AIMMS Language Reference - Numerical and Logical Expressions

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com.

Chapter 6

Numerical and Logical Expressions

AIMMS has a comprehensive set of built-in numerical and logical operators which allow you quickly and concisely express the details of your model. The subject of Macros, which are a parametric form of expression, is also explained. For expressions that evaluate to sets, set elements or strings, see Chapter 5.

This chapter

6.1 Numerical expressions

Like any expression in AIMMS, a numerical expression can either be a *constant* or a *symbolic* expression. Constant expressions are those that contain references to explicit set elements and values, but do not contain references to other identifiers. Constant expressions are mostly intended for the initialization of sets, parameters and variables. Such an initialization must conform to one of the following formats:

Constant numerical expressions

- a *scalar* value,
- a *list* expression,
- a *table* expression, or
- a *composite table.*

Table expressions and composite tables are mostly used for data initialization from *external* files. They are discussed in Chapter 28.

Symbolic expressions are those expressions that contain references to other AIMMS identifiers. They can be used in the Definition attributes of sets, parameters and variables, or as the right-hand side of assignment statements. AIMMS provides a powerful notation for expressions, and complicated numerical manipulations can be expressed in a clear and concise manner.

Symbolic numerical expressions

numerical-expression : *Syntax*
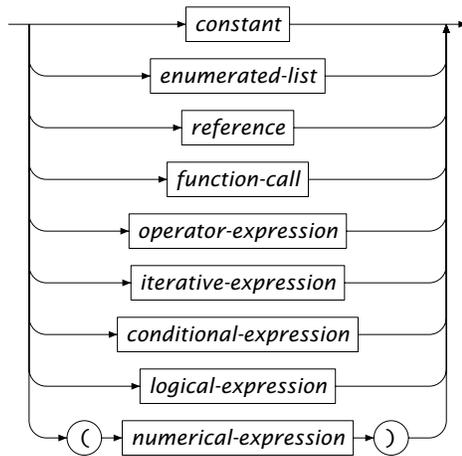
6.1.1 Real values and arithmetic extensions

Traditional arithmetic is defined on the real line, $\mathbb{R} = (-\infty, \infty)$, which does not contain either $+\infty$ or $-\infty$. AIMMS' arithmetic is defined on the set $\mathbb{R} \cup \{$-INF, INF, NA, UNDF, ZERO$\}$ and summarized in Table 6.1. The symbols INF and -INF are mostly used to model unbounded variables. The symbols NA and UNDF stand for *not available* and *undefined* data values respectively. The symbol ZERO denotes the numerical value zero, but has the logical value true (not zero).

Extension of the real line

Symbol	Description	Logical value	MapVal value
number	any valid real number		0
UNDF	undefined (result of an arithmetic error)	1	4
NA	not available	1	5
INF	$+\infty$	1	6
-INF	$-\infty$	1	7
ZERO	numerically indistinguishable from zero, but has the logical value of one.	1	8

Table 6.1: Extended values of the AIMMS language

AIMMS treats these special symbols as ordinary real numbers, and the results of the available arithmetic operations and functions on these symbols are defined. The values INF, -INF and ZERO are accessible by the user and are dealt with as expected: $1 + $ INF evaluates to INF, $1/$INF to 0, $1 + $ ZERO to 1, etc. However, the values of INF and -INF are undetermined and therefore, it makes no sense to consider INF/INF, $-$INF $+$ INF, etc. These expressions are therefore evaluated to UNDF. A runtime error will occur if the value UNDF is assigned to an identifier.

Numerical behavior

The symbol ZERO behaves like zero numerically, but its logical value is one. Using this symbol, you can make a distinction between the default value of 0 and an assigned ZERO. As an illustration, consider a distance matrix with distances between selected factory-depot combinations. A missing distance value evaluates to 0, and could mean that the particular factory-depot combination should not be considered. A ZERO value in that case could be used to indicate that the combination should be considered even though the corresponding distance is zero because the depot and factory happen to be one facility.

The symbol ZERO

Whenever the values 0 and ZERO appear in the same expression with equal priority, the value of ZERO prevails. For example, the expressions 0 + ZERO or max(0,ZERO) will both result in a numerical value of ZERO. In this way, the logically distinctive effect of ZERO is retained as long as possible. You should note, however, that AIMMS will evaluate the multiplication of 0 with *any* special number to 0.

Expressions with 0 and ZERO

The symbol NA can be used for missing data. The interpretation is "this number is not yet known". Any operation that uses NA and does not use the symbol UNDF will also produce the result NA. AIMMS can reason with this value as it propagates the value NA through its computations and assignments. The only exception is the condition in control flow statements where it must be known whether the result of that condition is equal to 0.0 or not, see also Section 8.3.

The symbol NA

The symbol UNDF cannot be input directly by a user, but is, besides an error message, the result of an undefined or illegal arithmetic operation. For example, 1/ZERO, 0/0, (-2)^0.1 all result in UNDF. Any operation containing the UNDF symbol evaluates to UNDF.

The symbol UNDF

6.1.2 List expressions

A *list* is a collection of *element-value pairs*. In a list a single element or range of elements is combined with a numerical, element-, or string-valued expression, separated by a *colon*. List expressions are the numerical extension of enumerated set expressions. The elements to which a value is assigned inside a list,

Element-value pairs

are specified in exactly the same manner as in an enumerated set expression as explained in Section 5.1.1.

enumerated-list : *Syntax*
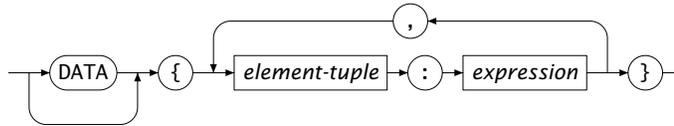
By preceding the list expression with the keyword DATA, it becomes a *constant* list expression, in a similar fashion as with constant set expressions (see Section 5.1.1). In a constant list expression, set elements need not be quoted and the assigned values must be constants. All other list expressions are symbolic, in which both the elements and the assigned values are the result of expression evaluation.

Constant versus symbolic

The following assignments illustrate the use of list expressions. *Example*

■ The following constant list expression assigns distances to tuples of cities.

```
Distance(i,j) := DATA {
    (Amsterdam, Rotterdam  ) : 85 [km] ,
    (Amsterdam, 'The Hague') : 65 [km] ,
    (Rotterdam, 'The Hague') : 25 [km]
} ;
```

■ The following symbolic list expression assigns a certain status to every node in a number of dynamically computed ranges.

```
NodeUsage(i) := {
    FirstNode            .. FirstNode + Batch - 1   : 'InUse'  ,
    FirstNode + Batch    .. FirstNode + 2*Batch - 1 : 'StandBy' ,
    FirstNode + 2*Batch  .. LastNode                : 'Reserve'
} ;
```

6.1.3 References

Sets, parameters and variables can be referred to by name resulting in a set-, set element-, string-valued, or numerical quantity. A reference can be scalar or multidimensional, and index positions may contain either indices or element expressions. By specifying a case reference in front, a reference can refer to data from cases that are not in memory.

References

reference :   *Syntax*
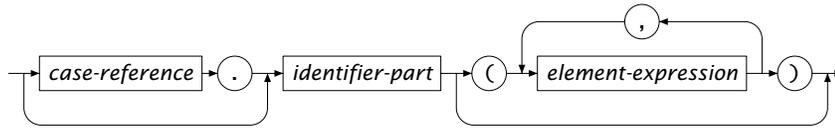
identifier-part :
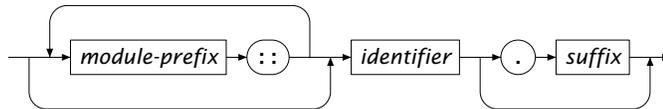
A *scalar* set, parameter or variable has no indexing (dimension) and is refer-   *Scalar versus*
enced simply by using its identifier. Indexed sets, parameters and variables   *indexed*
have dimensions equal to the number of indices.

The right-hand sides of the following assignments are examples of references   *Example*
to scalar and indexed identifiers.

```
MainCity                := 'Amsterdam' ;

DistanceFromMainCity(i) := Distance( MainCity, i );

SecondNextCity(i)       := NextCity( NextCity(i) );

NextPeriodStock(t)      := Stock( t + 1 );
```

The last two references, which make use of lag and lead operators and element   *Undefined*
parameters, may sometimes be undefined. When used in an expression such   *references*
undefined references evaluate to the empty set, zero, the empty element, or the
empty string, depending on the value type of the identifier. When an undefined
lag or lead operator or element parameter occurs on the left-hand side of an
assignment, the assignment is skipped. For more details, refer to Section 8.2.

When your model contains one or more Modules, your model will be supplied   *Referring to*
multiple additional namespaces besides the global namespace, one for each   *module*
module. Identifiers declared within a module are, by default, not contained   *identifiers*
in the global namespace. To refer to such identifiers outside the module, you
have to prefix the identifier name with a module-specific prefix and the ::
namespace resolution operator. Modules and the namespace resolution opera-
tor are discussed in full detail in Section 35.4.

When a reference is preceded by a *case reference*, AIMMS will not retrieve the *Referring to* requested identifier data from the case in memory, but from the case file as- *other cases* sociated with the case reference. Case references are elements of the (prede- fined) set `AllCases`, which contains all the cases available in the data manager of AIMMS. The AIMMS User's Guide describes all the mechanisms that are available and functions that you can use to let an end-user of your application select one or more cases from the set of all available cases. Case referencing is useful when you want to perform advanced case comparison over multiple cases.

The following computes the differences of the values of the variable `Transport` *Example* in the current case compared to its values in all cases in the set `CurrentCase-Selection`.

```
for ( c in CurrentCaseSelection ) do
    Difference(c,i,j) := c.Transport(i,j) - Transport(i,j) ;
endfor;
```

During execution, AIMMS will (temporarily) retrieve the values of `Transport` from all requested cases to compute the difference with the data of the current case.

6.1.4 Arithmetic functions

AIMMS provides the commonly used standard arithmetic functions such as the *Standard* trigonometric functions, logarithms, and exponentiations. Table 6.2 lists the *functions* available arithmetic functions with their arguments and result, where x is an extended range arithmetic expressions, m, n are integer expressions, i is an index, l is a set element, I is a set identifier, and e is a scalar reference.

Special caution is required when one or more of the arguments in the functions *Functions and* are special symbols of AIMMS' extended range arithmetic. If the value of any *extended* of the arguments is UNDF or NA, then the result will also be UNDF or NA. If the *arithmetic* value of any of the arguments is ZERO and the numerical value of the result is zero, the function will return ZERO.
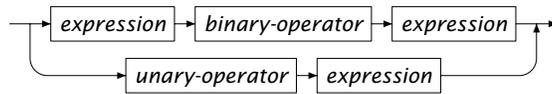
6.1.5 Numerical operators

Using unary or binary numerical operators you can construct numerical ex- pressions that consist of multiple terms and/or factors. The syntax follows.

Function	Meaning
$\texttt{Abs}(x)$	absolute value $|x|$
$\texttt{Exp}(x)$	e^x
$\texttt{Log}(x)$	$\log_e(x)$ for $x > 0$,UNDF otherwise
$\texttt{Log10}(x)$	$\log_{10}(x)$ for $x > 0$, UNDF otherwise
$\texttt{Max}(x_1,\ldots,x_n)$	$\max(x_1,\ldots,x_n)$ $(n > 1)$
$\texttt{Min}(x_1,\ldots,x_n)$	$\min(x_1,\ldots,x_n)$ $(n > 1)$
$\texttt{Mod}(x_1,x_2)$	$x_1 \bmod x_2 \in [0,x_2)$ for $x_2 > 0$ or $\in (x_2,0]$ for $x_2 < 0$
$\texttt{Div}(x_1,x_2)$	$x_1 \operatorname{div} x_2$
$\texttt{Sign}(x)$	$\operatorname{sign}(x) = +1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$
$\texttt{Sqr}(x)$	x^2
$\texttt{Sqrt}(x)$	$\sqrt{x}$ for $x \geq 0$, UNDF otherwise
$\texttt{Power}(x_1,x_2)$	$x_1^{x_2}$, alternative for $\texttt{x\^{}y}$ (see Section 6.1.5)
$\texttt{ErrorF}(x)$	$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}}\, dt$
$\texttt{Cos}(x)$	$\cos(x)$; x in radians
$\texttt{Sin}(x)$	$\sin(x)$; x in radians
$\texttt{Tan}(x)$	$\tan(x)$; x in radians
$\texttt{ArcCos}(x)$	$\arccos(x)$; result in radians
$\texttt{ArcSin}(x)$	$\arcsin(x)$; result in radians
$\texttt{ArcTan}(x)$	$\arctan(x)$; result in radians
$\texttt{Degrees}(x)$	converts x from radians to degrees
$\texttt{Radians}(x)$	converts x from degrees to radians
$\texttt{Cosh}(x)$	$\cosh(x)$
$\texttt{Sinh}(x)$	$\sinh(x)$
$\texttt{Tanh}(x)$	$\tanh(x)$
$\texttt{ArcCosh}(x)$	$\operatorname{arccosh}(x)$
$\texttt{ArcSinh}(x)$	$\operatorname{arcsinh}(x)$
$\texttt{ArcTanh}(x)$	$\operatorname{arctanh}(x)$
$\texttt{Card}(I[, \mathit{suffix}])$	cardinality of (suffix of) set, parameter or variable I
$\texttt{Ord}(i)$	ordinal number of index i in set I (see also Table 5.2)
$\texttt{Ord}(l[,I])$	ordinal number of element l in set I
$\texttt{Ceil}(x)$	$\lceil x \rceil$ = smallest integer $\geq x$
$\texttt{Floor}(x)$	$\lfloor x \rfloor$ = largest integer $\leq x$
$\texttt{Precision}(x,n)$	x rounded to n significant digits
$\texttt{Round}(x)$	x rounded to nearest integer
$\texttt{Round}(x,n)$	x rounded to n decimal places left $(n < 0)$ or right $(n > 0)$ of the decimal point
$\texttt{Trunc}(x)$	truncated value of x: $\texttt{Sign}(x)*\texttt{Floor}(\texttt{Abs}(x))$
$\texttt{NonDefault}(e)$	1 if e is not at its default value, 0 otherwise
$\texttt{MapVal}(x)$	$\texttt{MapVal}$ value of x according to Table 6.1

Table 6.2: Intrinsic numerical functions of AIMMS

operator-expression :   *Syntax*

The order of precedence of the standard numerical operators in AIMMS is given   *Standard*
in Table 6.3. Parentheses may be used to override the precedence order. Ex-   *numerical*
pression evaluation is from left to right.   *operators*

Operator	Meaning	Precedence
Unary		
+	positive	n/a
–	negative	n/a
Binary		
^	exponentiation	3 (high)
*	multiplication	2
/	division	2
+	addition	1
–	subtraction	1 (low)

Table 6.3: Numerical operators

The expression   *Example*

```
p1 + p2 * p3 / p4^p5
```

is parsed by AIMMS as if it had been written

```
p1 + [(p2 * p3) / (p4^p5)]
```

In general, it is better to use parentheses than to rely on the precedence and
associativity of the operators. Not only because it prevents you from making
unwanted mistakes, but also because it makes your intentions clearer.

Special restrictions apply to the exponential operator "^". AIMMS accepts the   *Exponential*
following combinations of left-hand side operand (called the *base*), and right-   *operator*
hand side operand (called the *exponent*):

- a positive base with a real exponent,
- a negative base with an integer exponent,
- a zero base with a positive exponent, and
- a zero base with a zero exponent results in one (as controlled by the
  option power_0_0).

6.1.6 Numerical iterative operators

Iterative operators are used to express repeated arithmetic operations, such as summation, in a concise manner. The arithmetic iterative operators supported by AIMMS are listed in Table 6.4. The second column in this table refers to the required number of expression arguments following the binding domain argument, while the last column refers to the result of the operator in case of an empty domain.

Arithmetic iterative operators

Name	# Expr.	Computes over all elements in the domain	Default
Sum	1	the sum of the expression	0
Prod	1	the product of the expression	1
Count	0	the total number of elements in the domain	0
Min	1	the minimum value of the expression	INF
Max	1	the maximum value of the expression	-INF

Table 6.4: Arithmetic iterative operators

The Min and Max operators return the minimum or maximum value of an expression. The allowed expressions are:

Compared expressions

- numerical expressions, in which case AIMMS returns the lowest or highest numerical values,
- string expressions, in which case AIMMS returns the strings which are first or last with respect to the normal alphabetic ordering, and
- element expressions, in which case AIMMS returns the elements with the lowest or highest ordinal numbers (see also Section 5.2.2).

The following assignments are valid examples of the use of the arithmetic iterative operators.

Example

```
NumberOfRoutes     := Count( (i,j) | Distance(i,j) ) ;
NettoTransport(i)  := Sum( j, Transport(i,j) - Transport(j,i) ) ;
MaximumTransport(i) := Max( j, Transport(i,j) ) ;
```

6.1.7 Statistical functions and operators

AIMMS provides the most commonly used distributions. They are listed in Table 6.5, together with the required type of arguments and a description of the result. You can find a more detailed description of these distributions in Appendices A.1 and A.2. When called as functions inside your model, they behave as random number generators.

Distributions

Distribution	Meaning
Binomial(p, n)	Binomial distribution with probability p and number of trials n
NegativeBinomial(p, r)	Negative Binomial distribution with probability p and number of successes r
Poisson(λ)	Poisson distribution with rate λ
Geometric(p)	Geometric distribution with probability p
HyperGeometric(p, n, N)	Hypergeometric distribution with initial probability of success p, number of trials n and population size N
Uniform(*min,max*)	Uniform distribution with lower bound *min* and upper bound *max*
Triangular(β, min, max)	Triangular distribution with shape β, lower bound *min*, and upper bound *max*, where $\beta = (x_{\text{peak}} - min)/(max - min)$
Beta(α, β, min, max)	Beta distribution with shapes α, β, lower bound *min*, and upper bound *max*
LogNormal(β, min, s)	Lognormal distribution with shape β, lower bound *min*, and scale s
Exponential(*min, s*)	Exponential distribution with lower bound *min* and scale s
Gamma(β, min, s)	Gamma distribution with shape β, lower bound *min*, and scale s
Weibull(β, min, s)	Weibull distribution with shape β, lower bound *min*, and scale s
Pareto(β, l, s)	Pareto distribution with shape β, location l, and scale s (lower bound $= l + s$)
Normal(μ, σ)	Normal distribution with mean μ and standard deviation σ
Logistic(μ, s)	Logistic distribution with mean μ and scale s
ExtremeValue(l, s)	Extreme Value distribution with location l and scale s

Table 6.5: Distributions available in AIMMS

You can set the seed of the random number generators for all distributions using the execution option seed. By setting the seed explicitly you can guarantee that your model results are reproducible.

Setting the seed

Each distribution in Table 6.5 can be used as an argument for four operators: DistributionCumulative and DistributionInverseCumulative, and their derivatives DistributionDensity and DistributionInverseDensity. In the explanation below it is assumed that $\alpha \in [0, 1]$, $x \in (-\infty, \infty)$, and X a random variable distributed according to the given distribution *distr*.

Cumulative distributions and their derivatives

- DistributionCumulative(*distr*,*x*) computes the probability $P(X \le x)$.
- DistributionInverseCumulative(*distr*,α) computes the smallest x such that the probability $P(X \le x) \ge \alpha$, except for $\alpha = 0$ which returns the lowest possible value for X.
- DistributionDensity(*distr*,*x*) computes for continuous distributions the probability density $\lim_{\alpha \downarrow 0} P(x \le X \le x+\alpha)/\alpha$. For discrete distributions, the operator is only defined for integer values of x and returns $P(X = x)$.
- DistributionInverseDensity(*distr*,α) is the derivative of DistributionInverseCumulative. For more details you are referred to Appendix A.3.

For the continuous distributions in Table 6.5 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.

Use in constraints

The following statements demonstrate how the distributions can be used to perform statistical tasks.

Example

1. Draw a random number from a distribution.

        Draw := Normal(0,1);
        Draw := Uniform(LowestValue, HighestValue);

2. Compute the probability of at most 10 successes out of 50 trials, with a 0.25 probability of success.

        Probability := DistributionCumulative(Binomial(0.25,50), 10);

3. Compute a two-sided 90% confidence interval of a Normal(0,1) distribution.

        LeftBound := DistributionInverseCumulative(Normal(0,1), 0.05);
        RightBound := DistributionInverseCumulative(Normal(0,1), 0.95);

The distributions, listed in Table 6.5, make it possible for you to execute a stochastic experiment based on your model representation. In order to analyze the subsequent results, Aimms provides a number of statistical iterative operators which are listed in Table 6.6. The second column in this table refers to the required number of expression arguments following the binding domain argument. For the most common sample operators, Aimms provides distribution operators to calculate the corresponding expected values, assuming the sample is drawn from a given distribution. These distribution operators are listed in Table 6.7. A more detailed description of these operators is provided in Appendix A.

Statistical operators

Name	# Expr.	Computes over all elements in the domain
Mean,	1	the (arithmetic) mean
GeometricMean	1	the geometric mean
HarmonicMean	1	the harmonic mean
RootMeanSquare	1	the root mean square
Median	1	the median
SampleDeviation	1	the standard deviation of a sample
PopulationDeviation	1	the standard deviation of a population
Skewness	1	the coefficient of skewness
Kurtosis	1	the coefficient of kurtosis
Correlation	2	the correlation coefficient
RankCorrelation	2	the rank correlation coefficient

Table 6.6: Statistical sample operators

Assume that p is an index into a set that has been used to index a number of experiments resulting in observables x(p) and y(p). Then the following assignments demonstrate the use of the statistical operators in AIMMS.

Example

```
MeanX         := Mean(p, x(p));
MeanX         := Mean(p | x(p), x(p));
DeviationX    := SampleDeviation(p, x(p));
CorrelationXY := Correlation(p, x(p), y(p));
```

In case the x values are drawn from a Binomial(0.6,8) distribution the expected value of MeanX is given by

```
ExpectedMeanX := DistributionMean(Binomial(0.6,8));
```

For all distributions, the units of measurement (see also Chapter 32) of parameters and result should be consistent. The unit relationships for each distribution are described in Appendix A in full detail. In the presence of units of measurement within your model, AIMMS will perform a unit consistency check.

Units of measurement

For easy visualization of statistical data, AIMMS offers support for creating histograms based on a large collection of observed values. Through a number of predefined procedures and functions, AIMMS allows you to flexibly create interval-based histogram data, which can easily be displayed, for instance, using the standard (graphical) AIMMS bar chart object. For further information about creating and displaying histograms, as well as an illustrative example, you are referred to section A.6 in the Appendix.

Histogram support

Name	Computes for a given distribution
DistributionMean	the (arithmetic) mean
DistributionDeviation	the (standard) deviation
DistributionVariance	the variance (the square of the deviation)
DistributionSkewness	the coefficient of skewness
DistributionKurtosis	the coefficient of kurtosis

Table 6.7: Statistical distribution operators

In addition to the distribution and statistical operators listed above, AIMMS also offers support for the most common combinatoric calculations. Table 6.8 contains the list of combinatoric functions that are available in AIMMS.

Combinatoric functions

Function	Meaning
Factorial(n)	$n!$
Combination(n, m)	$\binom{n}{m}$
Permutation(n, m)	$m! \cdot \binom{n}{m}$

Table 6.8: Combinatoric functions

6.1.8 Financial functions

AIMMS provides an extensive library of financial functions for a variety of financial applications. The available functions can be classified as follows.

Financial functions

- Functions for the computation of the depreciation of assets using various methods such as fixed-declining balance method, double-declining balance method, etc.
- Functions for computing various quantities regarding investments that consist of a series of constant or variable periodic cash flows. The computed quantities include present value, net present value, future value, internal rate of return, interest and principal payments, etc.
- Functions for computing various security-related quantities of, for instance, discounted securities, securities that pay periodic interest and securities that pay interest at maturity. The computed quantities include yield, interest rate, redemption, price, accrued interest, etc.

The precise description of all financial functions available in AIMMS is not included in this Language Reference. You can find a complete list of the available financial functions on pages 103 and further of the AIMMS Function Reference. The Function Reference provides a description as well as the prototype of every financial function present in AIMMS.

*Consult the online function reference*

6.1.9 Conditional expressions

There are two ways to specify expressions that adopt different values depending on one or more logical conditions. The ONLYIF operator is the simpler and operates as it sounds. The IF-THEN-ELSE expression is more powerful in its ability to distinguish several cases.

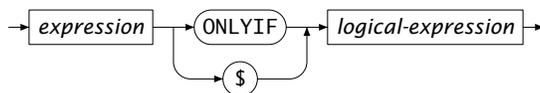*Two conditional expressions*

conditional-expression :

Syntax
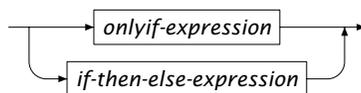
The simplest way of specifying a conditional expression is to use the ONLYIF operator. Its syntax is given by

The ONLYIF operator

onlyif-expression :

The ONLYIF expression evaluates to the arithmetic expression in the first argument if the logical condition of the second argument is true. Otherwise, it is zero. The "$" symbol can be used as a synonym for the ONLYIF operator.

A simple example of the use of the ONLYIF operator is given by the assignment

Example

```
AverageVelocity := (Distance / TravelTime) ONLYIF TravelTime ;
```

or equivalently, using the $ operator,

```
AverageVelocity := (Distance / TravelTime) $ TravelTime ;
```

Both expressions evaluate to Distance / TravelTime if TravelTime assumes a nonzero value, or to zero otherwise. In Section 12.2 you will see that this particular expression can be written even more concisely using the sparsity modifier "$".

A much more flexible way for specifying conditional expressions is given by the IF-THEN-ELSE operator. The syntax of the IF-THEN-ELSE expression is given below.

IF-THEN-ELSE expressions

if-then-else-expression :   *Syntax*
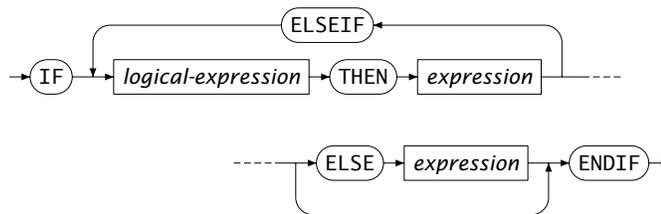
The IF-THEN-ELSE expression works like a *switch statement*—a series of ELSEIFs   *Explanation*
can be used to denote numerous special cases. The value of the IF-THEN-ELSE
expression is the first numerical expression for which the corresponding logi-
cal condition is true. If none of the conditions are true, then the value will be
the numerical expression after the ELSE keyword if present or zero otherwise.

A simple illustration of the use of the IF-THEN-ELSE construction is given by   *Example*
the assignments

```
AverageVelocity := IF TravelTime THEN Distance / TravelTime ENDIF ;
```

which is equivalent to the ONLYIF expression above. A more elaborate example
is given by the assignment

```
WeightedDistance(i) :=
    IF      Distance(i) <= 100 THEN Distance(i)
    ELSEIF Distance(i) <= 200 THEN (100 + Distance(i)) / 2
    ELSEIF Distance(i) <= 300 THEN (250 + Distance(i)) / 3
    ELSE   550 / 3
    ENDIF ;
```

The expression takes the value associated with the first logical expression that
is true.

6.2 Logical expressions

Logical expressions are expressions that evaluate to a logical value—0.0 for   *Logical*
false and 1.0 for true. Aimms supports several types of logical expressions.   *expressions*

logical-expression :
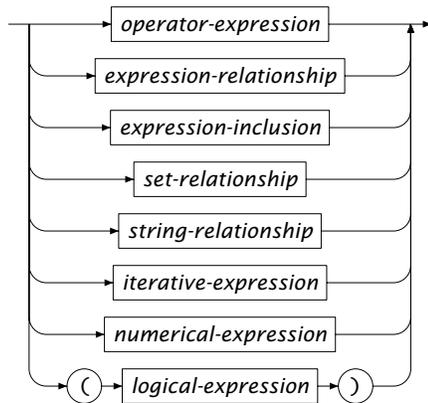
As AIMMS permits numerical expressions as logical expressions it is important to discuss how numerical expressions are interpreted logically, and how logical expressions are interpreted numerically. Numerical expressions that evaluate to zero (0.0) are false, while all others (including ZERO, NA and UNDF) are true. A false logical expression evaluates to zero (0.0), while a true logical expression evaluates to one (1.0). If one or more of the operands of a logical operator is UNDF or NA, the numerical value is also UNDF or NA. Note that AIMMS will not accept expressions that evaluate to UNDF or NA in the condition in control flow statements, where it must be known whether the result of that condition is equal to 0.0 or not (see also Section 8.3).

Numerical expressions as logical

Table 6.9 illustrates the different interpretation of a number of numerical and logical expressions as either a numerical or a logical expression. See also Table 6.10 for the results associated with the AND operator.

Example

Expression	Numerical value	Logical value
3*(2 > 1)	3.0	true
3*(1 > 2)	0.0	false
(1 < 2) + (2 < 3)	2.0	true
max((1 < 2),(2 < 3))	1.0	true
2 AND 0.0	0.0	false
2 AND ZERO	1.0	true
2 AND NA	NA	true
UNDF < 0	UNDF	true

Table 6.9: Numerical and logical values

6.2.1 Logical operator expressions

AIMMS supports the unary logical operator NOT and the binary logical operators AND, OR, and XOR. Table 6.10 gives the logical results of these operators for zero and nonzero operands.

Unary and binary logical operators

Operands		Result			
a	b	a AND b	a OR b	a XOR b	NOT a
0	0	0	0	0	1
0	nonzero	0	1	1	1
nonzero	0	0	1	1	0
nonzero	nonzero	1	1	0	0

Table 6.10: Logical operators

The precedence order of these operators from highest to lowest is given by NOT, AND, OR, and XOR respectively. Whenever the precedence order is not immediately clear, it is advisable to use parentheses. Besides preventing unwanted mistakes, it also make your model easier to understand and maintain.

Precedence order

The expression

Example

```
NOT a AND b XOR c OR d
```

is parsed by AIMMS as if it were written

```
((NOT a) AND b) XOR (c OR d).
```

Due to the sparse execution system underlying AIMMS it is not guaranteed that logical expressions containing binary logical operators are executed in a strict left-to-right order. If you are a C/C++ programmer (where logical conditions are executed in a strict left-to-right order), you should take extra care to ensure that your logical conditions do not depend on this assumption.

Execution order

6.2.2 Numerical comparison

Numerical relationships compare two numerical expressions, using one of the relational operators =, <>, >, >=, <, or <=. Numerical inclusions are equivalent to two numerical relationships, and indicate whether a given expression lies within two bounds.
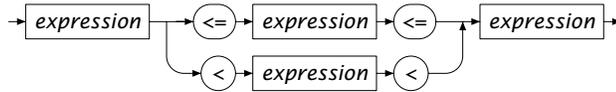
Numerical comparison

expression-relationship :   *Syntax*

expression-inclusion :

For two real numbers x and y the result of the comparison $x \gtreqless y$, where $\gtreqless$ *Numerical*
denotes any relational operator, depends on two tolerances   *tolerances*

- `Equality_Absolute_Tolerance` (denoted as ε_a), and
- `Equality_Relative_Tolerance` (denoted as ε_r).

You can set these tolerances through the options dialog box. Their default
values are 0 and 10^{-13}, respectively. If the number $\varepsilon_{x,y}$ is given by the formula

$$\varepsilon_{x,y} = \max(\varepsilon_a, \varepsilon_r \cdot x, \varepsilon_r \cdot y),$$

then the relational operators evaluate as shown in the Table 6.11.

AIMMS expression	Evaluates as
$x = y$	$|x - y| \le \varepsilon_{x,y}$
$x <> y$	$|x - y| > \varepsilon_{x,y}$
$x <= y$	$x - y \le \varepsilon_{x,y}$
$x < y$	$x - y < -\varepsilon_{x,y}$

Table 6.11: Interpretation of numerical tolerances

For any combination of an ordinary real number with one of the special sym-   *Comparison for*
bols ZERO, INF, and -INF, the relational operators behave as expected. If any *extended*
of the operands is either NA or UNDF, relationships other than = and <> also *arithmetic*
evaluate to NA or UNDF and hence, as a logical expression, to true. In addition,
the logical expressions INF = INF and -INF = -INF evaluate to true.

One can formulate numerous logical expressions to test for a zero value, and   *Testing for zero*
one should be clear on the desired result. The following example makes the   *value*
point.

```
p_inv(i)           := 1 / p(i);
p_inv(i | p(i))    := 1 / p(i);
p_inv(i | p(i) <> 0) := 1 / p(i);
```

The first assignment will produce a runtime error when p(i) assumes a value of 0 or ZERO. The second assignment will filter out the 0's, but not the ZERO values because ZERO evaluates to the logical value "true". The last assignment will never produce runtime errors, because of the *numerical* comparison to 0.
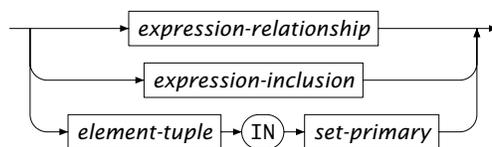

6.2.3 Set and element comparison

AIMMS features very powerful logical set comparison operators. Not only can sets and their elements be compared using relational operators, but you can also check for set membership with the IN operator.

Set relationships

set-relationship :

Syntax

Set elements that lie in the same set can be compared according to their relative position inside that set. You can also compare the positions of arbitrary set element expressions, as long as AIMMS is able to determine a unique domain set in which the comparison has to take place. The allowed relational operators are =, <>, <, <=, >, and >=. As with numerical expression, AIMMS also allows you to specify an inclusion relationship as a form of repeated comparison to verify whether an element lies within two boundary elements.

Element relationship and inclusion

The relational operators for element relationships are conveniently defined in terms of the Ord function. Let S be a simple set, i and j indices or element parameters in S, $\pm$ any of the lag or lead operators +, ++, - or --, m and n integer expressions, and $\gtrless$ one of the operators =, <>, <, <=, >, or >=. The relational operators $\gtrless$ have the following definition for set elements, provided that the set elements on both sides of the relational operator exist.

Element comparison

$$i \pm m \gtrless j \pm n \quad \Leftrightarrow \quad \begin{cases} i \pm m \text{ and } j \pm n \text{ are both defined, and} \\ \text{Ord}(i \pm m, S) \gtrless \text{Ord}(j \pm n, S) \end{cases}$$

Note that this type of relational expression evaluates to "false" if one or both of the operands do not refer to existing set elements.

Only elements that lie in the same set are comparable using the <, <=, >, and >= operators. The = and <> operators can also be used when the operands merely share the same root set.

Compare within the same set

The following set assignments demonstrate the correct use of element comparisons.

Example

```
FuturePeriods := { t in Periods | CurrentPeriod <= t <= PlanningHorizon } ;

BandMatrix := { (i,j) | i - BandWidth <= j <= i + BandWidth } ;
```

Set membership can be tested using the IN operator. This operator checks whether a set element or an element tuple on the left-hand side is a member of the set expression on the right-hand side. Both operands must have the same root set.

Set membership

Assume that all one-dimensional sets in the following two assignments share the same root set Cities. Then these statements illustrate the correct use of the logical IN operator.

Example

```
NeighborhoodRoutes := { (i,j) in Routes | j in NeighborhoodCities(i) } ;
ExcludedCities     := { i in ( SmallCities + ForeignCities ) } ;
```

Sets can be logically compared using any of the relational operators =, <>, <, <=, > and >=. The inequality operators denote the usual subset relationships. They replace the standard "contained in" operators $\subsetneq$, $\subseteq$, $\supsetneq$ and $\supseteq$ which are not available on many keyboards.

Set comparisons

The following statement illustrates a logical set comparison operator.

Example

```
IF ( RoutesWithTransport <= NeighborhoodRoutes ) THEN
    DialogMessage( "Solution only contains neighborhood transports" );
ENDIF;
```

6.2.4   String comparison

Besides their use for comparison of numerical, element- and set-valued expressions, the relational operators =, <>, <, <=, >, and >= can also be used for string comparison. When used for string comparison, AIMMS employs the usual lexicographical ordering. String comparison in AIMMS is case sensitive by default, i.e. strings that only differ in case are considered to be unequal. You can modify this behavior through the option Case_Sensitive_String_Comparison.

String comparison

All the following string comparisons evaluate to true.

```
"The city of Amsterdam" <> "the city of amsterdam"     ! Note case
"The city of Amsterdam" <> "The city of Amsterdam "    ! Note last space
"The city of Amsterdam" <  "The city of Rotterdam"
```

6.2.5 Logical iterative expressions

Logical iterative operators verify whether some or all elements in a domain satisfy a certain logical condition. Table 6.12 lists all logical iterative operators supported by Aimms. The second column in this table refers to the required number of expression arguments following the binding domain argument.

Logical iterative operators

Name	# Expr.	Meaning
Exists	0	true if the domain is not empty
Atleast	1	true if the domain contains at least n elements
Atmost	1	true if the domain contains at most n elements
Exactly	1	true if the domain contains at exactly n elements
ForAll	1	true if the expression is true for all elements in the domain

Table 6.12: Logical iterative operators

The following statements illustrate the use of some of the logical iterative operators listed in Table 6.12.

Example

```
MultipleSupplyCities := { i | Atleast( j | Transport(i,j), 2 ) } ;

IF ( ForAll( i, Exists( j | Transport(i,j) ) ) ) THEN
    DialogMessage( "There are no cities without a transport" );
ENDIF ;
```

6.3 Operator precedence

In the previous sections we have introduced unary and binary operators for several types of expressions, together with their relative precedence order. Table 6.13 provides an overview of all of them. The last column lists the expression types in which the operator is used, where the letters "N", "L", "E", and "S" stand for *N*umerical, *L*ogical, set *E*lement and *S*et expressions, respectively.

Combined precedence order

Precedence	Name	Type
14	`ONLYIF $`	N
13	`^`	N
12	`+ - (unary)`	N
11	`* /`	N,S
10	`+ - ++ -- (binary)`	N,E,S
9	`CROSS`	S
8	`IN`	L
7	`< <= > >= = <>`	L
6	`NOT`	L
5	`AND`	L
4	`OR`	L
3	`XOR`	L
2	`|`	S
1	`IF THEN ELSEIF ELSE ENDIF`	N

Table 6.13: Operator precedence (highest to lowest)

6.4 `MACRO` declaration and attributes

The `MACRO` facility offers a mechanism for parameterizing expressions. Macros are useful for enhancing the readability of models, and avoiding inconsistencies in frequently used expressions.

Macro facility

Macros are declared as ordinary identifiers in your model. They can have arguments. The attributes of a `Macro` declaration are listed in Table 6.14.

Declaration and attributes

Attribute	Value-type	See also page
`Text`	*string*	19
`Arguments`	*argument-list*	
`Comment`	*comment string*	19
`Definition`	*expression*	34

Table 6.14: `Macro` attributes

The `Definiton` attribute of a macro declaration is the replacement text that is substituted when a macro is used in the model text. The (optional) `Arguments` of a macro must be scalar entities. Unlike function arguments, however, you do not have to declare `Macro` arguments as local identifiers. The `Definition` of a macro must be a valid expression in its arguments.

The `Definition` attribute

When you define a macro with arguments, the actual replacement text depends on the arguments that are supplied to it, as illustrated in the following example. Using the macro declaration   *Example*

```
Macro MyAverage {
    Arguments  : (dom, expr);
    Definition : Sum(dom, expr) / Count(dom);
}
```

the assignments

```
  AverageTransport   := MyAverage( (i,j), Transport(i,j) );
  AverageNZTransport := MyAverage( (i,j) | Transport(i,j),  Transport(i,j) );
```

are compiled as if they read:

```
  AverageTransport   := Sum( (i,j), Transport(i,j) ) / Count( (i,j) );
  AverageNZTransport :=
        Sum  ( (i,j) | Transport(i,j),  Transport(i,j) ) /
        Count( (i,j) | Transport(i,j) );
```

When you use a macro with arguments, the actual arguments *must* be valid expressions. As a result, there is no need to add additional braces to the replacement text of the macro, like, for instance, in the C programming language. The following example illustrates this point.   *Expression substitution*

```
Macro MyMult {
    Arguments  : (x,y);
    Definition : x*y;
}
```

Using this macro, the expression

```
  a + MyMult(b+c,d+e) + f
```

will evaluate to

```
  a + ((b+c)*(d+ e)) + f
```

instead of

```
  a + b + c*d + e + f
```

In many execution statements you have a choice to use either macros or de-
fined parameters as a mechanism to replace complicated expressions by de-
scriptive names. While a macro is purely substituted by its replacement text,
the current value of a defined parameter is stored and looked up when needed.
When deciding whether to use a macro or a defined parameter, you should con-
sider both storage and computational consequences. Macros are recomputed
every time they are referenced, and therefore there may be an unnecessary
time penalty if the macro is called with identical arguments in more than one
place within your model. When storage considerations are important, a macro
may be attractive since it does not introduce additional parameters.

Macro versus defined parameters

You should also consider your choices when you use a macro with variables
as arguments in a constraint. In this case, you also have the option to use
a defined variable, or a defined `Inline` variable (see also Section 14.1). The
following considerations are of interest.

Macro versus defined variables

- A macro can produce different expressions of the same structure for dif-
  ferent identifier arguments, but does not allow you to specify a domain
  restriction that will reduce the number of generated columns in the ma-
  trix.
- Defined and `Inline` variables support an index domain to restrict the
  number of generated columns, but only allow an expression in terms of
  fixed identifiers. Compared to a macro or an `Inline` variable, the number
  of rows and columns increases for a defined variable, but if the variable
  is referenced more than once in the other constraints, it will result in a
  smaller number of nonzeros.
- An advantage of variables (both defined and `Inline`) over macros is that
  their final values are stored by Aimms, and can be retrieved in other
  execution statements or in the graphical user interface, whereas a macro
  has to be recomputed all the time.