October 9, 1994

The ASCEND language syntax and semantics

Arthur W. Westerberg
Kirk Abbott
Benjamin Allan
Peter C. Piela[1]
Department of Chemical Engineering
and the Engineering Design Research Center
Carnegie Mellon University

We shall present an informal bottom-up description of the ASCEND language. Being informal, we shall often include descriptions of the intended semantics along with the syntax of the items. At times the inclusion of semantics will seem to anticipate later definitions. We do this because we would also like this section to be used as a reference for the ASCEND language after one generally understands it. Often one will need to clarify a point about a particular item and will not wish to have to search in several places to do so.

Syntax is the form or structure for the statements in ASCEND, where one worries about the exact words one uses, their ordering, the punctuation, etc. *Semantics* describe the meaning of a statement.

To distinguish between syntax and semantics, consider the statement

```
y IS_A fraction;
```

Rules on the syntax for this statement tell us we need a user supplied instance name (here, y) followed by the ASCEND operator IS_A followed by a model name (fraction). The statement terminates with a semicolon. The statement semantics says we are declaring the existence of an instance called y of the type fraction as a part within the current model definition.

[1] Currently at AspenTech, Cambridge, MA.

The building blocks

One often describes the syntax for a computer language by using a Bachus-Naur formal description for it. The following is an informal description motivated by such a formal description.

Letters	a through z, A through Z (the underscore is used as a letter, but it cannot be the first letter in a name)
Digits	0 through 9
Parentheses	( ) Rounded parentheses. Used to enclose arguments for functions, to embed terms in complex arithmetic expressions. { } Curly braces. Used to enclose units. For example when one states that a quantity is 1 {kg_mole/s}, we enclose the units with curly braces. [ ] Square brackets. Used to enclose sets or elements of sets.
Punctuation	colon (:), semicolon (;)
Binary math relational operators	plus (+), minus (-), time(*), to the power (^), divides (/)
Unary math relational operator	minus (-) (There is no unary + operator.)
Procedural equals	:= *Procedural equals* differs from the ordinary equals (=) in that it means the left-hand-side (LHS) variables is to be replaced by the value of the right-hand-side (RHS) expression immediately when this statement is encountered, either while compiling or when executing a procedure on request through the ASCEND user interface. The order the system encounters these statements matters, therefore, with a later one overwriting an earlier one if both have the same LHS variable. We use an ordinary equals (=) when defining an equation to state that the LHS expression is to equal the RHS expression at the solution for the model.
Math comparators	less than or equals (<), equals (=), greater than or equals (>). (These operators are used in *non-procedural* arithmetic statements only.)
Symbol value	A symbol is text of any length. It can contain any number of any characters in it except the single quote (').
Model part name	A symbol value made up of letters, digits and underscore (_) only. It must have fewer than 80 characters in it.
Comments	Comments are any symbol value beginning with an open parenthesis followed immediately by an asterisk, (*, and ending with an asterisk followed immediately by a closed parenthesis, *). They may NOT be nested. Examples are: (* this is a comment *)

```
(* and so
is
this *)
```

Integer

An optional negative sign (-) followed by any number of consecutive digits up to the maximum that can be represented by the computer on which one is running ASCEND. Examples are

```
123
-5
10
```

Real

ASCEND represents reals almost exactly as any other mathematically oriented programming language does. The mantissa has an optional negative sign followed by a string of digits and at most one decimal point. The exponent is the letter *e* followed by an integer. The number must not exceed the largest the computer is able to handle. There can be no blank characters in a real. The following are legitimate reals in ASCEND:

```
-1
1.2
1.3e-2
7.888888e34
.6
```

while the following are not:

```
1.  2 (contains a blank within it)
1.3e2.0 (exponent has a decimal in it)
+1.3 (contains illegal unary + sign)
```

Symbol

A type of constant in ASCEND. The format for a symbol is that of a symbol value enclosed between two single quotes. The following are legal symbols in ASCEND:

```
'H2O'
'r1'
```

while the following are not:

```
"ethanol"  (double quotes not allowed)
water (no single quotes)
```

If an ASCEND variable is of type symbol, once it is given a value, that value can never be altered. The system will prevent one from trying.

ASCEND operators

IS_A
ARE_THE_SAME
ARE_ALIKE
IS_REFINED_TO

Basic variable types

REAL and BOOLEAN. These types are built into the ASCEND system, and all variables in ASCEND are of these types or *refinements* of these types. Variables of type *real* and *boolean* are the only ones at present whose values may be altered during the course of solving a model. Boolean variables may take on the values *TRUE* or *FALSE* only.

Basic constant types

INTEGER, BOOLEAN, SYMBOL These types are built into the system and all constants in the system are of this type or refinements of these basic types. Symbols are any set of characters enclosed within single quotes. For example, '34xyz' is a symbol.

Built-in constants

ASCEND has the following built-in constant.

REAL_MAX	the largest real the machine is capable of representing (IEEE value is 1.79e308)

Solver_var

The base-type for all *computable* variables in the current ASCEND system. It has wild card dimensionality. (Wild card means that until ASCEND can decide what its dimensionality is, it has none assigned. ASCEND can decide on dimensionality while compiling or executing.) ASCEND has defined the following parts with associated initial values for each:

attribute	type	default
lower_bound	real	0.0
upper_bound	real	0.0
nominal	real	0.0
fixed	boolean	FALSE

lower_bound and *upper_bound* are bounds for a variable which are monitored and maintained during solving. The nominal value the value used to scale a variable when solving. The flag *fixed* indicates if the variable is to be held fixed during solving.

A full definition of solver_var is in the system.lib.

generic_real

One should not declare a variable to be of type solver_var. The nominal value and bound values will get you into trouble when solving. If you are programming and do not wish to declare variable types, then declare them to be of type generic_real. This type has nominal value of 0.5 and lower and upper bounds of -1.0e50 and 1.0e50 respectively. It is dimensionless. Generic_real is a built in type.

MUTABLE_INTEGER and MUTABLE SYMBOL

Mutable integer and symbol are special variable types whose values can be altered in external procedures. Mutable integers cannot be used to set the limits of a range of a set (as you would be changing the size of the storage space needed for that set and anything else based on that set). You would use one of these only if you pass an integer to a procedure and the procedure needs to alter it, like with a flag or as the next value for the integer when you are going to call the external procedure the next time.

Variables

A variable in ASCEND is an instance of a variable type. The base_type of a computable variable is solver_var; thus, all variables have the parts defined for solver_var.

Dimensionality

Variables have dimensionality such as length/time for velocity. Dimensionality is to be distinguished from the units such as ft/s. Dimensionality is built up from the following base dimensions:

symbol	dimension	typical unit
L	length	meter, m
M	mass	kilogram, kg
T	time	second, s
E	e- current	ampere, A
Q	quantity	mole, mole
TMP	temperature	Kelvin, K
LUM	luminous intensity	candela, cd

P	plane angle	radian, rad
S	solid angle	steradian, srad
C	currency	currency, CR

The atom definitions in the library illustrate the use of dimensionality.

Dimensions may be any combination of these symbols along with rounded parentheses, (), and the operators *, ^ and /. Examples include *M/T* or *M*L^2/T^2/TMP* {this latter means (M*(L^2)/(T^2))/TMP. The second operand for the "to the power" operator ^ must be an integer constant (e.g., -2 or 3).

If the dimensionality for a type is undefined, then ASCEND gives it a wild card dimensionality. If ASCEND can later deduce its dimensionality from its use in a model definition or when values are given to it through the interface, it will do so. For example, suppose *a* has wild card dimensionality, *b* has dimensionality of *L/T* Then the statement:

a + *b* = *3* {ft/s};

requires that *a* have the same dimensionality as the other two terms, namely, *L/T* ASCEND will assign this dimensionality to *a*.

Unit expression

A unit expression may be composed of any combination of unit names defined to the system and any numerical constants combined with times (*), divide(/) and "to the power" (^) operators. The rhs of ^ must be of type integer. Parentheses can be used to group subexpressions EXCEPT a divide operator may not be followed by a grouped subexpression.

Units

A unit expression enclosed in curly brackets {}. When a real number is used in a mathematical expression in ASCEND, it must have a set of units expressed with it. If it does not, ASCEND assumes the number is dimensionless, which may not be the intent of the modeler. An example is shown just above where the number 3 has the units {ft/s} associated with it.

{kg_mole/s/m}	same as {(kg_mole/s)/m}
{m^3/yr}	
{3/100*ft}	same as {0.03*ft}
{s^-1}	same as {1/s}

Illegal examples are

{m/(K*kg_mole)}	grouped subexpression used in denominator (should be written {m/K/kg_mole})
{m^3.5}	power must be integer
{m^(2/3)}	ditto. Should be written {m^2/m^3}.

Sets

Sets are lists of elements, all of type integer or all of type symbol, enclosed between square brackets []. The following are examples of sets:

['methane', 'ethane', 'propane']
[1..5, 7, 15]

```
[2..n_stages]
[1, 4, 2, 1, 16]
[]
```

The symbols 1..5 is an allowable shorthand for the integers 1, 2, 3, 4 and 5 while the symbol 2..n_stages (where n_stages must be of type integer) means all integers from 2 to n_stages. If n_stages is less than 2, then the third set is empty. The repeated occurence of 1 in the fourth set is ignored. The fifth set is the empty set.

We use the term *set* in the pure mathematical sense. The elements have no order. One can only two thing of a set: (1) if an element is a member of it and (2) it cardinality (card(set)). Repeated elements used in defining a set are ignored. The elements of sets **cannot** themselved be sets in ASCEND; i.e., there can be no sets of set.

A set of integers may appear to be ordered to the modeler as the natural numbers have an order. One can find the successor to an integer (by adding one to it). It is the user imposing and using the ordering, not ASCEND. ASCEND sees these integers as elements in the set with NO ordering. Therefore, there are no operators in ASCEND such as successor or precursor member.

Arrays

An array is a list of instances indexed over a set. The instances are all of the same *base* type (as that is the only way they can be defined). An individual member of a list may be more refined than the other members (we shall illustrate that possibility later). The following are arrays in ASCEND.

```
stage[1..n_stages]
y[components]
column[areas][processes]
```

where *components*, *areas* and *processes* are sets. For example *components* could be the set of symbols ['ethylene','propylene'] , *areas* the set of symbols ['feed_prep','prod_purification'] while *processes* could be the set ['alcohol_manuf', 'poly_propropylene_manuf']. Note that the third example is a list of lists (the way that ASCEND permits a multiply subscripted list). There can be up to eight levels of subscripts for a list.

The following are elements in these arrays:

```
stage[1]
y['ethylene']
column['feed_prep'][alcohol_manuf']
```

provided that n_stages is 1 or larger.

Index variable

One can introduce a variable as an index which is to range over a set. Index variables are local to the statements in which they occur. An example of using an index variable is the following FOR statement (see later):

```
FOR i IN components CREATE
        VLE_equil[i]:  y[i] = K[i]*x[i];
END;
```

Label

One can label statements which define relationships (equalities and inequalities) in ASCEND. A label is a sequence of alphameric characters ending in a colon. An example of a labeled statement is

```
mass_balance: m_in = m_out;
```

If a relationship is defined within a FOR statement (see later), it must be indexed so each occurrence of the statement is distinguishable from the others. An example is:

```
FOR i IN components CREATE
        equil[i]: y[i] = K[i]*x[i];
END;
```

The ASCEND interactive user interface identifies relationships by their labels. If one has not provided such a label, the system generates the label

relation_$*model_name*$l*ine_number*

where model _name and line_number are that of the name of the model and the line number in the model on which the statement starts. An example is

```
relation_$mixture$14
```

for the unlabeled relation starting in the model *mixture* on line *14*.

List of names

Often in a statement one can include a list of names. A list is one or more names where multiple list entries are separated from each other by commas. Examples of a list of names are:

```
T1, inlet_T, outlet_T
y[components], y_in
stage[1..n_stages]
```

Single operand real ASCEND functions

exp	exponential (i.e., exp(x) = e^x)
ln	log to the base e
sin	sine
cos	cosine
tan	tangent
arcsin	inverse sine
arccos	inverse cosine
arctan	inverse tangent
erf	error function
sinh	hyperbolic sine
cosh	hyperbolic cosine
tanh	hyperbolic tangent
arcsinh	inverse hyperbolic sine
arccosh	inverse hyperbolic cosine
arctanh	inverse hyperbolic tangent
lnm	modified ln function (see following)

This lnm function is paramertized by a constant a, which is typically set to about 1.e-8. lnm(x) is defined as follows

$$\text{lnm}(x) = \begin{cases} \ln(x) & \text{for } x > a \\ (x-a)/a + \ln(a) & \text{otherwise} \end{cases}$$

Below the value a (default setting is 1.0e-8), lnm takes on the value given by the straight line passing through ln(a) and having the same slope as ln(a) has at a. The lnm function can tolerate a negative argument while the ln function cannot.

The base type real operands for ASCEND functions are enclosed within rounded parentheses, (). An example of use is:

```
y = A*exp(-B/T);
```

(Not implemented - only under consideration at this time.) The following function only works in procedures as it is not a smooth function that would destroy a Newton-based solution algorithm if used in defining a model equation:

ABS	absolute value of argument

Multiple operand real functions

SUM(list_of_expressions) Add all expressions in the list

For the SUM , the base type real items can be arbitrary arithmetic expressions. The resulting items must all be units compatible.

An examples of the use is:

```
SUM(y[components]) = 1;
```

or, equivalently, one could write:

```
SUM(y[i], i IN components) = 1;
```

(Not implemented - only under consideration at this time.) The following functions only work in procedures as they are not smooth function and would destroy a Newton-based solution algorithm if used in defining a model equation:

MAX	maximum value on list of arguments
MIN	minimum value on list of arguments

Multiple operand boolean functions

INTERSECTION(list_of_sets) Find the intersection (and) of the sets listed

UNION(list_of_sets) Find the union (or) of the sets listed

Set function UNION

A function taken over a list of sets. The result is a set whose members are the union of the sets in the list. The syntax is

```
UNION(list_of_sets)
```

Consider the following sets for the examples to follow.

```
A := [1, 2, 3, 5, 9];
B := [2, 4, 6, 8];
```

Then UNION(A, B) is equal to the set [1, 2, 3, 4, 5, 6, 8, 9] which equals [1..6, 8, 9].

Set function INTERSECTION

A function taken over a list of sets. The result is a set whose members are the intersection of the sets in the list. The syntax is

```
INTERSECTION(list_of_sets)
```

For the sets A and B defined just above, INTERSECTIION(A, B) is the set [2];

Set difference

One can subtract one set from another. The result is the first set less any members in the set union of the first and second set. The syntax is

```
first_set - second_set
```

For the sets A and B defined above, the set difference A - B is the set [1, 3, 5, 9] while the set difference
B - A is the set [4, 6, 8];

Set function CHOICE

The format for this instruction is

```
CHOICE(set)
```

The result of running the CHOICE function over a set is any arbitrary single element of that set.

The result of running CHOICE(A) is any member of the set A. The result is a member, not a set. To make the result into a set, it must be enclosed in square brackets. Thus [CHOICE(A)] is a set with a single element in it which could be any member of the set A.

To reduce a set by one element, one can use the following

```
A_less_one IS_A set OF integer;
A_less_one := A - [CHOICE(A)];
```

Statements

Statements

Statements in ASCEND terminate with a semicolon (;). Statements may extend over any number of lines. They may have blank lines in the middle of them. There may be several statements on a single line.

One constructs statements using the above building blocks.

Compound statements

Some statements in ASCEND can contain other statements as a part of them. The three examples are the FOR/CREATE, the FOR/DO and the IF statements.

In the following statement descriptions, we show keywords in capital letters. These words must appear in capital letters as shown in ASCEND statements. We show optional parts to a statement enclosed in double angle brackets (« ») and user supplied names in lower-case *italic* letters. (Remember that ASCEND treats the underscore (_) as a letter).

The user may substitute any name desired for these names. We use names that describe the kind of name the user should use.

Begin model statement

This statement has the syntax

MODEL *model_name* « REFINES
name_of_previously_defined_model»;

It is used to initiate MODEL definitions. (The REFINES part is optional.)

Examples are:

MODEL fancy_flash REFINES simple_flash;
MODEL storage_tank;

Begin atom statement

The syntax for the begin atom statement is

ATOM atom_name REFINES variable_type
«DIMENSION dimension_expression»
«DEFAULT value»;

The options for variable type are those built into the system (real, integer, boolean) or the name of any other previously defined atom.

The dimension attribute is for variables whose base type is real. It is an optional field. If not defined for any atom whose base type is real, the dimensions will be left as undefined. Any variable which is later declared to be one of these types will be given *wild card* dimensionality (represented in the interactive display by an asterisk (*). The system will deduce the dimensionality from its use in the relationships in which it appears or in the declaring of default values for it, if it can.

The default field is also optional. If the atom has a declared dimensionality, then this value must be expressed with units which are compatible with this dimensionality.

End model/atom statement

This syntax for this statement is

END *model_name*;

or

END *atom_name*;

It is used to terminate model and atom definitions.

IS_A statement

This statement has the syntax

list_of_instance_names IS_A *model_name*;

The IS_A statement allows us to declare *instances* of a given *type* to exist within a model definition.

Instances and types

This is an opportune time to emphasize again the distinction between the terms *instance* and *type*. A *type* in ASCEND is what we define when we declare an ASCEND model or atom. It is the definition of the attributes that something will have if the system creates one of them using this definition.

In ASCEND there are two meanings (closely related) of an instance. (1) An *instance* is a *named part* that exists within a model definition. (2) If one is in the context of the ASCEND interface, the system compiles an instance of a model to create an object with which one carries out computations. The system asks you to name this instance, this name being then the first part of the qualified name for the objects seen in the compiled object.

IS_REFINED_TO statement

This statement has the syntax

```
list_of_instances IS_REFINED_TO model_name;
```

We use this statement to change the type of the instances listed to being of a more refined type. The modeler has to have defined each member on the list of instances previously. The model_name has to be a type which is an ancestor in the refinement hierarchy for all of the instances on the list.

An example of its use is as follows. First we define the the parts called fl1, fl2 and fl3 which are of type flash.

```
fl1, fl2, fl3 IS_A flash;
```

Assume that there exists in the previously define model definition an adiabatic_flash as a refinement of flash. Then we can make fl1 and fl3 into more refined types by stating:

```
fl1, fl3 IS_REFINED_TO adiabatic_flash;
```

ARE_THE_SAME statement

The format for this instruction is

```
list_of_instance_names ARE_THE_SAME;
```

All items on the list must have compatible types. For the example in Fig. 1, consider a model where we define the following parts:

```
a1 IS_A a;
b1 IS_A b;
c1 IS_A c;
d1 IS_A d;
e1 IS_A e;
```

Then the following ARE_THE_SAME statement is legal

```
a1, b1, c1 ARE_THE_SAME;
```

while the following are not

```
b1, d1 ARE_THE_SAME;
a1, c1, d1 ARE_THE_SAME;
b1, e1 ARE_THE_SAME;
```

When compiling a model, ASCEND will put all of the instances mentioned as being the same into an ARE_THE_SAME "clique." ASCEND lists members of is this clique when you ask for the aliases of any object in a compiled model.

Merging any other item with a member of the clique makes it the same as all the other items in the clique, i.e., it adds the newly mentioned items to the existing clique.

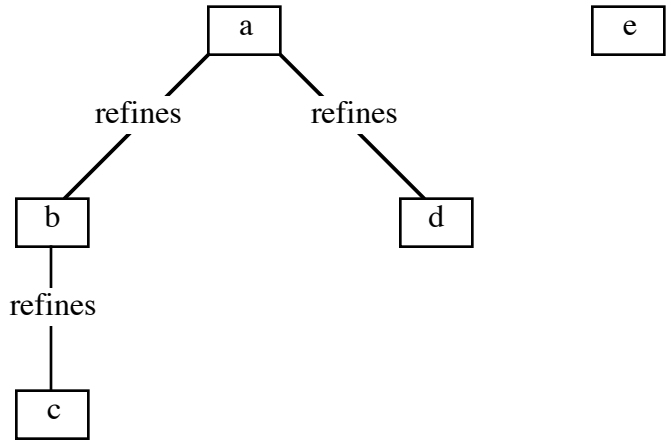

Figure 1. Diagram of the model hierarchy

ASCEND merges all members of a clique by first checking that all members on the clique are type compatible. It then changes the type designation for all clique members to that of the most refined type for any member on the clique.

It next looks inside each of the instances, all of which are now of the same type, and puts all of the parts with the same name into their respective ARE_THE_SAME cliques. The process repeats by processing these cliques until all parts of all parts of all parts, etc., are their respective most refined type or discovered to be type incompatible.

There are now lots of cliques associated with the instances being merged. The type associated with each such clique is now either a model or an atom (i.e., a variable). If a model, only one member of the clique generates its equations. If a variable, it assigns all members to the same storage location.

ARE_ALIKE statement

The format for this statement is

list_of_instance_names ARE_ALIKE;

ARE_ALIKE is similar in implementation to ARE_THE_SAME. The compiler places all instances on the list onto an ARE_ALIKE clique. It checks that the members are type compatible and then it converts each into the most refined type of any instance in the clique. At that point it stops. It does not continue by placing the parts into cliques nor does it assign common storage locations, etc. It may be thought of as a partial merge. There are important consequences in modeling of such a partial merge.

One consequence of ARE_ALIKE is to prevent model misuse when configuring models. For example, suppose a modeler creates a pressure changing model. The modeler is not concerned about the type of the streams into and out of the device but does care that these streams are of the same final type. For example, the modeler wants both to be liquid streams if either is or both to be vapor streams if either is. By declaring both to be streams only but declaring the two streams to be alike, the modeler accomplishes this intent. Suppose the modeler merges the inlet stream with a liquid outlet stream from a reactor. The merge operation makes the inlet stream into a liquid stream. The outlet stream, being in an ARE_ALIKE clique with the inlet stream, also becomes a liquid stream. Any subsequent merge of the outlet stream with a vapor stream will lead to an error due to type incompatibility when ASCEND attempts to compile that merge. Without the ARE_ALIKE statement, the compiler would detect no such incompatibility.

Another purpose is the propagation of type through a model which we just saw taking place in the previous paragraph. Altering the type of the inlet stream through merging it with a liquid stream automatically made the outlet stream into a liquid stream.

If all the liquid streams within a distillation column are alike, then the modeler can make them all into streams with a particular set of components in them and with the same method used for physical property evaluation by merging only one of them with a liquid stream of this type.

UNIVERSAL qualifier

The begin model and the begin atom statements can have a qualifier preceding them which causes all instances of the type being defined to be merged. The syntax for these statements are

```
UNIVERSAL model_begin_statement
UNIVERSAL atom_begin_statement
```

Examples are

```
UNIVERSAL MODEL methane REFINES
        components_constants;

UNIVERSAL ATOM circle_constant REFINES real
        DIMENSIONLESS;
```

In the former case, ASCEND will merge all instances of the physical property data for methane (such as it molecular weight, its normal boiling point, etc) into the same instance. In the later, all instances of circle_constant (3.1415926536) become the same instance.

Mathematical expression

The syntax for a mathematical expression is any legal combination of variable names and arithmetic operators in the normal notation. An expression may contain any number of matched rounded parentheses, ( ), to clarify meaning. The following is a legal arithmetic expression:

```
y^2+(sin(x)-tan(z))*q
```

Each term in a mathematical expression (terms are separated by + or - operators) must have the same dimensionality.

An expression may contain an index variable as a part of the calculation if that index variable is over s aset whose elements are of type integer. (See the FOR/CREATE and FOR/DO statements below.) An example is:

```
term[i] = a[i]*x^(i-1);
```

Logical expression

(The following is proposed and is not implemented at this time.) An expression whose value is TRUE or FALSE is a logical expression. The mathematical comparitor operators (=, <, >) are common in such expressions. Such expressions may also contain boolean variables. If laminar is a boolean, then the following is an example:

```
(a>0) + ((b<3) * laminar)
```

The plus operator acts like an OR among the terms while the times operator acts like an AND. Think of TRUE being equal to 1 and FALSE being equal to 0 with the 1+1=0+1=1+0=1 , 0+0=0, 1*1=1 and 0*1=1*0=0*0=0. If *a* in fact equals -5, *b*=2.99 and *laminar* is TRUE, this expression has the value

```
FALSE OR (TRUE AND TRUE) = TRUE
```

or in terms of ones and zeros

```
0 + (1 * 1) = 1
```

Mathematical relation

The syntax for a mathematical relation is

```
math_expression1  math_comparitor
        math_expression2;
```

where a math_comparator is one of the following: equals (=), less than or equals (<) or greater than or equals (>). Math relationships are equations or inequalities. The following are examples:

```
Ftot*y['methane'] = m['methane'];
y['ethanol'] > 0;
```

where this last one means that the mole fraction for ethanol is greater than or equal to 0. Equations must be dimensionally correct.

FOR/CREATE statement

The FOR/CREATE statement is a compound statement. Its format is:

```
FOR index_variable IN set CREATE
        list_of_statements;
END;
```

This statement must be in the nonprocedural part of the model definition only. Every statement in the list must have at least one variable which is indexed by the index variable, else the same statement will be produced multiple times. An example is

```
FOR i IN components CREATE
        a.y[i], b[i]  ARE_THE_SAME;
        y[i] = K[i]*x[i];
END;
```

INITIALIZATION statement

This statement introduces the procedure definitions in ASCEND. All statements following this statement are to define procedures in ASCEND while all before it are for the declarative part of ASCEND. The syntax for this statement is simply

```
INITIALIZATION
```

with no punctuation. The next statement must be a procedure heading statement. If there are no procedure definitions, this statement is omitted.

Begin PROCEDURE statement

The syntax for a begin procedure statement is

```
PROCEDURE procedure_name;
```

End PROCEDURE statement

The syntax of a procedure ending statement is

```
END procedure_name;
```

LHS name

A LHS_name is one of the following

```
instance_name
array_name[set_name]
set_name
```

RHS expression

A RHS expression is one of the following

```
mathematical, boolean or set expression
```

Procedural assignment statement

The syntax is

```
instance_name := mathematical_expression;
```
or
```
array_name[set_name] :=
        mathematical_expression
```
or
```
set_name := [list_of_symbols];
```
or
```
set_name := set_expression;
```

Its meaning is that the value for the variable on the LHS is set to the value of the expression on the RHS.

FOR/DO statement

This statement is similar to the FOR/CREATE statement except it can only appear in a procedure definition. An example would be

```
FOR i IN [1..n_stages] DO
        T[i] := T[1] + (i-1)*DT;
END;
```

IF statement

This statement can only appear in a procedure definition. Its syntax is

```
IF (logical_expression)
THEN
        list_of_statements
ELSE
        list_of_statements
```

```
END;
```

or

```
IF (logical_expression)
THEN
        list_of_statements
END;
```

If the logical expression has a value of TRUE, the statements ASCEND will execute the statements in the THEN part. If it is FALSE, it executes the statements in the optional ELSE part.

RUN statement

This statement can appear only in a procedure. Its format is

```
RUN name_of_procedure;
```

The procedure can be within the current model or within any of its parts.

PROCEDURE

A procedure in ASCEND must appear following the INITIALIZE statement within a model. The first statement must be a begin procedure statement and the last an end procedure statement. In-between exist procedural assignment, FOR/DO and IF statements. The systems executes procedure statements in the order they are encountered.

Specifically disallowed are IS_A, IS_REFINED_TO, ARE_THE_SAME and ARE_ALIKE statements as these "declare" the structure of the model and belong only in the delarative section.

ASCEND model

An ASCEND model begins with a *begin model* statement and ends with an *end model* statement. Both must have the identical model_name. It has a declarative part and an optional procedural part headed by an INITIALIZATION statement.