identifierName)      ! (input) scalar element parameter
```

Arguments:

identifierName
An element expression in the predefined set AllSymbols specifying the identifier for which the range should be obtained.

Return value:

This function returns the set, as an element in AllSymbols, 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.

See also:

- The functions DomainIndex, IdentifierDimension, and IndexRange.
- Section 25.4 of the Language Reference.
- The common example on page 670.

IdentifierText

The function `IdentifierText` returns the text 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` specifying the identifier for which the text should be obtained.

Return value:

This function returns a non-negative integer. If `identifierName` is not an identifier, an error message is issued. When the text is not specified, the name of the identifier is returned.

Remarks:

This function replaces the deprecated suffix `.txt`.

See also:

- The functions `IdentifierText`.
- Section 25.4 of the Language Reference.
- The common example on page 670.

IdentifierType

The function IdentifierType returns the type of identifierName as an element in AllIdentifierTypes.

```
IdentifierType(
     identifierName)       ! (input) scalar element parameter
```

Arguments:

identifierName
An element expression in the predefined set AllIdentifiers specifying the identifier for which the type should be obtained.

Return value:

This function returns a type as an element in AllIdentifierTypes. If identifierName is not an identifier, an error message is issued.

Remarks:

This function replaces the suffix .type; this suffix is deprecated.

See also:

- The functions IdentifierDimension and IdentifierUnit.
- Section 25.4 of the Language Reference.
- The common example on page 670.

IdentifierUnit

The function `IdentifierUnit` 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` 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.

Remarks:

This function complements the suffix `.unit`; 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.

See also:

- The functions `IdentifierDimension` and `IdentifierType`.
- Section 25.4 of the Language Reference.
- The common example on page 670.

IndexRange

The function `IndexRange` returns the range of an index as an element in `AllIdentifiers`.

```
IndexRange(
     indexName       ! (input) scalar element parameter
     )
```

Arguments:

indexName
: An element expression in the predefined set `AllIdentifiers` 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`. 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` and `DomainIndex`.

IsRuntimeIdentifier

The function `IsRuntimeIdentifier` 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` 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.

Remarks:

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`.

See also:

- The functions `StringToElement`, `DeclaredSubset` and `IndexRange`.
- Section 25.4 of the Language Reference.
- The common example on page 670.

ReferencedIdentifiers

The function `ReferencedIdentifiers` determines which identifiers are used in the specified attributes of a subset of `AllIdentifiers`.

```
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`.

searchAttrSet
: The set of attributes to search in for referenced identifiers. This is a subset of `AllAttributeNames`.

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` containing all the identifiers that are referenced in the attributes in *searchAttrSet* in one of the identifiers in *searchIdentSet*.

See also:

The function `ConstraintVariables` and `VariableConstraints`

SectionIdentifiers

The function SectionIdentifiers 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 specifying the section for which the identifiers should be listed.

Return value:

This function returns a subset of AllIdentifiers 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.

VariableConstraints

The function `VariableConstraints` 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`.

Remarks:

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`, containing the constraints found.

See also:

The functions `ConstraintVariables` and `ReferencedIdentifiers`.

Chapter 14

Model Edit Functions

AIMMS supports the following functions for model editing:

- `me::AllowedAttribute`
- `me::ChangeType`
- `me::ChangeTypeAllowed`
- `me::ChildTypeAllowed`
- `me::Children`
- `me::Compile`
- `me::Create`
- `me::CreateLibrary`
- `me::Delete`
- `me::ExportNode`
- `me::GetAttribute`
- `me::ImportLibrary`
- `me::ImportNode`
- `me::IsRunnable`
- `me::Move`
- `me::Parent`
- `me::Rename`
- `me::SetAttribute`

me::AllowedAttribute

The function `me::AllowedAttribute` 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` referencing a runtime identifier.

attr
: An element in the set `AllAttributeNames`

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.

See also:

The procedures `me::SetAttribute` and `me::Create`.

me::ChangeType

The procedure `me::ChangeType` 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` referencing a runtime identifier.

newType
: An element in the set `AllIdentifierTypes`.

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.

See also:

The functions `me::Create` and `me::Move`.

me::ChangeTypeAllowed

The function `me::ChangeTypeAllowed` 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` referencing a runtime identifier.

newType
An element in the set `AllIdentifierTypes`.

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.

See also:

The functions `me::Create` and `me::Move`.

me::ChildTypeAllowed

The function `me::ChildTypeAllowed` 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` referencing a runtime identifier.

newType
: An element in the set `AllIdentifierTypes`.

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.

See also:

The functions `me::Create` and `me::Move`.

me::Children

The procedure `me::Children` 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` 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`.

Return value:

This procedure returns the number of children of `runtimeId`. When `runtimeId` doesn't reference a runtime identifier an error will be raised.

See also:

The functions `me::Parent` and `me::GetAttribute`.

me::Compile

The procedure `me::Compile` 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` 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.

See also:

- The functions `me::IsRunnable` and the `APPLY statement` 10.3.1.
- The AIMMS blog post: Getting value of a dynamic identifier illustrates the use of model edit functions. The purpose of `me::Compile` in that post is to check the code in the runtime library and prepare it for execution.

me::Create

The function `me::Create` 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`.

parentId
: An element in the set `AllSymbols` 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` 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.

See also:

- The functions `me::Delete` and `me::SetAttribute`.
- The AIMMS blog post: Getting value of a dynamic identifier illustrates the use of model edit functions. The purpose of `me::Create` in that post is to create the procedure that does the actual retrieving of the data.

me::CreateLibrary

The function `me::CreateLibrary` 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` 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::ImportLibrary` and `me::Create`.
- The AIMMS blog post: Getting value of a dynamic identifier illustrates the use of model edit functions.

me::Delete

The procedure `me::Delete` a runtime identifier and all runtime identifiers below that identifier.

```
me::Delete(
        runtimeId  ! (input) an element
)
```

Arguments:

runtimeId
: An element in the set `AllIdentifiers` 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.

See also:

- The functions `me::Children` and `me::GetAttribute`.
- The AIMMS blog post: Getting value of a dynamic identifier illustrates the use of model edit functions. The purpose of `me::Delete` in that post is to remove an old existing library before creating a new one.

me::ExportNode

The procedure `me::ExportNode` writes a section to file.

```
me::ExportNode(
        esection,  ! (input) section element.
        filename)  ! (input) a string
```

Arguments:

esection
: An element in the set `AllIdentifiers` 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.

See also:

The functions `me::CreateLibrary`, `me::ImportLibrary` and `me::ImportNode`.

me::GetAttribute

The function `me::GetAttribute` 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` referencing a runtime identifier.

attr
: An element in the set `AllAttributeNames`

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.

See also:

The procedures `AttributeToString`, `me::SetAttribute` and `me::Create`.

me::ImportLibrary

The function `me::ImportLibrary` 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` 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`, `me::ImportNode` and `me::ExportNode`.

me::ImportNode

The procedure `me::ImportNode` reads a section from file.

```
me::ImportNode(
        esection,  ! (input) section element.
        filename)  ! (input) a string
```

Arguments:

esection
: An element in the set `AllIdentifiers` 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` and `me::ExportNode`.

me::IsRunnable

The function `me::IsRunnable` 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` 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` and `me::IsReadonly`.

me::Move

The procedure `me::Move` 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` referencing a runtime identifier.

parentid
: An element in the set `AllIdentifiers` 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.

Remarks:

The name change file is not supported for runtime libraries.

See also:

The functions `me::ChangeType` and `me::Rename`.

me::Parent

The function `me::Parent` returns the parent of a runtime identifier.

```
me::Parent(
        runtimeId   ! (input) an element
)
```

Arguments:

runtimeId
: An element in the set `AllIdentifiers` referencing a runtime identifier.

Return value:

The function returns an element in the set `AllIdentifiers` 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.

See also:

The functions `me::Children` and `me::GetAttribute`.

me::Rename

The procedure `me::Rename` 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` 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.

Remarks:

The name change file is not supported for runtime libraries.

See also:

The functions `me::ChangeType` and `me::Move`.

me::SetAttribute

The procedure `me::SetAttribute` changes the type 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` referencing a runtime identifier.

attr
: An element in the set `AllAttributeNames`

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.

See also:

- The procedures `me::Create` and `me::ChangeType`.
- The AIMMS blog post: Getting value of a dynamic identifier illustrates the use of model edit functions. The purpose of `me::SetAttribute` in that post is to specify the body of the procedure that does the actual work.

Part IV

Data Management

Chapter 15

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
- CaseFileMerge
- CaseFileSave
- CaseFileGetContentType
- CaseCompareIdentifier
- CaseCreateDifferenceFile
- CaseFileSectionExists
- CaseFileSectionGetContentType
- CaseFileSectionLoad
- CaseFileSectionMerge
- CaseFileSectionRemove
- CaseFileSectionSave
- CaseFileURLtoElement

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
- CaseCommandLoadAsActive
- CaseCommandLoadIntoActive
- CaseCommandMergeIntoActive
- CaseCommandNew
- CaseCommandSave
- CaseCommandSaveAs
- CaseDialogConfirmAndSave

- `CaseDialogSelectForLoad`
- `CaseDialogSelectForSave`
- `CaseDialogSelectMultiple`
- `DataManagementExit`

CaseFileLoad

With the function CaseFileLoad, 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.

Remarks:

- 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 AIMMSPRO 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.

See also:

The procedure CaseFileMerge.

CaseFileMerge

With the function `CaseFileMerge`, 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.

Remarks:

- 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 AIMMSPRO 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`.

See also:

The procedure `CaseFileLoad`

CaseFileSave

The function `CaseFileSave` 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` containing all the identifiers that must be saved. Preferrably, this set is an element of `AllCaseFileContentTypes` 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.
- When you save using `CaseFileSave` to an existing `.data` file with sections, the sections are removed.

See also:

The functions `CaseFileSectionSave` and `CaseFileLoad`

CaseCompareIdentifier

With the function `CaseCompareIdentifier` 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`

SecondCase
: An element in the set `AllCases`

Identifier
: An element in the set `AllIdentifiers`, refering to the specific identifier that you want to compare.

Suffix
: An element in the set `AllSuffixNames` with respect to which you want to compare the data.

Mode
: An element in the `AllCaseComparisonModes` 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.

CaseCreateDifferenceFile

With the procedure `CaseCreateDifferenceFile` 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` 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` with range the predeclared set `AllDifferencingModes`.

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.

CaseFileGetContentType

The procedure CaseFileGetContentType 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

See also:

The function CaseFileSave.

CaseFileSectionExists

The function `CaseFileSectionExists` 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

See also:

The functions `CaseFileSectionSave`, `CaseFileSectionLoad`, `CaseFileSectionMerge`, `CaseFileSectionRemove`

CaseFileSectionGetContentType

The procedure CaseFileSectionGetContentType 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

See also:

The functions CaseFileSectionSave, CaseFileGetContentType

CaseFileSectionLoad

With the function CaseFileSectionLoad, 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

See also:

The functions CaseFileLoad, CaseFileSectionSave, CaseFileSectionMerge, CaseFileSectionExists, CaseFileSectionRemove

CaseFileSectionMerge

With the function CaseFileSectionMerge, 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

See also:

The functions CaseFileMerge, CaseFileSectionSave, CaseFileSectionLoad, CaseFileSectionExists, CaseFileSectionRemove

CaseFileSectionRemove

The function CaseFileSectionRemove 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

See also:

The functions CaseFileSectionSave, CaseFileSectionLoad, CaseFileSectionMerge, CaseFileSectionExists

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`.

```
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` 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.

Remarks:

- 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 AIMMSPRO 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`, `CaseFileSectionLoad`, `CaseFileSectionMerge`, `CaseFileSectionExists`, `CaseFileSectionRemove`

CaseFileURLtoElement

For each case file that has been accessed during an AIMMS session, a new element is created in the predefined set AllCases. The predefined string parameter CaseFileURL 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. In that process, it may be useful to find the corresponding element in AllCases 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 a priori.

caseFileElement
: On return, this element parameter is set to the element in AllCases 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. 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.
- If *url* exists, but is not in CaseFileURL, an element will be added to AllCases.

- If *url* does not exist, but there is a corresponding entry `CaseFileURL`, 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`

CaseFileSetCurrent

The procedure CaseFileSetCurrent sets the predefined element parameter CurrentCase 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.

Remarks:

- 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 AIMMSPRO server, the url can also point to a case file stored at the server.

CaseCommandLoadAsActive

The procedure `CaseCommandLoadAsActive` 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.

Remarks:

- 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`, `CaseCommandMergeIntoActive`, `CaseCommandNew`, `CaseCommandSave`, `CaseCommandSaveAs`

CaseCommandLoadIntoActive

The procedure `CaseCommandLoadIntoActive` 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.

Remarks:

- 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`, `CaseCommandMergeIntoActive`, `CaseCommandNew`, `CaseCommandSave`, `CaseCommandSaveAs`

CaseCommandMergeIntoActive

The procedure `CaseCommandMergeIntoActive` 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.

Remarks:

- 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`, `CaseCommandLoadIntoActive`, `CaseCommandNew`, `CaseCommandSave`, `CaseCommandSaveAs`

CaseCommandNew

The procedure CaseCommandNew 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.

Remarks:

- 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 is calling CaseFileSetCurrent with an empty string. The latter will not check whether the current case should be saved first.

See also:

The procedures CaseCommandLoadAsActive, CaseCommandLoadIntoActive, CaseCommandMergeIntoActive, CaseCommandSave, CaseCommandSaveAs

CaseCommandSave

The procedure CaseCommandSave 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.

Remarks:

- 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, CaseCommandLoadIntoActive, CaseCommandMergeIntoActive, CaseCommandNew, CaseCommandSaveAs

CaseCommandSaveAs

The procedure CaseCommandSaveAs 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.

Remarks:

- 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, CaseCommandLoadIntoActive, CaseCommandMergeIntoActive, CaseCommandNew, CaseCommandSave

CaseDialogConfirmAndSave

The procedure `CaseDialogConfirmAndSave` 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.

Remarks:

- 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`

CaseDialogSelectForLoad

The procedure `CaseDialogSelectForLoad` 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.

Remarks:

- 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`, `CaseFileLoad`

CaseDialogSelectForSave

The procedure CaseDialogSelectForSave 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 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. 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.

Remarks:

- 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, CaseFileSave

CaseDialogSelectMultiple

The procedure `CaseDialogSelectMultiple` shows a case file selection dialog box in which you can select multiple case files. The result is a subset of `AllCases` 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`. 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.

Remarks:

- 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` 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`.
- 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`, the string parameter `CaseFileURL` and the set `AllCases`.

DataManagementExit

The function `DataManagementExit` 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.

Remarks:

- 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`

Chapter 16

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:

- `DataChangeMonitorCreate`
- `DataChangeMonitorDelete`
- `DataChangeMonitorHasChanged`
- `DataChangeMonitorReset`

DataChangeMonitorCreate

With the function DataChangeMonitorCreate, you can create a new data change monitor. With a data change monitor, you can determine whether any identifiers in a subset of AllIdentifiers have been changed since the latest call to DataChangeMonitorCreate or DataChangeMonitorReset. To check for any changes, you can use DataChangeMonitorHasChanged.

```
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.

Remarks:

- 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`, `DataChangeMonitorReset`, `DataChangeMonitorDelete`.

DataChangeMonitorDelete

With the function `DataChangeMonitorDelete`, 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`, `DataChangeMonitorReset`, `DataChangeMonitorHasChanged`.

DataChangeMonitorHasChanged

The function `DataChangeMonitorHasChanged` 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.

Remarks:

- Calling `DataChangeMonitorHasChanged` does not reset the data change monitor.

See also:

The functions `DataChangeMonitorCreate`, `DataChangeMonitorReset`, `DataChangeMonitorDelete`.

DataChangeMonitorReset

The function `DataChangeMonitorReset` 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`, `DataChangeMonitorHasChanged`, `DataChangeMonitorDelete`.

Chapter 17

Database Functions

AIMMS supports the following database related functions:

- CloseDataSource
- CommitTransaction
- DirectSQL
- LoadDatabaseStructure
- RollbackTransaction
- SaveDatabaseStructure
- StartTransaction
- TestDataSource
- TestDatabaseTable
- TestDatabaseColumn
- GetDataSourceProperty
- SQLCreateConnectionString
- SQLNumberOfColumns
- SQLNumberOfTables
- SQLNumberOfViews
- SQLNumberOfDrivers
- SQLColumnData
- SQLTableName
- SQLViewName
- SQLDriverName

CloseDataSource

With the procedure `CloseDataSource` 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.

Remarks:

When `CloseDataSource` is called during a transaction that was explicitly started by calling `StartTransaction` the transaction is rolled back before actually closing the data source. `CurrentErrorMessage` contains a message telling it did so.

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` 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`, `RollbackTransaction`.

DirectSQL

With the procedure `DirectSQL` 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`.

Remarks:

- 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` 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 Section 27.5 of the Language Reference.

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` 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`

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` 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`, `RollbackTransaction`.

SaveDatabaseStructure

With the procedure SaveDatabaseStructure you can save the database table structure information such that this information is quickly retrieved in subsequent AIMMS sessions. Please note that you should first make sure that you have connected to all datasources involved. Information for tables contained in non-connected datasources is not stored. In order to connect to a datasource, you should either run a read or write statement using one of its tables, or open the mapping wizard of one of its database tables.

```
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.

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` 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`, 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.

Remarks:

You cannot call `StartTransaction` recursively, i.e. you must call `CommitTransaction` or `RollbackTransaction` prior to the next call to `StartTransaction`.

See also:

The procedures `CommitTransaction` and `RollbackTransaction`.

TestDataSource

With the procedure `TestDataSource` 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` will contain a proper error message.

See also:

The procedures `TestDatabaseTable` and `TestDatabaseColumn`.

TestDatabaseTable

With the procedure `TestDatabaseTable` 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` will contain a proper error message.

Remarks:

The *Tablename* argument of the procedure `TestDatabaseTable` is case sensitive if the ODBC driver is case sensitive.

See also:

The procedures `TestDataSource` and `TestDatabaseColumn`.

TestDatabaseColumn

With the procedure TestDatabaseColumn 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 will contain a proper error message.

Remarks:

The *TableName* and *ColumnName* arguments of the procedure TestDatabaseColumn are case sensitive if the ODBC driver is case sensitive.

See also:

The procedures TestDataSource and TestDatabaseTable.

GetDataSourceProperty

With the function GetDataSourceProperty 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.

Return value:

The function returns a string with the requested datasource property in it.

Remarks:

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).

SQLNumberOfColumns

With the function `SQLNumberOfColumns` 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`, `SQLNumberOfTables` and `SQLColumnData`.

SQLNumberOfDrivers

With the function `SQLNumberOfDrivers` 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.

Remarks:

This function should be used in combination with the function `SQLDriverName`, to determine all ODBC drivers installed on your system.

See also:

The functions `SQLDriverName` and `SQLCreateConnectionString`.

SQLNumberOfTables

With the function `SQLNumberOfTables` 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`, `SQLNumberOfColumns` and `SQLTableName`.

SQLNumberOfViews

With the function `SQLNumberOfViews` 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`, `SQLNumberOfColumns` and `SQLViewName`.

SQLColumnData

With the function `SQLColumnData` 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`, 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`.

Remarks:

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` and `Val`.

SQLDriverName

With the function SQLDriverName 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 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 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.

Remarks:

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.

See also:

The functions SQLNumberOfDrivers and SQLCreateConnectionString.

SQLTableName

With the function `SQLTableName` 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.

Remarks:

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` and `SQLViewName`.

SQLViewName

With the function `SQLViewName` 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.

Remarks:

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` and `SQLTableName`.

SQLCreateConnectionString

The function `SQLCreateConnectionString` 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` and `SQLDriverName` 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.

Remarks:

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` and `SQLDriverName`.

Chapter 18

Spreadsheet Functions

AIMMS supports the following functions for reading from and writing to Excel and OpenOffice Calc workbooks:

- Spreadsheet::ColumnName
- Spreadsheet::ColumnNumber
- Spreadsheet::SetVisibility
- Spreadsheet::SetActiveSheet
- Spreadsheet::SetUpdateLinksBehavior
- Spreadsheet::SetOption
- Spreadsheet::AssignValue
- Spreadsheet::RetrieveValue
- Spreadsheet::AssignSet
- Spreadsheet::RetrieveSet
- Spreadsheet::AssignParameter
- Spreadsheet::RetrieveParameter
- Spreadsheet::AssignTable
- Spreadsheet::RetrieveTable
- Spreadsheet::ClearRange
- Spreadsheet::CopyRange
- Spreadsheet::AddNewSheet
- Spreadsheet::DeleteSheet
- Spreadsheet::GetAllSheets
- Spreadsheet::RunMacro
- Spreadsheet::CreateWorkbook
- Spreadsheet::SaveWorkbook
- Spreadsheet::CloseWorkbook
- Spreadsheet::Print

The functions operate on OpenOffice Calc workbooks, if the `WorkbookName` argument ends in `.ods`. In all other cases, the functions operate on Excel workbooks.

Spreadsheet::ColumnName

The function `Spreadsheet::ColumnName` 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.

Remarks:

- 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`.

Spreadsheet::ColumnNumber

The function `Spreadsheet::ColumnNumber` 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.

Remarks:

- 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`.

Spreadsheet::SetVisibility

The procedure `Spreadsheet::SetVisibility` 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 used.

Visibility
: A scalar element expression in the pre-defined AIMMS set `OnOff` specifying whether to show or hide the specified workbook.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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.

Spreadsheet::SetActiveSheet

The procedure `Spreadsheet::SetActiveSheet` 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 used.

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` contains a description of what went wrong.

Remarks:

- 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.

Spreadsheet::SetUpdateLinksBehavior

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` contains a description of what went wrong.

Remarks:

- 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.

Spreadsheet::SetOption

The procedure `Spreadsheet::SetOption` 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.

Spreadsheet::AssignValue

The procedure Spreadsheet::AssignValue 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 contains a description of what went wrong.

Remarks:

- By calling the procedure Spreadsheet::SetActiveSheet 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.

Spreadsheet::RetrieveValue

The procedure `Spreadsheet::RetrieveValue` 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 used.

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` contains a description of what went wrong.

Remarks:

- By calling the procedure `Spreadsheet::SetActiveSheet` 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.

Spreadsheet::AssignSet

The procedure `Spreadsheet::AssignSet` 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` contains a description of what went wrong.

Remarks:

- By calling the procedure `Spreadsheet::SetActiveSheet` 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.

Spreadsheet::RetrieveSet

The procedure Spreadsheet::RetrieveSet 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 used.

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. 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 contains a description of what went wrong.

Remarks:

- By calling the procedure Spreadsheet::SetActiveSheet 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.

Spreadsheet::AssignParameter

The procedure `Spreadsheet::AssignParameter` 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` contains a description of what went wrong.

Remarks:

- By calling the procedure Spreadsheet::SetActiveSheet 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.

Spreadsheet::RetrieveParameter

The procedure `Spreadsheet::RetrieveParameter` 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 used.

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` contains a description of what went wrong.

Remarks:

- By calling the procedure `Spreadsheet::SetActiveSheet` 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.

Spreadsheet::AssignTable

The procedure `Spreadsheet::AssignTable` 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`.
- 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`.
- 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` contains a description of what went wrong.

Remarks:

- By calling the procedure `Spreadsheet::SetActiveSheet` 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.

Spreadsheet::RetrieveTable

The procedure `Spreadsheet::RetrieveTable` 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 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` contains a description of what went wrong.

Remarks:

- By calling the procedure `Spreadsheet::SetActiveSheet` 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.

See also:

An example of the use of `ExcelRetrieveTable` is presented on the AIMMS blog post: Reading multi-dimensional Excel data with ExcelRetrieveTable including a pictorial explanation of the use of spreadsheet ranges.

Spreadsheet::ClearRange

The procedure Spreadsheet::ClearRange 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 contains a description of what went wrong.

Remarks:

- By calling the procedure Spreadsheet::SetActiveSheet 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.

Spreadsheet::CopyRange

The procedure Spreadsheet::CopyRange 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 contains a description of what went wrong.

Remarks:

- By calling the procedure Spreadsheet::SetActiveSheet 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.

- 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.

Spreadsheet::AddNewSheet

The procedure `Spreadsheet::AddNewSheet` 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
        [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` contains a description of what went wrong.

Remarks:

- Upto AIMMS 3.11 this function was known as `ExcelAddNewSheet`, which has become deprecated as of AIMMS 3.12.

Spreadsheet::DeleteSheet

The procedure Spreadsheet::DeleteSheet 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 used.

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 contains a description of what went wrong.

Remarks:

- Upto AIMMS 3.11 this function was known as ExcelDeleteSheet, which has become deprecated as of AIMMS 3.12.

Spreadsheet::GetAllSheets

The procedure `Spreadsheet::GetAllSheets` 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 used.

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` contains a description of what went wrong.

Remarks:

None.

Spreadsheet::RunMacro

The procedure `Spreadsheet::RunMacro` 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` contains a description of what went wrong.

Remarks:

- 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` 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.

Spreadsheet::CreateWorkbook

The procedure `Spreadsheet::CreateWorkbook` 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` contains a description of what went wrong.

Remarks:

- Upto AIMMS 3.11 this function was known as `ExcelCreateWorkbook`, which has become deprecated as of AIMMS 3.12.

Spreadsheet::SaveWorkbook

The procedure `Spreadsheet::SaveWorkbook` 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 used.

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` contains a description of what went wrong.

Remarks:

- Upto AIMMS 3.11 this function was known as `ExcelSaveWorkbook`, which has become deprecated as of AIMMS 3.12.

Spreadsheet::CloseWorkbook

The procedure Spreadsheet::CloseWorkbook 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 contains a description of what went wrong.

Remarks:

- Upto AIMMS 3.11 this function was known as ExcelCloseWorkbook, which has become deprecated as of AIMMS 3.12.

Spreadsheet::Print

The procedure Spreadsheet::Print 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 used.

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` contains a description of what went wrong.

Remarks:

- By calling the procedure `Spreadsheet::SetActiveSheet` 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.

Chapter 19

XML Functions

AIMMS supports the following functions for reading and writing XML files:

- `GenerateXML`
- `ReadGeneratedXML`
- `ReadXML`
- `WriteXML`

GenerateXML

The procedure GenerateXML 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, 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.

Remarks:

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, ReadXML, WriteXML. Generating XML data is discussed in full detail in Section 30.3 of the Language Reference.

ReadGeneratedXML

The procedure `ReadGeneratedXML` 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`, `ReadXML`, `WriteXML`. Generating XML data is discussed in full detail in Section 30.3 of the Language Reference.

ReadXML

The procedure ReadXML 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.

Remarks:

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, ReadGeneratedXML, WriteXML. Reading user-defined XML data is discussed in full detail in Section 30.4 of the Language Reference.

WriteXML

With the procedure `WriteXML` 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.

Remarks:

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`, `ReadGeneratedXML`, `ReadXML`. Writing user-defined XML data is discussed in full detail in Section 30.4 of the Language Reference.

Part V

User Interface Related Functions

Chapter 20

Dialog Functions

AIMMS supports the following functions for simple interaction with the end user.

- `DialogAsk`
- `DialogError`
- `DialogGetColor`
- `DialogGetDate`
- `DialogGetElement`
- `DialogGetElementByData`
- `DialogGetElementByText`
- `DialogGetNumber`
- `DialogGetPassword`
- `DialogGetString`
- `DialogMessage`
- `DialogProgress`
- `StatusMessage`

DialogAsk

The procedure `DialogAsk` 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.

Remarks:

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`, `DialogError`.

DialogError

The procedure `DialogError` 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.

Remarks:

The procedures `DialogMessage` and `DialogError` only differ in the icon that is displayed at the left side of the dialog box.

See also:

The procedures `DialogMessage`, `DialogAsk`, `DialogProgress`.

DialogGetColor

The procedure `DialogGetColor` 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.

DialogGetDate

The procedure `DialogGetDate` 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.

Remarks:

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 Section 33.7.1 of the Language Reference.

DialogGetElementByData

The procedure `DialogGetElementByData` 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`, `DialogGetElementByText`.

DialogGetElement

The procedure `DialogGetElement` 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`, `DialogGetElementByData`, `DialogGetNumber`.

DialogGetElementByText

The procedure `DialogGetElementByText` 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`, `DialogGetElementByData`.

DialogGetNumber

The procedure `DialogGetNumber` 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`, `DialogGetElement`.

DialogGetPassword

The procedure `DialogGetPassword` 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`.

DialogGetString

The procedure `DialogGetString` 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`, `DialogGetPassword`, `DialogGetElement`.

DialogMessage

The procedure `DialogMessage` 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.

Remarks:

The procedures `DialogMessage` and `DialogError` only differ in the icon that is displayed at the left side of the dialog box

See also:

The procedures `DialogError`, `DialogAsk`.

DialogProgress

The procedure `DialogProgress` 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.

Remarks:

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` with an empty message string: `DialogProgress("")`.

See also:

The procedures `DialogMessage`, `DialogError`, `DialogAsk`.

StatusMessage

With the procedure StatusMessage 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.

Remarks:

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, DialogProgress.

Chapter 21

Page Functions

AIMMS supports the following functions for opening, closing, and manipulating the pages in the interface:

- PageClose
- PageCopyTableToClipboard
- PageCopyTableToExcel
- PageGetActive
- PageGetAll
- PageGetChild
- PageGetFocus
- PageGetNext
- PageGetNextInTreeWalk
- PageGetParent
- PageGetPrevious
- PageGetTitle
- PageGetUsedIdentifiers
- PageOpen
- PageOpenSingle
- PageRefreshAll
- PageSetCursor
- PageSetFocus
- PivotTableDeleteState
- PivotTableReloadState
- PivotTableSaveState
- PrintEndReport
- PrintPage
- PrintPageCount
- PrintStartReport
- PrinterGetCurrentName
- PrinterSetupDialog
- ShowMessageWindow
- ShowProgressWindow

PageClose

With the procedure PageClose 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 closes the currently active page.

Return value:

The procedure returns 1 if the page is closed successfully, or a 0 otherwise.

Remarks:

The active page can be obtained by PageGetActive.

See also:

The procedures PageOpen, PageGetActive, and PageOpenSingle.

PageCopyTableToClipboard

With the procedure `PageCopyTableToClipboard` 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` will contain a proper error message.

Remarks:

You can specify a unique tag name for each page object via the object properties.

See also:

The procedure `PageCopyTableToExcel`.

PageCopyTableToExcel

With the procedure `PageCopyTableToExcel` 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` will contain a proper error message.

Remarks:

- 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`.

PageGetActive

With the procedure PageGetActive 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 and PageClose.

PageGetAll

With the procedure `PageGetAll` 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`, `PageGetPrevious`, `PageGetChild`, `PageGetParent`, `PageGetNextInTreeWalk`.

PageGetChild

The procedure PageGetChild 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, PageGetNext, PageGetPrevious, PageGetNextInTreeWalk, PageGetAll.

PageGetFocus

With the procedure `PageGetFocus` 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.

Remarks:

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`, `PageGetActive`.

PageGetNext

The procedure PageGetNext 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, PageGetChild, PageGetParent, PageGetNextInTreeWalk, PageGetAll.

PageGetNextInTreeWalk

The procedure PageGetNextInTreeWalk 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, PageGetPrevious, PageGetChild, PageGetParent, PageGetAll.

PageGetParent

The procedure `PageGetParent` 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`, `PageGetNext`, `PageGetPrevious`, `PageGetNextInTreeWalk`, `PageGetAll`.

PageGetPrevious

The procedure `PageGetPrevious` 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`, `PageGetChild`, `PageGetParent`, `PageGetNextInTreeWalk`, `PageGetAll`.

PageGetTitle

The procedure `PageGetTitle` 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.

PageGetUsedIdentifiers

The procedure `PageGetUsedIdentifiers` returns a subset of `AllIdentifiers` 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`.

PageOpen

With the procedure PageOpen 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 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 will contain a proper error message.

See also:

The procedures PageOpenSingle, PageClose.

PageOpenSingle

The procedure `PageOpenSingle` is similar to `PageOpen`, except that after successfull opening the page `PageOpenSingle` 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` will contain a proper error message.

See also:

The procedures `PageOpen`, `PageClose`.

PageRefreshAll

Normally, the data on all open pages is refreshed automatically each time AIMMS has finished executing a procedure. Via a call to `PageRefreshAll` you can refresh the data on all pages at any time during a procedure run (for example to show intermediate results).

```
PageRefreshAll
```

Arguments:

None

Remarks:

- 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` for a newly opened page.
- At the end of an button action, AIMMS will automatically refresh all pages.

See also:

The procedure `PageOpen`.

PageSetCursor

With the procedure PageSetCursor 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 will contain a proper error message.

Examples:

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'));
```

Remarks:

You can specify a unique tag name for each page object via the object properties.

See also:

The procedure PageSetFocus.

PageSetFocus

With the procedure PageSetFocus 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 will contain a proper error message.

Remarks:

You can specify a unique tag name for each page object via the object properties.

See also:

The procedures PageSetCursor, PageGetFocus.

PivotTableDeleteState

With the procedure `PivotTableDeleteState` 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` will contain a proper error message.

Remarks:

- 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`, `PivotTableSaveState`.

PivotTableReloadState

With the procedure `PivotTableReloadState` 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` will contain a proper error message.

Remarks:

- 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`, `PivotTableDeleteState`.

PivotTableSaveState

With the procedure PivotTableSaveState 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 will contain a proper error message.

Remarks:

- 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`, `PivotTableReloadState`.

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`, `PrintPage`.

PrintPage

With the procedure PrintPage 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` will contain a proper error message.

See also:

The procedures `PrintPageCount`, `PrintStartReport`.

PrintPageCount

The procedure `PrintPageCount` 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`.

PrintStartReport

With the procedure PrintStartReport 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 will contain a proper error message.

Remarks:

A successful call to PrintStartReport must be followed by a call to PrintEndReport, otherwise nothing is printed, and your printer may hang.

See also:

The procedures PrintEndReport, PrintPage.

PrinterGetCurrentName

With the procedure `PrinterGetCurrentName` 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.

Examples:

You can use the procedure `PrinterGetCurrentName` 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;
```

Remarks:

To change the current printer, you can use the menu item `File - Print Setup` or make a call to the procedure `PrinterSetupDialog`.

See also:

The procedures `PrinterSetupDialog`.

PrinterSetupDialog

With the procedure `PrinterSetupDialog` 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` returns 0. Otherwise it will return 1.

Examples:

You can use the procedure `PrinterSetupDialog` to make sure that a user selects a PDF printer:

```
isPDFPrinter := 0;
Repeat
PrinterGetCurrentName(currentPrinter);
if FindString(currentPrinter,"PDF") then
isPDFPrinter := 1;
break;
endif;
DialogMessage("Please select a PDF printer.");
break when PrinterSetupDialog() = 0;
EndRepeat;
```

See also:

The procedures `PrinterGetCurrentName`.

ShowMessageWindow

With the procedure `ShowMessageWindow` 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`.

ShowProgressWindow

With the procedure `ShowProgressWindow` 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`.

Chapter 22

User colors

- UserColorAdd
- UserColorDelete
- UserColorGetRGB
- UserColorModify

UserColorAdd

With the procedure UserColorAdd 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...255 indicating the red component in the RGB value of the color.

green
: An integer value in the range 0...255 indicating the green component in the RGB value of the color.

blue
: An integer value in the range 0...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.

Remarks:

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 do not persist, and, therefore, have to be added during the initialization of every project session.

See also:

UserColorDelete, UserColorGetRGB, UserColorModify. User colors are discussed in full detail in Section 11.4 of the User's Guide.

UserColorDelete

With the procedure `UserColorDelete` 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.

Remarks:

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`, `UserColorGetRGB`, `UserColorModify`. User colors are discussed in full detail in Section 11.4 of the User's Guide.

UserColorGetRGB

With the procedure `UserColorGetRGB` 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`, `UserColorDelete`, `UserColorModify`. User colors are discussed in full detail in Section 11.4 of the User's Guide.

UserColorModify

With the procedure `UserColorModify` 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...255 indicating the red component in the RGB value of the color.

green
: An integer value in the range 0...255 indicating the green component in the RGB value of the color.

blue
: An integer value in the range 0...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.

Remarks:

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`, `UserColorDelete`, `UserColorGetRGB`. User colors are discussed in full detail in Section 11.4 of the User's Guide.

Part VI

Development Support

Chapter 23

Profiler and Debugger

- DebuggerBreakPoint
- ProfilerStart
- ProfilerPause
- ProfilerContinue
- ProfilerRestart
- ProfilerCollectAllData

DebuggerBreakPoint

The procedure DebuggerBreakPoint 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.

Remarks:

- 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.

ProfilerStart

The procedure `ProfilerStart` starts measuring the execution time of statements and definitions.

```
ProfilerStart
```

Remarks:

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`, `ProfilerContinue` and `ProfilerRestart` and the predefined identifier `ProfilerData`.

ProfilerPause

The procedure `ProfilerPause` temporarily disables measuring the execution time of statements and definitions.

```
ProfilerPause
```

Remarks:

- 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` and `ProfilerRestart`.

ProfilerContinue

The procedure `ProfilerContinue` continues measuring the execution time of statements and definitions.

```
ProfilerContinue
```

Remarks:

- 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` and `ProfilerRestart`.

ProfilerRestart

The procedure `ProfilerRestart` clears the execution time measurement data of all statements and definitions.

```
ProfilerRestart
```

Remarks:

- 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` and `ProfilerPause`.

ProfilerCollectAllData

With the procedure `ProfilerCollectAllData` 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`, 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`.

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.

Remarks:

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,IndexProfilerValues);
}
```

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`.

Chapter 24

Application Information

AIMMS supports the following to help model development

- IdentifierGetUsedInformation
- IdentifierMemory
- IdentifierMemoryStatistics
- MemoryInUse
- MemoryStatistics
- ListExpressionSubstitutions
- ShowHelpTopic

IdentifierGetUsedInformation

With the procedure `IdentifierGetUsedInformation` 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`, 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.

Remarks:

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`.

IdentifierMemory

With the function `IdentifierMemory` 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` 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.

Remarks:

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.

IdentifierMemoryStatistics

With the procedure `IdentifierMemoryStatistics` 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
     doAggregate         ! (optional, default 0) scalar expression
     )
```

Arguments:

IdentSet
A subset of `AllIdentifiers` 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`.

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`.

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`.

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.

Remarks:

- The procedure prints a report of the statistics collected by AIMMS' memory manager since the last call to `IdentifierMemoryStatistics`.
- AIMMS will only collect memory statistics if the option `memory_statistics` is `on`.

ListExpressionSubstitutions

With the procedure ListExpressionSubstitutions, the expressions substituted are printed to the listing file.

```
ListExpressionSubstitutions()
```

Return value:

The procedure returns 1 on success, or 0 otherwise.

Example:

With the definition:

```
Parameter Conn3 {
    IndexDomain  :  (l1,l4);
    Definition   : {
        1 | sum( (l2,l3) | d(l1,l2) <= md and
                           d(l2,l3) <= md and
                           d(l3,l4) <= md
                           ,1 )
    }
}
```

The procedure ListExpressionSubstitutions will print to the listing file:

```
1: D(l1,l2) <= md has card 0, and is used 1 times
2: D(l2,l3) <= md has card 0, and is used 1 times
3: D(l3,l4) <= md has card 0, and is used 1 times
```

The card is the number of elements in the cache, here 0; when running this example, the definition of Conn3 was not evaluated, and the procedure ListExpressionSubstitutions does not force the evaluation of the caches either.

MemoryInUse

With the function `MemoryInUse` 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].

Remarks:

See also the functions `MemoryStatistics`, `IdentifierMemory`, `GMP::Instance::GetMemoryUsed`

MemoryStatistics

With the procedure `MemoryStatistics` 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
: 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 MemoryStatistics.

ShowSmallBlockUsage
: A 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`.

Remarks:

AIMMS will only collect memory statistics if the option `memory_statistics` is `on`.

ShowHelpTopic

With the procedure ShowHelpTopic 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.

Remarks:

AIMMS supports the following help file formats: WinHelp or WinHelp2000 (*.hlp), compiled HTML Help (*.chm), and Acrobat Reader (*.pdf).

Part VII

System Interaction

Chapter 25

Error Handling Functions

AIMMS supports the following functions for error handling:

- errh::Adapt
- errh::Attribute
- errh::Category
- errh::Code
- errh::Column
- errh::CreationTime
- errh::Filename
- errh::InsideCategory
- errh::IsMarkedAsHandled
- errh::Line
- errh::MarkAsHandled
- errh::Message
- errh::Multiplicity
- errh::Node
- errh::NumberOfLocations
- errh::Severity

errh::Adapt

The procedure `errh::Adapt` 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` referencing an error.

severity
: An element in the set `errh::AllErrorSeverities`.

message
: A string describing the problem and possibly suggestions for repairing the problem.

category
: An element in the set `errh::AllErrorCategories`, indicating the problem category to which the error belongs.

code
: An element with root set `errh::ErrorCodes`. The element will be added to the set `errh::ErrorCodes` if needed.

Return value:

Returns 1 if adapting the error is successful, 0 otherwise. In the latter case additional error(s) have been raised.

Remarks:

When `err` does not reference an error in the set `errh::PendingErrors` an additional error will be raised.
If the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Severity`, `errh::Message`, `errh::Category` and `errh::Code`.

errh::Attribute

The function `errh::Attribute` returns 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` referencing an error.

loc
An integer in the range { `1 .. errh::NumberOfLocations(err)` }.

Return value:

Returns an element in `AllAttributeNames` if the information is available and the empty element otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Node`, `errh::Line` and `errh::NumberOfLocations`.

errh::Category

The function `errh::Category` returns the error category to which the error belongs.

```
errh::Category(
        err  ! (input) an element
)
```

Arguments:

err
: An element in the set `errh::PendingErrors` referencing an error.

Return value:

Returns an element in `errh::AllErrorCategories` if the information is available and the empty element otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The function `errh::Code`, `errh::InsideCategory`.

errh::Code

The function `errh::Code` returns the identification code of the format string.

```
errh::Code(
        err  ! (input) an element
)
```

Arguments:

err
An element in the set `errh::PendingErrors` referencing an error.

Return value:

Returns an element in `errh::ErrorCodes` if the information is available and the empty element otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The function `errh::Category` and the procedure `errh::Adapt`. The predeclared identifier `errh::PendingErrors`.

errh::Column

The function `errh::Column` 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` referencing an error.

Return value:

Returns a column number if the information is available and 0 otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Line` and `errh::Filename`.

errh::CreationTime

The function `errh::CreationTime` 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` referencing an error.

fmt

A string that holds the date and time format used in the returned string. Valid format strings are described in Section 33.7. 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.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The function `CurrentToString`.

errh::Filename

The function `errh::Filename` 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` 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.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Line` and `errh::Column`.

errh::InsideCategory

The function `errh::InsideCategory` 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` referencing an error.

cat
: An element in the set `errh::AllErrorCategories` referencing an error.

Return value:

Returns 1 if `err` in inside the category `cat` and 0 otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Code` and `errh::Category`.

errh::IsMarkedAsHandled

The function `errh::IsMarkedAsHandled` 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` referencing an error.

Return value:

Returns 1 if the error is marked as handled and 0 otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The function `errh::MarkAsHandled`.

errh::Line

The function `errh::Line` 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` 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.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The function `errh::Column`, `errh::Filename`, `errh::Attribute`, `errh::Node` and `errh::NumberOfLocations`.

errh::Message

The function `errh::Message` returns a description of the error.

```
errh::Message(
        err  ! (input) an element
)
```

Arguments:

err
  An element in the set `errh::PendingErrors` referencing an error.

Return value:

Returns a string if the information is available and the empty string otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors`or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The procedure `errh::Adapt`.

errh::MarkAsHandled

The procedure `errh::MarkAsHandled` 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` 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.

Remarks:

When `err` doesn't reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The function `errh::IsMarkedAsHandled`.

errh::Multiplicity

The function `errh::Multiplicity` returns the number of occurrences of this error.

```
errh::Multiplicity(
        err  ! (input) an element
)
```

Arguments:

err

An element in the set `errh::PendingErrors` referencing an error.

Return value:

Returns the number of occurrences of this error.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Code`, `errh::Category`, `errh::Message` and `errh::Severity`.

errh::Node

The function `errh::Node` 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` referencing an error.

loc
: An integer in the range { `1 .. errh::NumberOfLocations(err)` }.

Return value:

Returns an element in `AllSymbols` if the information is available and the empty element otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Attribute`, `errh::Line` and `errh::NumberOfLocations`.

errh::NumberOfLocations

The function `errh::NumberOfLocations` 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` referencing an error.

Return value:

Returns the number locations for which this error is relevant.

Remarks:

When `err` doesn't reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The functions `errh::Node`, `errh::Attribute` and `errh::Line`.

errh::Severity

The function `errh::Severity` returns the severity of the error.

```
errh::Severity(
        err  ! (input) an element
)
```

Arguments:

err
: An element in the set `errh::PendingErrors` referencing an error.

Return value:

Returns an element in `errh::AllErrorSeverities` if the information is available and the empty element otherwise.

Remarks:

When `err` does not reference an element in `errh::PendingErrors` or when the current filter is the filter `To Global Collector` an additional error will be raised.

See also:

The procedures `errh::Adapt` and `errh::MarkAsHandled`.

Chapter 26

Option manipulation

OptionGetDefaultString

With the procedure `OptionGetDefaultString` 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`, `OptionGetKeywords`, `OptionGetString`.

OptionGetKeywords

With the procedure `OptionGetKeywords` 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.

Remarks:

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`, `OptionGetString`, `OptionSetString`.

OptionGetString

With the procedure `OptionGetString` 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.

Remarks:

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` and `OptionGetKeywords`.

See also:

`OptionGetValue`, `OptionGetKeywords`, `OptionSetString`.

OptionGetValue

With the procedure OptionGetValue 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.

Remarks:

- 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 and OptionGetKeywords.
- You can modify option values programmatically using the OPTION statement (see also Section 8.5 of the Language Reference), or using the procedures OptionSetValue and OptionSetString.

See also:

OptionGetString, OptionGetKeywords, OptionSetValue, OptionSetString.

OptionSetString

With the procedure `OptionSetString` 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.

Remarks:

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` and `OptionGetKeywords`.

See also:

`OptionSetValue`, `OptionGetValue`, `OptionGetKeywords`.

OptionSetValue

With the procedure `OptionSetValue` 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.

Remarks:

- 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` and `OptionGetKeywords`.
- You can also modify option values using the `OPTION` statement (see also Section 8.5 of the Language Reference).

See also:

`OptionGetString`, `OptionGetKeywords`, `OptionSetString`.

Chapter 27

Licensing Functions

AIMMS supports the following licensing functions:

- LicenseExpirationDate
- LicenseMaintenanceExpirationDate
- LicenseNumber
- LicenseStartDate
- LicenseType
- ProjectDeveloperMode
- SecurityGetGroups
- SecurityGetUsers
- SolverGetControl
- SolverReleaseControl

LicenseExpirationDate

The procedure `LicenseExpirationDate` 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.

Remarks:

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`, `LicenseMaintenanceExpirationDate`.

LicenseMaintenanceExpirationDate

The procedure `LicenseMaintenanceExpirationDate` 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.

Remarks:

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`, `LicenseExpirationDate`.

LicenseNumber

The procedure `LicenseNumber` 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.

Remarks:

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`.

LicenseStartDate

The procedure `LicenseStartDate` 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.

Remarks:

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`, `LicenseMaintenanceExpirationDate`.

LicenseType

The procedure LicenseType 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.

Remarks:

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.

ProjectDeveloperMode

The function `ProjectDeveloperMode` 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.

SecurityGetGroups

With the procedure `SecurityGetGroups` 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`.

SecurityGetUsers

With the procedure SecurityGetUsers 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. 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.

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 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.

Remarks:

- 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.

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 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.

Remarks:

- 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.

Chapter 28

Environment Functions

AIMMS supports the following system setting functions, which give access to, or allow modification of, various system settings:

- `AimmsRevisionString`
- `EnvironmentGetString`
- `EnvironmentSetString`
- `GeoFindCoordinates`
- `TestInternetConnection`

AimmsRevisionString

The procedure `AimmsRevisionString` 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.

Remarks:

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.

EnvironmentGetString

With the procedure EnvironmentGetString 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.

Remarks:

- 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`.

EnvironmentSetString

With the function `EnvironmentSetString` 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.

Remarks:

- With `EnvironmentSetString` you can change the value for existing environment variables as well as create new environment variables.
- Note that the function `EnvironmentSetString` will only change the values of variables in the environment associated with the AIMMS process.

See also:

`EnvironmentGetString`.

GeoFindCoordinates

The procedure GeoFindCoordinates 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 will contain a proper error message.

Examples:

The following calls to the procedure GeoFindCoordinates 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.

Remarks:

- 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 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.

TestInternetConnection

With the procedure `TestInternetConnection` 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` will contain a proper error message.

Remarks:

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.

Chapter 29

Invoking actions

Delay

With the procedure `Delay` 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`.

Execute

With the Execute 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. When running on Linux and the program is located in the AIMMSproject folder, this string must start with a '/.' (without the single quotes).

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).

Remarks:

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.

ExitAimms

With the procedure ExitAimms 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.

Remarks:

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.

OpenDocument

The procedure `OpenDocument` 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.

Examples:

```
OpenDocument( "http://www.aimms.com" );
OpenDocument( "mailto:info@aimms.com" );
OpenDocument( "anyfile.doc" );
OpenDocument( "c:\\windows" );
```

See also:

The procedure `Execute`.

ScheduleAt

With the procedure ScheduleAt 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. 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 will contain a proper error message.

Remarks:

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.

SessionArgument

With the procedure `SessionArgument` 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.

Remarks:

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` 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.

Chapter 30

File and Directory Functions

AIMMS supports the following functions for accessing disk files and directories:

- DirectoryCopy
- DirectoryCreate
- DirectoryDelete
- DirectoryExists
- DirectoryGetCurrent
- DirectoryGetFiles
- DirectoryGetSubdirectories
- DirectoryMove
- DirectorySelect
- FileAppend
- FileCopy
- FileDelete
- FileEdit
- FileExists
- FileGetSize
- FileMove
- FilePrint
- FileRead
- FileSelect
- FileSelectNew
- FileTime
- FileTouch
- FileView

DirectoryCopy

The procedure `DirectoryCopy` 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` will contain a proper error message.

Remarks:

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`, `FileCopy`, `DirectoryExists`.

DirectoryCreate

The procedure `DirectoryCreate` 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` will contain a proper error message.

Remarks:

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`, `DirectoryDelete`.

DirectoryDelete

The procedure `DirectoryDelete` 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` will contain a proper error message.

See also:

The procedures `DirectoryExists`, `FileDelete`.

DirectoryExists

With the procedure `DirectoryExists` 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.

Remarks:

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`.

DirectoryGetCurrent

The procedure `DirectoryGetCurrent` 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`.

DirectoryGetFiles

The procedure `DirectoryGetFiles` 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`. 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` 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`.

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` 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.

Remarks:

- 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` to find the names of the subdirectories in a particular directory.
- The procedures `DirectoryGetCurrent` and `DirectorySelect` to obtain the current directory and to select a particular directory.

DirectoryGetSubdirectories

The procedure `DirectoryGetSubdirectories` 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`. 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` 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`.

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` 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.

Remarks:

- 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` to find the names of the files in a particular directory.
- The procedures `DirectoryGetCurrent` and `DirectorySelect` to obtain the current directory and to select a particular directory.

DirectoryMove

The procedure `DirectoryMove` 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 wild-card 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` will contain a proper error message.

Remarks:

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`, `FileMove`, `DirectoryExists`.

DirectorySelect

With the procedure `DirectorySelect` 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.

Remarks:

If `DirectorySelect` 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`, `DirectoryGetCurrent`.

FileAppend

The procedure FileAppend 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 will contain a proper error message.

Remarks:

- 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, FileExists.

FileCopy

The procedure FileCopy 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 wild-card 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 will contain a proper error message.

See also:

The procedures FileMove, DirectoryCopy, FileExists.

FileDelete

The procedure `FileDelete` 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` will contain a proper error message.

See also:

The procedures `FileExists`, `DirectoryDelete`.

FileEdit

The procedure `FileEdit` 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`. 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.

Remarks:

If you want to use another external text editor to edit a specific file, then you can use the procedure `Execute`.

See also:

The procedures `FileView`, `Execute`.

FileExists

With the procedure `FileExists` 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.

Remarks:

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`

FileGetSize

The procedure FileGetSize 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 will contain a proper error message.

See also:

The procedure FileExists.

FileMove

The procedure FileMove 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 wild-card 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 will contain a proper error message.

See also:

The procedures FileCopy, DirectoryMove, FileExists.

FilePrint

The procedure `FilePrint` 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`. 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.

Remarks:

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`.

FileRead

With the procedure `FileRead` 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`. 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.

Remarks:

- 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`.

FileSelect

With the procedure `FileSelect` 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.

Remarks:

If `FileSelect` 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`.

FileSelectNew

With the procedure `FileSelectNew` 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.

Remarks:

If `FileSelectNew` 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`.

FileTime

The procedure `FileTime` 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` will contain a proper error message.

See also:

The procedure `FileExists`.

FileTouch

The procedure FileTouch 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 will contain a proper error message.

FileView

The procedure `FileView` 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`. 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.

Remarks:

If you want to use another external text editor to view a specific file, then you can use the procedure `Execute`.

See also:

The procedures `FileEdit`, `Execute`.

Part VIII

Predefined Identifiers

Chapter 31

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.

- AllAuthorizationLevels
- AllAvailableCharacterEncodings
- ASCIICharacterEncodings
- ASCIIUnicodeCharacterEncodings
- UnicodeCharacterEncodings
- AllCharacterEncodings
- AllColors
- AllIntrinsics
- AllKeywords
- AllOptions
- AllPredeclaredIdentifiers
- AllSolvers
- AllSymbols
- ProfilerData
- CurrentAuthorizationLevel
- CurrentGroup
- CurrentSolver
- CurrentUser
- AllAimmsStringConstantElements
- AimmsStringConstants

AllAuthorizationLevels

The predefined set `AllAuthorizationLevels` contains the names of all authorization levels associated with an AIMMS project.

```
Set AllAuthorizationLevels {
    Index      :  IndexAuthorizationLevels;
}
```

Definition:

The contents of the set `AllAuthorizationLevels` 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.

Remarks:

The set `AllAuthorizationLevels` 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`, 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 19.2).

AllAvailableCharacterEncodings

The predefined set `AllAvailableCharacterEncodings` 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` 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`.

ASCIICharacterEncodings

The predefined set `ASCIICharacterEncodings` 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` 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`.

ASCIIUnicodeCharacterEncodings

The predefined set `ASCIIUnicodeCharacterEncodings` is the union of `ASCIICharacterEncodings` and `UnicodeCharacterEncodings`.

```
Set ASCIIUnicodeCharacterEncodings {
    SubsetOf   :  AllCharacterEncodings;
    Index      :  IndexAvailableCharacterEncodings;
}
```

Definition:

The contents of the set `ASCIIUnicodeCharacterEncodings` 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`.

UnicodeCharacterEncodings

The predefined set `UnicodeCharacterEncodings` contains the names of Unicode character encodings.

```
Set UnicodeCharacterEncodings {
    SubsetOf    :  AllCharacterEncodings;
    Index       :  IndexAvailableCharacterEncodings;
}
```

Definition:

The contents of the set `UnicodeCharacterEncodings` 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`.

AllCharacterEncodings

The predefined set `AllCharacterEncodings` contains the names of all character encodings known to AIMMS.

```
Set AllCharacterEncodings {
    Index      :  IndexCharacterEncodings;
}
```

Definition:

The contents of the set `AllCharacterEncodings` 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

  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

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
```

Remarks:

Not all character encodings enumerated above may be available on your system. The subset of available character encodings is `AllAvailableCharacterEncodings`.
The set `AllCharacterEncodings` 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 page 496 of the Language Reference.
- The set of character encodings available to the current AIMMS session: AllAvailableCharacterEncodings.

AllColors

The predefined set `AllColors` contains the names of all users colors associated with an AIMMS project.

```
Set AllColors {
    Index      :  IndexColors;
}
```

Definition:

The contents of the set `AllColors` 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` and `UserColorDelete`.

Remarks:

The set `AllColors` 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 Section 11.4 of the User's Guide.

AllIntrinsics

The predefined set AllIntrinsics contains the names of all standard AIMMS functions and operators.

```
Set AllIntrinsics {
    SubsetOf   :  AllSymbols;
    Index      :  IndexIntrinsics;
}
```

Definition:

The contents of the set AllIntrinsics is the collection of all standard functions and operators.

Updatability:

The contents of the set cannot be modified.

See also:

The set AllSymbols.

AllKeywords

The predefined set AllKeywords contains the names of all keywords in AIMMS.

```
Set AllKeywords {
    SubsetOf   :  AllSymbols;
    Index      :  IndexKeywords;
}
```

Definition:

The contents of the set AllKeywords is the collection of all keywords.

Updatability:

The contents of the set cannot be modified.

See also:

The set AllSymbols.

AllOptions

The predefined set AllOptions contains the names of all options available in AIMMS.

```
Set AllOptions {
    Index      :  IndexOptions;
}
```

Definition:

The contents of the set AllOptions 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.

Remarks:

In the set AllOptions, the solver specific options are prefixed by the solver name and version.

See also:

Options in AIMMS is described in detail in Section 20.1 of the User's Guide.

AllPredeclaredIdentifiers

The predefined set `AllPredeclaredIdentifiers` contains the names of all predeclared identifiers in AIMMS.

```
Set AllPredeclaredIdentifiers {
    SubsetOf    :  AllSymbols;
    Index       :  IndexPredeclaredIdentifiers;
}
```

Definition:

The contents of the set `AllPredeclaredIdentifiers` is the collection of all predeclared identifier names.

Updatability:

The contents of the set cannot be modified.

See also:

The set `AllSymbols`.

AllSolvers

The predefined set `AllSolvers` 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` 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.

Remarks:

The set `AllSolvers` 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 Section 20.3 of the User's Guide.
- The parameter `CurrentSolver`.
- The functions `GMP::Instance::CreateSolverSession` and `GMP::Instance::GetSolver`

AllSymbols

The predefined set `AllSymbols` 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` 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`, `AllPredeclaredIdentifiers`, `AllKeywords`, and `AllIntrinsics`.

ProfilerData

The predefined parameter ProfilerData can be used by AIMMS to store profiling information about the execution of procedures and the updating of definitions.

```
Parameter ProfilerData {
    IndexDomain  :  ( IndexIdentifiers, IndexprofilerTypes );
}
```

Remarks:

- Profiling information is only stored in the parameter ProfilerData 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 and the predefined identifier AllProfilerTypes.

CurrentAuthorizationLevel

The predefined element parameter `CurrentAuthorizationLevel` 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` 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` 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.

Remarks:

The element parameter `CurrentAuthorizationLevel` 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`, 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 19.2).

See also:

The set `AllAuthorizationLevels`.

CurrentGroup

The predefined string parameter `CurrentGroup` 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` 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` 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.

Remarks:

The string parameter `CurrentGroup` 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 19.2).

See also:

The function `SecurityGetGroups`.

CurrentSolver

The predefined element parameter `CurrentSolver` 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` 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` 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` will, however, not modify the (default) settings in the **Solver Configuration** dialog box, that will be loaded at startup.

See also:

- The sets `AllMathematicalProgrammingTypes` and `AllSolvers`.
- Solver configuration is discussed in full detail in Section 20.3 of the User's Guide.

CurrentUser

The predefined string parameter `CurrentUser` contains the name of the user currently logged on to an AIMMS project.

```
StringParameter CurrentUser;
```

Definition:

The contents of the string parameter `CurrentUser` 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` can only be modified by logging on to the project as another user through the **File-Authorization-User** menu.

Remarks:

The string parameter `CurrentUser` 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 19.2).

See also:

The function `SecurityGetUsers`.

AllAimmsStringConstantElements

The predefined set `AllAimmsStringConstantElements` contains the elements for which the predeclared string parameter `AimmsStringConstants` has a value.

```
Set AllAimmsStringConstantElements {
    Index      :  IndexAimmsStringConstantElements;
}
```

Definition:

This set is fixed to { `Platform, Architecture, Flavor` }.

AimmsStringConstants

The predefined string parameter `AimmsStringConstants` 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` and the predeclared set `AllAimmsStringConstantElements`.

Chapter 32

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.

- AggregationTypes
- AllAttributeNames
- AllBasicValues
- AllCaseComparisonModes
- AllColumnTypes
- AllDataColumnCharacteristics
- AllDifferencingModes
- AllExecutionStatuses
- AllGMPExtensions
- AllIdentifierTypes
- AllIsolationLevels
- AllFileAttributes
- AllMathematicalProgrammingTypes
- AllMatrixManipulationDirections
- AllMatrixManipulationProgrammingTypes
- AllProfilerTypes
- AllRowTypes
- AllConstraintProgrammingRowTypes
- AllMathematicalProgrammingRowTypes
- AllSolutionStates
- AllSolverInterrupts
- AllStochasticGenerationModes
- AllSuffixNames
- AllValueKeywords
- AllViolationTypes
- ContinueAbort
- DiskWindowVoid
- Integers
- MaximizingMinimizing
- MergeReplace
- OnOff
- TimeSlotCharacteristics

- YesNo

AggregationTypes

The predefined set `AggregationTypes` contains the collection of all possible aggregation types supported by the `Aggregate` and `DisAggregate` functions.

```
Set AggregationTypes {
    Index      :  IndexAggregationTypes;
    Definition :  {
        data { summation, average,
               maximum, minimum,
               interpolation }
    }
}
```

Definition:

The set `AggregationTypes` contains the collection of all possible aggregation types supported by the `Aggregate` and `DisAggregate` functions.

Updatability:

The contents of the set cannot be modified.

Remarks:

ELement parameters into `AggregationTypes` can be used as the *type* argument of the `Aggregate` and `DisAggregate` functions.

See also:

The functions `Aggregate` and `DisAggregate`. Time-dependent aggregation and disaggregation is discussed in full detail in Section 33.5 of the Language Reference.

AllAttributeNames

The predefined set `AllAttributeNames` contains the names of all possible identifier attributes.

```
Set AllAttributeNames {
    Index      :  IndexAttributeNames;
}
```

Definition:

The predefined set `AllAttributeNames` contains the names of all possible identifier attributes.

Updatability:

The contents of the set cannot be modified.

See also:

- The sets `AllIdentifierTypes`, and `AllSuffixNames`.
- Model edit functions, see Section 35.6 of the Language Reference.
- The functions `me::AllowedAttribute` and `IdentifierAttributes`.

AllBasicValues

The predefined set `AllBasicValues` contains the names of all basic values available in Aimms.

```
Set AllBasicValues {
    Index      :  IndexBasicValues;
    Definition :  data { NonBasic, Basic, SuperBasic };
}
```

Definition:

The set `AllBasicValues` contains the names of all basic values in Aimms.

Updatability:

The contents of the set cannot be modified.

AllCaseComparisonModes

The predefined set `AllCaseComparisonModes` contains the collection of all possible modes supported by the `CaseCompareIdentifier` function.

```
Set AllCaseComparisonModes {
    Index      :  IndexCaseComparisonModes;
    Definition :  {
        data { min, max, sum
               average, count }
      }
}
```

Definition:

The predefined set `AllCaseComparisonModes` contains the collection of all possible modes supported by the `CaseCompareIdentifier` function.

Updatability:

The contents of the set cannot be modified.

Remarks:

Element parameters into `AllCaseComparisonModes` can be used as the *mode* argument of the `CaseCompareIdentifier` function.

See also:

The function `CaseCompareIdentifier`.

AllColumnTypes

The predefined set `AllColumnTypes` contains the names of all column types available in the matrix manipulation library of AIMMS.

```
Set AllColumnTypes {
    Index      :  IndexColumnTypes;
    Definition :  data { integer, continuous };
}
```

Definition:

The set `AllColumnTypes` contains the names of all column types available in the matrix manipulation library of AIMMS.

Updatability:

The contents of the set cannot be modified.

Remarks:

ELement parameters into `AllColumnTypes` can be used as the *type* argument of the `GMP::Column::SetType` function.

See also:

The function `GMP::Column::SetType`. Matrix manipulation is discussed in more detail in Chapter 16 of the Language Reference.

AllDataColumnCharacteristics

The predefined set `AllDataColumnCharacteristics` contains all possible column properties, which can be queried using the function `SQLColumnData`.

```
Set AllDataColumnCharacteristics {
    Index      :  IndexDataColumnCharacteristics;
    Definition :  {
        data { Name, DataType, Width,
               NumberOfDecimals, IsPrimaryKey,
               Nullable, DefaultValue, Remark }
    }
}
```

Definition:

The set `AllDataColumnCharacteristics` contains all possible column properties, which can be queried using the function `SQLColumnData`. 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`.

AllDataSourceProperties

The predefined set AllDataSourceProperties 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,
                SQL_DM_VER, SQL_DRIVER_VER, SQL_KEYWORDS,
                SQL_SERVER_NAME }
    }
}
```

Definition:

The set AllDataSourceProperties 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.

AllDifferencingModes

The predefined set `AllDifferencingModes` contains the collection of all possible differencing modes supported by the `CaseCreateDifferenceFile` function.

```
Set AllDifferencingModes {
    Index      :  IndexDifferencingModes;
    Definition :  {
         data { blockReplacement, elementReplacement,
                elementAddition, elementMultiplication  }
    }
}
```

Definition:

The predefined set `AllDifferencingModes` contains the collection of all possible differencing modes supported by the `CaseCreateDifferenceFile` 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.

Remarks:

Element parameters into `AllDifferencingModes` can be used as the *diffTypes* argument of the `CaseCreateDifferenceFile` function.

See also:

The function `CaseCreateDifferenceFile`.

AllExecutionStatuses

The predefined set `AllExecutionStatuses` contains the names of all execution statuses associated with asynchronous solves.

```
Set AllExecutionStatuses {
    Index       :  IndexExecutionStatus;
}
```

Definition:

The set `AllExecutionStatuses` 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`.

AllGMPExtensions

The predefined set `AllGMPExtensions` contains the collection of all possible extensions in the matrix manipulation library of AIMMS.

```
Set AllGMPExtensions {
    Index      :  IndexGMPExtensions;
    Definition :  {
         data { DualObjective, DualDefinition,
                DualLowerBound, DualUpperBound }
    }
}
```

Definition:

The predefined set `AllGMPExtensions` contains the collection of all possible extensions in the matrix manipulation library of AIMMS.

Updatability:

The contents of the set cannot be modified.

Remarks:

Together with the suffixes `.ExtendedConstraint` and `.ExtendedVariable`, element parameters into `AllGMPExtensions` can be used as the `extension` argument of a constraint, a variable, and a mathematical program.

See also:

The set `AllSuffixNames`. Matrix manipulation is discussed in more detail in Chapter 16 of the Language Reference.

AllIdentifierTypes

The predefined set `AllIdentifierTypes` contains the names of all possible identifier types.

```
Set AllIdentifierTypes {
    Index     :  IndexIdentifierTypes;
}
```

Definition:

The predefined set `AllIdentifierTypes` 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                       ,
  '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 and AllSuffixNames.
- Model edit functions, see Section 35.6 of the Language Reference.
- The functions me::ChangeType and IdentifierType.

AllIsolationLevels

The predefined set AllIsolationLevels contains the supported isolation levels for a database transaction, as started through the procedure StartTransaction.

```
Set AllIsolationLevels {
    Index      :  IndexIsolationLevels;
    Definition :  {
        data { ReadUncommitted, ReadCommitted,
               RepeatableRead, Serializable }
    }
}
```

Definition:

The predefined set AllIsolationLevels 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.

Remarks:

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.

AllFileAttributes

The predefined set `AllFileAttributes` 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` contains the attributes the intrinsic functions `DirectoryGetFiles`, 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`, and `DirectoryGetSubDirectories`.

AllMathematicalProgrammingTypes

The predefined set `AllMathematicalProgrammingTypes` contains the list of mathematical programming types supported by AIMMS.

```
Set AllMathematicalProgrammingTypes {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexMathematicalProgrammingTypes;
}
```

Definition:

The set `AllMathematicalProgrammingTypes` contains the list of mathematical programming types supported by AIMMS.

Updatability:

The contents of the set `AllMathematicalProgrammingTypes` is completely under the control of AIMMS, and cannot be modified.

Remarks:

Element parameters into the set `AllMathematicalProgrammingTypes` 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` defines the active solver for each mathematical programming type.

See also:

The set `AllValueKeywords`, `CurrentSolver`. Mathematical programs are discussed in full detail in Section 15.1 of the Language Reference, the `SOLVE` statement in Section 15.3.

AllMatrixManipulationDirections

The predefined set `AllMatrixManipulationDirections` contains the list of optimization directions supported by the matrix manipulation library of AIMMS.

```
Set AllMatrixManipulationDirections {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexMatrixManipulationDirections;
}
```

Definition:

The set `AllMatrixManipulationDirections` contains the list of optimization directions supported by the matrix manipulation library of AIMMS.

Updatability:

The contents of the set `AllMatrixManipulationDirections` is completely under the control of AIMMS, and cannot be modified.

Remarks:

Element parameters into the set `AllMatrixManipulationDirections` can be used as the *direction* argument of the `GMP::Instance::SetDirection` function.

See also:

The set `AllValueKeywords`, the function `GMP::Instance::SetDirection`. Matrix manipulation is discussed in more detail in Chapter 16 of the Language Reference.

AllMatrixManipulationProgrammingTypes

The predefined set `AllMatrixManipulationProgrammingTypes` contains the collection of mathematical programming types that can be used in conjunction with the matrix manipulation library of Aimms.

```
Set AllMatrixManipulationProgrammingTypes {
    SubsetOf   :  AllMathematicalProgrammingTypes;
    Index      :  IndexMatrixManipulationProgrammingTypes;
}
```

Definition:

The predefined set `AllMatrixManipulationProgrammingTypes` contains the collection of mathematical programming types that can be used in conjunction with the matrix manipulation library of Aimms.

Updatability:

The contents of the set `AllMatrixManipulationProgrammingTypes` is completely under the control of Aimms, and cannot be modified.

Remarks:

Element parameters into the set `AllMatrixManipulationDirections` can be used as the *type* argument of the `GMP::Instance::SetMathematicalProgrammingType` function.

See also:

The set `AllMathematicalProgrammingTypes`, the function `GMP::Instance::SetMathematicalProgrammingType`. Matrix manipulation is discussed in more detail in Chapter 16 of the Language Reference.

AllProfilerTypes

The predefined set AllProfilerTypes 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 currently contains the profiler types '*hits*', '*gross time*', and '*net time*'.

See also:

The function ProfilerStart and the predefined parameter ProfilerData.

AllRowTypes

The predefined set AllRowTypes contains the collection of all possible row types and is the superset of .

```
Set AllRowTypes {
    Index      :  IndexRowTypes;
    Definition :  data { '<=', '=', '>=', ranged, '<', '>', '<>' };
}
```

Definition:

The set AllRowTypes contains the collection of all possible row types available in the matrix manipulation library of Aimms.

Updatability:

The contents of the set cannot be modified.

Remarks:

ELement parameters into AllRowTypes can be used as the *type* argument of the GMP::Row::SetType function.

See also:

The function GMP::Row::SetType. Matrix manipulation is discussed in more detail in Chapter 16 of the Language Reference.

AllConstraintProgrammingRowTypes

The predefined set `AllConstraintProgrammingRowTypes` contains the collection of all possible row types available to be used by the Constraint Programming global constraint `cp::Count`.

```
Set AllConstraintProgrammingRowTypes {
    SubsetOf   :  AllRowTypes;
    Index      :  IndexConstraintProgrammingRowTypes;
    Definition :  data { '<=', '=', '>=', '<', '>', '<>' };
}
```

Definition:

The set `AllConstraintProgrammingRowTypes` contains the collection of all possible row types available as relation operator to the function `cp::Count`.

Updatability:

The contents of the set cannot be modified.

See also:

The function `cp::Count` and the set `AllRowTypes`

AllMathematicalProgrammingRowTypes

The predefined set `AllMathematicalProgrammingRowTypes` contains the collection of all possible row types available in the matrix manipulation library of Aimms.

```
Set AllMathematicalProgrammingRowTypes {
    Index      :  IndexMathematicalProgrammingRowTypes;
    Definition :  data { '<=', '=', '>=', ranged };
}
```

Definition:

The set `AllMathematicalProgrammingRowTypes` contains the collection of all possible row types available in the matrix manipulation library of Aimms.

Updatability:

The contents of the set cannot be modified.

Remarks:

ELement parameters into `AllMathematicalProgrammingRowTypes` can be used as the *type* argument of the `GMP::Row::SetType` function.

See also:

The function `GMP::Row::SetType` and the super set `AllRowTypes`. Matrix manipulation is discussed in more detail in Chapter 16 of the Language Reference.

AllSolutionStates

The predefined set `AllSolutionStates` contains the names of possible values of the program and solver status of a mathematical program.

```
Set AllSolutionStates {
    Index      :  IndexSolutionStates;
}
```

Definition:

The set `AllSolutionStates` 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.

Remarks:

The suffixes `ProgramStatus` and `SolverStatus` of a mathematical program take their values in the set `AllSolutionStates`.

See also:

The program status and solver status are discussed in more detail in Section 15.2 of the Language Reference.

AllSolverInterrupts

The predefined set `AllSolverInterrupts` contains the names of all causes for a callback.

```
Set AllSolverInterrupts {
    Index      :  IndexSolverInterrupts;
    Definition :  {
        data { AddCut, Branch, Candidate, Heuristic, Incumbent,
               Iterations, StatusChange, AddLazyConstraint,
               Finished, Time }
    }
}
```

Definition:

The set `AllSolverInterrupts` contains the names of all causes for a callback.

Updatability:

The contents of the set cannot be modified.

Remarks:

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`, to obtain the particular callback for which your callback procedure was called.

See also:

The routines `GMP::Instance::SetCallbackAddCut`, `GMP::Instance::SetCallbackAddLazyConstraint`, `GMP::Instance::SetCallbackBranch`, `GMP::Instance::SetCallbackCandidate`, `GMP::Instance::SetCallbackHeuristic`, `GMP::Instance::SetCallbackIncumbent`, `GMP::Instance::SetCallbackStatusChange`, `GMP::Instance::SetCallbackTime`, and `GMP::SolverSession::GetCallbackInterruptStatus`.

AllStochasticGenerationModes

The predefined set `AllStochasticGenerationModes` defines the modes in which `GMP::Instance::GenerateStochasticProgram` may generate a stochastic programming problem.

```
Set AllStochasticGenerationModes {
    Index      :  IndexStochasticGenerationModes;
    Definition :  {
        data { CreateNonAnticipativityConstraints,
               SubstituteStochasticVariables }
    }
}
```

Definition:

The predefined set `AllStochasticGenerationModes` defines the set of elements `CreateNonAnticipativityConstraints` and `SubstituteStochasticVariables`.

Updatability:

The contents of the set `AllStochasticGenerationModes` cannot be modified.

See also:

- Stochastic programming is discussed in Chapter 19 of the Language Reference.
- The intrinsic function `GMP::Instance::GenerateStochasticProgram`.

AllSuffixNames

The predefined set AllSuffixNames contains the names of all existing suffixes of all identifier types.

```
Set AllSuffixNames {
    Index       :  IndexSuffixNames;
}
```

Definition:

The set AllSuffixNames 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.
- The functions ScalarValue, ActiveCard, Card, CaseCompareIdentifier, and GMP::Solution::SendToModelSelection.

AllValueKeywords

The predefined set `AllValueKeywords` serves as the root set of various other predefined sets containing AIMMS keywords.

```
Set AllValueKeywords {
    Index      :  IndexValueKeywords;
    Definition :  {
        AllMathematicalProgrammingTypes +
        AllMatrixManipulationDirections +
        AllViolationTypes + YesNo +
        ContinueAbort + MergeReplace + OnOff +
        DiskWindowVoid + MaximizingMinimizing
    }
}
```

Definition:

The set `AllValueKeywords` contains keywords used in various other predefined sets containing AIMMS keywords.

Updatability:

The contents of the set `AllValueKeywords` is completely under the control of AIMMS, and cannot be modified.

Remarks:

The set `AllValueKeywords` is, in general, of little direct use in an AIMMS application.

See also:

The sets `AllMathematicalProgrammingTypes`, `AllMatrixManipulationDirections`, `AllViolationTypes`, `YesNo`, `ContinueAbort`, `DiskWindowVoid`, `MaximizingMinimizing`, `MergeReplace`, `OnOff`.

AllViolationTypes

The predefined set AllViolationTypes 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 contains the violation types for which violation penalties can be specified in a mathematical program declaration.

Updatability:

The contents of the set AllViolationTypes is completely under the control of AIMMS, and cannot be modified.

Remarks:

The set AllViolationTypes is typically used in the index domain of identifiers specified in the ViolationPenalties attribute of a MathematicalProgram.

See also:

The sets AllMathematicalProgrammingTypes, AllMatrixManipulationDirections, ContinueAbort, DiskWindowVoid, MaximizingMinimizing, MergeReplace, OnOff. The ViolationPenalties attribute of a mathematical programs is discussed in Section 15.4 of the Language Reference.

ContinueAbort

The predefined set `ContinueAbort` 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` defines the set of possible return statuses of solver callback procedures.

Updatability:

The contents of the set cannot be modified.

Remarks:

The elements of the set `ContinueAbort` can be assigned to the `CallbackReturnStatus` suffix of a mathematical program upon return of a solver callback procedure.

See also:

The set `AllValueKeywords`. Solver callback procedures are discussed in Section 15.2 of the Language Reference.

DiskWindowVoid

The predefined set DiskWindowVoid 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 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.

Remarks:

Element parameters into the set DiskWindowVoid can be entered in the Device attribute of File identifiers to allow dynamic device changes for a file.

See also:

The set AllValueKeywords. File identifiers are discussed in Section 31.1 of the Language Reference.

Integers

The predefined set Integers 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 defines the range of integers that can possibly serve as integer set elements in AIMMS.

Updatability:

The contents of the set cannot be modified.

Remarks:

Subsets of the sets Integers 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.

See also:

The sets AllDataFiles, AllCases, AllDataSets. Integer sets are discussed in Section 3.2.2 of the Language Reference.

MaximizingMinimizing

The predefined set `MaximizingMinimizing` 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` 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.

Remarks:

Element parameters into the set `MaximizingMinimizing` can be entered in the `Direction` attribute of mathematical programs to allow dynamic choices of the optimization direction.

See also:

The set `AllValueKeywords`. Mathematical programs are discussed in more detail in Section 15.1 of the Language Reference.

MergeReplace

The predefined set MergeReplace 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 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 cannot be modified.

Remarks:

Element parameters into the set MergeReplace can be used to dynamically indicate the mode of a READ, WRITE or SOLVE statement.

See also:

The set AllValueKeywords. The SOLVE statement is discussed in Section 15.3 of the Language Reference, the READ and WRITE statements in Section 26.2.

OnOff

The predefined set OnOff 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 defines the set of possibilities the PageMode suffix of File identifiers.

Updatability:

The contents of the set OnOff cannot be modified.

Remarks:

Element parameters into the set OnOff 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. The PageMode suffix of FILE identifiers is discussed in full detail in Section 31.4.

TimeSlotCharacteristics

The predefined set `TimeSlotCharacteristics` contains the collection of timeslot characteristic which can be used in conjunction with the function `TimeSlotCharacteristic`.

```
Set TimeSlotCharacteristics {
    Index      :  IndexTimeSlotCharacteristics;
    Definition :  {
        data { century, year, quarter, month
        weekday, yearday, monthday
        week, weekyear, weekcentury
        hour, minute, second, tick }
    }
}
```

Definition:

The set `TimeSlotCharacteristics` contains the collection of timeslot characteristic which can be used in conjunction with the function `TimeSlotCharacteristic`.

Updatability:

The contents of the set cannot be modified.

Remarks:

ELement parameters into `TimeSlotCharacteristics` can be used as the *characteristic* argument of the `TimeSlotCharacteristic` function.

See also:

The function `TimeSlotCharacteristic`. The use of the function `TimeSlotCharacteristic` is explained in more detail in Section 33.4 of the Language Reference.

YesNo

The predefined set YesNo defines the set of elements Yes and No.

```
Set YesNo {
    SubsetOf   :  AllValueKeywords;
    Index      :  IndexYesNo;
    Definition :  data { yes, no };
}
```

Definition:

The predefined set YesNo defines the set of elements Yes and No.

Updatability:

The contents of the set YesNo cannot be modified.

Remarks:

The set YesNo is not used by AIMMS anymore.

See also:

The set AllValueKeywords.

Chapter 33

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.

- AllAssertions
- AllConstraints
- AllConventions
- AllDatabaseTables
- AllDefinedParameters
- AllDefinedSets
- AllFiles
- AllFunctions
- AllGMPEvents
- AllIdentifiers
- AllIndices
- AllIntegerVariables
- AllMacros
- AllMathematicalPrograms
- AllNonLinearConstraints
- AllParameters
- AllProcedures
- AllQuantities
- AllSections
- AllSets
- AllSolverSessionCompletionObjects
- AllSolverSessions
- AllStochasticConstraints
- AllStochasticParameters
- AllStochasticVariables
- AllUpdatableIdentifiers
- AllVariables
- AllVariablesConstraints

AllAssertions

The predefined set AllAssertions contains the names of all assertions within an AIMMS model.

```
Set AllAssertions {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexAssertions;
}
```

Definition:

The contents of the set AllAssertions 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**.

Remarks:

The set AllAssertions or subsets thereof are typically used in the ASSERT statement in an AIMMS model.

See also:

The sets AllIdentifiers. Assertions are discussed in Section 25.2 of the Language Reference.

AllConstraints

The predefined set `AllConstraints` contains the names of all constraints within an AIMMS model.

```
Set AllConstraints {
    SubsetOf   :  AllVariablesConstraints;
    Index      :  IndexConstraints;
}
```

Definition:

The contents of the set `AllConstraints` 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**.

Remarks:

The set `AllConstraints` or subsets thereof are typically used in the `Constraints` attribute of `MathematicalProgram` declared within an AIMMS model.

See also:

The sets `AllIdentifiers`, `AllVariables`. Constraints are discussed in Section 14.2, mathematical programs in Section 15.1 of the Language Reference.

AllConventions

The predefined set AllConventions contains the names of all conventions defined within an AIMMS model.

```
Set AllConventions {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexConventions;
}
```

Definition:

The contents of the set AllConventions 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**.

Remarks:

Element parameters into the set AllConventions 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, AllQuantities. Conventions are discussed in full detail in Section 32.8 of the Language Reference.

AllDatabaseTables

The predefined set `AllDatabaseTables` 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` 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`. Database tables are discussed in Section 27.1 of the Language Reference.

AllDefinedParameters

The predefined set `AllDefinedParameters` contains the names of all defined parameters within an AIMMS model.

```
Set AllDefinedParameters {
    Subsetof   :  AllParameters;
    Index      :  IndexDefinedParameters;
}
```

Definition:

The contents of the set `AllDefinedParameters` 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`. Parameters are discussed in Section 4.1 of the Language Reference.

AllDefinedSets

The predefined set `AllDefinedSets` contains the names of all defined sets within an AIMMS model.

```
Set AllDefinedSets {
    SubsetOf   :  AllSets;
    Index      :  IndexDefinedSets;
}
```

Definition:

The contents of the set `AllDefinedSets` 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`. Sets are discussed in Section 3.2 of the Language Reference.

AllFiles

The predefined set `AllFiles` contains the names of all files declared within an AIMMS model.

```
Set AllFiles {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexFiles;
}
```

Definition:

The contents of the set `AllFiles` 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**.

Remarks:

The set `AllFiles` is the range of the element parameter `CurrentFile`.

See also:

The element parameter `CurrentFile`. Files are discussed in Section 31.1 of the Language Reference.

AllFunctions

The predefined set AllFunctions contains the names of all functions defined within an AIMMS model.

```
Set AllFunctions {
    SubsetOf    :  AllIdentifiers;
    Index       :  IndexFunctions;
}
```

Definition:

The contents of the set AllFunctions 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**.

Remarks:

Elements of the set AllFunctions are typically used in conjunction with the APPLY statement, to allow data-driven evaluation of functional expressions.

See also:

The sets AllIdentifiers. Functions are discussed in Section 10.2 of the Language Reference, the APPLY statement in Section 10.3.1.

AllGMPEvents

The predefined set `AllGMPEvents` contains all GMP Events.

```
Set AllGMPEvents {
    SubsetOf     :  AllSolverSessionCompletionObjects;
    Index        :  IndexGMPEvents;
}
```

Definition:

The set `AllGMPEvents` 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`, `GMP::Event::Delete`, `GMP::Event::Reset`, and `GMP::Event::Set`, and the predeclared identifier `AllSolverSessionCompletionObjects`.

AllIdentifiers

The predefined set `AllIdentifiers` 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` 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**.

Remarks:

Subsets of `AllIdentifiers` 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`. Data control statements are discussed in Section 25.3, the `READ` and `WRITE` statements in Section 26.2, and the `DISPLAY` statement in Section 31.3 of the Language Reference. Working with the set `AllIdentifiers` is described in more detail in Section 25.4.

AllIndices

The predefined set `AllIndices` contains the names of all indices defined within an AIMMS model.

```
Set AllIndices {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexIndices;
}
```

Definition:

The contents of the set `AllIndices` 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`, `AllIdentifiers`. Sets and their corresponding indices are discussed in Section 3.2 of the Language Reference.

AllIntegerVariables

The predefined set `AllIntegerVariables` contains the names of all integer variables within an AIMMS model.

```
Set AllIntegerVariables {
    SubsetOf   :  AllVariables;
    Index      :  IndexIntegerVariables;
}
```

Definition:

The contents of the set `AllIntegerVariables` 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`, `Integers`.

AllMacros

The predefined set AllMacros contains the names of all macros within an AIMMS model.

```
Set AllMacros {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexMacros;
}
```

Definition:

The contents of the set AllMacros 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 Section 6.4 of the Language Reference.

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` 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 Section 15.1 of the Language Reference.
- The functions `GMP::Instance::Generate`, `GMP::Instance::GenerateStochasticProgram`, and `GMP::Instance::GetSymbolicMathematicalProgram`.

AllNonLinearConstraints

The predefined set `AllNonLinearConstraints` 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` 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`.

AllParameters

The predefined set `AllParameters` contains the names of all parameters within an AIMMS model.

```
Set AllParameters {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexParameters;
}
```

Definition:

The contents of the set `AllParameters` 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**.

Remarks:

Subsets of `AllParameters` 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`, `AllIdentifiers`. Data control statements are discussed in Section 25.3, the `READ` and `WRITE` statements in Section 26.2, and the `DISPLAY` statement in Section 31.3 of the Language Reference.

AllProcedures

The predefined set `AllProcedures` contains the names of all procedures defined within an AIMMS model.

```
Set AllProcedures {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexProcedures;
}
```

Definition:

The contents of the set `AllProcedures` 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**.

Remarks:

Elements of the set `AllProcedures` are typically used in conjunction with the `APPLY` statement, to allow data-driven procedural execution.

See also:

The sets `AllIdentifiers`. Procedures are discussed in Section 10.1 of the Language Reference, the `APPLY` statement in Section 10.3.1.

AllQuantities

The predefined set AllQuantities contains the names of all quantities defined within an AIMMS model.

```
Set AllQuantities {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexQuantities;
}
```

Definition:

The contents of the set AllQuantities 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, AllConventions. Quantities are discussed in full detail in Section 32.2 of the Language Reference.

AllSections

The predefined set AllSections contains the names of all sections within an AIMMS model.

```
Set AllSections {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexSections;
}
```

Definition:

The contents of the set AllSections 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**.

Remarks:

Section names contained in AllSections 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. Model sections are discussed in Section 4.2 of the User's Guide. Data control statements are discussed in Section 25.3, the READ and WRITE statements in Section 26.2, and the DISPLAY statement in Section 31.3 of the Language Reference.

AllSets

The predefined set `AllSets` contains the names of all sets within an AIMMS model.

```
Set AllSets {
    SubsetOf   :  AllIdentifiers;
    Index      :  IndexSets;
}
```

Definition:

The contents of the set `AllSets` 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**.

Remarks:

Subsets of `AllSets` 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`, `AllIdentifiers`. Data control statements are discussed in Section 25.3, the `READ` and `WRITE` statements in Section 26.2, and the `DISPLAY` statement in Section 31.3 of the Language Reference.

AllSolverSessionCompletionObjects

The predefined set `AllSolverSessionCompletionObjects` is the root set containing both `AllGMPEvents` and `AllSolverSessions`.

```
Set AllSolverSessionCompletionObjects {
    Index       :  IndexSolverSessionCompletionObjects;
    Definition  :  AllGMPEvents + AllSolverSessions;
}
```

Definition:

The set `AllExecutionStatuses` is the root set containing both `AllGMPEvents` and `AllSolverSessions`.

See also:

The predeclared identifiers `AllGMPEvents` and `AllSolverSessions`.

AllSolverSessions

The predefined set AllSolverSessions 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 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 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, GMP::Instance::CreateSolverSession and GMP::Instance::DeleteSolverSession, and the predeclared identifier AllSolverSessionCompletionObjects.

AllStochasticConstraints

The predefined set `AllStochasticConstraints` 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` 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 Chapter 19 of the Language Reference.
- The intrinsic function `GMP::Instance::GenerateStochasticProgram`.
- The sets `AllConstraints`, `AllStochasticParameters` and `AllStochasticVariables`.
- Constraints are discussed in Chapter 14 of the Language Reference.

AllStochasticParameters

The predefined set `AllStochasticParameters` 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` 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 Chapter 19 of the Language Reference.
- The intrinsic function `GMP::Instance::GenerateStochasticProgram`.
- The sets `AllParameters`, `AllStochasticVariables` and `AllStochasticConstraints`.
- Parameters are discussed in Section 4.1 of the Language Reference.

AllStochasticVariables

The predefined set AllStochasticVariables 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 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 Chapter 19 of the Language Reference.
- The intrinsic function GMP::Instance::GenerateStochasticProgram.
- The sets AllVariables, AllStochasticParameters and AllStochasticConstraints.
- Variables are discussed in Chapter 14 of the Language Reference.

AllUpdatableIdentifiers

The predefined set AllUpdatableIdentifiers 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 contains the names of the model identifiers that are, in principle, considered updatable by AIMMS.

Updatability:

The contents of AllUpdatableIdentifiers can be modified programmatically from within an AIMMS model. The set cannot be updated from within the end-user interface.

Remarks:

- The set AllUpdatableIdentifiers determines which identifiers are updatable *in principle*. Which identifiers in AllUpdatableIdentifiers can *actually* be modified within the graphical end-user interface is determined by the set CurrentInputs.
- 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 and CurrentInputs.

See also:

The sets AllIdentifiers, CurrentInputs.

AllVariables

The predefined set AllVariables contains the names of all variables within an AIMMS model.

```
Set AllVariables {
    SubsetOf    :  AllVariablesConstraints;
    Index       :  IndexVariables;
}
```

Definition:

The contents of the set AllVariables 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**.

Remarks:

The set AllVariables or subsets thereof are typically used in the Variables attribute of MathematicalPrograms declared within an AIMMS model.

See also:

The sets AllIdentifiers, AllConstraints. Variables are discussed in Section 14.1, mathematical programs in Section 15.1 of the Language Reference.

AllVariablesConstraints

The predefined set `AllVariablesConstraints` 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` 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**.

Remarks:

The set `AllVariablesConstraints` 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`, `AllVariables`, `AllConstraints`. The `ViolationPenalties` attribute of a mathematical programs is discussed in Section 15.4 of the Language Reference.

Chapter 34

Execution State Related Identifiers

The following collection of predefined identifiers contains information about the current state of the AIMMS execution engine.

- `AllGeneratedMathematicalPrograms`
- `AllProgressCategories`
- `AllStochasticScenarios`
- `CurrentAutoUpdatedDefinitions`
- `CurrentErrorMessage`
- `CurrentFile`
- `CurrentFileName`
- `CurrentInputs`
- `CurrentMatrixBlockSizes`
- `CurrentMatrixColumnCount`
- `CurrentMatrixRowCount`
- `CurrentPageNumber`
- `ODBCDateTimeFormat`

AllGeneratedMathematicalPrograms

The predefined set `AllGeneratedMathematicalPrograms` 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` 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`, `GMP::Instance::Copy`, `GMP::Instance::Rename`, `GMP::Instance::Delete` and `GMP::Instance::CreateDual`.

AllProgressCategories

The predefined set `AllProgressCategories` contains the names of all created progress categories.

```
Set AllProgressCategories {
    Index       :  IndexProgressCategories;
}
```

Definition:

The contents of the set `AllProgressCategories` is the collection of all progress categories created by the functions `GMP::Instance::CreateProgressCategory` and `GMP::SolverSession::CreateProgressCategory`. 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`, `GMP::SolverSession::CreateProgressCategory` and `GMP::ProgressWindow::DeleteCategory`.

AllStochasticScenarios

The predefined set AllStochasticScenarios contains the names of all stochastic scenarios.

```
Set AllStochasticScenarios {
    Index       :  IndexStochasticScenarios;
}
```

Definition:

The contents of the set AllStochasticScenarios 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 Chapter 19 of the Language Reference.
- The intrinsic function GMP::Instance::GenerateStochasticProgram.

CurrentAutoUpdatedDefinitions

The predefined set CurrentAutoUpdatedDefinitions 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 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 can be modified programmatically from within an AIMMS model. The set cannot be modified from within the end-user interface.

Remarks:

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 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, CurrentInputs. The UPDATE statement and the set CurrentAutoUpdatedDefinitions are discussed in more detail in Section 7.3 of the Language Reference.

CurrentErrorMessage

The predefined string parameter `CurrentErrorMessage` contains a description of the last runtime error that occurred during the execution of an AIMMS model.

```
StringParameter CurrentErrorMessage;
```

Definition:

The string parameter `CurrentErrorMessage` 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` can be modified programmatically from within an AIMMS model. Its value cannot be modified from within the end-user interface.

Remarks:

- AIMMS never clears the contents `CurrentErrorMessage`, but only updates its value whenever an error occurs.
- When AIMMS is called through the AIMMS API, `CurrentErrorMessage` 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 Section 34.7 of the Language Reference. Error messages from interface functions are discussed in Section 17.3 from the User's Guide.

CurrentFile

The predefined element parameter CurrentFile contains the name of the file identifier to which output is currently directed.

```
ElementParameter CurrentFile {
    Range       :  AllFiles;
}
```

Definition:

The element parameter CurrentFile contains the name of the file identifier to which output from the PUT and DISPLAY statements is currently directed.

Updatability:

The value of CurrentFile 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.

Remarks:

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.

See also:

The string parameter CurrentFileName. The PUT statement is discussed in Section 31.2 of the Language Reference, the DISPLAY statement in Section 31.3.

CurrentFileName

The predefined string parameter `CurrentFileName` contains the file name associated with the file identifier to which output is currently directed.

```
StringParamter CurrentFileName;
```

Definition:

The string parameter `CurrentFileName` 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` 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.

Remarks:

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`.

See also:

The element parameter `CurrentFile`. File identifiers are discussed in Section 31.1 of the Language Reference.

CurrentInputs

The predefined set CurrentInputs 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 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 can be modified programmatically from within an Aimms model. The set cannot be updated from within the end-user interface.

Remarks:

- The set AllUpdatableIdentifiers determines which identifiers are updatable *in principle*. Therefore, you can only add identifiers to CurrentInputs which are already contained in the set AllUpdatableIdentifiers
- 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 and CurrentInputs.
- 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, CurrentInputs.

CurrentMatrixBlockSizes

The predefined parameter `CurrentMatrixBlockSizes` contains the number of non-zeros for the last mathematical program generated.

```
Parameter CurrentMatrixBlockSizes {
    IndexDomain  :  (IndexConstraints, IndexVariables);
}
```

Definition:

The parameter `CurrentMatrixBlockSizes` 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.

Remarks:

- You can use the parameter `CurrentMatrixBlockSizes`, 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`, `CurrentMatrixColumnCount` and `CurrentMatrixBlockSizes` are only set when the AIMMS option **Solvers General - Matrix Generation - Matrix_Block_Sizes** is set to `on`.

See also:

The sets `CurrentMatrixColumnCount`, `CurrentMatrixRowCount`.

CurrentMatrixColumnCount

The predefined parameter `CurrentMatrixColumnCount` contains the number of columns for the last mathematical program generated.

```
Parameter CurrentMatrixColumnCount {
    IndexDomain  :  IndexVariables;
}
```

Definition:

The parameter `CurrentMatrixColumnCount` contains the number of columns for the last mathematical program generated. The parameter counts the columns generated for each individual *symbolic* variable.

Remarks:

- You can use the parameter `CurrentMatrixColumnCount`, 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`, `CurrentMatrixColumnCount` and `CurrentMatrixBlockSizes` are only set when the AIMMS option **Solvers General - Matrix Generation - Matrix_Block_Sizes** is set to `on`.

See also:

The sets `CurrentMatrixRowCount`, `CurrentMatrixBlockSizes`.

CurrentMatrixRowCount

The predefined parameter `CurrentMatrixRowCount` contains the number of rows for the last mathematical program generated.

```
Parameter CurrentMatrixRowCount {
    IndexDomain  :  IndexConstraints;
}
```

Definition:

The parameter `CurrentMatrixRowCount` contains the number of rows for the last mathematical program generated. The parameter counts the rows generated for each individual *symbolic* constraint.

Remarks:

- You can use the parameter `CurrentMatrixRowCount`, 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`, `CurrentMatrixColumnCount` and `CurrentMatrixBlockSizes` are only set when the AIMMS option **Solvers General - Matrix Generation - Matrix_Block_Sizes** is set to `on`.

See also:

The sets `CurrentMatrixColumnCount`, `CurrentMatrixBlockSizes`.

CurrentPageNumber

The predefined parameter `CurrentPageNumber` contains current page number used by AIMMS when printing print pages.

```
Parameter CurrentPageNumber;
```

Definition:

The predefined parameter `CurrentPageNumber` contains current page number used by AIMMS when printing print pages.

Updatability:

AIMMS will automatically reset the value `CurrentPageNumber` to 1 at the following times:

- before printing a print page using the **File-Print** menu,
- before printing a print page using the `PrintPage` function outside of a pair of calls to the functions `PrintStartReport` and `PrintEndReport`, and
- just after calling the function `PrintStartReport`.

The value of `CurrentPageNumber` can be modified programmatically from within the AIMMS model.

Remarks:

According to the list of rules above, modifying the value of `CurrentPageNumber` will only have an effect of the page numbers printed on print pages within a pair of calls to `PrintStartReport` and `PrintEndReport`.

See also:

The functions `PrintPage`, `PrintStartReport`, `PrintEndReport`. Print pages are discussed in Section 14.1 of the User's Guide, print functions are discussed in more detail in Section 17.3.2.

ODBCDateTimeFormat

The predefined string parameter ODBCDateTimeFormat defines, for each identifier within an AIMMS model, the date-time conversion string.

```
StringParameter ODBCDateTimeFormat {
    IndexDomain  :  IndexIdentifiers;
}
```

Definition:

The string parameter ODBCDateTimeFormat 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 can be modified both from within the model and the end-user interface.

Remarks:

The use of ODBCDateTimeFormat 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 is discussed in more detail in Section 27.8 of the Language Reference. The format to which values of ODBCDateTimeFormat should comply are discussed in Section 33.7.

Chapter 35

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
- AllCaseTypes
- AllDataCategories
- AllDataFiles
- AllDataSets
- CurrentCase
- CurrentCaseSelection
- CurrentDataSet
- CurrentDefaultCaseType

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
- CurrentCase
- CurrentCaseSelection
- CurrentCaseFileContentType
- AllCaseFileContentTypes
- CaseFileURL

AllCases

The predefined set AllCases contains the references to all cases that are currently available in the AIMMS project.

```
Set AllCases {
    SubsetOf   :  AllDataFiles;
    Index      :  IndexCases;
}
```

Definition:

The set AllCases 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 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 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.

Remarks:

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, DataFileGetDescription, DataFileGetGroup, DataFileGetName, DataFileGetOwner, DataFileGetPath and DataFileGetTime.
- The integer case references stored in the set AllCases 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 can be obtained through the predeclared identifier CaseFileURL.
- The integer case references stored in the set AllCases 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`. Accessing cases from within an AIMMS model is discussed in full detail in Section 16.2 of the User's Guid.

AllCaseTypes

The predefined set AllCaseTypes contains the names of all case types declared within an AIMMS project.

```
Set AllCaseTypes {
    Index      :  IndexCaseTypes;
}
```

Definition:

The contents of the set AllCaseTypes 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.

Remarks:

- The function CaseGetType returns the case type of a case as an element of the set AllCaseTypes. The identifiers and data categories associated with a case type can be obtained through the functions CaseTypeContents and CaseTypeCategories. The default case type of a case when saving it is set through the predefined element parameter CurrentDefaultCaseType.
- This identifier is only relevant when the chosen Data_Management_style is single_data_manager_file.

AllDataCategories

The predefined set `AllDataCategories` contains the names of all data categories declared within an AIMMS project.

```
Set AllDataCategories {
    Index      :  IndexDataCategories;
}
```

Definition:

The contents of the set `AllDataCategories` 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.

Remarks:

- The function `DatasetGetCategory` returns the data category of a dataset as an element of the set `AllDataCategories`. The identifiers associated with a data category can be obtained through the function `DataCategoryContents`.
- This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

AllDataFiles

The predefined set `AllDataFiles` 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` 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.

Remarks:

- Elements of the set `AllDataFiles` are more commonly referenced through its subsets `AllCases` and `AllDataSets`.
- Further information about the integer data file references can be obtained through the functions `DataFileGetAcronym`, `DataFileGetDescription`, `DataFileGetGroup`, `DataFileGetName`, `DataFileGetOwner`, `DataFileGetPath` and `DataFileGetTime`.
- The integer data file references stored in the set `AllDataFiles` 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`, `AllDataSets`.

AllDataSets

The predefined set AllDataSets 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 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, DatasetDelete and DatasetSaveAs.

Remarks:

- Further information about the integer dataset references can be obtained through the functions DataFileGetAcronym, DataFileGetDescription, DataFileGetGroup, DataFileGetName, DataFileGetOwner, DataFileGetPath and DataFileGetTime.
- The integer dataset references stored in the set AllDataSets 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.

CurrentCase

The predefined element parameter `CurrentCase` contains a reference to the currently active case within an AIMMS project.

```
ElementParameter CurrentCase {
    Range      :  AllCases;
}
```

Definition:

The element parameter `CurrentCase` contains an (integer) case reference (as an element of `AllCases`) to the currently active case within an AIMMS project, or is empty if the active case is not named.

Updatability:

The element parameter `CurrentCase` 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` can only be modified by actively loading another case either in the **Data Manager**, through the **Data** menu, or using the functions `CaseLoadCurrent` and `CaseSaveAs`.
When using `Disk_files_and_folders`, the value of `CurrentCase` can only be modified by actively loading or saving a case through the **Data** menu, or by using the functions `CaseFileSetCurrent`, `CaseCommandLoadAsActive`, `CaseComandSave`, `CaseComandSaveAs` or `CaseCommandNew`.

See also:

The set `AllCases`, the element parameter `CurrentDataSet`. Loading and saving cases is discussed in full detail in Section 16.1 of the User's Guide.

CurrentCaseSelection

The predefined set CurrentCaseSelection contains the current multiple case selection within an AIMMS project.

```
Set CurrentCaseSelection {
    SubsetOf   :  AllCases;
    Index      :  IndexCurrentCaseSelection;
}
```

Definition:

The contents of the set CurrentCaseSelection is the collection of (integer) case references (as elements of AllCases) 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, or programmatically through a direct assignment within the model.

See also:

The set AllCases. Working with multiple cases is discussed in full detail in Section 16.2 of the User's Guide.

CurrentDataSet

The predefined element parameter `CurrentDataSet` 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` contains, for every data category, an (integer) dataset reference (as an element of `AllDataSets`) 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` 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` and `DatasetSaveAs`.

Remarks:

This identifier is only relevant when the chosen `Data_Management_style` is `single_data_manager_file`.

CurrentDefaultCaseType

The predefined element parameter AllCaseTypes contains the name of the current default case type.

```
ElementParameter CurrentDefaultCaseType {
    Range      :  AllCaseTypes;
}
```

Definition:

The value of the element parameter CurrentDefaultCaseType, 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 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.

Remarks:

This identifier is only relevant when the chosen Data_Management_style is single_data_manager_file.

CurrentCaseFileContentType

The predefined element parameter CurrentCaseFileContentType 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 is a references to a subset of AllIdentifiers, 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.

Remarks:

- 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.

AllCaseFileContentTypes

The predefined set AllCaseFileContentTypes 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 is a subset of AllIdentifiers. 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'. 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.

Remarks:

- 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.

CaseFileURL

The string parameter CaseFileURL holds the url (i.e. the full path name) of the file that corresponds to each element in AllCases.

```
StringParameter CaseFileURL {
    IndexDomain  :  AllCases;
}
```

Definition:

The contents of the set AllCases 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 helps you to get the location of each of these cases.

Updatability:

The contents of the set AllCases as well as their corresponding values in CaseFileURL are maintained by AIMMS itself and cannot be modified directly. They are modified when you load or save cases, or through the function CaseFileURLtoElement.

Remarks:

- 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 are only guaranteed to be unique within a single AIMMS session.

See also:

The set AllCases and the function CaseFileURLtoElement.

Chapter 36

Date-Time Related Identifiers

The following collection of predefined identifiers contains data used in representing

- AllAbbrMonths
- AllAbbrWeekdays
- AllMonths
- AllTimeZones
- AllWeekdays
- LocaleAllAbbrMonths
- LocaleAllAbbrWeekdays
- LocaleAllMonths
- LocaleAllWeekdays
- LocaleLongDateFormat
- LocaleShortDateFormat
- LocaleTimeFormat
- LocaleTimeZoneName
- LocaleTimeZoneNameDST

AllAbbrMonths

The predefined set `AllAbbrMonths` 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` contains the abbreviated English names of all months.

Updatability:

The contents of the set cannot be modified.

Remarks:

The set `AllAbbrMonths` can be used to construct a date-time format specification as specified in Section 33.7. Such date-time format specifications are required, for instance, in the `TimeslotFormat` attribute of a `Calendar`.

See also:

The sets `AllMonths`, `LocaleAllAbbrMonths`, `LocaleAllMonths`. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

AllAbbrWeekdays

The predefined set AllAbbrWeekdays 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 contains the abbreviated English names of all weekdays.

Updatability:

The contents of the set cannot be modified.

Remarks:

The set AllAbbrWeekdays can be used to construct a date-time format specification as specified in Section 33.7. Such date-time format specifications are required, for instance, in the TimeslotFormat attribute of a Calendar.

See also:

The sets AllWeekdays, LocaleAllAbbrWeekdays, LocaleAllWeekdays. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

AllMonths

The predefined set AllMonths 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 contains the unabbreviated English names of all months.

Updatability:

The contents of the set cannot be modified.

Remarks:

The set AllMonths can be used to construct a date-time format specification as specified in Section 33.7. Such date-time format specifications are required, for instance, in the TimeslotFormat attribute of a Calendar.

See also:

The sets AllAbbrMonths, LocaleAllMonths, LocaleAllMonths. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

AllTimeZones

The predefined set AllTimeZones contains the set of all available time zones.

```
Set AllTimeZones {
    Index      :  IndexTimeZones;
}
```

Definition:

The set AllTimeZones 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.

Remarks:

The set AllTimeZones 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 Section 33.2 of the Language Reference, the time zone specific part of a date-time format in Section 33.7.4.

AllWeekdays

The predefined set AllWeekdays 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 contains the unabbreviated English names of all weekdays.

Updatability:

The contents of the set cannot be modified.

Remarks:

The set AllWeekdays can be used to construct a date-time format specification as specified in Section 33.7. Such date-time format specifications are required, for instance, in the TimeslotFormat attribute of a Calendar.

See also:

The sets AllAbbrWeekdays, LocaleAllWeekdays, LocaleAllWeekdays. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleAllAbbrMonths

The predefined set `LocaleAllAbbrMonths` contains the abbreviated names of all months according the current system locale.

```
Set LocaleAllAbbrMonths {
    Index      :  LocaleIndexAbbrMonths;
}
```

Definition:

The set `LocaleAllAbbrMonths` contains the abbreviated names of all months according to the current system locale.

Updatability:

During system startup, the set `LocaleAllAbbrMonths` is filled with the set of abbreviated month names according to the current system locale. The contents of the set cannot be modified.

Remarks:

The set `LocaleAllAbbrMonths` can be used to construct a date-time format specification as specified in Section 33.7. 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`, `AllMonths`, `LocaleAllMonths`. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleAllAbbrWeekdays

The predefined set LocaleAllAbbrWeekdays contains the abbreviated names of all weekdays according the current system locale.

```
Set LocaleAllAbbrWeekdays {
    Index      :  LocaleIndexAbbrWeekdays;
}
```

Definition:

The set LocaleAllAbbrWeekdays contains the abbreviated names of all weekdays according to the current system locale.

Updatability:

During system startup, the set LocaleAllAbbrWeekdays is filled with the set of abbreviated weekday names according to the current system locale. The contents of the set cannot be modified.

Remarks:

The set LocaleAllAbbrWeekdays can be used to construct a date-time format specification as specified in Section 33.7. 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, AllWeekdays, LocaleAllWeekdays. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleAllMonths

The predefined set `LocaleAllMonths` contains the unabbreviated names of all months according the current system locale.

```
Set LocaleAllMonths {
    Index      :  LocaleIndexMonths;
}
```

Definition:

The set `LocaleAllMonths` contains the unabbreviated names of all months according to the current system locale.

Updatability:

During system startup, the set `LocaleAllMonths` is filled with the set of unabbreviated month names according to the current system locale. The contents of the set cannot be modified.

Remarks:

The set `LocaleAllMonths` can be used to construct a date-time format specification as specified in Section 33.7. 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`, `AllMonths`, `LocaleAllAbbrMonths`. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleAllWeekdays

The predefined set LocaleAllWeekdays contains the unabbreviated names of all weekdays according the current system locale.

```
Set LocaleAllWeekdays {
    Index      :  LocaleIndexWeekdays;
}
```

Definition:

The set LocaleAllWeekdays contains the unabbreviated names of all weekdays according to the current system locale.

Updatability:

During system startup, the set LocaleAllWeekdays is filled with the set of unabbreviated weekday names according to the current system locale. The contents of the set cannot be modified.

Remarks:

The set LocaleAllWeekdays can be used to construct a date-time format specification as specified in Section 33.7. 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, AllWeekdays, LocaleAllAbbrWeekdays. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleLongDateFormat

The predefined string parameter `LocaleLongDateFormat` 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` 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` is computed on the basis of the information in the current system locale. The contents of the string parameter cannot be modified.

Remarks:

The string parameter `LocaleLongDateFormat` 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`, `LocaleTimeFormat`. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleShortDateFormat

The predefined string parameter `LocaleShortDateFormat` 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` 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` is computed on the basis of the information in the current system locale. The contents of the string parameter cannot be modified.

Remarks:

The string parameter `LocaleShortDateFormat` 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`, `LocaleTimeFormat`. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleTimeFormat

The predefined string parameter LocaleTimeFormat contains the AIMMS date-time format equivalent with the time format specified in the current system locale.

```
StringParameter LocaleTimeFormat;
```

Definition:

The string parameter LocaleTimeFormat 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 is computed on the basis of the information in the current system locale. The contents of the string parameter cannot be modified.

Remarks:

The string parameter LocaleTimeFormat 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, LocaleShortDateFormat. Calendars are discussed in full detail in Section 33.2 of the Language Reference, date-time formats in Section 33.7.

LocaleTimeZoneName

The predefined string parameter LocaleTimeZoneName contains the local name of all standard time zones.

```
StringParameter LocaleTimeZoneName {
     IndexDomain :  IndexTimeZones;
}
```

See also:

The predeclared identifier LocaleTimeZoneNameDST contains the local name of all daylight saving time zones.

LocaleTimeZoneNameDST

The predefined string parameter `LocaleTimeZoneNameDST` contains the local name of all daylight saving time zones.

```
StringParameter LocaleTimeZoneNameDST {
    IndexDomain  :  IndexTimeZones;
}
```

See also:

The predeclared identifier `LocaleTimeZoneName` contains the local name of all standard time zones.

Chapter 37

Error Handling Related Identifiers

The following collection of predefined identifiers contains data regarding the Error Handling functions.

- `errh::PendingErrors`
- `errh::ErrorCodes`
- `errh::AllErrorCategories`
- `errh::AllErrorSeverities`

errh::PendingErrors

The predefined set `errh::PendingErrors` 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.

errh::ErrorCodes

The predefined set `errh::ErrorCodes` 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.

errh::AllErrorCategories

The predefined set `errh::AllErrorCategories` 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.
  - **Licensing:** Protecting the intellectual property of the developers of the AIMMS system.
  - **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.

errh::AllErrorSeverities

The predefined set `errh::AllErrorSeverities` 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.

Updatability:

The contents of this set cannot be modified.

Part IX

Suffices

Chapter 38

Common Suffices

The following collection of suffices are common to all identifier types.

- .dim
- .txt
- .type
- .unit

38.1 Example

These suffixes are typically appended to an index into the set AllIdentifiers 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:

```
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
        si.txt, /              ! Documenting text.
endfor ;

putclose ;
```

Remarks:

Note that the suffixes .dim, .txt and .type are deprecated.
See also Section 25.4 of the Language Reference.

.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.

Remarks:

- This suffix is deprecated and it is advised to use the intrinsic function `IdentifierDimension` instead.
- See also Section 25.4 of the Language Reference.

.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.

Remarks:

- This suffix is typically used with an index into the set AllIdentifiers, as illustrated in the common example on page 1103.
- See also Section 25.4 of the Language Reference.
- The GAMS equivalent name is .ts.
- This suffix is deprecated.

.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`.

Dimension:

The `.type` suffix is scalar.

Remarks:

- This suffix is typically used with an index into the set `AllIdentifiers`, as illustrated in the common example on page 1103.
- See also Section 25.4 of the Language Reference.
- This suffix is deprecated, see `IdentifierType`.

.unit

Definition:

The `.unit` suffix returns the unit of the identifier at hand.

Datatype:

The datatype of the u̇nit suffix is string

Dimension:

The `.unit` suffix is scalar.

Remarks:

- This suffix is typically used with an index into the set `AllIdentifiers`, as illustrated in the common example on page 1103.
- See also the function `IdentifierUnit`
- See also Section 25.4 of the Language Reference.

Chapter 39

Horizon Suffices

The collection of suffices available to a horizon are.

- `.past`
- `.planning`
- `.beyond`

See also section 33.3 of the Language Reference.

.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.

Remarks:

See also section 33.3 of the Language Reference.

.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.

Remarks:

See also section 33.3 of the Language Reference.

.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.

Remarks:

See also section 33.3 of the Language Reference.

Chapter 40

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:

- `.Basic`
- `.Level`
- `.Lower`
- `.Stochastic`
- `.Upper`
- `.Violation`
- `.ExtendedConstraint`
- `.ExtendedVariable`

.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.

Dimension:

The .Basic suffix has the same dimension and domain as that of the constraint or variable at hand.

Remarks:

- 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 Section 14.1 of the Language Reference

.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.

Remarks:

- When a variable without a suffix is used inside an assignment statement or a parameter definition the `.Level` suffix is automatically used.
- See also Section 14.2.5 of the Language Reference.
- The GAMS and AIMMS 2 equivalent suffix name is `.l`.

.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.

Remarks:

- When the .lower suffix of a variable is equal to the .upper suffix (see .Upper) 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 14.1).
- In order to access the lower bound of the definition of a variable X use the notation X_definition.Lower.
- See also Sections 14.1 and 14.2.5 of the Language Reference.
- The GAMS and AIMMS 2 equivalent suffix name is .lo.

.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` 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.

Remarks:

- See also Chapter 19 of the Language Reference.

.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.

Remarks:

- When the .Lower suffix (see .Lower) 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 14.1).
- In order to access the upper bound of the definition of a variable X use the notation X_definition.Upper.
- See also Sections 14.1 and 14.2.5 of the Language Reference.
- The GAMS and AIMMS 2 equivalent suffix name is .up.

.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.

Remarks:

- 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 Section 15.4.2 of the Language Reference and `.DefinitionViolation`.

.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.

Remarks:

- See also Section 16.3.6 of the Language Reference.

.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` followed by the domain of the identifier at hand.

Remarks:

- See also Section 16.3.6 of the Language Reference.

Chapter 41

Variable Suffices

AIMMS variables support the following collection of suffixes.

- .ReducedCost
- .Nonvar
- .Relax
- .Complement
- .DefinitionViolation
- .Derivative
- .Priority
- .SmallestCoefficient
- .NominalCoefficient
- .LargestCoefficient
- .SmallestValue
- .LargestValue

.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.

Remarks:

- The GAMS equivalent suffix name is .m.
- The default of the option Always_store_marginals is off.
- See also Section 14.1 of the Language Reference.

.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.

Remarks:

- 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 Section 14.1 of the Language Reference.
- The AIMMS 2 equivalent suffix name is `.freeze`.
- 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`.

.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.

Remarks:

- See also Section 14.1 of the Language Reference.

.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.

Remarks:

- The `Complement` suffix is only applicable for complementarity variables.
- See also Section 23.1 of the Language Reference.

.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.

Remarks:

- See also section 15.4 of the Language Reference.

.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.

Remarks:

- See also section 11.4.1 of the Language Reference.

.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.

Remarks:

- See also Section 14.1 of the Language Reference.
- The GAMS equivalent suffix name is `.prior`.

.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.

Remarks:

- The default of the option Calculate_Sensitivity_Ranges is on.
- See also Section 14.1 of the Language Reference.

.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.

Remarks:

- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Section 14.1 of the Language Reference.

.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.

Remarks:

- The default of the option Calculate_Sensitivity_Ranges is on.
- See also Section 14.1 of the Language Reference.

.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.

Remarks:

- The default of the option `Calculate_Sensitivity_Ranges` is `on`.
- See also Section 14.1 of the Language Reference.

.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.

Remarks:

- The default of the option Calculate_Sensitivity_Ranges is on.
- See also Section 14.1 of the Language Reference.

Chapter 42

Constraint Suffices

AIMMS constraints support the following collection of suffices.

- `.ShadowPrice`
- `.Convex`
- `.RelaxationOnly`
- `.SmallestShadowPrice`
- `.LargestShadowPrice`
- `.SmallestRightHandSide`
- `.NominalRightHandSide`
- `.LargestRightHandSide`

See also Section 14.2 of the Language Reference.

.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.

Remarks:

- 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 Section 14.2 of the Language Reference.

.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.

Remarks:

- See also Section 14.2.6 of the Language Reference.

.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.

Remarks:

- See also Section 14.2.6 of the Language Reference.

.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.

Remarks:

- 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 Section 14.2 of the Language Reference.

.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.

Remarks:

- 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 Section 14.2 of the Language Reference.

.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.

Remarks:

- 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 Section 14.2 of the Language Reference.

.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.

Remarks:

- 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 Section 14.2 of the Language Reference.

.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.

Remarks:

- 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 Section 14.2 of the Language Reference.

Chapter 43

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
- .cutoff
- .domlim
- .iterlim
- .limrow
- .nodlim
- .optca
- .optcr
- .reslim
- .tolinfrep
- .workspace

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
- .ProgramStatus
- .SolverCalls
- .objective
- .incumbent
- .BestBound
- .GenTime
- .SolutionTime
- .Iterations
- .NumberOfBranches
- .NumberOfConstraints
- .NumberOfFails
- .NumberOfNonzeros
- .NumberOfVariables
- .NumberOfInfeasibilities

- .SumOfInfeasibilities

The third group of suffixes control which AIMMS procedure should be called during the solution process and whether this calling should take place.

- .CallbackProcedure
- .CallbackIterations
- .CallbackTime
- .CallbackStatusChange
- .CallbackIncumbent
- .CallbackReturnStatus
- .CallbackAddCut
- .CallbackAOA

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

.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.

Remarks:

- The suffix `.bratio` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix `.cutoff` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix `.domlim` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix `.iterlim` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix `.limrow` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix `.nodlim` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix .optca is initialized to NA. Aimms considers it specified when its value is not equal to NA.

.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.

Remarks:

- The suffix .optcr is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

.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.

Remarks:

- The suffix `.optcr` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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.

Remarks:

- The suffix .tolinfrep is initialized to NA. AIMMS considers it specified when its value is not equal to NA.

.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.

Remarks:

- The suffix `.workspace` is initialized to `NA`. AIMMS considers it specified when its value is not equal to `NA`.

.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`.

Remarks:

- The related GAMS and AIMMS 2 name is `.SolveStat` but that value is a numeric code.
- The `.SolverStatus` suffix is also mentioned in Table 15.3 of the Language Reference.

.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`.

Remarks:

- The related GAMS and AIMMS 2 name is `.modelstat` but that value is a numeric code.
- The `.ProgramStatus` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.number`.
- The `.SolverCalls` suffix is also mentioned in Table 15.5 of the Language Reference.

.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`.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.objval`.
- The `.objective` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The `.Incumbent` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The .BestBound suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The equivalent `GAMS` and AIMMS 2 name is `.nodusd`.
- The `.Nodes` suffix is also mentioned in Table 15.3.

.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.

Remarks:

- 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 15.3.

.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.

Remarks:

- 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 15.3 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.itrusd`.
- The `.Iterations` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The `.NumberOfBranches` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.numequ`.
- The `.NumberOfConstraints` suffix is also mentioned in Table 15.5 of the Language Reference.

.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.

Remarks:

- The `.NumberOfFails` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.numnz`.
- The `.NumberOfNonzeros` suffix is also mentioned in Table 15.5 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.numvar`.
- The `.NumberOfVariables` suffix is also mentioned in Table 15.5 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.numinfes`.
- The `.NumberOfInfeasibilities` suffix is also mentioned in Table 15.3 of the Language Reference.

.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.

Remarks:

- The GAMS and AIMMS 2 equivalent name is `.suminfes`.
- The `.SumOfInfeasibilities` suffix is also mentioned in Table 15.3 of the Language Reference.

.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` and the default is the empty element `''`.

Remarks:

- See also Section 15.2 of the Language Reference.

.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.

Remarks:

- See also Section 15.2 of the Language Reference.

.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` and the default is the empty element `''`.

Remarks:

- See also Section 15.2 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`.

.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` and the default is the empty element `''`.

Remarks:

- See also Section 15.2 of the Language Reference.

.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` and the default is the empty element `''`.

Remarks:

- See also Section 15.2 of the Language Reference.

.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`.

Remarks:

- See also Section 15.2 of the Language Reference.

.CallbackAOA

Definition:

The mathematical program suffix `.CallbackAOA` contains the name of the AIMMS procedure to be called by the `AOA` open solver.

Datatype:

The value of the `.CallbackAOA` suffix is an element in the set of `AllProcedures` and the default is the empty element `''`.

Remarks:

- See also Section 15.2 of the Language Reference.

.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` and the default is the empty element `''`.

Remarks:

- See also Section 15.2 of the Language Reference.

Chapter 44

File Suffices

AIMMS files support the following three collections of suffices.
File suffix group 1: the suffixes that apply to the entire file.

- .Ap
- .blankzeros
- .case
- .PageNumber
- .PageMode
- .PageSize
- .PageWidth

File suffix group 2: the suffixes that control page layout.

- .TopMargin
- .LeftMargin
- .BottomMargin
- .BodyCurrrentColumn
- .BodyCurrentRow
- .BodySize
- .FooterCurrentColumn
- .FooterCurrentRow
- .FooterSize
- .HeaderCurrentColumn
- .HeaderCurrentRow
- .HeaderSize

File suffix group 3: the suffixes that control the formatting of individual elements.

- .lj
- .lw
- .nd
- .nj
- .nr
- .nw
- .nz
- .sj

- .sw
- .tf
- .tj
- .tw

.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

Remarks:

- The file attribute *mode* should be used instead.

.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.

.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.

.PageNumber

Definition:

The file suffix `.PageNumber` contains the number of the current page.

Datatype:

The value of the `.PageNumber` suffix is numeric.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.lp`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is *.pc* but this value is numeric.
- See also Section 31.4 of the Language Reference.

.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}.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.ps`.
- See also Section 31.4 of the Language Reference.

.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}.

Remarks:

- 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 Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name up to AIMMS 3.3 is *.tm*.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name up to AIMMS 3.3 is `.lm`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name up to AIMMS 3.3 is `.bm`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.cc`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.cr`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.ll`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.ftcc`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.ftcr`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.ftll`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.hdcc`.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is .hdcr.
- See also Section 31.4 of the Language Reference.

.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.

Remarks:

- The equivalent GAMS and AIMMS 2 name is `.hdll`.
- See also Section 31.4 of the Language Reference.

.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

Remarks:

- 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.

.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.

Remarks:

- 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.

.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.

Remarks:

- 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.

.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

Remarks:

- 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.

.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

Remarks:

- 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.

.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.

Remarks:

- 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.

.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.

Remarks:

- 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.

.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

Remarks:

- 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.

.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.

Remarks:

- 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.

.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.

Remarks:

- The suffix `.tf` is a legacy from GAMS and AIMMS 2.

.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

Remarks:

- 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.

.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.

Remarks:

- 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.

Part X

Deprecated

Chapter 45

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.

45.1 Deprecated keywords

The keywords for which direct replacements are available are documented in Table 45.1.

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

Table 45.1: AIMMS deprecated keywords and their modern equivalents

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 Section 31.3,
- halt See Section 8.3.6, and
- raise error See Section 8.4.2.

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 45.2.

Deprecated	Modern equivalent
.date	CurrentToString("%Am|AllAbbrMonths| %d, %c%y")
.time	CurrentToString("%H:%M:%S")
.version	AimmsRevisionString(string parameter, 4);
.page	currentOutputFile.PageNumber

Table 45.2: The keyword system and selected suffixes with their modern counterparts

The system suffixes .ifile, .ofile, .rdate, .rfile, .rtime, .sfile, and .title are pointless within the AIMMS environment.

45.2 Deprecated intrinsic procedures and functions

The mapping of the matrix manipulation procedures to GMP procedures and functions is documented in Table 46.1 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`.
- One may replace `SQLDirect` with `DirectSQL`
- One may replace `StringToLabel` with `StringToElement`

The deprecated iterative operators are documented in Table 45.3.

Deprecated	Modern equivalent
`smax`	`max`
`smin`	`min`
`arg`	`nth`

Table 45.3: AIMMS deprecated iterative operators and their modern equivalents

45.3 Deprecated suffixes

Most deprecated suffixes can be directly translated into their modern equivalents, as documented in Table 45.4.
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 Section 31.1.
- `.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 Table 15.6 and `AllSolutionStates`.
- `.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 Table 15.6 and `AllSolutionStates`.

- `.dim` This should be replaced by a call to `IdentifierDimension`.
- `.txt` This should be replaced by a call to `IdentifierText`.
- `.type` This should be replaced by a call to `IdentifierType`.

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

Table 45.4: Aimms deprecated suffixes and their modern equivalents

Chapter 46

Matrix Manipulation Functions

AIMMS supports the following matrix manipulation functions:

- MatrixActivateRow
- MatrixAddColumn
- MatrixAddRow
- MatrixDeactivateRow
- MatrixFreezeColumn
- MatrixGenerate
- MatrixModifyCoefficient
- MatrixModifyColumnType
- MatrixModifyDirection
- MatrixModifyLeftHandSide
- MatrixModifyLowerBound
- MatrixModifyQuadraticCoefficient
- MatrixModifyRightHandSide
- MatrixModifyRowType
- MatrixModifyType
- MatrixModifyUpperBound
- MatrixRegenerateRow
- MatrixRestoreState
- MatrixSaveState
- MatrixSolve
- MatrixUnfreezeColumn

In addition, the following function can be used the add cuts during the solution process of a mixed integer program:

- GenerateCut

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP functions.
AIMMS versions prior to version 3.5, also supported a collection of matrix manipulation procedures with more limited functionality. Although these procedures will remain to be supported for all AIMMS 3.x versions, they

have become deprecated. The deprecated manipulation procedures and their GMP counterparts in AIMMS 3.5 and higher are listed in Table 46.1.

Deprecated procedure	GMP counterpart
MatrixModifyCoefficient	GMP::Coefficient::Set
MatrixModifyQuadraticCoefficient	GMP::Coefficient::SetQuadratic
MatrixModifyRightHandSide	GMP::Row::SetRightHandSide
MatrixModifyLeftHandSide	GMP::Row::SetLeftHandSide
MatrixModifyRowType	GMP::Row::SetType
MatrixAddRow	GMP::Row::Add
MatrixRegenerateRow	GMP::Row::Generate
MatrixDeactivateRow	GMP::Row::Deactivate
MatrixActivateRow	GMP::Row::Activate
MatrixModifyLowerBound	GMP::Column::SetLowerBound
MatrixModifyUpperBound	GMP::Column::SetUpperBound
MatrixModifyColumnType	GMP::Column::SetType
MatrixAddColumn	GMP::Column::Add
MatrixFreezeColumn	GMP::Column::Freeze
MatrixUnfreezeColumn	GMP::Column::Unfreeze
MatrixModifyType	GMP::Instance::SetMathematicalProgrammingType
MatrixModifyDirection	GMP::Instance::SetDirection
MatrixGenerate	GMP::Instance::Generate
MatrixSolve	GMP::Instance::Solve, GMP::SolverSession::Execute
MatrixSaveState	GMP::Instance::Copy
MatrixRestoreState	GMP::Instance::Copy

Table 46.1: Deprecated matrix manipulation procedures

MatrixActivateRow

The procedure `MatrixActivateRow` activates a row in the matrix that was previously deactivated.

```
MatrixActivateRow(
     MP,            ! (input) a mathematical program
     row            ! (input) a scalar value
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row in the matrix; this can not be the objective row.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of Aimms release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the gmp library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding gmp procedures.

See also:

The procedure `MatrixDeactivateRow`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixAddColumn

The procedure `MatrixAddColumn` adds a column to the matrix.

```
MatrixAddColumn(
     MP,              ! (input) a mathematical program
     column           ! (input) a scalar value
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

column
: A scalar reference to an existing column name in the model.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- Coefficients for this column can be added to the matrix by using the procedure `MatrixModifyCoefficient`. After calling `MatrixAddColumn` 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 `MatrixModifyColumnType`, `MatrixModifyLowerBound` and `MatrixModifyUpperBound` the column type and the lower and upper bound can be changed.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixModifyCoefficient`, `MatrixModifyColumnType`, `MatrixModifyLowerBound`, `MatrixModifyUpperBound`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixAddRow

The procedure `MatrixAddRow` adds a row to the matrix.

```
MatrixAddRow(
     MP,            ! (input) a mathematical program
     row            ! (input) a scalar value
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row name in the model.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- Initially, the row will added with zero coefficients, regardless of whether the symbolic AIMMS constraint has a definition or not. Regeneration of all of the coefficients of the row according to its definition can be achieved through the procedure `MatrixRegenerateRow`. Individual coefficients of the row can be added by calling the procedure `MatrixModifyCoefficient`.
- After calling `MatrixAddRow` the type of the row is set to '<=' and the right-hand-side value to `INF` (the left-hand-side value is set to `-INF`). By using the procedures `MatrixModifyRowType` and `MatrixModifyRightHandSide` the row type and right-hand-side value can be changed.
- After a call to `MatrixAddRow` or `MatrixRegenerateRow` for a certain row it is not allowed to do another call to `MatrixAddRow` for that row.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixModifyCoefficient`, `MatrixModifyLeftHandSide`, `MatrixModifyRightHandSide`, `MatrixModifyRowType`, `MatrixRegenerateRow`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixDeactivateRow

The procedure `MatrixDeactivateRow` deactivates a row in the matrix. The row will be ignored by the solver until it is activated again.

```
MatrixDeactivateRow(
     MP,            ! (input) a mathematical program
     row            ! (input) a scalar value
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row in the matrix; this can not be the objective row.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- Deactivating a row results in changing the type of that row into '<' and the right hand side value into `INF` (the row coefficients do not change).
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedure `MatrixActivateRow`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixFreezeColumn

The procedure `MatrixFreezeColumn` fixes the value of a column in the model. The column can be freed by using `MatrixUnfreezeColumn`.

```
MatrixFreezeColumn(
     MP,             ! (input) a mathematical program
     column,         ! (input) a scalar value
     value           ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing column in the matrix.

value
: The value to which the column should be fixed.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- Fixing a column to a certain value has the same effect as changing the lower and upper bound into that value.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixModifyLowerBound`, `MatrixModifyUpperBound`, `MatrixUnfreezeColumn`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixGenerate

The procedure `MatrixGenerate` instructs AIMMS to generate a mathematical program without actually solving it.

```
MatrixGenerate(
     MP             ! (input) a mathematical program
     )
```

Arguments:

MP
: A mathematical program to be generated. The mathematical program should be a linear, mixed-integer linear or quadratic programming model.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- The procedure `MatrixGenerate` can be used to generate a mathematical program, if your algorithm does not call the `SOLVE` statement to solve it initially, prior using the matrix manipulation routines.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyCoefficient

The procedure `MatrixModifyCoefficient` changes a coefficient in the matrix. This procedure can also be used to modify a coefficient in the objective row. The value for the coefficient can be equal to 0.0 prior to calling this procedure.

```
MatrixModifyCoefficient(
     MP,            ! (input) a mathematical program
     row,           ! (input) a scalar value
     column,        ! (input) a scalar value
     value          ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row in the matrix; this might be the objective row.

column
: A scalar reference to an existing column in the matrix.

value
: The new value that should be assigned to the coefficient corresponding to *row* and *column* in the matrix. This value should be unequal to `NA`, `INF`, `-INF` and `UNDF`.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

As of Aimms release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyColumnType

The procedure `MatrixModifyColumnType` changes the type of a column in the matrix into either `'continuous'` or `'integer'`.

```
MatrixModifyColumnType(
     MP,            ! (input) a mathematical program
     column,        ! (input) a scalar value
     type           ! (input) a column type
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

column
: A scalar reference to an existing column in the matrix.

type
: One of the column types `'continuous'` or `'integer'` that should be assigned to the column.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyDirection

The procedure `MatrixModifyDirection` changes the direction of a mathematical program to `'maximize'`, `'minimize'` or `'none'`. The direction `'none'` is the instruction to the solver to find a feasible solution. If the type of the mathematical program is `'MIP'` then the solver will try to find an integer feasible solution.

```
MatrixModifyDirection(
     MP,             ! (input) a mathematical program
     direction       ! (input) a direction
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

direction
: One of the directions `'maximize'`, `'minimize'` or `'none'`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The set `AllMatrixManipulationDirections`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyLeftHandSide

The procedure `MatrixModifyLeftHandSide` changes the left-hand-side of a row in the matrix.

```
MatrixModifyLeftHandSide(
     MP,            ! (input) a mathematical program
     row,           ! (input) a scalar value
     value          ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing ranged row in the matrix.

value
: The new value that should be assigned to the left-hand-side of the row. This value should be unequal to `NA`, `UNDF` and `INF` (but might be `-INF`).

Remarks:

- After a call to `MatrixSolve` AIMMS checks for each modified ranged row whether or not the left-hand-side value is valid, that is, the left-hand-side value should be unequal to `INF`.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixModifyRightHandSide`, `MatrixSolve`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyLowerBound

The procedure `MatrixModifyLowerBound` changes the lower bound of a column in the matrix.

```
MatrixModifyLowerBound(
     MP,            ! (input) a mathematical program
     column,        ! (input) a scalar value
     value          ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing column in the matrix.

value
: The new value that should be assigned to the lower bound of the column.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedure `MatrixModifyUpperBound`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyQuadraticCoefficient

The procedure `MatrixModifyQuadraticCoefficient` changes a quadratic coefficient in the objective row of a quadratic mathematical program. The value for the coefficient can be equal to 0.0 prior to calling this procedure.

```
MatrixModifyQuadraticCoefficient(
     MP,            ! (input) a mathematical program
     col1,          ! (input) a scalar value
     col2,          ! (input) a scalar value
     value          ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a quadratic programming model.

col1
: A scalar reference to an existing column.

col2
: A scalar reference to an existing column.

value
: The new value that should be assigned to the quadratic coefficient corresponding to *col1* and *col2* in the objective row. This value should be unequal to `NA`, `INF`, `-INF` and `UNDF`.

Return value:

The procedure returns 1 on success, and 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyRightHandSide

The procedure `MatrixModifyRightHandSide` changes the right-hand-side of a row in the matrix.

```
MatrixModifyRightHandSide(
     MP,            ! (input) a mathematical program
     row,           ! (input) a scalar value
     value          ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row in the matrix; this can not be the objective row.

value
: The new value that should be assigned to the right-hand-side of the row. This value should be unequal to `NA` and `UNDF` (but might be `INF` or `-INF`).

Remarks:

- If you assign INF to the right-hand-side value of a row with type '=', `MatrixModifyRightHandSide` will not produce an error, since you might want to change the type of this row into '<=' (using `MatrixModifyRowType`) immediately thereafter.
- After a call to `MatrixSolve` AIMMS checks for each modified row whether or not the right-hand-side value is valid for the current row type. If the row type is '=' then the right-hand-side value should be unequal to `INF` and `-INF`; if the row type is '<=' or 'ranged' then it should be unequal to `-INF`.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixModifyLeftHandSide`, `MatrixModifyRowType`, `MatrixSolve`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyRowType

The procedure `MatrixModifyRowType` changes the type of a row in the matrix.

```
MatrixModifyRowType(
     MP,            ! (input) a mathematical program
     row,           ! (input) a scalar value
     type           ! (input) a row type
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row in the matrix; this can not be the objective row.

type
: One of the row types `'<='`, `'='`, `'>='` or `'ranged'` that should be assigned to the row.

Remarks:

- The following examples show what happens if we change the row type into `'ranged'`:

`a(x) <= 3`	modified into `'ranged'` results in	`-inf <= a(x) <= 3`
`a(x) >= 3`	modified into `'ranged'` results in	`3 <= a(x) <= inf`
`a(x)  = 3`	modified into `'ranged'` results in	`3 <= a(x) <= 3`

  The next examples show what happens if we change the row type of a `'ranged'` row:

`2 <= a(x) <= 4`	modified into `'<='` results in	`a(x) <= 4`
`2 <= a(x) <= 4`	modified into `'>='` results in	`a(x) >= 2`
`2 <= a(x) <= 4`	modified into `'='` results in	`a(x)  = 4`

- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyType

The procedure `MatrixModifyType` changes the type of a mathematical program from `'MIP'` into `'RMIP'`, or vice versa.

```
MatrixModifyType(
     MP,            ! (input) a mathematical program
     type           ! (input) a mathematical programming type
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

type
: One of the types `'MIP'` or `'RMIP'`.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixModifyUpperBound

The procedure `MatrixModifyUpperBound` changes the upper bound of a column in the matrix.

```
MatrixModifyUpperBound(
     MP,             ! (input) a mathematical program
     column,         ! (input) a scalar value
     value           ! (input) a numerical expression
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

column
: A scalar reference to an existing column in the matrix.

value
: The new value that should be assigned to the upper bound of the column.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedure `MatrixModifyLowerBound`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixRegenerateRow

The procedure `MatrixRegenerateRow` regenerates the coefficients of a row according to the definition of its associated symbolic constraint in the model.

```
MatrixRegenerateRow(
     MP,            ! (input) a mathematical program
     row            ! (input) a scalar value
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

row
: A scalar reference to an existing row name in the model.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- If the row does not exist yet, it will be automatically added to the matrix before generating its coefficients.
- Before regenerating the row, the procedure first removes all existing matrix coefficients.
- This procedure will automatically add columns that are not in the matrix.
- 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.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixAddRow`, `MatrixModifyCoefficient`, `MatrixModifyLeftHandSide`, `MatrixModifyRightHandSide`, `MatrixModifyRowType`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixRestoreState

With procedure `MatrixRestoreState` you can restore the state of your mathematical program as it was on the moment that you called `MatrixSaveState`.

```
MatrixRestoreState(
     MP,            ! (input) a mathematical program
     state          ! (input) an integer scalar parameter
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

state
: The value corresponding to a state that you want to restore.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedure `MatrixSaveState`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixSaveState

With the procedure `MatrixSaveState` you can save the current state of a mathematical program. Later on, after manipulating the mathematical program, you can restore this state by calling `MatrixRestoreState`.

```
MatrixSaveState(
     MP,             ! (input)  a mathematical program
     state           ! (output) an integer scalar parameter
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

state
: On return, contains a positive integer value assigned to the state.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- States are numbered from 1 upwards by AIMMS.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedure `MatrixRestoreState`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixSolve

The procedure MatrixSolve instructs the solver to solve a mathematical program in its current state after being modified by using several matrix manipulation procedures.

```
MatrixSolve(
     MP             ! (input) a mathematical program
     )
```

Arguments:

MP

A mathematical program that was previously solved or generated. The mathematical program should be a linear, mixed-integer linear or quadratic programming model.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- After a call to MatrixSolve AIMMS will first check if all modifications performed by calling matrix manipulation procedures are all valid, before actually calling the solver. Most errors, however, will be filtered out by the matrix manipulation procedures themselves.
- As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedure MatrixGenerate. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

MatrixUnfreezeColumn

The procedure `MatrixUnfreezeColumn` frees a column that was fixed with `MatrixFreezeColumn`. After calling `MatrixUnfreezeColumn` the value of the column can vary again between its lower and upper bound.

```
MatrixUnfreezeColumn(
     MP,              ! (input) a mathematical program
     column           ! (input) a scalar value
     )
```

Arguments:

MP
: A mathematical program that was previously solved. The mathematical program should be a linear or mixed-integer linear programming model.

column
: A scalar reference to an existing fixed column in the matrix.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

As of AIMMS release 3.5, the matrix manipulation procedures have become deprecated. New projects should use the GMP library instead. Please refer to Table 46.1 of the Language Reference for a mapping of the matrix manipulation procedures to corresponding GMP procedures.

See also:

The procedures `MatrixFreezeColumn`, `MatrixModifyLowerBound`, `MatrixModifyUpperBound`. Matrix manipulation routines are discussed in more detail in Chapter 16 of the Language Reference.

GenerateCut

The procedure GenerateCut adds a row to the matrix during the solution process of a mixed integer proghram.

```
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.

Remarks:

- 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 Section 15.2 of the Language Reference for more details on how to install a callback procedure to add cuts.

Chapter 47

Outer Approximation Functions

The AIMMS Outer Approximation functions allow you to solve MINLP problems through a sequence of MIP and NLP solves. The following Outer Approximation functions are available.

Master MIP functions

AIMMS supports the following Outer Approximation functions for solving and managing the master MIP problem:

- MasterMIPAddLinearizations
- MasterMIPDeleteIntegerEliminationCut
- MasterMIPDeleteLinearizations
- MasterMIPEliminateIntegerSolution
- MasterMIPGetCPUTime
- MasterMIPGetIterationCount
- MasterMIPGetNumberOfColumns
- MasterMIPGetNumberOfNonZeros
- MasterMIPGetNumberOfRows
- MasterMIPGetObjectiveValue
- MasterMIPGetProgramStatus
- MasterMIPGetSolverStatus
- MasterMIPGetSumOfPenalties
- MasterMIPIsFeasible
- MasterMIPLinearizationCommand
- MasterMIPSetCallback
- MasterMIPSolve

MINLP functions

AIMMS supports the following Outer Approximation functions for managing the global MINLP problem:

- MINLPGetIncumbentObjectiveValue
- MINLPGetOptimizationDirection
- MINLPIncumbentIsFeasible
- MINLPIncumbentSolutionHasBeenFound
- MINLPSetIncumbentSolution
- MINLPSetIterationCount
- MINLPSetProgramStatus
- MINLPSolutionDelete

- MINLPSolutionRetrieve
- MINLPSolutionSave

NLP functions

AIMMS supports the following Outer Approximation functions for managing and solving the NLP problem:

- NLPGetCPUTime
- NLPGetIterationCount
- NLPGetNumberOfColumns
- NLPGetNumberOfNonZeros
- NLPGetNumberOfRows
- NLPGetObjectiveValue
- NLPGetProgramStatus
- NLPGetSolverStatus
- NLPIsFeasible
- NLPLinearizationPointHasBeenFound
- NLPSolutionIsInteger
- NLPSolve

MasterMIPAddLinearizations

The procedure `MasterMIPAddLinearizations` adds a linearization for a subset of `AllNonlinearConstraints`. The linearizations are created by using the solution present at that time inside the AIMMS Outer Approximation solver interface. Normally the solution that is returned by the NLP solver is used.
When permitted, variables are introduced to allow for deviations from each linearized constraint. These deviation variables are penalized in the objective function using the penalty multipliers times the corresponding shadow prices (Lagrange multipliers). The procedure returns the updated linearization counter in the output argument n.

```
MasterMIPAddLinearizations(
     IncludedConstraints, ! (input) subset of the set AllNonlinearConstraints
     DeviationsPermitted, ! (input) 0-1 parameter over AllNonlinearConstraints
     PenaltyMultiplier,   ! (input) parameter over AllNonlinearConstraints
     n                    ! (output) integer scalar parameter
     )
```

Arguments:

IncludedConstraints
: Set of nonlinear constraints for which linearizations have to be added.

DeviationsPermitted
: Parameter that indicates whether or not variables should be introduced to allow for deviations from each linearized constraint. If so, the corresponding entry in this parameter should be 1, otherwise 0.

PenaltyMultiplier
: The deviation variables (if any) are penalized in the objective function by using the values in this parameter times the corresponding shadow prices (Lagrange multipliers).

n
: The updated linearization counter.

Return value:

`MasterMIPAddLinearizations` has no return value.

MasterMIPDeleteIntegerEliminationCut

The procedure `MasterMIPDeleteIntegerEliminationCut` deletes a set of integer solution elimination cuts and variables that was previously added by the `MasterMIPEliminateIntegerSolution` procedure.

```
MasterMIPDeleteIntegerEliminationCut(
     n       ! (input) integer scalar value
     )
```

Arguments:

n

The cut counter for the set of cuts and variables that has to be deleted. It was returned by `MasterMIPEliminateIntegerSolution` when these cuts and variables were added.

Return value:

`MasterMIPDeleteIntegerEliminationCut` has no return value.

MasterMIPDeleteLinearizations

The procedure `MasterMIPDeleteLinearizations` deletes a set of linearizations that was previously added by the `MasterMIPAddLinearizations` procedure for a certain solution.

```
MasterMIPDeleteLinearizations(
     n      ! (input) integer scalar value
     )
```

Arguments:

n

The linearization counter for the set of linearizations that has to be deleted. It was returned by `MasterMIPAddLinearizations` when these linearizations were added.

Return value:

`MasterMIPDeleteLinearizations` has no return value.

MasterMIPEliminateIntegerSolution

The procedure `MasterMIPEliminateIntegerSolution` adds a set of cuts and variables to the master MIP model instance which eliminates the current integer solution inside the AIMMS Outer Approximation solver interface.

```
MasterMIPEliminateIntegerSolution(
     n       ! (output) integer scalar parameter
     )
```

Arguments:

n
: The updated cut counter.

Return value:

`MasterMIPEliminateIntegerSolution` has no return value.

Remarks:

To eliminate the current integer solution, 3 variables (2 continuous, 1 binary) and 3 constraints are added for each integer variable whose level value is between its bounds. Also one main cut constraint is added. In case all integer variables are binary, only this main cut constraint is added.

MasterMIPGetCPUTime

The function `MasterMIPGetCPUTime` returns the CPU time needed to solve the master MIP problem.

```
MasterMIPGetCPUTime
```

Arguments:

None

Return value:

The function `MasterMIPGetCPUTime` returns the double value of the CPU time (in seconds) needed to solve the last master MIP problem.

MasterMIPGetIterationCount

The function `MasterMIPGetIterationCount` returns the iteration count associated with the last master MIP problem solved.

```
MasterMIPGetIterationCount
```

Arguments:

None

Return value:

The function `MasterMIPGetIterationCount` returns the iteration count associated with the last master MIP problem solved.

MasterMIPGetNumberOfColumns

The function `MasterMIPGetNumberOfColumns` returns the number of columns in the last master MIP problem solved.

```
MasterMIPGetNumberOfColumns
```

Arguments:

None

Return value:

The function `MasterMIPGetNumberOfColumns` returns the number of columns in the last master MIP problem solved.

MasterMIPGetNumberOfNonZeros

The function `MasterMIPGetNumberOfNonZeros` returns the number of nonzeros in the last master MIP problem solved.

```
MasterMIPGetNumberOfNonZeros
```

Arguments:

None

Return value:

The function `MasterMIPGetNumberOfNonZeros` returns the number of nonzeros in the last master MIP problem solved.

MasterMIPGetNumberOfRows

The function `MasterMIPGetNumberOfRows` returns the number of rows in the last master MIP problemm solved.

```
MasterMIPGetNumberOfRows
```

Arguments:

None

Return value:

The function `MasterMIPGetNumberOfRows` returns the number of rows in the last master MIP problem solved.

MasterMIPGetObjectiveValue

The function `MasterMIPGetObjectiveValue` returns the objective value of the last solved master MIP.

```
MasterMIPGetObjectiveValue
```

Arguments:

None

Return value:

The function `MasterMIPGetObjectiveValue` returns the objective value of the last solved master MIP.

MasterMIPGetProgramStatus

The function `MasterMIPGetProgramStatus` returns the program (or model) status associated with the last master MIP problem solved.

```
MasterMIPGetProgramStatus
```

Arguments:

None

Return value:

The function `MasterMIPGetProgramStatus` returns the program (or model) status associated with the last master MIP problem solved. The return value will be an element in the set `AllSolutionStates`.

MasterMIPGetSolverStatus

The function `MasterMIPGetSolverStatus` returns the solver status associated with the last master MIP problem solved.

```
MasterMIPGetSolverStatus
```

Arguments:

None

Return value:

The function `MasterMIPGetSolverStatus` returns the solver status associated with the last master MIP problem solved. The return value will be an element in the set `AllSolutionStates`.

MasterMIPGetSumOfPenalties

The function `MasterMIPGetSumOfPenalties` returns the sum of the penalties in the solution of the last solved master MIP.

```
MasterMIPGetSumOfPenalties
```

Arguments:

None

Return value:

The function `MasterMIPGetSumOfPenalties` returns the sum of the penalties in the solution of the last solved master MIP.

MasterMIPIsFeasible

The function `MasterMIPIsFeasible` indicates whether the solution found for the last solved master MIP is feasible or not.

```
MasterMIPIsFeasible
```

Arguments:

None

Return value:

The function `MasterMIPIsFeasible` returns 1 if the solution of the last master MIP is feasible, or 0 otherwise.

MasterMIPLinearizationCommand

The procedure `MasterMIPLinearizationCommand` allows you to retrieve or modify certain aspects of the linearization of a constraint added for linearization counter `n` at the individual level. The argument `Command` specifies which data (e.g. GetDeviation) should be retrieved or modified. The retrieved or modified value is passed through the `CommandData` argument.

```
MasterMIPLinearizationCommand(
     n,                 ! (input) integer scalar value
     ModelConstraint,   ! (input) scalar value
     Command,           ! (input) element parameter into
                        !         MasterMIPLinearizationCommands
     CommandData        ! (inout) scalar value (in) or parameter (out)
     )
```

Arguments:

n

The linearization counter as returned by `MasterMIPAddLinearizations` when adding this linearization.

ModelConstraint

Scalar reference to a constraint for which certain aspects of the linearization have to be retrieved or modified.

Command

Element parameter into `MasterMIPLinearizationCommands` that specifies which data should be retrieved or modified. Possible values are:

Command	**Description**
`GetDeviation`	Get the value of the deviation variable.
`RemoveDeviation`	Delete the deviation variable.
`GetWeight`	Get the objective coefficient of the deviation variable.
`SetWeight`	Set the objective coefficient of the deviation variable.
`GetDeviationBound`	Get the upper bound of the deviation variable.
`SetDeviationBound`	Get the upper bound of the deviation variable.
`GetLagrangeMultiplier`	Get value of the shadow price (Lagrange multiplier) of constraint for last solved NLP.

CommandData

The retrieved or modified value.

Return value:

`MasterMIPLinearizationCommand` has no return value.

Remarks:

- Normally, the weight obtained with 'GetWeight' equals the value of the penalty multiplier, as passed to `MasterMIPAddLinearizations`, times the shadow price (Lagrange multiplier) of the constraint. With 'SetWeight' this weight can be changed.
- Note that 'SetWeight' can be used to create a deviation variable (slack) if the linearization does not have one. To do so the value filled in for `CommandData` should be unequal to 0.
- The lower bound of a deviation variable always equals 0.

MasterMIPSetCallback

The procedure `MasterMIPSetCallback` allows the user to set a callback procedure that will be called during the solve of the master MIP. It will be called either for every new incumbent value found by the MIP solver or after a certain number of iterations. This is determined by the argument `Iterations`.

```
MasterMIPSetCallback(
     ProcedureName,      ! (input) scalar string expression
     Iterations          ! (input) integer scalar value
     )
```

Arguments:

ProcedureName
: The name of the AIMMS procedure that will be used as callback procedure.

Iterations
: If Iterations ≥ 1 then the callback procedure will be called after this number of iterations; else it will be called for every new incumbent value found by the MIP solver.

Return value:

`MasterMIPSetCallback()` has no return value.

MasterMIPSolve

The procedure `MasterMIPSolve` calls the MIP solver to solve the master MIP problem. Any modifications that have been made since the last call to `MasterMIPSolve` will be added to the master MIP prior to solving. Examples of such modifications are additions of linearizations and cuts that eliminate integer solutions.

```
MasterMIPSolve
```

Arguments:

None

Return value:

`MasterMIPSolve()` has no return value.

MINLPGetIncumbentObjectiveValue

The function `MINLPGetIncumbentObjectiveValue` returns the objective value associated with the incumbent solution.

```
MINLPGetIncumbentObjectiveValue
```

Arguments:

None

Return value:

The function `MINLPGetIncumbentObjectiveValue` returns the objective value associated with the incumbent solution.

MINLPGetOptimizationDirection

The function `MINLPGetOptimizationDirection` returns the optimization direction: 1 for maximization and -1 for minimization.

```
MINLPGetOptimizationDirection
```

Arguments:

None

Return value:

The function `MINLPGetOptimizationDirection` returns 1 for maximization and -1 for minimization.

MINLPIncumbentIsFeasible

The function `MINLPIncumbentIsFeasible` indicates whether the current incumbent solution is feasible or not for the MINLP problem.

```
MINLPIncumbentIsFeasible
```

Arguments:

None

Return value:

The function `MINLPIncumbentIsFeasible` returns 1 if the current incumbent solution is feasible for the MINLP problem, or 0 otherwise.

MINLPIncumbentSolutionHasBeenFound

The function `MINLPIncumbentSolutionHasBeenFound` indicates whether an incumbent has already been specified.

```
MINLPIncumbentSolutionHasBeenFound
```

Arguments:

None

Return value:

The function `MINLPIncumbentSolutionHasBeenFound` returns 1 if an incumbent has already been specified, or 0 otherwise.

MINLPSetIncumbentSolution

The procedure `MINLPSetIncumbentSolution` marks the current values of the decision variables as an incumbent solution for the MINLP problem.

```
MINLPSetIncumbentSolution
```

Arguments:

None

Return value:

`MINLPSetIncumbentSolution()` has no return value.

MINLPSetIterationCount

The procedure `MINLPSetIterationCount` sets the iteration count for the MINLP problem.

```
MINLPSetIterationCount(
     IterationCount      ! (input) integer scalar value
     )
```

Arguments:

IterationCount
The iteration number that should be set for the MINLP problem.

Return value:

`MINLPSetIterationCount()` has no return value.

MINLPSetProgramStatus

The procedure `MINLPSetProgramStatus` sets the program status for the MINLP problem.

```
MINLPSetProgramStatus(
     ProgramStatus     ! (input) element parameter into AllSolutionStates
     )
```

Arguments:

ProgramStatus
: Element parameter into `AllSolutionStates` that sets the program status for the MINLP problem.

Return value:

`MINLPSetProgramStatus()` has no return value.

MINLPSolutionDelete

The procedure `MINLPSolutionDelete` deletes the solution inside the AIMMS Outer Approximation solver interface that was previously saved by a call to `MINLPSolutionSave` with solution number `n`.

```
MINLPSolutionDelete(
     n      ! (input) integer scalar value
     )
```

Arguments:

n

The solution number corresponding to the solution that has to be deleted. The solution number was passed to `MINLPSolutionSave` before to label the solution.

Return value:

`MINLPSolutionDelete` has no return value.

MINLPSolutionRetrieve

The procedure `MINLPSolutionRetrieve` retrieves the solution previously saved by a call to `MINLPSolutionSave` with solution number `n`, and stores it as the current solution inside the AIMMS Outer Approximation solver interface.

```
MINLPSolutionRetrieve(
     n       ! (input) integer scalar value
     )
```

Arguments:

n
: The solution number corresponding to the solution that has to be retrieved. The solution number was passed to `MINLPSolutionSave` before to label the solution.

Return value:

`MINLPSolutionRetrieve` has no return value.

MINLPSolutionSave

The procedure MINLPSolutionSave saves the current solution that is present inside the AIMMS Outer Approximation solver interface, and stores it as solution number n for later retrieval.

```
MINLPSolutionSave(
     n      ! (input) integer scalar value
     )
```

Arguments:

n

The solution number used to label the saved solution.

Return value:

MINLPSolutionSave has no return value.

Remarks:

If as solution was saved before with the same value for n then that solution will be replaced by this new solution.

NLPGetCPUTime

The function `NLPGetCPUTime` returns the CPU time needed to solve the last NLP subproblem.

```
NLPGetCPUTime
```

Arguments:

None

Return value:

The function `NLPGetCPUTime` returns the double value of the CPU time (in seconds) needed to solve the last NLP subproblem.

NLPGetIterationCount

The function `NLPGetIterationCount` returns the iteration count associated with the last NLP subproblem solved.

```
NLPGetIterationCount
```

Arguments:

None

Return value:

The function `NLPGetIterationCount` returns the iteration count associated with the last NLP subproblem solved.

NLPGetNumberOfColumns

The function `NLPGetNumberOfColumns` returns the number of columns in the last NLP subproblem solved.

```
NLPGetNumberOfColumns
```

Arguments:

None

Return value:

The function `NLPGetNumberOfColumns` returns the number of columns in the last NLP subproblem solved.

NLPGetNumberOfNonZeros

The function `NLPGetNumberOfNonZeros` returns the number of nonzeros in the last NLP subproblem solved.

```
NLPGetNumberOfNonZeros
```

Arguments:

None

Return value:

The function `NLPGetNumberOfNonZeros` returns the number of nonzeros in the last NLP subproblem solved.

NLPGetNumberOfRows

The function `NLPGetNumberOfRows` returns the number of rows in the last NLP subproblem solved.

```
NLPGetNumberOfRows
```

Arguments:

None

Return value:

The function `NLPGetNumberOfRows` returns the number of rows in the last NLP subproblem solved.

NLPGetObjectiveValue

The function `NLPGetObjectiveValue` returns the objective value of the last solved NLP.

```
NLPGetObjectiveValue
```

Arguments:

None

Return value:

The function `NLPGetObjectiveValue` returns the objective value of the last solved NLP.

NLPGetProgramStatus

The function `NLPGetProgramStatus` returns the program (or model) status associated with the last NLP subproblem solved.

```
NLPGetProgramStatus
```

Arguments:

None

Return value:

The function `NLPGetProgramStatus` returns the program (or model) status associated with the last NLP subproblem solved. The return value will be an element in the set `AllSolutionStates`.

NLPGetSolverStatus

The function `NLPGetSolverStatus` returns the solver status associated with the last NLP subproblem solved.

```
NLPGetSolverStatus
```

Arguments:

None

Return value:

The function `NLPGetSolverStatus` returns the solver status associated with the last NLP subproblem solved. The return value will be an element in the set `AllSolutionStates`.

NLPIsFeasible

The function `NLPIsFeasible` indicates whether the solution found for the last solved NLP is feasible or not.

```
NLPIsFeasible
```

Arguments:

None

Return value:

The function `NLPIsFeasible` returns 1 if the solution of the last NLP is feasible, or 0 otherwise.

NLPLinearizationPointHasBeenFound

The function `NLPLinearizationPointHasBeenFound` indicates whether the NLP solver has found a point that can be used to linearize the nonlinear constraints. If the NLP problem is infeasible then usually the NLP solver provides a point that solves the so-called feasibility problem (i.e., a point that minimizes the sum of the infeasibilities).

```
NLPLinearizationPointHasBeenFound
```

Arguments:

None

Return value:

The function `NLPLinearizationPointHasBeenFound` returns 1 if the NLP solver has found a point that can be used to linearize the nonlinear constraints. It returns 0 otherwise.

Remarks:

This function always returns 1 if the NLP has found a feasible solution.

NLPSolutionIsInteger

The function `NLPSolutionIsInteger` indicates whether the solution found for the last NLP is integer and feasible, or not.

```
NLPSolutionIsInteger
```

Arguments:

None

Return value:

The function `NLPSolutionIsInteger` returns 1 if the solution of the last NLP is integer feasible, or 0 otherwise.

NLPSolve

The procedure `NLPSolve` calls the NLP solver to solve the NLP subproblem in which the (symbolic) integer variables in the set FrozenVariables remain frozen during the solve, and all other integer variables are considered to be continuous between their bounds.

```
NLPSolve(
     FrozenVariables      ! (input) subset of the set AllIntegerVariables
     )
```

Arguments:

FrozenVariables
The set of (symbolic) integer variables that remain frozen during the solve of the NLP. This is a subset of `AllIntegerVariables`.

Return value:

`NLPSolve()` has no return value.

Chapter 48

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:

- Cases
- Data categories
- Datasets

48.1 Cases

- CaseDelete
- CaseFind
- CaseGetChangedStatus
- CaseGetDatasetReference
- CaseGetType
- CaseLoadCurrent
- CaseLoadIntoCurrent
- CaseMerge
- CaseNew
- CaseSave
- CaseSaveAll
- CaseSaveAs
- CaseSelect
- CaseSelectMultiple
- CaseSelectNew
- CaseSetChangedStatus
- CaseSetCurrent
- CaseReadFromSingleFile
- CaseWriteToSingleFile

CaseCreate

The procedure CaseCreate 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. On successful return this parameter will refer to the newly created element in AllCases.

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.

Remarks:

- 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, CaseDelete.

CaseDelete

The procedure CaseDelete deletes a specific case node from the Data Management tree.

```
CaseDelete(
         case    ! (input) element parameter into AllCases
         )
```

Arguments:

case
: An element parameter into AllCases, representing the case that you want to delete.

Return value:

The procedure returns 1 if the case is deleted successfully, or 0 otherwise.

Remarks:

- 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 instead.

See also:

The procedure CaseFind.

CaseFind

The procedure CaseFind 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. 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.

Remarks:

- 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, CaseDelete.

CaseGetChangedStatus

The function `CaseGetChangedStatus` 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.

Remarks:

- 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` instead.

See also:

The functions `CaseSetChangedStatus`, `CaseSave`.

CaseGetDatasetReference

With the function CaseGetDatasetReference 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, refering to the case for which you want to retrieve the dataset reference.

data-category
: An element in the set AllDataCategories, refering to the specific data category for which you want to obtain the dataset reference.

dataset
: An element parameter into AllDataSets, 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 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.

Remarks:

- 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, CaseTypeCategories.

CaseGetType

The procedure CaseGetType 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, refering to the case for which you want to retrieve its case type.

case_type
: An element parameter into AllCaseTypes, on successfull return this argument will contain the case type for the given case.

Return value:

The procedure returns 1 on success, 0 otherwise.

Remarks:

- 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 instead.

CaseLoadCurrent

The procedure CaseLoadCurrent 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. 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 will contain a proper error message.

Remarks:

- 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.
- If the option Data_Management_style is set to disk_files_and_folders, please use the function CaseCommandLoadAsActive instead.

See also:

The procedures `CaseLoadIntoCurrent`, `CaseMerge`, `CaseSave`, `CaseSetChangedStatus`.

CaseLoadIntoCurrent

The procedure `CaseLoadIntoCurrent` 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`. 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` will contain a proper error message.

Remarks:

- 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` instead.

See also:

The procedures CaseLoadCurrent, CaseMerge, CaseSave, CaseSetChangedStatus.

CaseMerge

The procedure CaseMerge 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. 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 will contain a proper error message.

Remarks:

- 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 instead.

See also:

The procedures CaseLoadCurrent, CaseLoadIntoCurrent, CaseSave, CaseGetChangedStatus.

CaseNew

The procedure CaseNew 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 will contain a proper error message.

Remarks:

- If the option Data_Management_style is set to disk_files_and_folders, please use the function CaseCommandNew instead.
- 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 CaseLoadCurrent, CaseSave, CaseSelectNew, CaseSetCurrent.

CaseSave

The procedure CaseSave 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 will contain an error message.

Remarks:

- 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 or CaseFileSave instead.

See also:

The procedures CaseSaveAs, CaseSaveAll, CaseLoadCurrent, CaseGetChangedStatus.

CaseSaveAll

With the procedure CaseSaveAll 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.

Remarks:

- 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 and CaseCommandSave instead.

See also:

The procedures CaseSave, DatasetSave.

CaseSaveAs

The procedure CaseSaveAs 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 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. 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 will contain an error message.

Remarks:

- 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 instead.

See also:

The procedures CaseSave, CaseSaveAll, CaseLoadCurrent, CaseGetChangedStatus.

CaseSelect

The procedure CaseSelect 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. 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.

Remarks:

- 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 or CaseDialogSelectForSave instead.

See also:

The procedure CaseSelectNew.

CaseSelectMultiple

The procedure CaseSelectMultiple 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**.

Remarks:

- 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 instead.

CaseSelectNew

The procedure CaseSelectNew 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. 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.

Remarks:

- 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 or CaseDialogSelectForSave instead.

See also:

The procedures CaseSelect, CaseSetCurrent, CaseSave.

CaseSetChangedStatus

The procedure `CaseSetChangedStatus` 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.

Remarks:

- 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` or `DataChangeMonitorReset` instead.

See also:

The procedures `CaseGetChangedStatus`, `DatasetSetChangedStatus`.

CaseSetCurrent

The procedure CaseSetCurrent 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, refering to the case that should become the current case.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- 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 instead.

See also:

The procedures CaseNew, CaseCreate, CaseSelectNew, CaseSave.

CaseReadFromSingleFile

The procedure `CaseReadFromSingleFile` 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`, `CaseSave`.

CaseWriteToSingleFile

The procedure `CaseWriteToSingleFile` 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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.
- The procedure `CaseWriteToSingleFile` 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` can only be read by `CaseReadFromSingleFile`.

See also:

The procedures `CaseReadFromSingleFile`, `CaseSave`.

48.2 Datasets

AIMMS supports the following functions for accessing the datasets in the **Data Manager**:

- `DatasetCreate`
- `DatasetDelete`
- `DatasetFind`
- `DatasetGetCategory`
- `DatasetGetChangedStatus`
- `DatasetLoadCurrent`
- `DatasetLoadIntoCurrent`
- `DatasetMerge`
- `DatasetNew`
- `DatasetSave`
- `DatasetSaveAll`
- `DatasetSaveAs`
- `DatasetSelect`
- `DatasetSelectNew`
- `DatasetSetChangedStatus`
- `DatasetSetCurrent`

DatasetCreate

The procedure DatasetCreate 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, 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. On successful return this parameter will refer to the newly created element in AllDataSets.

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.

Remarks:

- 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, DatasetDelete.

DatasetDelete

The procedure DatasetDelete 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, specifying the data category for which a dataset is be deleted.

dataset
An element parameter into AllDataSets, representing the dataset that you want to delete.

Return value:

The procedure returns 1 if the dataset is deleted successfully, or 0 otherwise.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedure DatasetFind.

DatasetFind

The procedure DatasetFind 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, 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. 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetCreate, DatasetDelete.

DatasetGetCategory

The procedure `DatasetGetCategory` 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`, refering to the dataset for which you want to retrieve its data category.

data_category
: An element parameter into `AllDataCategories`, on successfull return this argument will contain the data category of the given dataset.

Return value:

The procedure returns 1 on success, 0 otherwise.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

DatasetGetChangedStatus

The function DatasetGetChangedStatus 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, 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The functions DatasetSetChangedStatus, DatasetSave.

DatasetLoadCurrent

The procedure DatasetLoadCurrent 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, specifying the data category for which a dataset is loaded.

dataset
: An element parameter in the set AllDataSets. 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 will contain a proper error message.

Remarks:

- 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.

See also:

The procedures DatasetLoadIntoCurrent, DatasetMerge, DatasetSave, DatasetSetChangedStatus.

DatasetLoadIntoCurrent

The procedure DatasetLoadIntoCurrent 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, specifying the data category for which a dataset is loaded.

dataset
: An element parameter in the set AllDataSets. 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 will contain a proper error message.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetLoadCurrent, DatasetMerge, DatasetSave, DatasetSetChangedStatus.

DatasetMerge

The procedure DatasetMerge 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, specifying the data category for which a dataset is loaded.

dataset
An element parameter in the set AllDataSets. 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 will contain a proper error message.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetLoadCurrent, DatasetLoadIntoCurrent, DatasetSave, DatasetGetChangedStatus.

DatasetNew

The procedure DatasetNew 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, 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 will contain a proper error message.

Remarks:

- 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, DatasetSave, DatasetSelectNew, DatasetSetCurrent.

DatasetSave

The procedure DatasetSave 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, 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 will contain an error message.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetSaveAs, DatasetSaveAll, DatasetLoadCurrent and function DatasetGetChangedStatus.

DatasetSaveAll

The procedure DatasetSaveAll 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 will contain an error message.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetSaveAs, DatasetSave, DatasetLoadCurrent, DatasetGetChangedStatus.

DatasetSaveAs

The procedure DatasetSaveAs 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, specifying the data category for which you want to save the data.

dataset
An element parameter in AllDataSets. 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 will contain an error message.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetSave, DatasetSaveAll, DatasetLoadCurrent, DatasetGetChangedStatus.

DatasetSelect

The procedure DatasetSelect 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, specifying the data category for which you want to the user to select a dataset.

dataset
: An element parameter in AllDataSets. 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedure DatasetSelectNew.

DatasetSelectNew

The procedure `DatasetSelectNew` 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`, specifying the data category for which you want to the user to select a new dataset.

dataset
: An element parameter in `AllDataSets`. 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.

Remarks:

- 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`, `DatasetSetCurrent`, `DatasetSave`.

DatasetSetChangedStatus

The procedure DatasetSetChangedStatus 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, 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The function DatasetGetChangedStatus.

DatasetSetCurrent

The procedure DatasetSetCurrent 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, specifying the data category for which you want to set the current dataset.

dataset
: An element of the set AllDataSets, refering to the dataset that should become the current dataset.

Return value:

The procedure returns 1 on success, or 0 otherwise.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DatasetNew, DatasetCreate, DatasetSelectNew, DatasetSave.

48.3 Data Manager files

AIMMS supports the following Data Manager functions, that are not specific for cases or datasets only:

- CaseTypeCategories
- CaseTypeContents
- DataCategoryContents
- DataFileCopy
- DataFileExists
- DataFileGetAcronym
- DataFileGetComment
- DataFileGetDescription
- DataFileGetGroup
- DataFileGetName
- DataFileGetOwner
- DataFileGetPath
- DataFileGetTime
- DataFileReadPermitted
- DataFileSetAcronym
- DataFileSetComment
- DataFileWritePermitted
- DataImport220
- DataManagerFileNew
- DataManagerFileOpen
- DataManagerFileGetCurrent
- DataManagerExport
- DataManagerImport
- DataManagementExit

CaseTypeCategories

The procedure CaseTypeCategories 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, refering to the case type for which you want to retrieve the included data categories.

category_set
: A subset of the set AllDataCategories, 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures CaseGetType, CaseTypeContents, DataCategoryContents.

CaseTypeContents

The procedure CaseTypeContents 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, refering to the case type for which you want to retrieve the contents.

identifier_set
: A subset of the set AllIdentifiers, 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.

Remarks:

- 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, CaseTypeCategories, DataCategoryContents.

DataCategoryContents

The procedure `DataCategoryContents` 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`, refering to the data category for which you want to retrieve the contents.

identifier_set
A subset of the set `AllIdentifiers`, 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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures `CaseTypeCategories`, `CaseTypeContents`.

DataFileCopy

With the procedure DataFileCopy 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, AllCases or AllDataSets.

acronym
: The name of the new data file to be created

copiedDatafile
: On success, contains the element in AllDataFiles 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.

Remarks:

- 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 will fail.

DataFileExists

With the procedure DataFileExists you can check whether a specific element from the set AllDataFiles 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, AllCases or AllDataSets.

Return value:

The procedure returns 1 if the given datafile still exists, and 0 otherwise.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.
- Note that AllCases and AllDataSets are subsets of AllDataFiles.

See also:

The procedure DataFileGetName.

DataFileGetAcronym

The predefined set AllDataFiles (and its subsets AllCases and AllDataSets), 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 you can obtain the acronym that is specified in the data manager for any element of the set AllDataFiles (cases or datasets).

```
DataFileGetAcronym(
        datafile,     ! (input) element in the set AllDataFiles
        acronym       ! (output) scalar string parameter
        )
```

Arguments:

datafile
: An element in the set AllDataFiles.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetName.

DataFileGetComment

The predefined set AllDataFiles (and its subsets AllCases and AllDataSets), 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 you can obtain the comment that is specified in the data manager for any element of the set AllDataFiles (cases or datasets).

```
DataFileGetComment(
        datafile,     ! (input) element in the set AllDataFiles
        comment       ! (output) scalar string parameter
        )
```

Arguments:

datafile
An element in the set AllDataFiles.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetName.

DataFileGetDescription

The predefined set `AllDataFiles` (and its subsets `AllCases` and `AllDataSets`), 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` 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`.

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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures `DataFileExists`, `DataFileGetName`, `DataFileGetAcronym`.

DataFileGetGroup

With the procedure DataFileGetGroup 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.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetOwner.

DataFileGetName

The predefined set `AllDataFiles` (and its subsets `AllCases` and `AllDataSets`), 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` you can obtain the name in the data manager for any element of the set `AllDataFiles` (cases or datasets).

```
DataFileGetName(
        datafile,     ! (input) element in the set AllDataFiles
        name          ! (output) scalar string parameter
        )
```

Arguments:

datafile
An element in the set `AllDataFiles`.

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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures `DataFileExists`, `DataFileGetPath`, `DataFileGetAcronym`.

DataFileGetOwner

With the procedure DataFileGetOwner 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.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetGroup.

DataFileGetPath

The predefined set AllDataFiles (and its subsets AllCases and AllDataSets), 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 you can obtain the path in the data manager for any element of the set AllDataFiles (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.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetName, DataFileGetAcronym.

DataFileGetTime

With the procedure `DataFileGetTime` 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`.

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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures `DataFileExists`, `FileTime`.

DataFileReadPermitted

With the procedure DataFileReadPermitted 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.

Return value:

The procedure returns 1 if the current user does have read permission, and 0 otherwise.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedure DataFileWritePermitted.

DataFileSetAcronym

The predefined set AllDataFiles (and its subsets AllCases and AllDataSets), 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 you can set the acronym for the data file corresponding to any element of the set AllDataFiles (cases or datasets).

```
DataFileSetAcronym(
        datafile,     ! (input) element in the set AllDataFiles
        acronym       ! (input) scalar string parameter
        )
```

Arguments:

datafile
: An element in the set AllDataFiles.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetAcronym.

DataFileSetComment

The predefined set AllDataFiles (and its subsets AllCases and AllDataSets), 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 you can set the comment for the data file corresponding to any element of the set AllDataFiles (cases or datasets).

```
DataFileSetComment(
        datafile,     ! (input) element in the set AllDataFiles
        comment       ! (input) scalar string parameter
        )
```

Arguments:

datafile
An element in the set AllDataFiles.

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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataFileExists, DataFileGetComment.

DataFileWritePermitted

With the procedure DataFileWritePermitted 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.

Return value:

The procedure returns 1 if the current user does have write permission, and 0 otherwise.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedure DataFileReadPermitted.

DataImport220

With the procedure DataImport220 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 will contain a proper error message.

Remarks:

- 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.

DataManagerFileNew

With the procedure `DataManagerFileNew` 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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures `DataManagerFileOpen`, `DataManagerFileGetCurrent`.

DataManagerFileOpen

With the procedure DataManagerFileOpen 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedures DataManagerFileNew, DataManagerFileGetCurrent.

DataManagerFileGetCurrent

With the procedure `DataManagerFileGetCurrent` 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.

Remarks:

- This function is only applicable if the project option `Data_Management_style` is set to `Single_Data_Manager_file`.

See also:

The procedures `DataManagerFileNew`, `DataManagerFileOpen`.

DataManagerExport

With the procedure DataManagerExport 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, 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.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedure DataManagerImport.

DataManagerImport

With the procedure DataManagerImport 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 will contain a proper error message.

Remarks:

- This function is only applicable if the project option Data_Management_style is set to Single_Data_Manager_file.

See also:

The procedure DataManagerExport.

Chapter 49

Deprecated AIMMS 220 Functions

AIMMS supports the following deprecated functions originating from AIMMS 220.

- ListingFileCopy
- ListingFileDelete

ListingFileCopy

With the procedure ListingFileCopy 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.

ListingFileDelete

The function ListingFileDelete 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.

Part XI

Appendices

Index

L

M

N

S

T

U

V

W

Y