AIMMS Language Reference - Set, Set Element and String Expressions

AIMMS 4

This documentation was typeset by AIMMS B.V. using LATEX and the LUCIDA font family.

Chapter 5

Set, Set Element and String Expressions

Several types of expressions

Expressions are organized arrangements of operators, constants, sets, indices, parameters, and variables that evaluate to either a set, a set element, a numerical value, a logical value, a string value, or a unit value. Expressions form the core of the AIMMS language. In the previous chapters you already have seen some elementary examples of expressions.

This chapter

In this chapter, set, set element and string expressions are presented in detail. For expressions that evaluate to either numerical or logical values, you are referred to Chapter 6. Expressions that evaluate to unit values are discussed in Section 32.6

5.1 Set expressions

Set expressions

Set expressions play an important role in the construction of index domains of indexed identifiers, as well as in constructing the domain of execution of particular indexed statements. The AIMMS language offers a powerful set of set expressions, allowing you to express complex set constructs in a clear and concise manner.

Constant set expressions

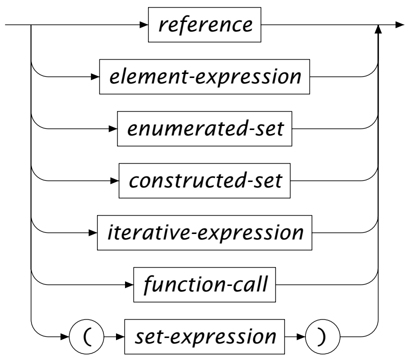
A set expression is evaluated to yield the value of a set. As with all expressions in AIMMS, set expressions come in two forms, *constant* and *symbolic*. Constant set expressions refer to explicit set elements directly, and are mainly intended for set initialization. The tabular format of set initialization is treated in Section 28.2.

Symbolic set expressions

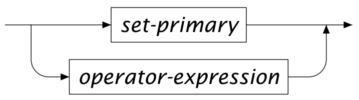
Symbolic set expressions are formulas that can be executed to result in a set. The contents of this set can vary throughout the execution of your model depending on the values of the other model identifiers referenced inside the symbolic formulas. Symbolic set expressions are typically used for specifying index domains. In this section various forms of set expressions will be treated.

set-primary :

Syntax



set-expression :



Set references

The simplest form of set expression is the reference to a set. The reference can be scalar or indexed, and evaluates to the current contents of that set.

The .Domain suffix

Through the `.Domain` suffix you have access to the current contents of the index domain of any indexed identifier. You can use the `.Domain` suffix as if it were a set identifier.

Example

Consider the indexed variable `Transport(i,j)` defined over (a subset of) the set `Cities` × `Cities`. You can use the reference

```
Transport.Domain
```

as a two-dimensional subset of `Cities` × `Cities`, containing all tuples that currently lie within the `IndexDomain` of the variable `Transport(i,j)`.

5.1.1 Enumerated sets

Enumerated sets

An *enumerated* set is a set defined by an explicit enumeration of its elements. Such an enumeration includes literal elements, set element expressions, and (constant or symbolic) element ranges. An enumerated set can be either a simple or a compound set. If you use an *integer element range*, an integer set will result.

Constant enumerated sets

Enumerated sets come in two flavors: *constant* and *symbolic*. Constant enumerated sets are preceded by the keyword DATA, and must only contain literal set elements. These set elements do not have to be contained in single quotes unless they contain characters other than the alpha-numeric characters, the underscore, the plus or the minus sign.

Example

The following simple and compound set assignments illustrate constant enumerated set expressions.

```
Cities := DATA { Amsterdam, Rotterdam, 'The Hague', London, Paris, Berlin, Madrid } ;

DutchRoutes := DATA { (Amsterdam, Rotterdam  ), (Amsterdam, 'The Hague'),
                      (Rotterdam, Amsterdam  ), (Rotterdam, 'The Hague') } ;
```

Symbolic enumerated sets

Any enumerated set not preceded by the keyword DATA is considered symbolic. Symbolic enumerated sets can also contain element parameters. In order to distinguish between literal set elements and element parameters, all literal elements inside symbolic enumerated sets must be quoted.

Examples

The following two set assignments illustrate the use of enumerated sets that depend on the value of the element parameters SmallestCity, LargestCity and AirportCity.

```
ExtremeCities := { SmallestCity, LargestCity } ;

Routes        := { (LargestCity, SmallestCity), (AirportCity, LargestCity )  } ;
```

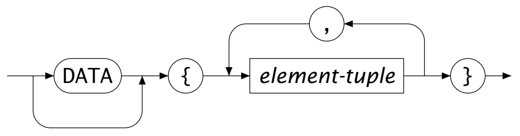
The following two set assignments contrast the semantics between constant and symbolic enumerated sets.

```
SillyExtremes := DATA { SmallestCity, LargestCity } ;
! contents equals { 'SmallestCity', 'LargestCity' }

ExtremeCities := { SmallestCity, LargestCity, 'Amsterdam' };
! contents equals e.g. { 'The Hague', 'London', 'Amsterdam' }
```

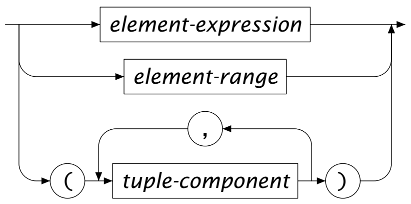
The syntax of enumerated set expressions is as follows.

Syntax

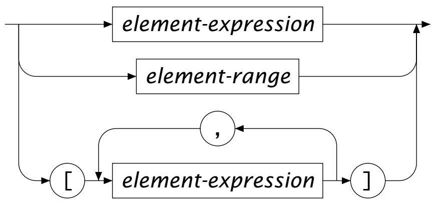
enumerated-set :



element-tuple :



tuple-component :



All elements in an enumerated set must have the same dimension.

Element range

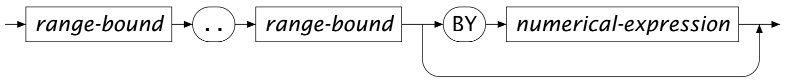
By using the .. operator, you can specify an *element range*. An element range is a sequence of consecutively numbered elements. The following set assignments illustrate both constant and symbolic element ranges. Their difference is explained below.

```
NodeSet       := DATA { node1 .. node100 } ;

FirstNode     := 1;
LastNode      := 100;

IntegerNodes  := { FirstNode .. LastNode } ;
```

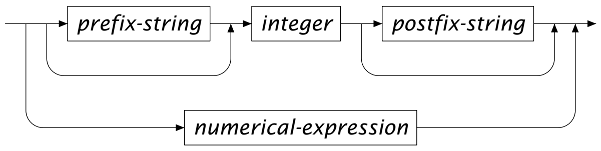
The syntax of element ranges is as follows.

Syntax

element-range :



range-bound :



Prefix and postfix strings

A range bound must consists of an integer number, and can be preceded or followed by a common prefix or postfix string, respectively. The prefix and postfix strings used in the lower and upper range bounds must coincide.

Constant range

If you use an element range in a static enumerated set expression (i.e. preceded by the keyword DATA), the range can only refer to explicitly numbered elements, which need not be quoted. By padding the numbered elements with zeroes, you indicate that AIMMS should create all elements with the same element length.

Constant range versus `ElementRange`

As the begin and end elements of a constant element range are literal elements, you cannot use a constant element range to create sets with dynamically changing border elements. If you want to accomplish this, you should use the `ElementRange` function, which is explained in detail in Section 5.1.4. Its use in the following example is self-explanatory. The following set assignments illustrate a constant element range and its equivalent formulation using the `ElementRange` function.

```
NodeSet     := DATA { node1   .. node100 } ;
PaddedNodes := DATA { node001 .. node100 } ;

NodeSet     := ElementRange( 1, 100, prefix: "node", fill: 0 );
PaddedNodes := ElementRange( 1, 100, prefix: "node", fill: 1 );
```

Symbolic integer range

Element ranges in a symbolic enumerated set can be used to create integer ranges. Now, both bounds can be numerical expressions. Such a construct will result in the *dynamic* creation of a number of *integer* elements based on the value of the numerical expressions at the range bounds. Such integer element ranges can only be assigned to *integer* sets (see Section 3.2.2). An example of a dynamic integer range follows.

```
IntegerNodes := { FirstNode .. LastNode } ;
```

In this example `IntegerNodes` must be an integer set.

Nonconsecutive range

If the elements in the range are not consecutive but lie at regular intervals from one another, you can indicate this by adding a `BY` modifier with the proper interval length. For static enumerated sets the interval length must be a constant, for dynamic enumerated sets it can be any numerical expression. The following set assignments illustrate a constant and symbolic element range with nonconsecutive elements.

```
EvenNodes         := DATA { node2 .. node100 by 2 } ;

StepSize          := 2;
EvenIntegerNodes  := { FirstNode .. LastNode by StepSize } ;
```

Element tuples

When specifying element tuples in an enumerated set expression, it is possible to create multiple tuples in a concise manner using cross products. You can specify multiple elements for a particular tuple component in the cross product either by grouping single elements using the [and] operators or by using an element range, as shown below.

```
DutchRoutes := DATA { ( Amsterdam, [Rotterdam, 'The Hague'] ),
                      ( Rotterdam, [Amsterdam, 'The Hague'] )  } ;
! creates  { ( 'Amsterdam', 'Rotterdam' ), ( 'Amsterdam', 'The Hague' ),
!            ( 'Rotterdam', 'Amsterdam' ), ( 'Rotterdam', 'The Hague' ) }

Network     := DATA { ( node1 .. node100, node1 .. node100 ) } ;
```

The assignment to the set `Network` will create a set with 10,000 elements.

5.1.2 Constructed sets

Constructed sets

A *constructed set* expression is one in which the selection of elements is constructed through filtering on the basis of a particular condition. When a constructed set expression contains an index, AIMMS will consider the resulting tuples for every element in the binding set.

Example

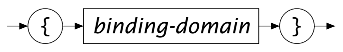
The following set assignments illustrate some constructed set expressions, assuming that `i` and `j` are indices into the set `Cities`.

```
LargeCities := { i | Population(i) > 500000 } ;

Routes := { (i,j) | Distance(i,j) } ;

RoutesFromLargestCity := { (LargestCity, j) in Routes } ;
```

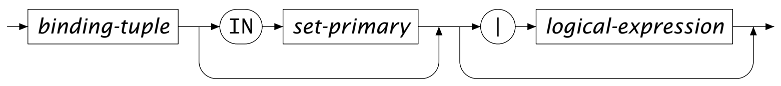
In the latter assignment route tuples are constructed from `LargestCity` (an element-valued parameter) to every city `j`, where additionally each created tuple is required to lie in the set `Routes`.

Syntax

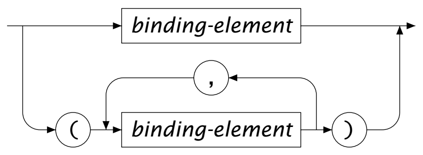
constructed-set :



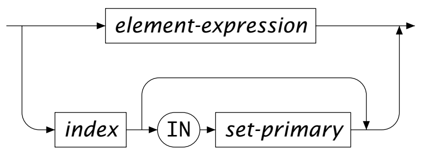
binding-domain :



binding-tuple :



binding-element :



Binding domain

The tuple selection in a constructed set expression behaves exactly the same as the tuple selection on the left-hand side of an assignment to an indexed parameter. This means that all tuple components can be either an explicit quoted set element, a general set element expression, or a binding index. The tuple can be subject to a logical condition, further restricting the number of elements constructed.

5.1.3 Set operators

Four set operators

There are four binary set operators in AIMMS: Cartesian product, intersection, union, and difference. Their notation and precedence are given in Table 5.1. Expressions containing these set operators are read from left to right and the operands can be any set expression. There are no unary set operators.

Operator	Notation	Precedence
intersection	*	3 (high)
difference	-	2
union	+	2
Cartesian product	CROSS	1 (low)

Table 5.1: Set operators

Example

The following set assignments to integer sets and Cartesian products of integer sets illustrate the use of all available set operators.

```
S := {1,2,3,4} * {3,4,5,6} ;     ! Intersection of integer sets: {3,4}.

S := {1,2} + {3,4} ;             ! Union of simple sets:
S := {1,3,4} + {2} + {1,2} ;     ! {1,2,3,4}

S := {1,2,3,4} - {2,4,5,7} ;     ! Difference of integer sets: {1,3}.

T := {1,2} cross {1,2} ;         ! The cross of two integer sets:
                                 ! {(1,1),(1,2),(2,1),(2,2)}.

U := {(1,2),(1,3)} cross {4,5} ; ! The cross of a compound and integer set:
                                 ! {(1,2,4),(1,2,5),(1,3,4),(1,3,5)}.
```

The precedence and associativity of the operators is demonstrated by the assignments

```
T := A cross B - C ;      ! Same as A cross (B - C).
T := A - B * C + D ;      ! Same as (A - (B * C)) + D.
T := A - B * C + D * E ;  ! Same as (A - (B * C)) + (D * E).
```

The operands of union, difference, and intersection must have the same dimensions.

```
T := {(1,2),(1,3)} * {(1,3)} ;              ! Same as {(1,3)}.

T := {(1,2),(1,3)} + {(i,j) | a(i,j) > 1} ; ! Union of enumerated
                                            ! and constructed set of
                                            ! the same dimension.

T := {(1,2),(1,3)} + {(1,2,3)} ;            ! ERROR: dimensions differ.
```

5.1.4 Set functions

Set functions

A special type of set expression is a call to one of the following set-valued functions

- `ElementRange`,
- `SubRange`,
- `ConstraintVariables`,
- `VariableConstraints`, or
- A user-defined function.

The `ElementRange` and `SubRange` functions are discussed in this section, while the functions `ConstraintVariables` and `VariableConstraints` are discussed in Section 15.1. The syntax of and use of tags in function calls is discussed in Section 10.2.

The function `ElementRange`

The `ElementRange` function allows you to *dynamically* create or change the contents of a set of non-integer elements based on the value of integer-valued scalars expressions.

Arguments

The `ElementRange` function has two mandatory integer arguments.

- *first*, the integer value for which the first element must be created, and
- *last*, the integer value for which the last element must be created.

In addition, it allows the following four optional arguments.

- *incr*, the integer-valued interval length between two consecutive elements (default value 1),
- *prefix*, the prefix string for every element (by default, the empty string),
- *postfix*, the postfix string (by default, the empty string), and
- *fill*, a logical indicator (0 or 1) whether the numbers must be padded with zeroes (default value 1).

If you use any of the optional arguments you must use their formal argument names as tags.

Example

Consider the sets `S` and `T` initialized by the constant set expressions

```
NodeSet     := DATA { node1   .. node100 } ;
PaddedNodes := DATA { node001 .. node100 } ;
EvenNodes   := DATA { node2   .. node100 by 2 } ;
```

These sets can also be created in a dynamic manner by the following applications of the `ElementRange` function.

```
NodeSet     := ElementRange( 1, 100, prefix: "node", fill: 0 );
PaddedNodes := ElementRange( 1, 100, prefix: "node", fill: 1 );
EvenNodes   := ElementRange( 2, 100, prefix: "node", fill: 0, incr: 2 );
```

The SubRange function

The SubRange function has three arguments:

- a simple or compound *set*,
- the *first* element, and
- the *last* element.

The result of the function is the subset ranging from the *first* to the *last* element. If the first element is positioned after the last element, the empty set will result.

Example

Assume that the set Cities is organized such that all foreign cities are consecutive, and that FirstForeignCity and LastForeignCity are element-valued parameters into the set Cities. Then the following assignment will create the subset ForeignCities of Cities

```
ForeignCities := SubRange( Cities, FirstForeignCity, LastForeignCity ) ;
```

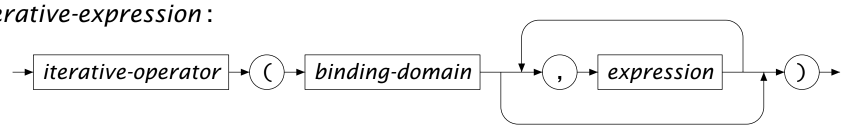
5.1.5 Iterative set operators

Iterative operators

Iterative operators form an important class of operators that are especially designed for indexed expressions in Aimms. There are set, element-valued, arithmetic, statistical, and logical iterative operators. The syntax is always similar.

Syntax

iterative-expression :



Explanation

The first argument of all iterative operators is a *binding domain*. It consists of a single index or tuple of indices, optionally qualified by a logical condition. The second argument and further arguments must be expressions. These expressions are evaluated for every index or tuple in the binding domain, and the result is input for the particular iterative operator at hand. Indices in the expressions that are not part of the binding domain of the iterative operators are referred to as *outer indices*, and must be bound elsewhere.

Set-related iterative operators

Aimms possesses the following set-related iterative operators:

- the Sort operator for sorting the elements in a domain,
- the NBest operator for obtaining the n best elements in a domain according to a certain criterion, and
- the Intersection and Union operators for repeated intersection or union of indexed sets.

Reordering your data

Sorting the elements of a set is a useful tool for controlling the flow of execution and for presenting reordered data in the graphical user interface. There are two mechanism available to you for sorting set elements

- the `OrderBy` attribute of a set, and
- the `Sort` operator.

Sorting semantics

The second and further operands of the `Sort` operator must be numerical, element-valued or string expressions. The result of the `Sort` operator will consist of precisely those elements that satisfy the domain condition, sorted according to the single or multiple ordering criteria specified by the second and further operands. Section 3.2 discusses the expressions that can be used for specifying an ordering principle.

Receiving set

Note that the set to which the result of the `Sort` operator is assigned must have the `OrderBy` attribute set to `User` (see also Section 3.2.1) for the operation to be useful. Without this setting Aimms will store the elements of the result set of the `Sort` operator, but will discard the underlying ordering.

Example

The following assignments will result in the same set orderings as in the example of the `OrderBy` attribute in Section 3.2.

```
LexicographicSupplyCities := Sort( i in SupplyCities, i ) ;

ReverseLexicographicSupplyCities := Sort( i in SupplyCities, -i );

SupplyCitiesByIncreasingTransport :=
    Sort( i in SupplyCities, Sum( j, Transport(i,j) );

SupplyCitiesByDecreasingTransportThenLexicographic :=
    Sort( i in SupplyCities, - Sum( j, Transport(i,j) ), i );
```

Sorting root sets

Aimms will even allow you to sort the elements of a root set. Because the entire execution system of Aimms is built around a fixed ordering of the root sets, sorting root sets may influence the overall execution in a negative manner. Section 13.2.7 explains the efficiency considerations regarding root set ordering in more detail.

Obtaining the n best elements

You can use the `NBest` operator, when you need the n best elements in a set according to a single ordering criterion. The syntax of the `NBest` is similar to that of the `Sort` operator. The first expression after the binding domain is the criterion with respect to which you want elements in the binding domain to be ordered. The second expression refers to the number of elements n in which you are interested.

Example

The following assignment will, for every city i, select the three cities to which the largest transports emanating from i take place. The result is stored in the indexed set LargestTransportCities(i).

```
    LargestTransportCities(i) := NBest( j, Transport(i,j), 3 );
```

Repeated intersection and union

With the Intersection and Union operators you can perform repeated set intersection or union respectively. A typical application is to take the repeated intersection or union of all instances of an indexed set. However, any set valued expression can be used on the second argument.

Example

Consider the following indexed set declarations.

```
Set IndSet1 {
    IndexDomain : s1;
    SubsetOf    : S;
}
Set IndSet2 {
    IndexDomain : s1;
    SubsetOf    : S;
}
```

With these declarations, the following assignments illustrate valid uses of the Union and Intersection operators.

```
      SubS := Union( s1, IndSet1(s1) );
      SubS := Intersection( s1, IndSet1(s1) + IndSet2(s1) );
```

5.1.6 Set element expressions as singleton sets

Element expressions . . .

Element expressions can be used in a set expression as well. In the context of a set expression, AIMMS will interpret an element expression as the singleton set containing only the element represented by the element expression. Set element expressions are discussed in full detail in Section 5.2.

. . . versus enumerated sets

Using an element expression as a set expression can equivalently be expressed as a symbolic enumerated set containing the element expression as its sole element. Whenever there is no need to group multiple elements, AIMMS allows you to omit the surrounding braces.

Example

The following set assignment illustrate some simple set element expressions used as a singleton set expression.

```
    ! Remove LargestCity from the set of Cities
    Cities -= LargestCity ;
```

```
! Remove first element from the set of Cities
Cities -= Element(Cities,1) ;

! Remove LargestCity and SmallestCity from Cities
Cities -= LargestCity + SmallestCity ;

! The set of Cities minus the CapitalCity
NonCapitalCities := Cities - CapitalCity ;
```

5.2 Set element expressions

Use of set element expressions

Set element expressions reference a particular element or element tuple model from a set or a tuple domain. Set element expressions allow for *sliced assignment*—executing an assignment only for a lesser-dimensional subdomain by fixing certain dimensions to a specific set element. Potentially, this may lead to a vast reduction in execution times for time-consuming calculations.

Passing elements from the GUI

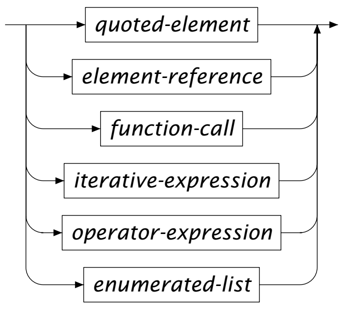
The most elementary form of a set element expression is an element parameter, which turns out to be a useful device for communicating set element information with the graphical interface. You can instruct AIMMS to locate the position in a table or other object where an end-user made changes to a numerical value, and have AIMMS pass the corresponding set element(s) to an element parameter. As a result, you can execute data input checks defined over these element parameters, thereby limiting the amount of computation. This issue is discussed in more detail in the help regarding the Identifier Selection dialog.

Element expressions

AIMMS supports several types of set element expressions, including *references* to parameters and (bound) indices, *lag-lead-expressions*, element-valued *functions*, and *iterative-expressions*. The last category turns out to be a useful device for computing the proper value of element parameters in your model.

Syntax

element-expression :



The format of list expressions are the same for element and numerical expressions. They are discussed in Section 6.1.2.

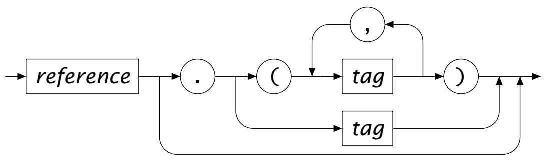
5.2.1 Element references

Element references

An element reference is any reference to either an element parameter or a (bound) index. An element reference can be followed by a tag or tuple of tags as a way to refer to a component or subtuple of components.

Syntax

element-reference :



Use of tags

You can use tags when you want to refer to a subtuple of an element of a compound set. The tags must be declared as attributes in the corresponding compound root set. Tags and their use are explained in detail in Section 3.2.3.

Example

Assume that `orig` and `dest` (in this order!) are tags in the compound set `Routes`, and that the element parameter `MainRoute` contains a two-tuple from the compound set `Cities`×`Cities`. The following statement computes the reverse route using a tagged element reference.

```
ReverseRoute := MainRoute.(dest,orig) ;
```

5.2.2 Intrinsic functions for sets and set elements

The element-related functions...

Aimms supports functions to obtain the position of an element within a set, the cardinality (i.e. number of elements) of a set, the n-th element in a set, the element in a non-compatible set with the identical string representation, and the numerical value represented by a set element. If `S` is a set identifier, `i` an index bound to `S`, l an element, and n a positive integer, then possible calls to the `Ord`, `Card`, `Element`, `ElementCast` and `Val` functions are given in Table 5.2.

... for simple and compound sets

The `Ord`, `Card` and `Element` functions can be applied to both simple and compound sets. In fact you can even apply `Card` to parameters and variables—it simply returns the number of nondefault elements associated with a certain data structure.

Function	**Value**	**Meaning**
`Ord(i)`	*integer*	Ordinal, returns the relative position of the index `i` in the set S. Does *not* bind `i`.
`Ord(`l`,S)`	*integer*	Returns the relative position of the element l in set S. Returns zero if l is not an element of S.
`Card(S)`	*integer*	Cardinality of set S.
`Element(S,`n`)`	*element*	Returns the element in set S at relative position n. Returns the empty element tuple if S contains less then n elements.
`ElementCast(S,`l`)`	*element*	Returns the element in set S, which corresponds to the textual representation of an element l in any other index set.
`Val(`l`)`	*numerical*	Returns the numerical value represented by l, or a runtime error if l cannot be interpreted as a number
`Max(`e_1`,...,`e_n`)`	*Max*	Returns the set element with the highest ordinal
`Min(`e_1`,...,`e_n`)`	*Min*	Returns the set element with the lowest ordinal

Table 5.2: Intrinsic functions operating on sets and set elements

Crossing root set boundaries

By default, Aimms does not allow you to use indices associated with one root set hierarchy in your model, in references to index domains associated with another root set hierarchy of your model. The function `ElementCast` allows you to cross root set boundaries, by returning the set element in the root set associated with the first (set) argument that has the identical name as the element (in another root set) passed as the second argument. The function `ElementCast` has an optional third argument *create* (values 0 or 1, with a default of 0), through which you can indicate whether you want elements which cannot be cast to the indicated set must be created within that set. In this case, a call to `ElementCast` will never fail. You can find more information about root sets, as well as an illustrative example of the use of `ElementCast`, in Section 9.1.

Example

In this example, we again use the set `Cities` initialized through the statement

```
Cities := DATA { Amsterdam, Rotterdam, 'The Hague', London, Paris, Berlin, Madrid } ;
```

The following table illustrates the intrinsic element-valued functions.

The `Val` function

If your model contains a set with elements that represent numerical values, you cannot directly use such elements as a numerical value in numerical expressions, unless the set is an integer set (see Section 3.2.2). To obtain the

Expression	**Result**
`Ord('Amsterdam', Cities)`	1
`Ord('New York', Cities)`	0 (i.e. not in the set)
`Card(Cities)`	7
`Element(Cities, 1)`	`'Amsterdam'`
`Element(Cities, 8)`	`''` (i.e. no 8-th element)

numerical value of such set elements, you can use the `Val` function. You can also apply the `Val` function to strings that represent a numerical value. In both cases, a runtime error will occur if the element or string argument of the `Val` function cannot be interpreted as a numerical value.

The `Min` and `Max` functions

The element-valued `Min` and `Max` functions operate on two or more element-valued expressions *in the same (sub-)set hierarchy*. If the arguments are references to element parameters (or bound indices), then the `Range` attributes of these element parameters or indices must be sets in a single set hierarchy. Through these functions you can obtain the elements with the lowest and highest ordinal relative to the set equal to highest ranking range set in the subset hierarchy of all its arguments. If one or more of the arguments are explicit labels, then AIMMS will verify that these labels are contained in that set, or will return an error otherwise. A compiler error will result, if no such set can be determined (i.e., when the function call refers to explicit labels only).

The `Tuple` function

The `Tuple` function has a variable number of input arguments, all of which must be element expressions. The value of this function is a literal element of a compound set. The dimension of the created tuple should match the dimension of the context in which it is used, and AIMMS must be able to determine the compound set from which the presented tuple is supposed to be an element from the context. When the result of the `Tuple` function is assigned to an element parameter, no assignment is made when the result is not an element of the range set of the parameter.

Example

The following assignment creates a reference to a particular route, namely between the cities represented by the element parameters `HarborCity` and `AirportCity`.

```
MainRoute := Tuple( AirportCity, HarborCity );
```

You can also assign a tuple with literal elements.

```
MainRoute := Tuple( 'Amsterdam', 'Rotterdam' );
```

In both cases the created tuple should be an element of the range set `Routes` of the parameter `MainRoute`.

5.2.3 Element-valued iterative expressions

Selecting elements

Aimms offers special iterative operators that let you select a specific element from a simple or compound domain. Table 5.3 shows all such operators that result in a set element value. The syntax of iterative operators is explained in Section 5.1.5. The second column in this table refers to the required number of expression arguments following the binding domain argument.

Name	# Expr.	Computes for all elements in the domain
`First`	0	the first element (tuple)
`Last`	0	the last element (tuple)
`Nth`	1	the n-th element (tuple)
`Min`	1	the value of the element expression for which the expression reaches its minimum ordinal value
`Max`	1	the value of the element expression for which the expression reaches its maximum ordinal value
`ArgMin`	1	the first element (tuple) for which the expression reaches its minimum value
`ArgMax`	1	the first element (tuple) for which the expression reaches its maximum value

Table 5.3: Element-valued iterative operators

Single index

The binding domain of the `First`, `Last`, `Nth`, `Min`, `Max`, `ArgMin`, and `ArgMax` operator can only consist of a single index in either a simple or compound set, and the result is a single element in that domain. You can use this result directly for indexing or referencing an indexed parameter or variable. Alternatively, you can assign it to an element parameter in the appropriate domain.

Compared expressions

The `ArgMin` and `ArgMax` operators return the element for which an expression reaches its minimum or maximum value. The allowed expressions are:

- numerical expressions, in which case Aimms performs a numerical comparison,
- string expressions, in which case Aimms uses the normal alphabetic ordering, and
- element expressions, in which case Aimms compares the ordinal numbers of the resulting elements.

For element expressions, the iterative `Min` and `Max` operators return expression *values* with the minimum and maximum ordinal value.

Example

The following assignments illustrate the use of some of the domain related iterative operators. The identifiers on the left are all element parameters.

```
FirstNonSupplyCity    := First ( i | not Exists(j | Transport(i,j)) ) ;
SecondSupplyCity      := Nth   ( i | Exists(j | Transport(i,j)), 2  ) ;
SmallestSupplyCity    := ArgMin( i, Sum(j, Transport(i,j))          ) ;
LargestTransportRoute := ArgMax( r, Transport(r)                    ) ;
```

Note that the iterative operators Exists and Sum are used here for illustrative purposes, and are not set- or element-related. They are treated in Sections 6.2.5 and 6.1.6, respectively.

5.2.4 Lag and lead element operators

Lag and lead operators...

There are four binary element operators, namely the lag and lead operators +, ++, - and --. The first operand of each of these operators must be an element reference (such as an index or element parameter), while the second operand must be an integer numerical expression. There are no unary element operators.

... explained

Lag and lead operators are used to relate an index or element parameter to preceding and subsequent elements in a set. Such correspondence is well-defined, except when a request extends beyond the bounds of the set.

Noncircular versus circular

There are two kinds of lag and lead operators, namely *noncircular* and *circular* operators which behave differently when pushed beyond the beginning and the end of a set.

- The noncircular operators (+ and -) consider the ordered set elements as a *sequence* with no elements before the first element or after the last element.
- The circular operators (++ and --) consider ordered set elements as a *circular chain*, in which the first and last elements are linked.

Definition

Let S be a set, i a set element expression, and k an integer-valued expression. The lag and lead operators +, ++, -, -- return the element of S as defined in Table 5.4. Please note that these operators are also available in the form of +=, -=, ++= and --=. The operators in this form can be used in statements like:

```
CurrentCity := 'Amsterdam';
CurrentCity --= 1; ! Equal to CurrentCity := CurrentCity -- 1;
```

Lag/lead expr.	Meaning
`i` + k	The element of `S` positioned k elements after `i`; the empty element if there is no such element.
`i` ++ k	The circular version of `i` + k.
`i` − k	The member of `S` positioned k elements before `i`; the empty element if there is no such element.
`i` −− k	The circular version of `i` − k.

Table 5.4: Lag and lead operators

Not for literal elements

You cannot always use lag and lead operators in combination with literal set elements. The reason for this is clear: a literal element can be an element of more than one set, and in general, unless the context in which the lag or lead operator is used dictates a particular (domain) set, it is impossible for AIMMS to determine which set to work with.

Verify the effect of lags and leads

Lag and lead operators are frequently used in indexed parameters and variables, and may appear on the left- and right-hand side of assignments. You should be careful to check the correct use of the lag and lead operators to avoid making conceptual errors. For more specific information on the lag and lead operators refer to Section 8.2, which treats assignments to parameters and variables.

Example

Consider the set `Cities` initialized through the assignment

```
Cities := DATA { Amsterdam, Rotterdam, 'The Hague', London, Paris, Berlin, Madrid } ;
```

Assuming that the index `i` and the element parameter `CurrentCity` both currently refer to `'Rotterdam'`, Table 5.5 illustrates the results of various lag/lead expressions.

Lag/lead expression	Result
`i+1`	`'The Hague'`
`i+6`	`''`
`i++6`	`'Amsterdam'`
`i++7`	`'Rotterdam'`
`i-2`	`''`
`i--2`	`'Madrid'`
`CurrentCity+2`	`'London'`
`'Rotterdam' + 1`	ERROR

Table 5.5: Example of lag and lead operators

5.3 String expressions

String expressions

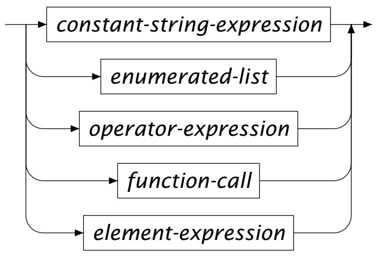
String expressions are useful for

- creating descriptive texts associated with particular set elements and identifiers, or
- forming customized messages for display in the graphical user interface or in output reports.

This section discusses all available string expressions in AIMMS.

Syntax

string-expression :



The format of list expressions are the same for string-valued and numerical expressions. They are discussed in Section 6.1.2.

5.3.1 String operators

String operators

There are three binary string operators in AIMMS, string concatenation (+ operator), string subtraction (- operator), and string repetition (* operator). There are no unary string operators.

String concatenation

The simplest form of composing strings in AIMMS is by the concatenation of two existing strings. String concatenation is represented as a simple addition of strings by means of the + operator.

String subtraction

In addition to string concatenation, AIMMS also supports subtraction of two strings by means of the - operator. The result of the operation $s_1 - s_2$ where s_1 and s_2 are string expressions will be the substring of s_1 obtained by

- omitting s_2 on the right of s_1 when s_1 ends in the string s_2, or
- just s_1 otherwise.

String repetition

You can use the multiplication operator * to obtain the string that is the result of a given number of repetitions of a string. The left-hand operand of the repetition operator * must be a string expression, while the right-hand operand must be an integer numerical expression.

Examples

The following examples illustrate some basic string manipulations in AIMMS.

```
"This is "     + "a string"           ! "This is a string"
"Filename.txt" - ".txt"               ! "Filename"
"Filename"     - ".txt"               ! "Filename"
"--"           * 5                    ! "----------"
```

5.3.2 Formatting strings

The function `FormatString`

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

- a *format string*, which specifies how the string is composed, and
- one or more *arguments* (number, string or element) which are used to form the string as specified.

The format string

The first argument of the function `FormatString` is a mixture of ordinary text plus *conversion specifiers* for each of the subsequent arguments. A conversion specifier is a code to indicate that data of a specified type is to be inserted as text. Each conversion specifier starts with the `%` character followed by a letter indicating its type. The conversion specifier for every argument type are given in Table 5.6.

Conversion specifiers	Argument type
`%s`	String expression
`%e`	Element expression
`%f`	Floating point number
`%g`	Exponential format number
`%i`	Integer expression
`%n`	Numerical expression
`%u`	Unit expression
`%%`	% sign

Table 5.6: Conversion codes for the `FormatString` function

Example

In the example below, the current value of the parameter `SmallVal` and `LargeVal` are 10 and 20, the current value of `CapitalCity` is the element `'Amsterdam'`, and `UnitPar` is a unit-valued parameter with value `kton/hr`. The following calls to `FormatString` illustrate its use.

```
FormatString("The numbers %i and %i", 10, 20)              ! "The numbers 10 and 20"
FormatString("The numbers %i and %i", SmallVal, LargeVal)  ! "The numbers 10 and 20"
FormatString("The number %n", 4*ArcTan(1))                 ! "The number 3.14156"
FormatString("The string %s", "is printed")                ! "The string is printed"
FormatString("The element %e", CapitalCity)                ! "The element Amsterdam"
FormatString("The unit is %u", UnitPar)                    ! "The unit is kton/hr"
```

Modification flags

By default, AIMMS will use a default representation for arguments of each type. By modifying the conversion specifier, you further dictate the manner in which a particular argument of the `FormatString` function is printed. *This is done by inserting modification flags in between the %-sign and the conversion character.* The following modification directives can be added:

- *flags*:

`<`	for left alignment
`<>`	for centered alignment
`>`	for right alignment
`+`	add a plus sign (nonnegative numbers)
`␣`	add a space (instead of the above + sign)
`0`	fill with zeroes (right-aligned numbers only)
`t`	print number using thousand separators, using local convention for both the thousand separator and decimal separator. Controlling these separators is via the options `Number 1000 separator` and `Number decimal separator`.

- *field width*: the converted argument will be printed in a field of at least this width, or wider if necessary
- *dot*: separating the field width from the precision
- *precision*: the number of decimals for numbers, or the maximal number of characters for strings or set elements.

Note the order

It is important to note that the modification flags must be inserted in the order as described above.

Field width and precision

Both the field width and precision of a conversion specifier can be either an integer constant, or a wildcard, `*`. In the latter case the `FormatString` expects one additional integer argument for each wildcard just before the argument of the associated conversion specifier. This allows you to compute and specify either the field width or precision in a dynamic manner.

Floating point vs. exponential format

When using the %n conversion specifier for a floating point number, AIMMS chooses whether to use a floating point or exponential format based on the global option put_number_style. Through the %f or %g conversion specifiers you can force the use of a floating point or exponential format, respectively.

Example

The following calls to FormatString illustrate the use of modification flags.

```
FormatString("The number %>+08i", 10)             ! "The number +0000010"
FormatString("The number %>t8i", 100000)          ! "The number  100,000"
FormatString("The number %> 8.2n", 4*ArcTan(1))     ! "The number     3.14"
FormatString("The number %> *.*n", 8,2,4*ArcTan(1)) ! "The number     3.14"
FormatString("The element %<5e", CapitalCity)       ! "The element Amsterdam"
FormatString("The element %<>5.3e", CapitalCity)    ! "The element  Ams "
```

Special characters

AIMMS offers a number of special characters to allow you to use the full range of characters in composing strings. These special characters are contained in Table 5.7.

Special character	text code	Meaning
\f	FF	Form feed
\t	HT	Horizontal tab
\n	LF	Newline character
\"	"	Double quote
\\	\	Backslash
\\n	n	character n ($001 \leq n \leq 65535$)

Table 5.7: Special characters

Example

Examples of the use of special characters within FormatString follow.

```
FormatString("%i \037 \t %i %%", 10, 11)     ! "10 %       11 %"
FormatString("This is a \"%s\" ", "string")  ! "This is a "string" "
```

Case conversion functions

With the functions StringToUpper, StringToLower and StringCapitalize you can convert the case of a string to upper case, to lower case, or capitalize it, as illustrated in the following example.

```
StringToUpper("Convert to upper case")    ! "CONVERT TO UPPER CASE"
StringToLower("CONVERT to lower case")    ! "convert to lower case"
StringCapitalize("capitaLIZED senTENCE")  ! "Capitalized sentence"
```

5.3.3 String manipulation

Other string related functions

In addition to the FormatString function, AIMMS offers a number of other functions for string manipulation. They are:

- Substring to obtain a substring of a particular string,
- StringLength to determine the length of a particular string,
- FindString to obtain the position of the first occurrence of a particular substring,
- FindNthString to obtain the position of the n-th occurrence of a particular substring, and
- StringOccurrences to obtain the number of occurrences of a particular substring.

The function SubString

With the SubString function you can obtain a substring from a particular begin position m to an end position n (or to the end of the string if the requested end position exceeds the total string length). The positions m and n can both be negative (but with $m \le n$), in which case AIMMS will start counting backwards from the end of the string. Examples are:

```
SubString("Take a substring of me", 8, 16)    ! returns "substring"
SubString("Take a substring of me", 18, 100)  ! returns "of me"
SubString("Take a substring of me", -5, -1)   ! returns "of me"
```

The function StringLength

The function StringLength can be used to determine the length of a string in AIMMS. The function will return 0 for an empty string, and the total number of characters for a nonempty string. An example follows.

```
StringLength("Guess my length")              ! returns 15
```

The functions FindString and FindNthString

With the functions FindString and FindNthString you can determine the position of the second argument, the *key*, within the first argument, the *search* string. The functions return zero if the key is not contained in the search string. The function FindString returns the position of the first occurrence of the key in the search string starting from the left, while the function FindNth-String will return the position of the n-th appearance of the key. If n is negative, the function FindNthString will search backwards starting from the right. Examples are:

```
FindString      ("Find a string in a string", "string"     )  ! returns 8
FindNthString   ("Find a string in a string", "string",  2 )  ! returns 20
FindNthString   ("Find a string in a string", "string", -1 )  ! returns 20

FindString      ("Find a string in a string", "this string")  ! returns 0
FindNthString   ("Find a string in a string", "string",  3 )  ! returns 0
```

Case sensitivity

By default, the functions FindString and FindNthString will use a case sensitive string comparison when searching for the key. You can modify this behavior through the option Case_Sensitive_String_Comparison.

The function StringOccurrences

The function StringOccurrences allows you to determine the number of occurrences of the second argument, the key, within the first argument, the *search* string. You can use this function, for instance, to delimit the number of calls to the function FindNthString a priori. An example follows.

```
StringOccurrences("Find a string in a string", "string"     )  ! returns 2
```

5.3.4 Converting strings to set elements

Converting strings to set elements

Converting strings to new elements to or renaming existing elements in a set is not an uncommon action when end-users of your application are entering new element interactively or when you are obtaining strings (to be used as set elements) from other applications through external procedures. Aimms offers the following support for dealing with such situations:

- the procedure SetElementAdd to add a new element to a set,
- the procedure SetElementRename to rename an existing element in a set, and
- the function StringToElement to convert strings to set elements.

Adding new set elements

The procedure SetElementAdd lets you add new elements to a set. Its arguments are:

- the *set* to which you want to add the new element,
- an *element parameter* into *set* which holds the new element after addition, and
- the *stringname* of the new element to be added.

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.

Renaming set elements

Through the procedure SetElementRename you can provide a new name for an existing element in a particular set whenever this is necessary in your application. Its arguments are:

- the *set* which contains the element to be renamed,
- the *element* to be renamed, and
- the *stringname* to which the element should be renamed.

After renaming the element, all data defined over the old element name will be available under the new element name.

The function `StringToElement`

With the function `StringToElement` you can convert string arguments into (existing) elements of a set. If there is no such element, the function evaluates to the empty element. Its arguments are:

- the *set* from which the element corresponding to *stringname* must be returned,
- the *stringname* for which you want to retrieve the corresponding element, and
- the optional *create* argument (values 0 or 1, with a default of 0) indicating whether nonexisting elements must be added to the set.

With the *create* argument set to 1, a call to `StringToElement` will always return an element in *set*. Alternatively to setting the *create* argument to 1, you can call the procedure `SetElementAdd` to add the element to the set.

Example

The following example illustrates the combined use of `StringToElement` and `SetElementAdd`. It checks for the existence of the string parameter `CityString` in the set `Cities`, and adds it if necessary.

```
ThisCity := StringToElement( Cities, CityString );
if ( not ThisCity ) then
   SetElementAdd( Cities, ThisCity, CityString );
endif;
```

Alternatively, you can combine both statements by setting the optional *create* argument of the function `StringToElement` to 1.

```
ThisCity := StringToElement( Cities, CityString, create: 1 );
```

Converting element to string

Reversely, you can use the `%e` specifier in the `FormatString` function to get a pure textual representation of a set element, as illustrated in the following assignment.

```
CityString := FormatString("%e", ThisCity );
```