

---

## AIMMS Language Reference - Execution Statements

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](http://www.aimms.com) or order your hard-copy at [www.lulu.com/aimms](http://www.lulu.com/aimms).

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

Email: [info@aimms.com](mailto:info@aimms.com)  
WWW: [www.aimms.com](http://www.aimms.com)

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation.  $\text{T}_{\text{E}}\text{X}$ ,  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ , and  $\text{A}_{\text{M}}\text{S}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$  are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

**Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.**

**In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.**

This documentation was typeset by Paragon Decision Technology B.V. using  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  and the LUCIDA font family.

## **Part III**

---

# **Procedural Language Components**

# Chapter 8

## Execution Statements

This chapter describes the interaction between the nonprocedural and procedural execution mechanisms in AIMMS. In addition, the major execution statements like the assignment statement, the flow control statements, and the OPTION statement are discussed. Other important execution statements such as procedure calls, the SOLVE statement, as well as data control and display statements are discussed in various other chapters.

*This chapter*

---

### 8.1 Procedural and nonprocedural execution

The definitions specified inside the declarations of sets and parameters together form a system of functional relationships. As discussed in Chapter 7 AIMMS automatically determines the dependency between the identifiers that are used inside these relationships. Based on the (required) *a-cyclic* dependency structure between identifiers (see also Section 7.1), AIMMS knows the exact order in which identifiers need to be computed. Execution based on definitions is not controlled by the user, but takes place automatically when values are needed.

*Execution based on definitions*

Procedures are self-contained programs with a body consisting of execution statements. These statements typically determine the value of those identifiers which cannot be defined using a single functional relationship. Execution using procedures proceeds according to the order of execution statements encountered inside each procedure, and is therefore controlled by the user.

*Execution based on procedures*

Whenever a set or a parameter with a definition is used in an execution statement inside a procedure, and its value is not up-to-date due to previous data changes, AIMMS will compute its current value just prior to executing the corresponding statement. This updating facility in AIMMS forms the necessary and powerful connection between automatic execution based on definitions and user-initiated execution based on procedures.

*Relating definitions and procedures*

Procedural and nonprocedural execution both have their own natural role in an AIMMS application. Identifier definitions are the most convenient way to define unique functional relationships between various identifiers in your model—and keep them up-to-date at all times. Procedures provide a powerful tool to specify the algorithms that are needed to compute the identifier values without a direct functional relationship. Procedural statements are also required to communicate data between AIMMS and external data sources such as files and databases.

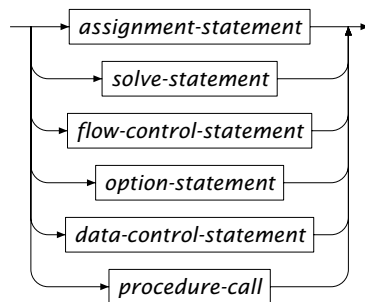
*Definitions and algorithms*

AIMMS provides a rich set of execution statements that you can use to compose your procedures. Available statements include a versatile assignment statement, statements for data and option management, the most common flow control statements, calls to other procedures, and a powerful SOLVE statement to solve various types of optimization programs.

*Execution statements*

*statement :*

*Syntax*



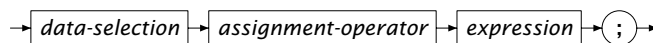
## 8.2 Assignment statements

Assignment statements are used to set or change the values of sets, parameters and variables during the execution of a procedure or a function. The syntax of an assignment statement is straightforward.

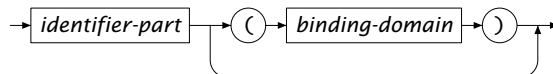
*Assignment*

*assignment-statement :*

*Syntax*



*data-selection :*



AIMMS offers several assignment operators. The standard *replacement* assignment operator `:=` replaces the value of all elements specified on the left hand side with the value of the expression on the right hand side. The *arithmetic* assignment operators `+=`, `-=`, `*=`, `/=` and `^=` combine an assignment with an arithmetic operation. Thus, the assignments

*Assignment operators*

$$a += b, \quad a -= b, \quad a *= b, \quad a /= b, \quad a ^= b$$

form a shorthand notation for the assignments

$$a := a + b, \quad a := a - b, \quad a := a * b, \quad a := a / b, \quad a := a ^ b.$$

Assignment is an *index binding* statement. AIMMS also binds unbound indices in (nested) references to element-valued parameters that are used for indexing the left-hand side. AIMMS will execute the assignment repeatedly for *all* elements in the binding domain, and in the order as specified by the declaration(s) of the binding set(s). The precise rules for index binding are explained in Section 9.1.

*Index binding*

In contrast to the binding domain of iterative operators and the FOR statements, the binding domain of an indexed assignment can contain the full range of element expressions:

*Allowed binding domains*

- references to unbound indices, which will be bound by the assignment,
- references to scalar element parameters and bound indices,
- references to indexed element parameters, for which any nested unbound index will be bound as well,
- calls to element-valued functions, and
- element-valued iterative operators.

If the element expression inside the binding domain of an indexed assignment is too lengthy, it may be better to use an intermediate element parameter to improve readability.

Like any binding domain, the binding domain of an indexed assignment can be subject to a logical condition. Such an assignment is referred to as a *conditional assignment*, and is only executed for those elements in the binding domain that satisfy the logical condition.

*Conditional assignments*

In addition, if the identifier on the left-hand side of the assignment has its own domain restriction, then the assignment is limited to those elements of the binding domain that satisfy this restriction. Assignments to elements outside the restricted domain are not considered.

*Domain checking*

The following five examples illustrate some simple assignment statements. In all examples we assume that  $i$  and  $j$  are unbound indices into a set `Cities`, and that `LargestCity` is an element parameter into `Cities`.

*Example*

1. The first example illustrates a simple *scalar assignment*.

```
TotalTransportCost := sum[(i,j), UnitTransportCost(i,j)*Transport(i,j)];
```

The value of the scalar identifier on the left-hand side is replaced with the value of the expression on the right-hand side.

2. The second example illustrates an *index binding assignment*.

```
UnitTransportCost(i,j) *= CostWeightFactor(i,j) ;
```

For *all* cities  $i$  and  $j$  in the index domain of `UnitTransportCost`, the old values of the identifier `UnitTransportCost(i,j)` are multiplied with the values of the identifier `CostWeightFactor(i,j)` and then used to replace the old values.

3. The third example illustrates a *conditional assignment*.

```
Transport((i,j) | UnitTransportCost(i,j) > 100) := 0;
```

The zero assignment to `Transport` is made to only those cities  $i$  and  $j$  for which the `UnitTransportCost` is too high.

4. The fourth example illustrates a *sliced assignment*, i.e. an assignment that only changes the values of a lower-dimensional subspace of the index domain of the left-hand side identifier.

```
Transport(LargestCity,j) := 0;
```

The sliced assignment in this example binds only the index  $j$ . The values of the parameter `Transport` are set to zero from the city `LargestCity` to *every* city  $j$ , but the values from every other city  $i$  to all cities  $j$  remain unchanged.

5. The fifth example illustrates a *nested index binding statement*.

```
PreviousCity( NextCity(i) ) := i;
```

The index  $i$  is bound, because it is used in the nested reference of the element parameter `NextCity(i)`, which in turn is used for indexing the identifier `PreviousCity`. Note that, in a tour, city  $i$  by definition is the previous city of the specific (next) city it is linked with.

Indexed assignments are executed in a sequential manner, i.e. as if it was replaced by a sequence of individual assignments to every element in the binding domain. Thus, if `Periods` is the integer set `{0 .. 3}` with index `t`, then the indexed assignment

*Sequential  
execution*

```
Stock( t | t > 0 ) := Stock(t-1) + Supply(t) - Demand(t);
```

is executed (conceptually) as the sequence of individual statements

```
Stock(1) := Stock(0) + Supply(1) - Demand(1);
Stock(2) := Stock(1) + Supply(2) - Demand(2);
Stock(3) := Stock(2) + Supply(3) - Demand(3);
```

Therefore, in the right hand side expression it is possible to refer to elements of the identifier on the left which have received their value prior to the execution of the current individual assignment. This type of behavior is typically observed and wanted in stock balance type applications which use lag references as shown above. The same argument also applies to assignments that use element parameters for indexing on either the left- or right-hand side of the assignment.

In addition to the indexed assignment, AIMMS also possesses a more general FOR statement which repeatedly executes a group of statements for all elements in its binding domain (see also Section 8.3.4). If you are familiar with programming languages like C or PASCAL you might be tempted to embed every indexed assignment into one or more FOR statements with the proper domain. Although this will conceptually produce the same results, we strongly recommend against it for two reasons.

*Indexed  
assignment  
versus FOR*

- By omitting the FOR statements you improve to the readability and maintainability of your model code.
- By including the FOR statement unnecessarily you are effectively degrading the performance of your model, because AIMMS can execute an indexed assignment much more efficiently than the equivalent FOR statement.

Whenever you use a FOR statement unnecessarily, AIMMS will produce a compile time warning to tell you that the code would be more efficient by removing the FOR statement.

Consider the indexed assignment

*Example*

```
Transport((i,j) | UnitTransportCost(i,j) > 100) := 0;
```

and the equivalent FOR statement

```

for ((i,j) | UnitTransportCost(i,j) > 100) do
  Transport(i,j) := 0;
endfor;

```

Notice that the indexed assignment is more compact than the FOR statement and is easier to read. In this example AIMMS will warn against this use of the FOR statement, because it can be removed without any change in semantics, and will lead to more efficient execution.

When there are undefined references with lag and lead operators on the left-hand side of an assignment (i.e. references that evaluate to the empty element), the corresponding assignments will be skipped. The same is true if the identifier on the left contains undefined references to element parameters. Notice that this behavior is different from the behavior of a reference containing undefined lag and lead expressions on the right-hand side of an assignment. These evaluate to zero.

*Undefined  
left-hand  
references*

Consider the assignment to the parameter Stock above. It could also have been written as

*Example*

```

Stock(t+1) := Stock(t) + Supply(t+1) - Demand(t+1);

```

In this case, there is no need to add a condition to the assignment for  $t = 3$ . The reference to  $t+1$  is undefined, and hence the assignment will be skipped. Similarly, the assignment

```

PreviousCity( NextCity(i) ) := i;

```

will only be executed for those cities  $i$  for which  $\text{NextCity}(i)$  is defined.

---

### 8.3 Flow control statements

Execution statements such as assignment statements, SOLVE statements or data management statements are normally executed in their order of appearance in the body of a procedure. However, the presence of control flow statements can redirect the flow of execution as the need arises. AIMMS provides six forms of flow control:

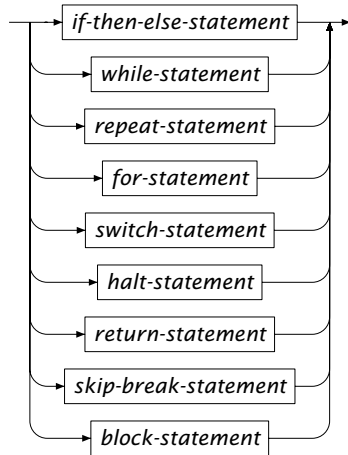
*Six forms of  
flow control*

- the IF-THEN-ELSE statement for conditional execution,
- the WHILE statement for repetitive conditional execution,
- the REPEAT statement for repetitive unconditional execution,
- the FOR statement for repetitive domain-driven execution,
- the SWITCH statement for branching on set and integer values,
- the HALT and RETURN statement for terminating the current execution,

- the SKIP and BREAK statements for terminating the current repetitive execution, and
- the BLOCK statement for visually grouping together multiple statements.

*flow-control-statement :*

*Syntax*



In the condition of flow control statements such as IF-THEN-ELSE, WHILE and REPEAT it is needed to know whether the result is equal to 0.0 or not in order to take the appropriate branch of execution. The special number NA has the interpretation “not yet available” thus it is also not yet known whether it is equal to 0.0 or not. The special number UNDF is the result of an illegal operation, so its value cannot be known. Therefore, AIMMS will issue an error message if the result of a condition in these statements evaluates to NA or UNDF. Special numbers and their interpretation as logical values are discussed in full detail in Sections 6.1.1 and 6.2.

*Flow control statements and special numbers*

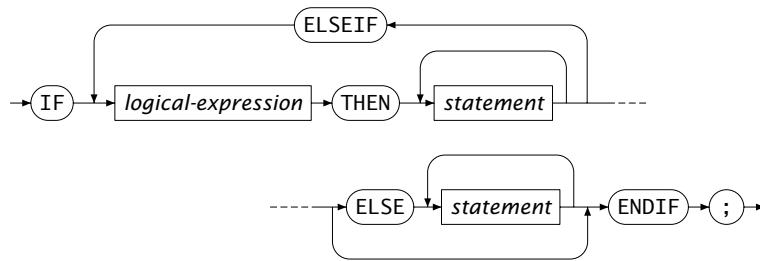
---

### 8.3.1 The IF-THEN-ELSE statement

The conditional IF-THEN-ELSE statement is used to choose between the execution of several groups of statements depending on the outcome of one or more logical conditions. The syntax of the IF-THEN-ELSE statement is given in the following diagram.

*Syntax*

*if-then-else-statement :*



AIMMS will evaluate all logical conditions in succession and stops at the first condition that is satisfied. The statements associated with that particular branch are executed. If none of the conditions is satisfied, the statements of the ELSE branch, if present, will be executed.

The following code illustrates the use of the IF-THEN-ELSE statement.

*Example*

```

if ( not SupplyDepot ) then
  DialogMessage( "Select a supply depot before solving the model" );
elseif ( Exists[ p, Supply(SupplyDepot,p) < Sum( i, Demand(i,p) ) ] ) then
  DialogMessage( "The selected supply depot has insufficient capacity" );
else
  solve TransportModel ;
endif ;

```

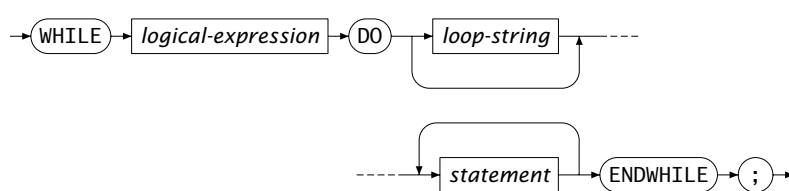
Note that in this particular example the evaluation of the ELSEIF condition only makes sense when a SupplyDepot exists. This is automatically enforced because the IF condition is not satisfied. Similarly, successful execution of the ELSE branch apparently depends on the failure of both the IF and ELSEIF conditions.

### 8.3.2 The WHILE and REPEAT statements

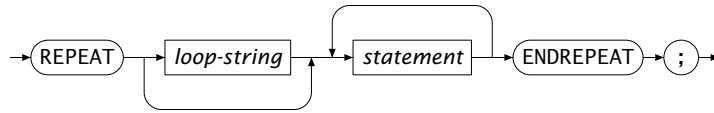
The WHILE and REPEAT statements group a series of execution statements and execute them repeatedly. The execution of the repetitive loop can be terminated by a logical condition that is part of the WHILE statement, or by means of a BREAK statement from within both the WHILE and REPEAT statements.

*while-statement :*

*Syntax*



*repeat-statement :*



Loop strings are discussed in Section 8.3.3.

The execution of a WHILE statement is subject to a logical condition that is verified each time the statements in the loop are executed. If the condition is false initially, the statements in the loop will never be executed. In case the WHILE loop does not contain a BREAK, HALT or RETURN statement, the statements inside the loop must in some way influence the outcome of the logical condition for the loop to terminate.

*Termination by  
WHILE condition*

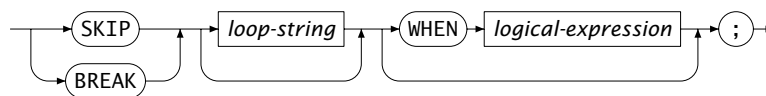
An alternative way to terminate a WHILE or REPEAT statement is the use of a BREAK statement inside the loop. BREAK statements make it possible to abort the execution at any position inside the loop. This freedom allows you to formulate more natural termination conditions than would otherwise be possible with just the logical condition in the WHILE statement. After aborting the loop, AIMMS will continue with the first statement following it.

*Termination by  
a BREAK  
statement*

In addition to the BREAK statement, AIMMS also offers a SKIP statement. With it you instruct AIMMS to skip the remaining statements inside the current iteration of the loop, and immediately return to the top of the WHILE or REPEAT statement to execute the next iteration. The SKIP statement is an elegant alternative to placing the statements inside the loop following the SKIP statement in a conditional IF statement.

*Skipping the  
remainder of a  
loop*

*skip-break-statement :*



*Syntax*

By adding a WHEN clause to either a BREAK or SKIP statement, you make its execution conditional to a logical expression. In practice, the execution of a BREAK or SKIP statement is almost always subject to some condition.

*The WHEN clause*

This example computes the *machine epsilon*, which is the smallest number that, when added to 1.0, gives a value different from 1.0. It is a measure of the accuracy of the floating point arithmetic, and it is machine dependent.

*Example WHILE  
statement*

We assume that `meps` is a scalar parameter, and that the numeric comparison tolerances are set to zero (see also Section 6.2.2).

```
meps := 1.0;
while (1.0 + meps/2 > 1.0) do
  meps /= 2;
endwhile;
```

Since the parameter `meps` is determined iteratively, and the loop condition will eventually be satisfied, this example illustrates an appropriate use of the `WHILE` loop.

By applying a `BREAK` statement, the machine epsilon can be computed equivalently using the following `REPEAT` statement.

*Example REPEAT statement*

```
meps := 1.0;
repeat
  break when (1.0 + meps/2 = 1.0) ;
  meps /= 2;
endrepeat;
```

The `BREAK` statement could also have been formulated in an equivalent but less elegant manner without a `WHEN` clause:

```
if (1.0 + meps/2 = 1.0) then
  break;
endif;
```

---

### 8.3.3 Advanced use of `WHILE` and `REPEAT`

Next to the common use of the `WHILE` and `REPEAT` statements described in the previous section, AIMMS offers some special constructs that help you

*Advanced uses*

- keep track of the number executed iterations automatically, and
- control nested arrangements of `WHILE` and `REPEAT` statements.

There are practical examples in which the terminating condition of a repetitive statement may not be met at all or at least not within a reasonable amount of work or time. A good example of this behavior are solution algorithms for which convergence is likely but not guaranteed. In these cases, it is common practice to terminate the execution of the loop when the total number of iterations exceeds a certain limit.

*Nonconvergent loops*

In AIMMS, such conditions can be formulated easily without the need to

- introduce an additional parameter,
- add a statement to initialize it, and
- increase the parameter every iteration of the loop.

Each repetitive statement keeps track of its iteration count automatically and makes the number of times the loop is entered available by means of the pre-defined operator `LoopCount`. Upon entering a repetitive statement AIMMS will set its value to 1, and will increase it by 1 at the end of every iteration.

*The LoopCount operator*

Whether the following sequence will converge depends on the initial value of `x`. In the case where there is no convergence or if convergence is too slow, the loop in the following example will terminate after 100 iterations.

*Example*

```
while ( Abs(x-OldValue) >= Tolerance and LoopCount <= 100 ) do
  OldValue := x ;
  x       := x^2 - x ;
endwhile ;
```

So far, we have considered single loops. However, in practice it is quite common that repetitive statements appear in nested arrangements. To provide finer control over the flow of execution in such situations, AIMMS allows you to label a particular repetitive statement with a *loop string*.

*Naming nested loops*

Using a loop string in conjunction with the `BREAK` and `SKIP` statements, it is possible to break out from several nested repetitive statements with a single `BREAK` statement. The loop string argument can also be supplied to the `LoopCount` operator so the break can be conditional on the number of iterations of any loop. Without specifying a loop string, `BREAK`, `SKIP` and `LoopCount` refer to the current loop by default.

*Use of loop strings*

The following example illustrates the use of loop strings and the `LoopCount` operator in nested repetitive statements. It outlines an algorithm in which the domain of definition of a particular problem is extended in every loop based on the current solution, after which the new problem is solved by means of a sequential solution process.

*Example*

```
repeat "OuterLoop"
  ... ! Determine initial settings for sequential solution process

  while( Abs( Solution - OldSolution ) <= Tolerance ) do
    OldSolution := Solution ;

    ... ! Set up and solve next sequential step ...

    ! ... but terminate algorithm when convergence is too slow
    break "OuterLoop" when LoopCount >= LoopCount("OuterLoop")^2 ;
```

```

endwhile;

... ! Extend the domain of definition based on current solution,
    ! or break from the loop when no extension is possible anymore.
endrepeat;

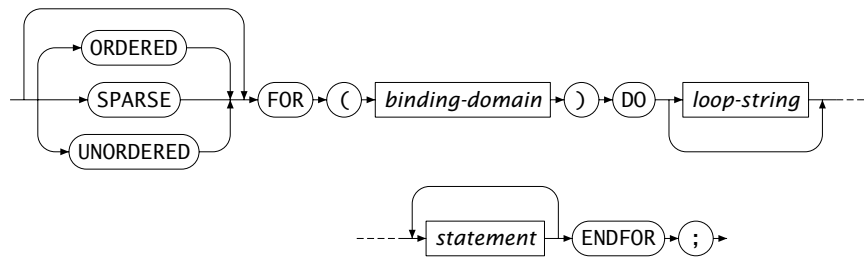
```

### 8.3.4 The FOR statement

The FOR statement is related to the use of iterative operators in expressions. An iterative operator such as SUM or MIN applies a particular operation to all expressions defined over a particular domain. Similarly, the FOR statement executes a group of execution statements for all elements in its domain. The syntax of the FOR statement is given in the following diagram.

*for-statement :*

*Syntax*



The binding domain of a FOR statement can only contain free indices, which are then bound by the statement. All statements inside a FOR statement are executed in sequence for the specific elements in the binding domain. Unless specified otherwise, the ordering of elements in the binding domain, and hence the execution order of the FOR statement, is the same as the order of the corresponding binding set(s).

*Execution is sequential*

FOR statements with an integer domain in the form of an enumerated set behave in a similar manner as the FOR statement in programming languages like C or Pascal. Like the example below, FOR statements of this type are mostly of an algorithmic nature, and the indices bound by the FOR statement typically serve as an iteration count.

*Integer domains*

```

for ( n in { 1 .. MaxPriority } ) do

    x.NonVar( i | x.Priority(i) < n ) := 1;
    x.Relax ( i | x.Priority(i) = n ) := 0;
    x.Relax ( i | x.Priority(i) > n ) := 1;

    Solve IntegerModel;
endfor;

```

*Example*

This example tries to solve a mixed-integer mathematical program heuristically in stages. The algorithm first only solves for those integer variables that have a particular integer priority, and then changes them to non-variables before going on to the next priority. The suffices used in this example are discussed in Section 14.1.

FOR statements with non-integer binding domains are typically used to process the data of a model for all elements in a data-related domain. The use of a FOR statement in such a situation is only necessary if the statements inside it form a unit, for which sequential execution for each element in the domain of the entire group of statements is essential. An example follows.

*Non-integer domains*

```
for ( i in Cities ) do
  SmallestTransportCity := ArgMin( j, Transport(i,j) ) ;
  DiscardedTransports += Transport( i, SmallestTransportCity ) ;
  Transport( i, SmallestTransportCity ) := 0 ;
endfor;
```

*Example*

In this example the three assignments form an inseparable unit. For each particular value of *i*, the second and third assignment depend on the correct value of *SmallestTransport* in the first assignment.

If you are familiar with programming language like PASCAL and C, then the use of FOR statements will seem quite natural. In AIMMS, however, FOR statements are often not needed, especially in the context of indexed assignments. Indexed assignments bind the free indices in their domain implicitly, resulting in sequential execution of that particular assignment for all elements in its domain. In general, such an index binding assignment is executed much more efficiently than the same assignment placed inside an equivalent FOR statement. In general, you should use FOR statements only when really necessary.

*Use FOR only when needed*

AIMMS will provide a warning when it detects unnecessary FOR statements in your model. Typically FOR statement are not required when the loop only contains assignments that do not refer to scalar identifiers (either numeric or element-valued) to which assignments have been made inside the loop as well. For instance, in the last example the FOR statement is essential, because the assignment and use of the element parameter *LargestTransportCity* is inside the loop.

*AIMMS issues a warning*

The following example shows an unnecessary use of the FOR statement.

*Example*

```
solve OptimizationModel;

! Mark variables with large marginal values
for (i) do
  if ( Abs[x.Marginal(i)] > HighPrice ) then
```

```

    Mark(i) := x.Marginal(i);
  else
    Mark(i) := 0.0;
  endif;
endfor;

```

While this statement may seem very natural to C or Pascal programmers, in a sparse execution language like AIMMS it should preferably be written by the following simpler, and faster, assignment statement.

```
Mark(i) := x.Marginal(i) OnlyIf ( Abs[x.Marginal(i)] > HighPrice );
```

With the optional keywords SPARSE, ORDERED and UNORDERED you can indicate that AIMMS should follow one of three possible strategies to execute the FOR statement. If you do not explicitly specify a strategy, AIMMS will follow the SPARSE strategy by default, and issue a warning when this strategy leads to severe inefficiencies. You can find an explanation of each of the strategies, as well as a description of the cases in which you may want to choose a specific strategy in Section 13.2.2.

*The SPARSE, ORDERED and UNORDERED keywords*

Like the WHILE and the REPEAT statements, FOR is a repetitive statement. Thus, you can use the SKIP and BREAK statements and the LoopCount operator. In addition, you can identify a FOR statement with a loop string thereby controlling execution in nested arrangements as discussed in the previous section.

*FOR as a repetitive statement*

The SKIP statement skips the remaining statements in the FOR loop and continues to execute the loop for the next element in the binding domain. The BREAK statement will abort the execution of the FOR statement all together.

*Use of SKIP and BREAK*

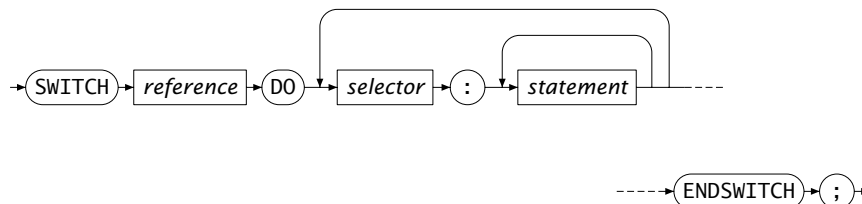
### 8.3.5 The SWITCH statement

The SWITCH statement is used to choose between the execution of different groups of statements depending on the value of a scalar parameter reference. The syntax of the SWITCH statement is given in the following two diagrams.

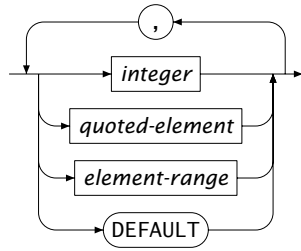
*The SWITCH statement*

*switch-statement :*

*Syntax*



*selector :*



The SWITCH statement can switch on two types of scalar parameter references: set element-valued or integer-valued. When you try to switch on references to string-valued or non-integer numerical parameters, AIMMS will issue a compile time error

*Integers and set element*

Each selector in a SWITCH statement must be a comma-separated list of values or value ranges, matching the type of the selecting scalar parameter. Expressions and ranges used in a SWITCH statement must only contain constant integers and set elements. Set elements used in a switch selector must be known at compile time, i.e. the data initialization of the corresponding set must be a part of the model description.

*Switch selectors*

The optional DEFAULT selector matches every reference. Since AIMMS executes only those statements associated with the *first* selector matching the value of the scalar reference, it is clear that the DEFAULT selector should be placed last.

*The DEFAULT selector last*

The following SWITCH statement takes different actions based on the model status returned by a SOLVE statement.

*Example*

```

solve OptimizationModel;

switch OptimizationModel.ProgramStatus do
  'Optimal', 'LocallyOptimal' :
    ObservedModelStatus := 'Solved' ;

  'Unbounded', 'Infeasible', 'IntegerInfeasible', 'LocallyInfeasible' :
    ObservedModelStatus := 'Infeasible' ;

  'IntermediateInfeasible', 'IntermediateNonInteger', 'IntermediateNonOptimal' :
    ObservedModelStatus := 'Interrupted' ;

  default :
    ObservedModelStatus := 'Not solved' ;
endswitch ;

```

### 8.3.6 The HALT statement

With a HALT statement you can stop the current execution. You can use it, for example, if your model has run into an unrecoverable error condition during its execution, or if you simply want to skip the remaining statements because they are no longer relevant in a particular situation.

*Terminating execution*

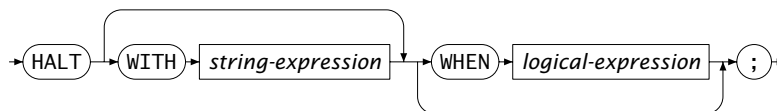
Instead of the HALT statement you can also use the RETURN statement (see also Section 10.1) to terminate the current execution. The HALT statement directly jumps back to the user interface, but a RETURN statement in a procedure only passes back control to the calling procedure and continues execution from there.

*Compare to RETURN*

The syntax of the HALT statement follows.

*Syntax*

*halt-statement :*



You can optionally specify a string in the HALT statement that will be printed in a message dialog box when execution has stopped. This is useful, for instance, to pass on an appropriate message to the user when a particular error condition has occurred.

*Printing a message*

You can make the execution of the HALT statement conditional on a WHEN clause. If present, the current run will only be aborted if the condition after the WHEN clause is satisfied.

*The WHEN clause*

The following example terminates the current run if the SOLVE statement does not solve to optimality. When aborting, the user will be notified with an explanatory message.

*Example*

```
solve LinearOptimizationModel;

halt with "Execution aborted: model not solved to optimality"
      when OptimizationModel.ProgramStatus <> 'Optimal' ;
```

Note that the type of model termination initiated by calling the HALT statement cannot be guarded against using AIMMS' error handling facilities (see Section 8.4). An alternative to the HALT statement, which enables error handling, is the RAISE statement discussed in Section 8.4.2. When you want to let the HALT

*Alternative*

act as a RAISE statement, you can switch the option `halt_acts_as_raise_error` on.

### 8.3.7 The BLOCK statement

A sequence of statements are grouped together into a single statement with the BLOCK statement possibly serving one or more of the following purposes:

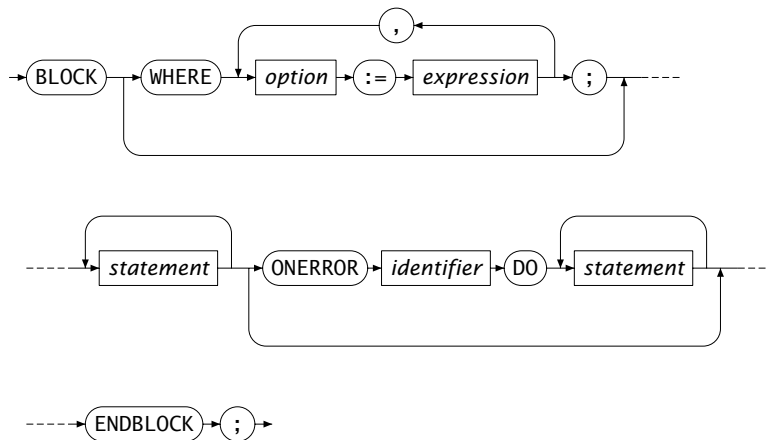
*The BLOCK statement*

- to emphasize the logical structure in the model,
- to execute a group of statements with different option settings, or
- to use dedicated error handling on a group of statements, see Section 8.4.

The syntax of the BLOCK statement is as follows.

*block-statement :*

*Syntax*



Consider the following BLOCK statement containing a group of statements.

*Emphasizing logical structure in the model*

```

block ! Initialize measured compositions as observable.
  CompositionObservable(nmf,c in MeasuredComponents(nmf)) := 1;
  CompositionObservable(mf,mc) := 0;

  if ( not CheckComputableFlows ) then
    UnobservableComposition(nmf,c) := 1$(not CompositionObservable(nmf,c));
    return 0;
  endif;

  CompositionCount(pu,c) :=
    Count((f,g) | Admissable(pu,c,f,g) and CompositionObservable(g,c));
  NewCount := Card ( CompositionObservable );
endblock ;
  
```

In the AIMMS syntax editor, the block can be displayed in either a collapsed or expanded state. When collapsed, the block will be displayed as follows, using the single line comment following the BLOCK keyword as its description.

```
Initialize measured compositions as observable. ;
```

When in a collapsed state, AIMMS will show the contents of the block in a tooltip when the mouse pointer hovers over the collapsed block, as illustrated in the figure below.

```
Initialize measured compositions as observable. ;
Block ! Initialize measured compositions as observable.
CompositionObservable(nmf,c in MeasuredComponents(nmf)) := 1;
CompositionObservable(nmf,mc) := 0;

if ( not CheckComputableFlows ) then
  UnobservableComposition(nmf,c) := 1 $ (not CompositionObservable(nmf,c));
  return 0;
endif;
...

```

During the execution of a block statement, the options in the WHERE clause will have the specified values by setting them at the beginning of the block statement and restoring the old values at the end of the block statement. More on the format of option names and value settings can be found in Section 8.5. The example below prints various parameters using various settings of the option Listing\_number\_precision.

*Executing with different option settings*

```
! The default value of the option Listing_number_precision is 3.
block ! Start printing numbers using 6 decimals.
  where Listing_number_precision := 6 ;

  display A, B ;

  block ! Start printing numbers without decimals.
    where Listing_number_precision := 0 ;
    display C, D ; ! The output looks as if C and D are integers.
  endblock ;

  display E, F ; ! Back to printing numbers using 6 decimals.

endblock ;

display G, H ; ! Back to printing numbers using 3 decimals.

```

In the above example a nested block statement is used to scope option settings; the inner block statement temporarily overrides the option setting of the outer block statement, which overrides the global option settings.

The OnError clause provides one of the means for handling runtime errors in AIMMS. It is discussed in full detail in Section 8.4.1.

*The OnError clause*

---

## 8.4 Raising and handling warnings and errors

During the development and deployment of an AIMMS application, unexpected, possibly harmful, situations can arise. These situations are divided into errors and warnings. An error is a situation that cannot be handled by the procedure encountering it. A warning is a situation that can be handled by the procedure encountering it, but might warrant further inspection by the model developer or model user anyway. Note that, even when a procedure cannot handle an error itself, it should be able to recover from that error. In this section, you will find AIMMS facilities

*Errors and warnings*

- to handle errors; to handle an error, AIMMS will give you access to the information therein. A handler is a piece of AIMMS code that handles selected errors and warnings. Errors and warnings can be communicated to handlers higher in the execution stack.
- to raise an error; not only AIMMS may detect situations warranting an error or warning message, but also the application itself. For such situations AIMMS provides a facility to raise custom errors from within your model.
- to handle a legacy situation; external and intrinsic AIMMS procedures may return a status code indicating success or failure. Whenever a failure status of an external and intrinsic procedure remains unnoticed, AIMMS can automatically raise an error in such situations.
- for extensive code checking; AIMMS can check your application for many different kinds of situations warranting a warning. It usually pays off to apply all these checks occasionally to your application.

---

### 8.4.1 Handling errors

In this subsection you will find an introduction to both the global and local error handling mechanism available in AIMMS. Global error handling, by means of specifying a single handler procedure, is used to handle runtime errors occurring inside the entire model that are not handled elsewhere. Local error handling, by means of the `OnError` clause in a `BLOCK` statement, allows for error handling of the runtime errors occurring in a specific block of code. Global and local error handling are the building blocks on which the error handling framework in AIMMS is built. At the end of this subsection, you will find a description of all intrinsic functions available for accessing and manipulating information regarding errors.

*Subsection overview*

To activate global error handling, the name of a handling procedure in your model must be assigned to the option `Global_error_handler`. Such a procedure must have a single element parameter argument `err` into the predeclared set `errh::PendingErrors`. The global error handling procedure will be executed for each pending error whenever an execution run has been terminated because of errors that have not been handled elsewhere in the model. The global error handler will also be called at the end of a finished execution run with unhandled warnings. In this context, an execution run is any call to an AIMMS procedure initiated either through the AIMMS GUI or through the AIMMS API.

*Global error  
handling*

Below a global error handling procedure `MyErrorHandler` is illustrated. The lines in the body of the procedure are numbered to facilitate the explanation of the example after the procedure listing.

*Example*

```
PROCEDURE:
  identifier : MyErrorHandler
  arguments : err
  DECLARATION SECTION
    ELEMENT PARAMETER:
      identifier : err
      range      : errh::PendingErrors
      property   : Input ;
  ENDSECTION ;
  body        :
    1 if errh::Node(err) = 'DefP' then
    2   DialogMessage(errh::Message(err) + "; resetting P to its default.");
    3   Empty P ;
    4   errh::MarkAsHandled(err);
    5 elseif errh::InsideCategory(err,'IO') then
    6   errh::Adapt(err,message:"IO error: please consult ...; "
    7     + errh::Message(err) ); ! Pass adapted message on to next handler.
    8 else
    9   ! Errors not handled will be passed on to the error/warning window.
    10 endif ;
  ENDPROCEDURE ;
```

The procedure starts with declaring the argument `err` as an element parameter with the predeclared set `errh::PendingErrors`, which is a subset of the predeclared set `Integers`, as its range. During an execution run this set is filled with the numbers of the errors and warnings raised. Each number refers to an error or warning with various pieces of information contained therein, such as its error description, the node in which the error or warning occurred and its severity. In addition, each error belongs to a category. All this information can be accessed via intrinsic functions. The body of the procedure is explained line by line:

*Example  
explanation*

- **line 1:** The intrinsic function `errh::Node` is used to determine whether or not the error occurred inside the procedure `DefP`. This intrinsic function returns the identifier or node in which the error occurred as an element in the predeclared set `AllSymbols`.

- **lines 2, 3:** If the error did happen inside the procedure DefP, the application user is notified and P is reset to its default. The notification uses the original error description obtained by using the intrinsic function `errh::Message(err)`.
- **line 4:** Each handled error should be marked as such. When an error handler finishes, it will delete the errors that have been marked as handled from the predeclared set `errh::PendingErrors`.
- **line 5:** To discern the type of an error, errors are divided into categories. For each error, the category to which it belongs can be obtained using the function `errh::Category(err)`. The error categories form a nested structure. For instance, both `FileIO` and `DatabaseIO` errors are IO errors. The intrinsic function `errh::InsideCategory(err)` can be used to determine whether or not an error is inside a particular category.
- **lines 6, 7:** Translate the error by adapting information. In this example, only the message is actually adapted, but most parts of an error can be adapted. Note that in this `else` branch, the function `errh::MarkAsHandled` is not called, the result of which is that the adapted error message will appear in the messages/errors window.
- **line 8:** In this branch, the error is not handled. An error not handled when the error handler finishes will not be deleted. Instead, it ends up being displayed in the messages/errors window.

The following template of a BLOCK statement illustrates local error handling by means of the `OnError` clause.

```

1  BLOCK
2    statement_1 ;
3    ...
4    statement_n ;
5  ONERROR err DO
6    ...
7    ...
8  ENDBLOCK ;
```

All errors occurring inside `statement_1 ... statement_n` on lines 2 ... 4 are handled by the error handler on lines 6 and 7, where `err` is an element parameter into the set `errh::PendingErrors`. Block statements can be nested, either directly in a single body or in other procedures called from within block statements. This gives rise to a stack of error handlers as illustrated below. A detailed example of a local error handler is given in Section 33.6.

The global error handlers and the `OnError` error handlers are essential building blocks of the error handling framework of AIMMS. This error handling framework is illustrated in Figure 8.1.

*Local error handling by means of the OnError clause*

*Error flow architecture*

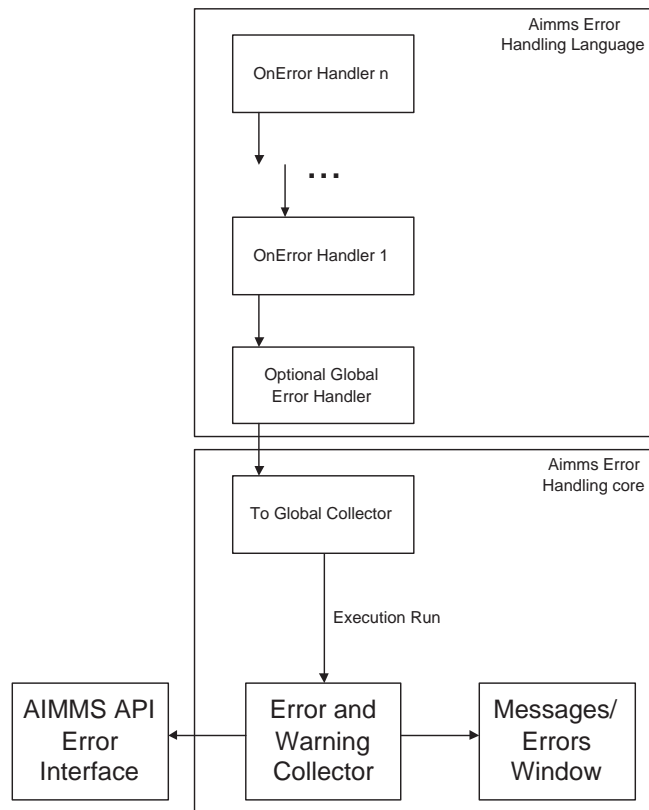


Figure 8.1: Error flow through handlers

At the start of each execution run, a new stack of error handlers is created. At the bottom of this stack is the standard handler To Global Collector. When the option `Global_error_handler` is set, the specified procedure is placed on top of this new stack. Additional handlers are placed on the stack by each `OnError` clause in a (nested) `BLOCK` statement.

*Construction of the error handler stack*

When raised, each error is set aside for being handled by the topmost error handler. When the number of errors set aside reaches the limit specified by the option `Errors_until_execution_interrupt`, the execution is interrupted and resumes by executing the code in the topmost error handler. When the execution is not interrupted, but there are pending errors or warnings, the error handling code is executed as well after the completion of the last statement prior to the `BLOCK` statement.

*Errors flowing through a handler stack*

For each error, the error handling code should decide whether to handle that error itself, let another handler handle the error, or ignore the error (as was already illustrated in the example above).

*What to do with an error*

Errors may also occur during the execution of the `OnError` clause of a `BLOCK` statement or the global error handling procedure. These errors are handled by the next error handler in the stack of error handlers.

*Handling an error inside a handler*

When an error reaches the handler `To Global Collector`, it is sent to the **Error and Warning Collector**; this object collects all errors that have fallen through the various handlers (if any). Errors in the **Error and Warning Collector** can be queried from within the AIMMS API or viewed from within the messages/errors window of the AIMMS GUI.

*Error collector*

Errors to be handled can be queried using the following predeclared identifiers and intrinsic functions from the module `ErrorHandling` with prefix `errh`:

*The predeclared module ErrorHandling*

- **`errh::PendingErrors`**: A predeclared set filled with the numbers of the errors that can be handled at this point.
- **`errh::IndexPendingErrors`**: An index into the above predeclared set.
- **`error parts`**: An error is made up of several parts; each of which can be obtained separately using the intrinsic functions below. Each of the functions below will raise an error of their own if `err` is not a valid error that can be handled at that point.
  - **`errh::Severity(err)`**: An element in `errh::AllErrorSeverities` is returned indicating the severity of the error.
  - **`errh::Message(err)`**: A string containing the error description is returned. This string is not empty.
  - **`errh::Category(err)`**: An element in `errh::AllErrorCategories` is returned indicating the category of the error.
  - **`errh::Code(err)`**: The element in `errh::ErrorCodes` that is returned by this function identifies the message code of the error. This element name may be cryptic.
  - **`errh::NumberOfLocations(err)`**: The number of locations relevant to this error. For compilation errors, there is typically only one location relevant. For an AIMMS initialization error there are no locations relevant. For an execution error the positions in all active procedures are recorded. For an error during file read, at least the positions in the data file and the read statement are recorded. Similarly, for an error during the generation of a constraint, at least the constraint and the `SOLVE` statement are recorded as relevant positions.
  - **`errh::Node(err, loc)`**: An element in `AllSymbols` is returned for an error location inside the model. The optional argument `loc` defaults to 1 and should be in the range `{ 1 .. NumberOfLocations }`. The element returned by this function is non-empty except for the first location when reading data from a file.
  - **`errh::Attribute(err, loc)`**: An element in `AllAttributeNames`.
  - **`errh::Line(err, loc)`**: An integer indicating the line number of the error in the attribute or file, or 0 if not known.

- **errh::Column(err)**: An integer indicating the column position in an erroneous line being read from a data file. All errors when reading a data file are reported separately, therefore the `loc` argument is not applicable.
- **errh::Filename(err)**: A non-empty string is returned when reading from a data file. All errors when reading a data file are reported separately, therefore the `loc` argument is not applicable.
- **errh::Multiplicity(err)**: An integer indicating the number of occurrences of this error. Two errors are considered equal if they are equal in all of the following parts: Severity, Message, Category, Code and the first location (if available). The first location is the location in the file being read when the error occurs during a read statement, otherwise it is the statement being executed.
- **errh::CreationTime(err,fmt)**: A string representing the creation time of the first occurrence of the error, formatting according to time format `fmt`.
- **errh::InsideCategory(err,cat)**: Returns 1 if the error code of `err` falls inside the category `cat`.
- **errh::IsMarkedAsHandled(err)**: Returns 1 if the error is marked as handled.
- **errh::Adapt(err,severity,message,category,code)**: Adapt an existing error. Several parts of an error can be adapted. Besides the mandatory argument `err` there should be at least one other argument.
- **errh::MarkAsHandled(err,actually)**: The error `err` is marked as handled if the argument `actually` is non-zero. Marked errors will not be passed to the next error handler. The default of the optional argument `actually` is 1. Using 0 will remove the mark from the error.

AIMMS logs all errors and warnings to the file `aimms.err` when they are raised. The folder in which this file resides is controlled by the option `Listing_and_temporary_files`. The number of backups retained of this file is controlled by the option `Number_of_log_file_backups`.

*The log file  
aimms.err*

---

### 8.4.2 Raising errors and warnings

The RAISE statement is used to

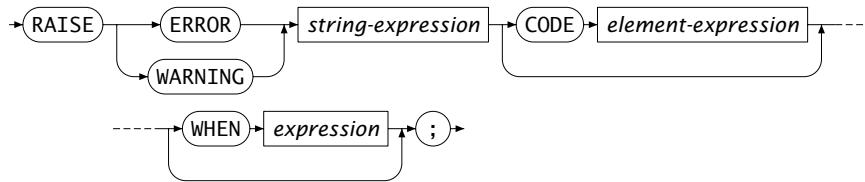
*Raising errors*

- raise an error regarding a situation that cannot be handled, or to
- raise a warning regarding a situation that can be handled but might warrant further investigation.

The syntax of the RAISE statement is straightforward.

*raise-statement* :

*Syntax*



In the following example an error is raised when the inflow of a node exceeds its capacity.

*Example*

```
if inflow > stockCap then
  RAISE ERROR "Inflow exceeds stock capacity" CODE 'TooMuchInflow' ;
endif ;
```

In order to enable an error handler to recognize the type of error raised by a RAISE statement, that statement allows for an optional error code to be specified. This is an element in the set `errh::ErrorCodes`. If the specified element does not yet exist, it is created and added to that set. The category of an error raised by the RAISE statement is fixed to 'User'.

*Error code and category*

AIMMS uses the line/procedure on which the RAISE statement is specified as the position information associated with the error. This permits the messages/errors window to open the attribute window of the procedure and put the cursor on the statement where the problematic situation is detected.

*Position information*

Not only AIMMS itself, but also procedures written in AIMMS may recognize situations that can be handled but might warrant closer inspection by the application user. For this purpose, the RAISE statement can raise a warning, for example:

*Raising warnings*

```
if card( RawMaterialTraders ) = 0 then
  RAISE WARNING "There are no raw material traders, this may lead to " +
    "infeasibilities in the case of too many accepted deliveries." ;
endif ;
```

The handling of warnings raised by a RAISE statement is controlled by the option `Warning_user` with default `common_warning_default`. The controlling of warning handling is further explained in Subsection [8.4.4](#).

### 8.4.3 Legacy: intrinsics with a return status

AIMMS external procedures and intrinsic procedures can return a status code indicating whether or not they were successful. A return value  $\leq 0.0$  is interpreted as not successful, whereas a return value  $> 0.0$  is successful. In addition, when they are not successful, the error message is often left in `CurrentErrorMessage`. This, however, is only a guideline. The return value of a call to an intrinsic procedure is either

*Legacy situation*

- **checked:** As illustrated in the example:

```
retval := PageOpen(...);
if retval <= 0 then
  ... use CurrentErrorMessage ...
endif;
```

- **not checked:** As illustrated in the example:

```
PageOpen(...);
```

In the context of the error handling facility available in AIMMS, how should the “checked” and “not checked” procedure calls be handled in case the return value is 0 and these procedures have not raised an error themselves? There are five error handling methods available to choose from:

*Available error handling methods*

- **ignore:** An error is never raised for an error occurring inside such a procedure whether or not the return status is checked.
- **raise\_warning\_when\_not\_checked:** A warning is only raised if the return status of an intrinsic procedure is not checked.
- **raise\_when\_not\_checked:** An error is only raised if the return status of an intrinsic procedure is not checked.
- **raise\_always\_warning:** A warning is raised whether the return status is checked or not.
- **raise\_always:** An error is raised whether the return status is checked or not.

Which choice of error handling method is best depends on the application and can be controlled by the options:

- **Intrinsic\_procedure\_error\_handling:** for the procedures with a return status supplied by AIMMS and
- **External\_procedure\_error\_handling:** for externally supplied procedures.

The values of these options are the names of the error handling methods described above. The default of both these options is `raise_when_not_checked`. For

projects created prior to the introduction of the error handling facilities in AIMMS (i.e. created in AIMMS 3.9 and lower), these options get the non-default value `raise_warning_when_not_checked` in order to notify the model developer but not change the existing behavior of these projects significantly.

---

#### 8.4.4 Warnings

AIMMS recognizes and warns against several types of possibly problematic situations. These situations might warrant further investigation. Like most other languages, AIMMS warns against the use of identifiers before initializing them. But unlike other languages, AIMMS also warns against the inconsistent use of units of measurement (such as a comparison of a volume against a weight), or against model formulations for which AIMMS can detect either compile- or runtime that they lead to sub-optimal performance or ambiguous results. A selection of performance-related warnings is discussed in Section [13.2.8](#).

*Warnings*

The desired handling of each of these situations depends on the developer and application; varying from treating it as an error to completely ignoring them. To permit complete flexibility, there is separate option controlling the reporting of each type of problematic situation recognized.

*Complete flexibility*

Although all warnings can be controlled individually, this is not the most convenient way to employ the diagnostics provided by these warnings. When entertaining a new idea (quick prototyping), most modelers understandably do not want to be bothered by various warnings and want to be able to turn them all off. To facilitate this, all warnings have been grouped into common and strict warnings, and the associated options assume the default value for common and strict warnings. Thus, all diagnostic warnings can be simply switched off by just changing the options that control these defaults. For normal development work it is advised to turn at least the common warnings on. In addition, it is encouraged to turn the strict warnings on during application tests.

*Grouping Warning options*

In order to implement the above scheme and still permit full flexibility, each option controlling the detection of a type of problematic situation can take on one of the following values:

*Choosing the option setting*

- **error:** The situation is marked as an error and treated as an error.
- **warning\_handle:** The warning is raised at the current error handler, but does not count in the interruption of normal execution.
- **common\_warning\_default:** The value of the option `Common_warning_default` is used.
- **warning\_collect:** The warning is raised at the `Global_error_collector`, bypassing the stack of error handlers.

- **strict\_warning\_default:** The value of the option `Strict_warning_default` is used.
- **off:** The warning is ignored.

The default of these options is either `common_warning_default` or `strict_warning_default`, thereby effectively dividing these options into a common and a strict group. The range of options `common_warning_default` and `strict_warning_default` is `{off, warning_collect, warning_handle, error}`. The default of the option `common_warning_default` is `warning_handle` and the default of the option `strict_warning_default` is `off`.

---

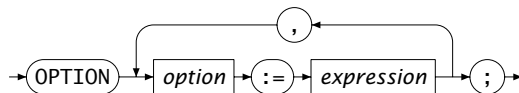
## 8.5 The OPTION and PROPERTY statements

Options are directives to AIMMS or to the solvers to execute a task in a particular manner. Options have a name and can assume a value that is either numeric or string-valued. You can modify the value of an option from within the graphical interface. The assigned value is stored along with the project. All global options are set to their stored values at the beginning of each session. During execution you can change option settings using the `OPTION` statement.

*Options*

*option-statement :*

*Syntax*



You can find a complete list of global options for AIMMS and its solvers in the help system.

The right-hand side of an `OPTION` statement must be a scalar expression of the proper type. If the option expects a string value, AIMMS will accept both string- or element-valued expressions. An example follows.

*Option values*

```
option Bound_Tolerance := 1.0e-6,
       Iteration_Limit := UserSettings('IterationLimit');
```

Some solver options are available for more than one solver. If you modify such a solver option per se, AIMMS will modify the option for all solver that support it. If you want to restrict the change to only a single solver, you can prefix the option name by the name of the solver followed by a dot ".", as illustrated in the example below.

*Solver options*

```
option 'Cplex 10.1'.lp_method := 'dual simplex';
```

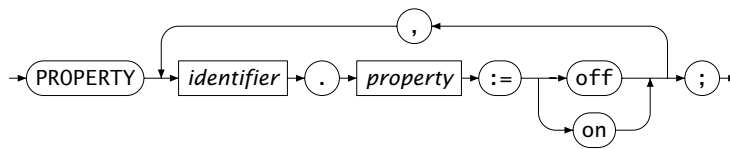
This statement will set the option `lp_method` of the solver that is known to the system as 'Cplex 10.1' equal to 'dual simplex'. The solver name can be either a quoted solver name, or an element parameter into the predefined set `AllSolvers`.

Identifier properties can be turned on or off. All properties default to off, unless they are turned on—either in the declaration of the identifier or in a `PROPERTY` statement. During the execution of your model you can dynamically change the default values of properties through the `PROPERTY` execution statements.

*Identifier properties*

*property-statement :*

*Syntax*



The properties of all identifier types can be found in the identifier declaration sections. Not all property settings can be changed, e.g. you cannot dynamically change the Input or Output property of arguments of functions and procedures. In such cases, AIMMS will produce a runtime error. An example of the `PROPERTY` statement follows.

*Resetting properties*

```
if ( Card(Cities) > 100 ) then
  property IntermediateTransport.NoSave := on;
endif;
```

Once the set of `Cities` contains more than 100 elements, the identifier `IntermediateTransport` is no longer saved as part of a case file.

When the `PROPERTY` statement is applied to an index into a subset of the predefined set `AllIdentifiers`, AIMMS will change the corresponding property for all identifiers in that subset.

*Multiple identifiers*

The following example illustrates how the `PROPERTY` statement can be used to obtain additional sensitivity data for a set `SensitivityVariables` of (symbolic) variables that has been previously determined.

*Example*

```
for ( var in SensitivityVariables ) do
  property var.CoefficientRanges := on;
endfor;
```

Here, you request AIMMS to determine the smallest and largest values for the objective coefficient of each variable in `SensitivityVariables` during the execution of a `SOLVE` statement such that the optimal basis remains constant (see also Section 14.1.2).