CHAPTER 19THE ASCEND IV LANGUAGE SYNTAX AND SEMANTICS

Benjamin Allan[1]

Arthur W. Westerberg[1]

Department of Chemical Engineering
and the Engineering Design Research Center /
Institute for Complex Engineered Systems

Carnegie Mellon University

We shall present an informal description of the ASCEND IV language. Being informal, we shall usually include examples and 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 chapter to be used as a reference for the ASCEND language even 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, `y`, followed by the ASCEND operator `IS_A`, followed by a type name (fraction). The statement terminates with a semicolon. The statement semantics says we are declaring the existence of an instance,

1. The ASCEND language has evolved from the combined efforts of several generations of users and implementors. We wish to particularly acknowledge the contributions of ASCEND III implementors Kirk Abbott, Tom Epperly, Peter Piela, Boyd Safrit, Karl Westerberg, and Joe Zaher, and of the ASCEND IV crew: Duncan Coffey, Chad Farschman, Jennifer Perry, Vicente Rico-Ramirez, Mark Thomas and Ken Tyner.

locally named y, of the type fraction as a part within the current model definition and it is to be constructed when an instance of the current model definition is constructed.

The syntax for a computer language is often defined by using a Bachus-Naur formal (BNF) description. The complete YACC and FLEX description of the language described (as presently implemented) is available by FTP[2] and via the World Wide Web[3]. The semantics of a very high level modeling language such as ASCEND IV are generally much more restrictive than the syntax. For this reason we do not include a BNF description in this paper. ASCEND IV is an experiment. The language is under constant scrutiny and improvement, so this document is under constant revision. Contact the authors for the latest version.

19.1 PRELIMINARIES

We will start off with some background information and some tips that make the rest of the chapter easier to read. ASCEND is an object-oriented (OO) language for hierarchical modeling that has been somewhat specialized for mathematical models. Most of the specialization is in the implementation and the user interface rather than the language definition.

We feel the single most distinguishing feature of mathematical models is that solving them efficiently requires that the solving algorithms be able to address the entire problem either simultaneously or in a decomposition of the natural problem structure that the algorithm determines is best for the machine(s) in use. In the ASCEND language object-orientation is used to organize natural structures and make them easier to understand. It is not used to hide the details of the objects. The user (or machine) is free to ignore uninteresting details, and the ASCEND environment provides tools for the runtime suppression of these.

ASCEND is well into its 4th generation. Some features we will describe are not yet implemented (some merely speculative) and these are clearly marked (* 4+ *). Any feature not marked (* 4+ *)has been completely implemented, and thus any mismatch between the description given here and the software we distribute is a bug we want you to tell us about.

The syntax and semantics of ASCEND may seem at first a bit unusual. However, do not be afraid to just try what comes naturally if what we write here is unclear. The parser and compiler of ASCEND IV really will help you get things right. Of course if what

2. In the directory ftp.cs.cmu.edu:project/ascend/gnu-ascend/ see the file README.
3. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ascend/ftp/gnu-ascend/README

we write here is unclear, please ask us about it because we aim to continuously improve both this document and the language system it describes.

We will describe, starting in Section 19.1.2, the higher level concepts of ASCEND, but first some important punctuation rules.

ASCEND is cAsE sensitive!

The keywords that are shown capitalized (or in lower case) in this chapter are that way because ASCEND is case sensitive. IS_A is an ASCEND keyword; isa, Is_a, and all the other permutations you can think of are NOT equivalent to IS_A. In declaring new types of models and variables the user is free to use any style of capitalization he or she may prefer, however, they must remain consistent or undefined types and instances will result.

This case restriction makes our code very readable, but hard to type without a smart editor. We have kept the case-sensitivity because, like all mathematicians, we find ourselves running out of good variable names if we are restricted to a 26 letter alphabet. We have developed smart add-ins for two UNIX editors, EMACS and vi, for handling the upper case keywords and some other syntax elements. The use of these editors is described in another chapter.

The ASCEND IV parser is very picky and pedantic. It also tries to give helpful messages and occasionally even suggestions. New users should just dive in and make errors, letting the system help them learn how to avoid errors.

19.1.1 PUNCTUATION

This section covers both the punctuation that must be understood to read this document and the punctuation of ASCEND code.

keywords:

ASCEND keywords and type names are given in the left column in **bold** format. It is generally clear from the main text which are keywords and which are type names.

Minor items:

Minor headings that are helpful in finding details are given in the left column in underline format.

Tips:

Special notes and hints are sometimes placed on the left.

3:

This indicates that what follows is specific to ASCEND IIIc and may disappear in a future version of ASCEND IV. Generally ASCEND IV will provide some equivalent functionality at 1/10th of the ASCEND III price.

4	This indicates that what follows is specific to ASCEND IV and may not be available in ASCEND IIIc. Generally ASCEND III may provide some very klugey equivalent functionality, often at a very high price in terms of increased compilation time or debugging difficulty.
4+	ASCEND IV functionality that is not fully implemented at the time of this writing. The precise syntax of the final implementation may vary slightly from what is presented here. A revision of this document will be made at the time of implementation.
LHS:	Left Hand Side. Abbreviation used frequently.
RHS:	Right Hand Side. Abbreviation used frequently.
Simple Names:	In ASCEND simple names are made of the characters a through z, A through Z, _, (*4+*: $). The underscore is used as a letter, but it cannot be the first letter in a name. The “$” character is used exclusively as the first character in the name of system defined built-in parts. "$" is explained in more detail in Section 19.6.2. Simple names should be no more than 80 characters long.
Compound names:	Compound names are simple names strung together with dots (.). See the description of "." below.
Groupings:	
« »	In documentation optional fields are surrounded by these markers.
`(*  *)`	Comment. *3* Anything inside these is a comment. Comments DO NOT nest in ASCEND IIIc. Comments may extend over many lines. *4* Comments DO nest in ASCEND IV.
`(  )`	Rounded parentheses. Used to enclose arguments for functions or models where the order of the arguments matters. Also used to group terms in complex arithmetic, logical, or set expressions where the order of operations needs to be specified.
Efficiency tip:	The compiler can simplify relation definitions in a particularly efficient manner if constants are grouped together.
`{  }`	Curly braces. Used to enclose units. For example, 1 {kg_mole/s}. Also used to enclose the body of annotations. **Note**: Curly braces are also used in TCL, the language of the ASCEND user interface, about which we will say more in another chapter.

`[ ]`	Square brackets. Used to enclose sets or elements of sets. Examples: my_integer_set :== [1,2,3], demonstrates the use of square brackets in the assignment of a set. My_array[1] demonstrates the use of square brackets in naming an array object indexed over an integer set which includes the element 1.
`.`	Dot. The dot is used, as in PASCAL and C, to construct the names of nested objects. Examples: if object a has a part b, then the way to refer to b is as a.b. Tray[1].vle shows a dot following a square bracket; here Tray[1] has a part named vle.
`..`	Dot-dot or double dot. Integer range shorthand. For example, my_integer_set :== [1,2,3] and my_integer_set :== [1..3] are equivalent. If .. appears in a context requiring (), such as the ALIASES/IS_A statement, then the range is expanded and ordered as we would naturally expect.
`:`	Colon. A separator used in various ways, principally to set the name of an arithmetic relation apart from the definition.
`::`	Double colon. A separator used in the methods section for accessing methods defined on types other than the type the method is part of. Explained in Section 19.4.
`;`	Semicolon. The separator of statements.

19.1.2 BASIC ELEMENTS

<u>Boolean value</u>	TRUE or FALSE. Can’t get much simpler, eh? In the language definition TRUE and FALSE do not map to 1 and 0 or any other type of numeric value. (In the implementation, of course, they do.)
User interface tip:	The ASCEND user interface programmers have found it very convenient, however, to allow T/F, 1/0, Y/N, and other obvious boolean conventions as interactive input when assigning boolean values. We are lazy users.

<u>Integer value</u>	A signed whole number up to the maximum that can be represented by the computer on which one is running ASCEND. MAX_INTEGER is machine dependent. Examples are:
	
	`123`
	`-5`
	`MAX_INTEGER, typically 2147483647.`

Real value

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* or 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. MAX_REAL is machine dependent. The following are legitimate reals in ASCEND:

```
-1
1.2
1.3e-2
7.888888e+34
.6E21
MAX_REAL, typically about 1.79E+308.
```

while the following are not:

```
1.  2 (*contains a blank within it*)
1.3e2.0 (*exponent has a decimal in it*)
+1.3 (* illegal unary + sign. x = +1.3 not allowed*)
```

Reals stored in SI units

We store all real values as double precision numbers in the MKS system of units. This eliminates many common errors in the modeling of physical systems. Since we also place the burden of scaling equations on system routines and a simple modeling methodology, the internal units are not of concern to most users.

Dimensionality:

Real values have dimensionality such as length/time for velocity. Dimensionality is to be distinguished from the units such as ft/s. ASCEND takes care of mapping between units and dimensions. A value without units (this includes integer values) is taken to be dimensionless. Dimensionality is built up from the following base dimensions:

Name	definition	typical units
L	length	meter, m
M	mass	kilogram, kg
T	time	second, s
E	electric 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 and constant 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 value (e.g., -2 or 3) because fractional powers of dimensional numbers are physically undefined.

If the dimensionality for a real value is undefined, then ASCEND gives it a wild card dimensionality. If ASCEND can later deduce its dimensionality from its use in a model definition it will do so. For example consider the real variable *a*, suppose *a* has wild card dimensionality, *b* has dimensionality of *L/T*. Then the statement:

Example of a dimensionally consistent equation.

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*. The user will be warned of dimensionally inconsistent equations.

Unit expression

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

So, {kg/m/s} is fine, but {kg/(m*s)} is not. Although the two expressions are mathematically equivalent, it makes the system programming and output formatting easier to code and faster to execute if we disallow expressions of the latter sort.

The units understood by the system are defined in Chapter 20. Note that several "units" defined are really values of interesting constants in SI, e.g. R :== 1{GAS_C} yields the correct value of the thermodynamic gas constant. Users can define additional units.

Units

A unit expression must be enclosed in curly braces {}. 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 in the dimensionally consistent equation above where the number 3 has the units {ft/s} associated with it.

Examples:

```
{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 unit examples are

`{m/(K*kg_mole)}` grouped subexpression used in the denominator; should be written `{m/K/kg_mole}`.
`{m^3.5}` power of units or dimensions must be integer.

Symbol Value

The format for a symbol is that of an arbitrary character string enclosed between two single quotes. There is no way to embed a single quote in a symbol: we are not in the escape sequence business at this time. The following are legal symbols in ASCEND:

```
’H2O'
’r1'
’Bill said,”foo” to whom?’
```

while the following are not legal symbol values:

```
"ethanol" (double quotes not allowed)
water (no single quotes given)
’i can’t do this’ (no embedded quotes)
```

There is an arbitrary upper limit to the number of characters in a symbol (something like 10,000) so that we may detect a missing close quote in a bad input file without crashing.

Sets values

Set values are lists of elements, all of type integer_constant or all of type symbol_constant, 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]
[]
```

We will say more about sets in 19.2.2.

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

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

Sets are unordered.

A set of integers may appear to be ordered to the modeler as the natural numbers have an order. However, 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 of a set.

Arrays

An array is a list of instances indexed over a set, in computer-speak, an associative array of objects. 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 later be more refined than the other members (we shall illustrate that possibility). 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 (column) is a list of lists (the way that ASCEND permits a multiply subscripted array).

The following are elements in the above arrays:

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

provided that n_stages is 1 or larger.

There can be any number of subscripts for an array. We point out, however, that in virtually every application of arrays requiring more than two subscripts, there is usually a some underlying concept that is much better modeled as an object than as part of a deeply subscripted array. In the following jagged array example, there are really the concepts of unit operation and stream that would be better understood if made explicit.

Arrays can be jagged

(* 4 *) Arrays can be ’sparse’ or jagged. For example:

```
process[1..3] IS_A set OF integer;
process[1] :== [2];
process[2] :== [7,5,3];
process[3] :== [4,6];
FOR i in [1..3] CREATE
   FOR j IN process[i] CREATE
      flow[i][j] IS_A mass;
   END FOR;
END FOR;
```

`process` is an array of sets (not to be confused with a set of sets which ASCEND does not have) and `flow` is an array with six elements spread over three rows:

```
flow[1][2]
flow[2][7], flow[2][3], flow[2][5]
flow[3][4], flow[3][6]
```

Sparse arrays of models and variables are new to ASCEND IV.

Arrays are also instances

Each array is itself an object. That is, when you write "`a[1..2]IS_A real;`" three objects get created: `a[1]`, `a[2]`, and `a`. *a* is an `array` instance which has parts named [1] and [2] that are `real` instances. When a parameterized model requires an array, you pass it the single item `a`, not the elements `a[1..2]`.

Not contiguous storage

Just in case you still have not caught on, ASCEND arrays are not blocks of memory such as are seen in low-level languages like C, FORTRAN, and Matlab. The *modeling* language does not provide things like MatMult, Transpose, and Inverse because these are *procedural* solving tools. If you are dedicated, you could write METHODs that implement matrix algebra, but this is a really dumb idea. We aim to structure our software so that it can interact openly with separate, dedicated tools (such as Matlab) when those tools are needed.

Index variable

One can introduce a variable as an index ranging 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:

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

In this example `i` implicitly is of the same type as the values in the set `components.`If another object `i` exists in the model containing the FOR loop, it is ignored while executing the statements in that loop. This may cause unexpected results and the compiler will generate warnings about loop index shadowed variables.

Label:

One can label statements which define arithmetic relationships (objective functions, equalities, and inequalities) in ASCEND. Labeling is highly recommended because it makes models much more readable and more easily debugged. Labels are also necessary for relations which are going to be used in conditional modeling or differentiation functions. A label is a sequence of alphanumeric characters ending in a colon. An example of a labeled equation is:

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

An example of a labeled objective function is:

```
obj1: MAXIMIZE revenue - cost;
```

If a relation is defined within a FOR statement, it must have an array indexed label so that each instance created using the statement is distinguishable from the others. An example is:

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

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

modelname_equationnumber

where *modelname* and *equationnumber* are the name of the model and the equation number in the model. An example is

```
mixture_14
```

for the unlabeled 14th relation in the mixture definition. If there is a conflict caused with an existing name, the generated name has enough letters added after *equationnumber* to make it a unique name. Remember that each model in a refinement hierarchy inherits the equations of its less refined ancestors, so the first equation appearing in the source code of a refining model may actually be the nth relation in that model.

Lists

Often in a statement one can include a list of names or expression. A name 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]
```

Ordered lists:

If the ordering of names in a list matters, that list is enclosed in (). Order matters in: calling externally defined methods or models, calling most real-valued functions, passing parameters to ASCEND models or methods, and declaring the controlling parameters that SELECT, SWITCH, and WHEN statements make decisions on.

19.1.3 BASIC CONCEPTS

Instances and types

This is an opportune time to emphasize 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 formal definition of the attributes (parts) and attribute default values that an object will have if it is created using the type definition. Methods are associated with types.

In ASCEND there are two meanings (closely related) of an instance.

- An *instance* is a *named part* that exists within a type definition.
- An *instance* is a compiled object.

If one is in the context of the ASCEND interface, the system compiles an instance of a model type to create an object with which one carries out computations. The system requires the user to give a simple name for this simulation instance. This name given is then the first part of the qualified name for all the parts of the compiled object.

Implicit types

It is possible to create an instance that does not have a corresponding type definition in the library. The type of such an instance is said to be *implicit.* (Some people use the word *anonymous*. However, no computable type is anonymous and the implicit type of an instance is theoretically computable). The simplest example of an implicit type is the type of an instance compiled from the built-in definition `integer_constant`. For example:

```
i, j IS_A integer_constant;
i:== 2;
j:== 3;
```

Instances i and j, though of the same formal type, are implicit type incompatible because they have been assigned distinct values.

Instances which are either formally or implicitly type incompatible cannot be merged. This will be discussed further in Section 19.3.

Parsing

Most errors in the declaration of an ASCEND model can be caught at parse time because the object type of any well-formed name in an ASCEND definition can be resolved or proved ambiguous. We cannot prove at parse time whether a specific array element will exist, but we can know that should such an element exist, it must be of the type with which the array is defined.

Ambiguity is warned about loudly because it is caused by either misspelling or poor modeling style. The simplest example of ambiguity follows.

Assume a type, `stream`, and a refinement of `stream`, `heat_stream`, which adds the new variable H. Now, if we write:

```
MODEL mixer;
input[1..2] IS_A stream;
output IS_A heat_stream;
input[1].H + input[2].H = output.H;
END mixer;
```

We see the parser can find the definition of `H` in the type `heat_stream`, so `output.H` is well defined. The author of the mixer model may intend to refine input[1] and input[2] to be objects of different types, say `steam_stream` and `electric_stream`, where each defines an `H` suitable for use in the equation. The parser cannot read the author's mind, so it warns that input[1].H and input[2].H are ambiguous in the mixer definition. The mixer model is not highly reusable except by the author, but sometimes reusability is not a high priority objective. The mixer definition is allowed, but it may cause problems in instantiation if the author has forgotten the assumption that is not explicitly stated in the model and neglects to refine the input streams appropriately.

Instantiation

Creating an simulation based on a type definition is a multi-phase process called compiling (or instantiation). When an instantiation cannot be completed because some structural parameter (a symbol_constant, real_constant, boolean_constant, integer_constant, or set) does not have a value there will be PENDING statements. The user interface will warn that something is incomplete.

In phase 1 all statements that create instance structures or assign constant values are executed. This phase theoretically requires an infinite number of passes through the structural statements of a definition. We allow a maximum of 5 and have never needed more than 3. There may be pending statements at the end of phase 1. The compiler or interface will issue warnings about pending statements, starting with warnings about unassigned constants.

Phase 2 compiles as many real arithmetic relation definitions as possible. Some relations may be impossible to compile because the constants or sets they depend on do not have values assigned. Other relations may be impossible because they reference variables that do not exist. This is determined in a single pass.

Phase 3 compiles as many logical arithmetic relation definitions as possible. Some relations may be impossible to compile because the constants or sets they depend on do not have values assigned. Other relations may be impossible because they reference real arithmetic relations that do not exist. This is determined in a single pass.

Phase 4 compiles as many conditional programming statements (WHENs) as possible. Some WHEN relations may be impossible to compile because the discrete variables, models, or relations they depend on do not exist. This is determined in a single pass.

Phase 5 executes the variable defaulting statements made in the declarative section of each model IF AND ONLY IF there are no pending statements from phases 1-4 anywhere in the simulation.

default_self

After all phases are done, the method *default_self* is called in the top-most model of the simulation, if this method exists.

The first occurrence of each impossible statement will be explained during a failed compilation. Impossible statements include:

- Relations containing undefinable variables (often misspellings).
- Assignments that are dimensionally inconsistent or containing mismatched types.
- Structure building or modifying statements that refer to model parts which cannot exist or that require a type-incompatible argument, refinement, or merge.

19.2 DATA TYPE DECLARATIONS

In the spectrum of OO languages, ASCEND is best considered as being class-based, though it is rather more a hybrid. We have atom and model definitions, called *types*, and the compiled objects themselves, called *instances*. ASCEND instances have a record of what type they were constructed from.

Type qualifiers:

UNIVERSAL

Universal is an optional modifier of all ATOM, CONSTANT. and MODEL definitions. If UNIVERSAL precedes the definition, then ALL instances of that type will actually refer to the first instance of the type that is created. This saves memory and ensures global consistency of data.

Examples of universal type definitions are

```
UNIVERSAL MODEL methane REFINES
generic_component_model;

UNIVERSAL CONSTANT circle_constant REFINES
real_constant :== 1{PI};

UNIVERSAL ATOM counter_1 REFINES integer;
```

Tip: Don’t use UNIVERSAL variables in relations.

It is important to note that, because variables must store information about which relations they occur in, it is a very bad idea to use UNIVERSAL typed variables in relations. The construction and maintenance of the relation list becomes very expensive for universal variables. UNIVERSAL constants are fine to use, though, because there are no relation links for constants.

19.2.1 MODELS

MODEL

An ASCEND model has a declarative part and an optional procedural part headed by the METHODS word. Models are essentially containers for variables and relations. We will explain the various statements that can be made within models in Section 19.3 and Section 19.4.

Simple models:

foo

```
MODEL foo;
   (* statements about foo go here*)
METHODS
   (* METHODs for foo go here*)
END foo;
```

bar

```
MODEL bar REFINES foo;
   (*additional statements about foo *)
METHODS
   (* additional METHODs for bar *)
```

```
END bar;
```

Parameterized Models

(* 4 *) Parameterizing models makes them easier to understand and faster for the system to compile. The syntax for a parameterized model vaguely resembles a function call in imperative languages, but it is NOT. When constructing a reusable model, all the constants that determine the sizes of arrays and other structures should be declared in the parameter list so that

- the user knows what is required to reuse the model.
- the compiler knows what values must be set before it should bother attempting to compile the model.

There is no reason that other items could not also go in the parameter list, such as key variables which might be considered inputs or outputs or control parameters in the mathematical application of the model. A simple example of parameterization would be:

column(n,s)

```
MODEL column(
ntrays WILL_BE integer_constant;
components IS_A set of symbol_constant;
);
   stage[1..ntrays] IS_A simple_tray;
END column;
```

flowsheet

```
MODEL flowsheet;
   tower4size IS_A integer_constant;
   tower4size :== 22;
   ct IS_A column(tower4size,['c5','c6']);
   (* additional flowsheet statements *)
END flowsheet;
```

In this example, the column model takes the first argument, ntrays, by reference. That is, `ct.ntrays` is an alias for the flowsheet instance `tower4size`. `tower4size` must be compiled and assigned a value before we will attempt to compile the column model instance ct. The second argument is taken by value, `['c5','c6'],` and assigned to `components`, a column part that was declared with IS_A in the parameter list. There is only one name for this set, `ct.components`. Note that in the flowsheet model there is no part that is a set of `symbol_constant`.

The use of parameters in ASCEND modeling requires some thought, and we will present that set of thoughts in Section 19.5.

Beginners may wish to create new models without parameters until they are comfortable using the existing parameterized library definitions. Parameters are intended to support model reuse and efficient compilation which are not issues in the very earliest phase of developing novel models.

19.2.2 SETS

Arrays in ASCEND, as already discussed in Section 19.1.2, are defined over sets. A set is simply an instance with a set value. The elements of sets are NOT instances or sets.

Set Declaration:

A set is made of either symbol_constants or integer_constants, so a set object is declared in one of two ways:

```
my_integer_set IS_A set OF integer_constant;
or
my_symbol_set IS_A set OF symbol_constant;
```

:==

A set is assigned a value like so:

```
my_integer_set :== [1,4];
```

The RHS of such an assignment must be either the name of another set instance or an expression enclosed in square brackets and made up of only set operators, other sets, and the names of integer_constants or symbol_constants. Sets can only be assigned once.

Set Operations

UNION[setlist]

A function taken over a list of sets. The result is the set that includes all the members of all the sets in the list. Note that the result of the UNION operation is an unordered set and the argument order to the union function does not matter. The syntax is:

+

```
UNION[list_of_sets]
```

A+B is shorthand for UNION[A,B]

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] which equals [[1..9] - [7]].

INTERSECTION[]

INTERSECTION[list of set expressions]. Finds the intersection (and) of the sets listed.

*****

`Equivalent to INTERSECTION[list_of_sets].`

A*B is shorthand for INTERSECTION[A,B]

For the sets A and B defined just above, `INTERSECTION[A, B]` is the set `[2]`. The * shorthand for intersection is NOT recommended for use except in libraries no one will look at.

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]`.

CARD[set]

Cardinality. Returns an integer constant value that is the number of items in the set.

CHOICE[set]

Choose one. The result of running the CHOICE function over a set is an arbitrary (but consistent: for any set instance you always get the same result) single element of that set.

Running `CHOICE[A]` gives any member from 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 arbitrarily chosen from the set A. Good modelers do not leave modeling decisions to the compiler; they do not use CHOICE[]. We are stuck with it for historical reasons.

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]];
```

IN

lhs IN rhs can only be well explained by examples. IN is used in index expressions. If lhs is a simple and not previously defined name, it is created as a temporary loop index which will take on the values of the rhs set definition. If lhs is something that already exists, the result of lhs IN rhs is a boolean value; stare at the model `set_example` below which demonstrates both IN and SUCH_THAT. If you still are not satisfied, you might examine [[westerbergksets]].

SUCH_THAT (* 4 *)

Set expressions can be rather clever. We will give a detailed example from chemistry because unordered sets are unfamiliar to most people and set arithmetic is quite powerful. In this example we see arrays of sets and sparse arrays.

```
MODEL set_example;
   (* we define a sparse matrix of reaction coefficient information
   * and the species balance equations. *)
   rxns IS_A set OF integer_constant;
   rxns :== [1..3];
   species IS_A set OF symbol_constant;
   species :== ['A','B','C','D'];

   reactants[rxns] IS_A set OF symbol_constant; (* species in each rxn_j *)
   reactants[1] :== ['A','B','C'];
   reactants[2] :== ['A','C'];
   reactants[3] :== ['A','B','D'];

   reactions[species] IS_A set OF integer_constant;
   FOR i IN species CREATE (* rxns for each species i *)
      reactions[i] :== [j IN rxns SUCH_THAT i IN reactants[j]];
   END FOR;
   (* Define sparse stoichiometric matrix. Values of eta_ij set later.*)
   FOR j IN rxns CREATE
      FOR i IN reactants[j] CREATE
      (* eta_ij --> mole i/mole rxn j*)
         eta[i][j] IS_A real_constant;
      END FOR;
   END FOR;
   production[species] IS_A molar_rate;
   rate[rxns] IS_A molar_rate; (* mole rxn j/time *)
   FOR i IN species CREATE
   gen_eqn[i]: production[i] =
         SUM[eta[i][j]*rate[j] | j IN reactions[i]];
   END FOR;
END set_example;
```

"|" is shorthand for SUCH_THAT.

The array eta has only 8 elements, and we defined those elements in a set for each reaction. The equation needs to know about the set of reactions for a species i, and that set is calculated automatically in the model's first FOR/CREATE statement.

|

The | symbol is the ASCEND III notation for SUCH_THAT. We noted that "|" is often read as "for all", which is different in that "for all" makes one think of a FOR loop where the loop index is on the left of an IN operator. For example, the j loop in the SUM of gen_eqn[i] above.

19.2.3 CONSTANTS

ASCEND supports real, integer, boolean and character string constants. Constants in ASCEND do not have any attributes other than their value. Constants are scalar quantities that can be assigned exactly once. Constants may only be assigned using the :== operator and the RHS expression they are assigned from must itself be constant. Constants do not have subparts. Integer and symbol constants may be used in determining the definitions of sets.

Explicit refinements of the built-in constant types may be defined as exemplified in the description of real_constant. Implicit type refinements may be done by instantiating an incompletely defined constant and assigning its final value.

Sets could be considered constant because they are assigned only once, however sets are described separately because they are not quite scalar quantities.

`real_constant` Real number with dimensionality. Note that the dimensionality of a real constant can be specified via the type definition without immediately defining the value, as in the following pair of definitions.

CONSTANT declaration example:

```
CONSTANT molar_weight REFINES real_constant DIMENSION
M/Q;
CONSTANT hydrogen_weight REFINES molar_weight :==
1.004{g/mole};
```

`integer_constant` Integer number. Principally used in determining model structure. If appearing in equations, integers are evaluated as dimensionless reals. Typical use is inside a MODEL definition and looks like:

```
n_trays IS_A integer_constant;
n_trays :== 50;
tray[1..n_trays] IS_A vl_equilibrium_tray;
```

`symbol_constant` Object with a symbol value. May be used in determining model structure.

`boolean_constant` Logical value. May be used in determining model structure.

Setting constants

:==

Constant and set assignment operator.

It is suggested, but not required, that names of all types that refine the built-in constant types have names that end in _constant.

```
LHS_list :== RHS;
```

Here it is required that the one or more items in the LHS be of the same constant type and that RHS is a single-valued expression made up of values, operators, and other constants. The :== is used to make clear to both the user and the system what scalar objects are constants.

19.2.4 VARIABLES

There are four built-in types which may be used to construct variables: symbol, boolean, integer, and real. At this time symbol types have special restrictions. Refinements of these variable base types are defined with the ATOM statement. Atom types may declare attribute fields with types real, integer, boolean, symbol, and set. These attributes are NOT independent objects and therefore cannot be refined, merged, or put in a refinement clique (ARE_ALIKEd).

ATOM

The syntax for declaring a new atom type is

```
ATOM atom_type_name REFINES variable_type
   «DIMENSION dimension_expression»
   «DEFAULT value»; (* note the ; *)
   «initial attribute assignment;»
END atom_type_name;
```

DEFAULT, DIMENSION, and DIMENSIONLESS

The DIMENSION attribute is for variables whose base type is real. It is an optional field. If not defined for any atom with base type 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 possible.

solver_var is a special case of ATOM and we will say much more about it in Section 19.6.1.

```
ATOM solver_var REFINES real DEFAULT 0.5 {?};
   lower_bound IS_A real;
   upper_bound IS_A real;
   nominal     IS_A real;
   fixed       IS_A boolean;
   fixed := FALSE;
   lower_bound := -1e20 {?};
   upper_bound := 1e20 {?};
   nominal := 0.5 {?};
END solver_var;
```

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. In the `solver_var` example, we see a DEFAULT value of 0.5 with the unspecified unit {?} which leaves the dimensionality wild.

`real` Real valued variable quantity. At present, all variables that you want to be attended to by solver tools must be refinements of the type solver_var. This is so that modifiable parametric values can be included in equations without treating them as variables. Strictly speaking, this is a characteristic of the solver interface and not the ASCEND language. Each tool in the total ASCEND system may have its own semantics that go beyond the ASCEND object definition language.

`integer` Integer valued variable quantity. We find these mighty convenient for use in certain procedural computations and as attributes of solver_var atoms.

`boolean` Truth valued variable quantity. These are principally used as flags on solver_vars and relations. They can also be used procedurally and as variables in logical programming models, subject to the logical solver tool’s semantics. (Compare `solver_boolean` and `boolean_var` in Section 19.6.)

`symbol` *4* Symbol valued variable quantity. We do not yet have operators for building symbols out of other symbols.

Setting variables

`:=` Procedural equals differs from the ordinary equals (=) in that it means the left-hand-side (LHS) variables are to be assigned the value of the right-hand-side (RHS) expression when this statement is processed. Processing happens in the last phase of compiling (INSTANTIATION on page 177) or when executing a method interactively through the ASCEND user interface. The order the system encounters these statements matters, therefore, with a later result overwriting an earlier one if both statements have the same the same LHS variable.

Note that variable assignments (also known as “defaulting statements”) written in the declarative section are executed only after an instance has been fully created. This is a frequent source of confusion and errors, therefore we recommend that you DO NOT ASSIGN VARIABLES IN THE DECLARATIVE SECTION.

Note that := IS NOT =.

We use an ordinary equals (=) when defining a real valued equation to state that the LHS expression is to equal the RHS expression at the solution for the model. We use == for logical equations.

<u>Tabular assignments</u>

(* 4+ *) Assigning values en masse to arrays of variables that are defined associatively on sets without order presents a minor challenge. The solution proposed in ASCEND IV (but not yet implemented as we’ve not had time or significant user demand) is to allow a tabular data statement to be used to assign the elements of arrays of variables or constants. The DATA statement may be used to assign variables in the declarative or methods section of a model (though we discourage its use declaratively for variable initialization) or to assign constant arrays of any type, including sets, in the declarative section. Here are some examples:

DATA (* 4+ *)

```
MODEL tabular_ex;
lset,rset,cset IS_A set OF integer_constant;
rset :== [1..3];
cset :== rset - [2];
lset :== [5,7];
a[rset][cset] IS_A real;
b[lset][cset][rset] IS_A real_constant;

(* rectangle table *)
DATA FOR a:
COLUMNS 1,3; (*order last subscript cset*)
UNITS {kg/s}, {s}; (* columnar units *)
(* give leading subscripts *)
[1]   2.8,    0.3;
[2]   2.7,    1.3;
[3]   3.3,    0.6;
END DATA;

(* 2 layer rectangle table *)
CONSTANT DATA FOR b:
COLUMNS 1..3; (* order last subscript rset *)
(* UNITS omitted, so either the user gives value in the
table or values given are DIMENSIONLESS. *)
(* ordering over [lset][cset] required *)
[5][1] 3 {m}, 2{m}, 1{m};
```

```
[5][3] 0.1, 0.2, 0.3;
[7][1] -3 {m/s}, -2{m/s}, -1{m/s};
[7][3] 4.1 {1/s}, 4.2 {1/s}, 4.3 {1/s};
END DATA;
```

END tabular_ex;

For sparse arrays of variables or constants, the COLUMNS and (possibly) UNITS keywords are omitted and the array subscripts are simply enumerated along with the values to be assigned.

19.2.5 RELATIONS

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 additive 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 a set 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);

Numerical relations The syntax for a numeric relation is either

```
optional_label: LHS relational_operator RHS;
or
optional_label: objective_type LHS;
```

`Objective_type` is either `MAXIMIZE` or `MINIMIZE`. RHS and LHS must be one or more variables, constants, and operators in a normal algebraic expression. The operators allowed are defined below and in Section 19.6.3. Variable integers, booleans, and symbols are not allowed as operands in numerical relations, nor are boolean constants. Integer indices declared in FOR/CREATE loops are allowed in relations, and they are treated as integer constants.

Relational operators:

=, >=, <=, <, >, <> These are the numerical relational operators for declarative use.

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

Equations must be dimensionally correct.

MAXIMIZE, MINIMIZE Objective function indicators.

Binary Operators: +, -, *, /, ^. We follow the usual algebraic order of operations for binary operators.

+ Plus. Numerical addition or set union.

- Minus. Numerical subtraction or set difference.

***** Times. Numerical multiplication or set intersection.

/ Divide. Numeric division. In most cases it implies real division and not integer division.

^ Power. Numeric exponentiation. If the value of y in x^y is not integer, then x must be greater than 0.0 and dimensionless.

Unary Operators: -, *ordered_function*()

- Unary minus. Numeric negation. There is no unary + operator.

***ordered_function*()** unary real valued functions. The unary real functions we support are given in section Section 19.6.3.

Real functions of sets of real terms:

SUM[term set] Add all expressions in the function’s list.

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

An examples of the use is:

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

or, equivalently, one could write:

```
SUM[y[i] | i IN components] = 1;
```

Empty SUM[] yields wild 0.

When a SUM is compiled over a list which is empty it generates a wild dimensioned 0. This will sometimes cause our dimension checking routines to fail. The best way to prevent this is to make sure the SUM never actually encounters an empty list. For example:

```
SUM[Q[possibly_empty_set], 0{watt}];
```

In the above, the variables `Q[i]` (if they exist) have the dimensionality associated with an energy rate. When the set is empty, the 0 is the only term in the SUM and establishes the dimensionality of the result. When the set is NOT empty the compiler will simplify away the *trailing* 0 in the sum.

`PROD[term set]`

Multiply all the expressions in the product's list. The product of an empty list is a dimensionless value, 1.0.

Possible future functions:

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

`MAX[term set]` (* 4+ *) maximum value on list of arguments

`MIN[term set]` (* 4+ *) minimum value on list of arguments

19.2.6 DERIVATIVES IN RELATIONS (* 4+ *)

Simply put, we would like to have general partial and full derivatives usable in writing equations, as there are many mathematically interesting things that can be said about both. We have not implemented such things yet for lack of time and because with several implementations (see gPROMS and OMOLA, among others) already out there we can't see too many research points to be gained by more derivative work.

19.2.7 EXTERNAL RELATIONS

We cannot document these at the present time. The only reference for them is [[abbottthesis]].

19.2.8 CONDITIONAL RELATIONS (* 4 *)

The syntax is CONDITIONAL list_of_relation_statements END CONDITIONAL;

A CONDITIONAL statement can appear anywhere in the declarative portion of the model and it contains only relations to be used as boundaries. That is, these real arithmetic equations are used in expressing logical condition equations via the SATISFIED operator. See LOGICAL FUNCTIONS on page 215.

19.2.9 LOGICAL RELATIONS (* 4 *)

Logical expression

An expression whose value is TRUE or FALSE is a logical expression. Such expressions may contain boolean variables. If `A,B`, and `laminar` are `boolean`, then the following is a logical expression:

```
A + (B * laminar)
```

as is (and probably more clearly)

A OR (B AND 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 = FALSE, B=TRUE* 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.

Logical relations are then made by putting together logical expressions with the boolean relational operators == and !=. Since we have no logical solving engine we have not pushed the implementation of logical relations very hard yet.

19.2.10 NOTES (* 4 *)

Within a MODEL(or METHOD) definition annotations (hereafter called notes) may be made on a part declared in the MODEL, or on the MODEL (or METHOD) itself. Short notes may be made when defining or refining an object by enclosing the note in "double quotes." Longer notes may be made in a block statement.

Each note is entered in a database with the name of the file, name of MODEL, name of METHOD if applicable, and the language (a kind of keyword) in which the note is written. Users, user interfaces, and other programs may query this database for information on models and simulations. The block notes may include code fragments in other languages that you wish to embed in your MODEL or any other kind of text.

Short notes should be included as you write any model to clarify the roles of parts and variables. All short notes have the language 'inline.' Here are some examples of short notes:

```
L[1..10] "L[i] is the length of the ith rod"
   IS_A distance;
thetaM "angle between horizon and moon",
thetaJ "angle between horizon and jupiter"
   IS_A angle;
car.tires "using car in Minnesota, you betcha"
   IS_REFINED_TO snow_tire;
```

In the second IS_A statement concerning two angles, we see that a short note in double quotes goes with the name immediately to its left. We also see that the note comes before the comma if the name is part of a list of names. In the third statement, we see that not only simple names but also qualified names may be annotated.

Longer notes are made in block statements of the form below. These blocks can appear in a METHOD or MODEL. These blocks can also be written separately before or after a model as we shall see.

```
NOTES
'language or keyword' list.of, names {
free-form block of text to store in the database
exactly as written.
}
some.other.name {
  this note has the same language or keyword as the
first since we didn't define a new keyword in single
quotes before the name list.
}
'another language' some.other.name {
  en espanol
}
'fortran' SELF {
This model should be solved with subroutine LSODE.
This note demonstrates that "SELF" can be used to
annotate the entire model instead of a named part.
}
END NOTES;
```

Notes made outside the scope of a model definition look like one of the following:

```
ADD NOTES IN name_of_model;
'language or keyword' list.of, names {
  more text
} (* more than one note may be made in this block if
desired. *)
END NOTES;
ADD NOTES IN name_of_model METHOD name_of_method;
'language or keyword'  SELF {
This method proves Fermat's last theorem and makes
toast.
}
'humor' SELF {
ASCEND is not expected to make either proving FLT or
toasting possible.
}
END NOTES;
```

We can add notes to the database before or after defining the annotated model. This is handy for several reasons including:

- Lengthy notes mixed with model and method code can make that code very hard to read.
- Separate notes describing a family of models can be loaded and browsed before loading that library family.
- Users other than the author of a model can annotate that model without fear of introducing typographical errors into the model.

These advantages come with a disadvantage that all documentation has. If you change the model, you ought to change the documentation at the same time. To make finding these documentation locations in need of change easier, the name of the file containing each note is included in the loaded database.

Experience has shown that even documentation embedded directly in models or in other computer programs gets out-dated if the person changing the program is in a hurry and is not required to document properly as part of the task at hand. Neither ASCEND nor any other software system can eliminate the garbage code and documentation that results from undisciplined modeling.

19.3 DECLARATIVE STATEMENTS

We have already seen several examples that included declarative statements. Here we will be more systematic in defining things. The statements we describe are legal within the declarative portion of an ATOM or MODEL definition. The declarative portion stops at the keyword METHODS if it is present in the definition or at the end of the definition.

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.

Compound statements

Some statements in ASCEND can contain other statements as a part of them. The declarative compound statements are the ALIASES/ IS_A, CONDITIONAL, FOR/CREATE, SELECT/CASE, and WHEN/CASE statements. The procedural compound statements allowed only in methods are the FOR/DO, FOR/CHECK, SWITCH (* 4 *) and the IF statements. Compound statements end with "`END` *`word`*`;`", where *`word`* matches the beginning of the syntax block, e.g. `END FOR.` and they can be nested, with some exceptions which are noted later.

CASE statements are here, finally!

(*4*) WHEN/CASE, CONDITIONAL, and SELECT/CASE handle modeling alternatives within a single definition. The easy way to remember the difference is that the first picks which equations to solve WHEN discrete *variables* have certain values, while the second SELECTs which statements to compile based on discrete *constants*. (* 4 *) SWITCH statements handle flow of control in methods, in a slightly more generalized form than the C language switch statement.

Type declarations are not compound statements.

MODEL and ATOM type definitions and METHOD definitions are not really compound statements because they require a name following their END word that matches the name given at the beginning of the definition. These definitions cannot be nested.

ASCEND operator synopses:	We’ll start with an extremely brief synopsis of what each does and then give detailed descriptions. It is helpful to remember that an instance may have many names, even in the same scope, but each name may only be defined once.
IS_A	Constructor. Calls for one or more named instances to be compiled using the type specified. (* 4 *) If the type is one that requires parameters, the parameters must be supplied in () following the type name.
IS_REFINED_TO	Reconstructor. Causes the already compiled instance(s) named to have their type changed to a more refined type. This causes an incremental recompilation of the instance(s). IS_REFINED_TO is not a redefinition of the named instances because refinement can only *add* compatible information. The instances retain all the structure that originally defined them. (* 4 *) If the type being refined to requires arguments, these must be supplied, even if the same arguments were required in the IS_A of the originally less refined declaration of the instance.
ALIASES (* 4 *)	Part alternate naming statement. Establishes another name for an instance at the same scope or in a child instance. The equivalent of an ALIASES in ASCEND III is to create another part with the desired name and merge it immediately via ARE_THE_SAME with the part being renamed, a rather expensive and unintuitive process.
ALIASES/IS_A (*4*)	Creates an array of alternate names for a list of existing instances with some common base type and creates the set over which the elements of the array are indexed. Useful for making collections of related objects in ways the original author of the model didn’t anticipate. Also useful for assembling array arguments to parameterized type definitions.
WILL_BE (* 4 *)	Forward declaration statement. Promises that a part with the given type will be constructed by an as yet unknown IS_A statement above the current scope. At present WILL_BE is legal only in defining parameters. Were it legal in the body of a model, compiling models would be very expensive.
ARE_THE_SAME	Merge. Calls for two or more instances already compiled to be merged recursively. This essentially means combining all the values in the instances into the most refined of the instances and then destroying all the extra, possibly less refined, instances. The remaining instance has its original name and also all the names of the instances destroyed during the merge.

WILL_BE_THE_SAME (* 4 *)	Structural condition statement restricting objects in a forward declaration. The objects passed to a parameterized type definition can be constrained to have arbitrary parts in common before the parameterized object is constructed.
WILL_NOT_BE_THE_SAME (* 4 *)	Structural condition statement restricting objects in a forward declaration. We apologize for the length of this key word, but we bet it is easy to remember. The objects passed to a parameterized type definition can be constrained to have arbitrary parts be distinct instances before the parameterized object is constructed. At present the constraint is only enforced when the objects are being passed.
ARE_NOT_THE_SAME (* 4+ *)	Cannot be merged. We believe it is useful to say that two objects cannot be merged and still represent a valid model. This is not yet implemented, however, mainly for lack of time. The implementation is simple.
ARE_ALIKE	Refinement clique constructor. Causes a group of instances to always be of the same formal type. Refining one of them causes a refinement of all the others. Does not propagate *implicit* type information, such as assignments to constants or part refinements made from a scope other than the scope of the formal definition.
FOR/CREATE	Indexed execution of other declarative statements. Required for creating arrays of relations and sparse arrays of other types.
FOR/CHECK	Indexed checking of the conditions (WHERE statements) of a parameterized model.
SELECT/CASE (*4*)	Select a subset of statements to compile. Given the values of the specified *constants*, SELECT compiles all cases that match those values. A name cannot be defined two different ways inside the SELECT statement, but it may be defined outside the case statement and then *refined* in different ways in separate cases.
CONDITIONAL (*4*)	Describe bounding relations. The relations written inside a CONDITIONAL statement must all be labelled. These relations can be used to define regions in which alternate sets of equations apply using the WHEN statement.
WHEN/CASE (* 4 *)	When logical *variables* have certain values, use certain relations or model parts in defining a mathematical problem. The relations are not defined inside the WHEN statement because all the relations must be compiled regardless of which values the logical variables have at any given moment.

Reminder: In the following detailed 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.

Operators in detail:

IS_A

This statement has the syntax

list_of_instance_names IS_A
model_name «(arguments_if_needed)»;

The IS_A statement allows us to declare *instances* of a given *type* to exist within a model definition. If *type* has not been defined (loaded in the ASCEND environment) then this statement is an error and the MODEL it appears in is irreparably damaged (at least until you delete the type definitions and reload a corrected file). Similarly, if the arguments needed are not supplied or if provably incorrect arguments are supplied, the statement is in error. The construction of the instances does not occur until all the arguments satisfy the definition of *type*.

If a name is used twice in WILL_BE/IS_A/ALIASES statements of the same model, ASCEND will complain bitterly when the definition is parsed. Duplicate naming is a serious error. Labels on relations share the same name space as other objects.

IS_REFINED_TO

This statement has the syntax

```
list_of_instances IS_REFINED_TO
type_name «(arguments_if_needed)»;
```

We use this statement to change the type of each of the instances listed to the type *type_name*. The modeler has to have defined each member on the list of instances. The *type_name* has to be a type which refines the types of all the instances on the list.

An example of its use is as follows. First we define 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 defined model definitions the type adiabatic_flash that is a refinement of flash. Then we can make fl1 and fl3 into more refined types by stating:

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

This reconstruction does not occur until the arguments to the type satisfy the definition *type_name*.

ALIASES (* 4 *)

This statement has the syntax

```
list_of_instances ALIASES instance_name;
```

We use this statement to point at an already existing instance of any type other than `relation`, `logical_relation`, or `when`. For example, say we want a flash tank model to have a variable T, the temperature of the vapor-liquid equilibrium mixture in the tank.

```
MODEL tank;
   feed, liquid, vapor IS_A stream;
   state IS_A VLE_mixture;
   T ALIASES state.T;
   liquor_temperature ALIASES T;
END tank;
```

We might also want a more descriptive name than T, so ALIASES can also be used to establish a second name at the same scope, e.g. `liquor_temperature`.

An ALIASES statement will not be executed until the RHS instance has been created with an IS_A. The compiler schedules ALIASES instructions appropriately and issues warnings if recursion is detected. An array of aliases, e.g.

```
b[1..n], c ALIASES a;
```

is permitted (though we can’t think why anyone would want such an array), and the sets over which the array is defined must be completed before the statement is executed. So, in the example of b and c, the array b will not be created until a exists and n is assigned a value. b and c will be created at the same time since they are defined in the same statement. This suggests the following rule: if you must use an array of aliases, do not declare it in the same statement with a scalar alias.

The ALIASES RHS can be an element or portion of a larger array with the following exception. The existing RHS instance cannot be a relation or array of relations (including logical relations and whens) because of the rule in the language that a relation instance is associated with exactly one model.

ALIASES/IS_A (*4*)

The ALIASES/IS_A statement syntax is subject to change, though some equivalent will always exist. We take a set of `symbol_constant` or `integer_constant` and pair it with a list of instances to create an array. For the moment, the syntax and semantics is as follows.

alias_array_instance[aset] ALIASES (*list_of_instances*) WHERE *aset* IS_A set OF *settype*;

or

alias_array_instance[aset] ALIASES (*list_of_instances*) WHERE *aset* IS_A set OF *settype* WITH_VALUE (*value_list_matching_settype*);

`aset` is the name of the set that will be created by the IS_A to index the array of aliases. If `value_list_matching_set_type` is not given, the compiler will make one up out of the integers (1..number of names in `list_of_instances`) or symbols derived from the individual names given. If the value list is given, it must have the same number of elements as the list of instances does. The value list elements must be unique because they form a set. The list of instances can contain duplicates. If any of these conditions are not met properly, the statement is in error.

ALIASES/IS_A can be used inside a FOR statement. When this occurs, the definition of `aset` must be indexed and it must be the last subscript of `alias_array_instance`. The statement must look like:

array_instance[FOR_index][aset[FORindex]] ALIASES (*list_of_instances*) WHERE *aset[FORindex]* IS_A set OF *settype* WITH_VALUE (*value_list_matching_settype*);

Here, as with the unindexed version, the WITH_VALUE portion is optional.

If this explanation is unclear, just try it out. The compiler error messages for ALIASES/IS_A are particularly good because we know it is a bit tricky to explain.

WILL_BE (* 4 *)

instance WILL_BE *type_name*;

The most common use of this forward declaration is as a statement within the parameter list of a model definition. In parameter lists, *list_of_instances* must contain exactly one instance. When a model definition includes a parameter defined by WILL_BE, that model cannot be compiled until a compiled instance at least as refined as the type specified by *type_name* is passed to it.

(* 4+ *) The second potential use of WILL_BE is to establish that an array of a common base type exists and its elements will be filled in individually by IS_A or ARE_THE_SAME or ALIASES statements. WILL_BE allows us to avoid costly reconstruction or merge operations by establishing a placeholder instance which contains just enough type information to let us check the validity of other statements that require type compatibility while delaying construction until it is called for by the filling in statements. Instances declared with WILL_BE are never compiled if they are not ultimately resolved to another instance created with IS_A. Unresolved WILL_BE instances will appear in the user interface as objects of type PENDING_INSTANCE_*model_nam*e. Because of the many implementation and explanation difficulties this usage of WILL_BE creates, it is not allowed. The ALIASES/IS_A construct does the same job in a much simpler way.

ARE_THE_SAME

The format for this instruction is

list_of_instances 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 this clique when one asks via the interface 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.

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

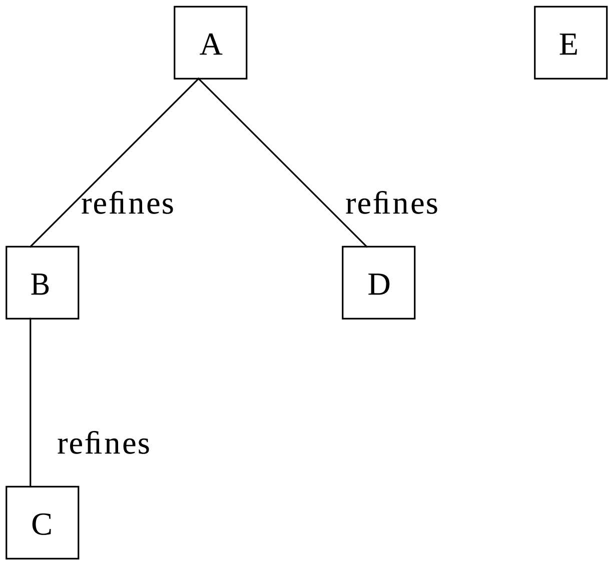

Figure 1. Diagram of the model type hierarchy A,B,C,D,E

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, an array, or an atom (i.e., a variable, constant, or set). If a model, only one member of the clique generates its equations. If a variable, it assigns all members to the same storage location.

Note that the values of constants and sets are essentially *type* information, so merging two already assigned constants is only possible if merging them does not force one of them to be assigned a new value. Merging arrays with mismatching ranges of elements is an error.

WILL_BE_THE_SAME (* 4 *)

There is no further explanation of WILL_BE_THE_SAME.

WILL_NOT_BE_THE_SAME (* 4 *)

There is no further explanation of WILL_NOT_BE_THE_SAME.

ARE_NOT_THE_SAME (* 4+ *)

ARE_NOT_THE_SAME will be documented further when it is implemented.

ARE_ALIKE

The format for this statement is

```
list_of_instance_names ARE_ALIKE;
```

The compiler places all instances in the list into an ARE_ALIKE clique. It checks that the members are formally type compatible and then it converts each into the most refined type of any instance in the clique. At that point the compiler stops. It does not continue by placing the parts into cliques nor does it merge anything.

There are important consequences of modeling with such a partial merge. The consequences we are about to describe can be much more reliably achieved by use of parameterized types, *when the types are well understood*. When we are exploring new ways of modeling, ARE_ALIKE still has its uses. When a model and its initial uses are understood well enough to be put into a reusable library, then parameterization and the explicit statement of structural constraints by operators such as WILL_NOT_BE_THE_SAME should be the preferred method of ensuring correct use.

One consequence of ARE_ALIKE is to prevent extreme model misuse when configuring models. For example, suppose a modeler creates a new pressure changing model. The modeler is not yet 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 can detect no such incompatibility unless parameterized models are used.

Another purpose is the propagation of types through a model. 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. This is *the primary example* which has been used to justify the existence of ARE_ALIKE. We have observed that its use makes a column library very difficult to compile efficiently. But since we now have parameterized types to help us keep the column library semantically consistent, ARE_ALIKE can be left to its proper role: the rapid prototyping of partially understood models. We have yet to see anyone use ARE_ALIKE in a prototyping context, however.

Finally, because ARE_ALIKE does not recursively put the parts of ARE_ALIKEd instances into ARE_ALIKE cliques, it is possible to ARE_ALIKE model instances which have compatible formal types but incompatible *implicit* types. This can lead to unexpected problems later and makes the ARE_ALIKE instruction a source of non-reusability.

FOR/CREATE

The FOR/CREATE statement is a compound statement that looks like a loop. It isn't, however, necessarily compiled as a loop. What FOR really does is specify an index set value. Its format is:

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

This statement can be in the declarative part of the model definition only. Every statement in the list should have at least one occurrence of the index variable, or the statement should be moved outside the FOR to avoid redundant execution. A correct example is

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

FOR loops can be nested to produce sparse arrays as illustrated in ARRAYS CAN BE JAGGED on page 173. IS_A and ALIASES statements are allowed in FOR loops, provided the statements are properly indexed, a new feature in ASCEND IV.

SELECT/CASE (*4*) Declarative. Order does not matter. All matching cases are executed. The OTHERWISE is executed if present and no other CASEs match. SELECT is not allowed inside FOR. Writing FOR statements inside SELECT is allowed.

CONDITIONAL (*4*) Both real and logical relations are allowed in CONDITIONAL statements. CONDITIONAL is really just a shorthand for setting the $boundary flag on a whole batch of relations, since $boundary is a write-once attribute invisible through the user interface and methods at this time.

WHEN/CASE (* 4 *) Inside each CASE, relations or model parts to be used are specified by writing, for example, USE mass_balance_1;. The method of dealing with the combined logical/nonlinear model is left to the solver. All matching CASEs are included in the problem to be solved.

19.4 PROCEDURAL STATEMENTS

METHODS This statement separates the method definitions in ASCEND from the declarative statements. All statements following this statement are to define methods in ASCEND while all before it are for the declarative part of ASCEND. The syntax for this statement is simply

METHODS

with no punctuation. The next code must be a METHOD or the END of the type being defined. If there are no method definitions, this statement may be omitted.

METHOD definitions for a type can also be added or replaced after the type has been defined. This is to make creating and debugging of methods as interactive as possible. In ASCEND III an instance must be destroyed and recreated each time a new or revised method is added to the type definition. This is a very expensive process when working with models of significant size.

The detailed semantics of method inheritance, addition, and replacement of methods are given at the end of this section.

`ADD METHODS IN type_name; (*4*)`

This statement allows new methods to be added to an already loaded type definition. The next code must be a METHOD or the END METHODS; statement. If a method of the same name already exists in `type_name`, the statement is in error. If other types refine `type_name` then the addition follows the method inheritance rules. Any type which inherited methods from `type_name` now inherits the methods added to `type_name`. If a refinement of `type_name` already defines a method ADDed to `type_name`, then the existing method in the more refined type is not disturbed.

`REPLACE METHODS IN type_name; (*4*)`

This statement allows existing methods to be replaced in an already loaded type definition. The next code must be a METHOD or the END METHODS; statement. If a method of the same name does not exist in `type_name`, the statement is in error. If other types refine `type_name` then the replacement follows the method inheritance rules. Any type which inherited the old method now inherits the replacment method instead.

`ADD METHODS IN DEFINITION MODEL;`

This statement allows methods to be added globally. It should be used very sparingly. Library basemodel.a4l contains the example of this statement. Methods in the global model definition are inherited by all models. There is no actual global model definition, but it has a method list for practical purposes.

Initialization routines:

`METHOD`

A method in ASCEND must appear following the METHODS statement within a model. The system executes procedural statements of the method in the order they are written.

At present, there are no local variables or other structures in methods except loop indices. A method may be written recursively, but there is an arbitrary stack depth limit (currently set to 20 in compiler/initialize.h) to prevent the system from crashing on infinite recursions.

Specifically disallowed in ASCEND III methods are IS_A, ALIASES, WILL_BE, IS, IS_REFINED_TO, ARE_THE_SAME and ARE_ALIKE statements as these “declare” the structure of the model and belong only in the declarative section.

(* 4+ *) In the near future, declarations of local instances (which are automatically destroyed when the method exits) will be allowed. Since methods are imperative, these local structure definitions are processed in the order they are written. Local structures are not allowed to shadow structures in the model context with which the method is called. When local structures are allowed, it will also be possible to define methods which take parameters and return values, thereby making the imperative ASCEND methods a rapid prototyping tool every bit as powerful and easy to use as the declarative ASCEND language.

The syntax for a method declaration is

```
METHOD method_name;
   «procedural statement;» (*one or more*)
END method_name;
```

Procedural assignment

The syntax is

```
instance_name := mathematical_expression;
or
array_name[set_name] := expression;
or
list_of_instance_names := expression.
```

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

DATA statements (DATA (* 4+ *) on page 187) can (should, rather) also appear in methods.

FOR/DO statement

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

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

```
END FOR;
```

Here we actually execute using the values of i *in the sequence given.* So,

```
FOR i IN [n_stages..1] DO ... END FOR;
```

is an empty loop, while

```
FOR i IN [n_stages..1] DECREASING DO ... END FOR;
```

is a backward loop.

IF

The IF statement can only appear in a method definition. Its syntax is

```
IF logical_expression THEN
   list_of_statements
ELSE
   list_of_statements
END IF;
```

or

```
IF logical_expression THEN
   list_of_statements
END IF;
```

If the logical expression has a value of TRUE, ASCEND will execute the statements in the THEN part. If the value is FALSE, ASCEND executes the statements in the optional ELSE part. Please use () to make the precedence of AND, OR, NOT, ==, and != clear to both the user and the system.

SWITCH (* 4 *)

Essentially roughly equivalent to the C switch statement, except that ASCEND allows wildcard matches, allows any number of controlling variables to be given in a list, and assumes BREAK at the end of each CASE.

CALL

External calls are not presently well defined, pending debugging of the EXTERNAL connection prototype originally created by Kirk Abbott.

RUN

This statement can appear only in a method. Its format is:

```
RUN name_of_method;
```

or
RUN *part_name.name_of_method*;
or
RUN *model_type*::*name_of_method*;

The named method can be defined in the current model (the first syntax), or in any of its parts (the second syntax). Methods defined in a part will be run in the scope of that part, not at the scope of the RUN statement.

Type access to methods: When *model_type*:: appears, the type named must be a type that the current model is refined from. In this way, methods may be defined incrementally. For example:

```
MODEL foo;
   x IS_A generic_real;
METHODS
METHOD specify;
   x.fixed:= TRUE;
END specify;
END foo;

MODEL bar REFINES foo;
   y IS_A generic_real;
METHODS
METHOD specify;
   RUN foo::specify;
   y.fixed := TRUE;
END specify;
END bar;
```

19.5 PARAMETERIZED MODELS

Parameterized model definitions have the following general form.

```
MODEL new_type(parameter_list;)
«WHERE (where_list;)»
«REFINES existing_type «(assignment_list;)»»;
```

19.5.1 THE PARAMETER LIST

A parameter list is a list of statements about the objects that will be passed into the model being defined when an instance of that model is created by IS_A or IS_REFINED_TO. The parameter list is designed to allow a complete statement of the necessary and sufficient conditions to construct the parameterized model. The mechanism implemented is general, however, so it is possible to put less than the necessary information in the parameter list if one seeks to confuse the model's reusers. To make parameters easy to understand for users with experience in other computer languages (and to make the implementation much simpler), we define the parameter list as ordered. All the statements in a parameter list, including the last one, must end with a ";". A parameter list looks like:

```
MODEL test (
   x WILL_BE real;
   n IS_A integer_constant;
   p[1..n] IS_A integer_constant;
   q[0..2*n-1] WILL_BE widget;
);
```

Each WILL_BE statement corresponds to a single object that the user must create and pass into the definition of `test`. We will establish the local name `x` for the first object passed to the definition of `test`. `n` is handled similarly, and it must preceed the definition of `p[1..n]`, because it defines the set for the array `p`. Constant types can also be defined with WILL_BE, though we have used IS_A for the example `test`.

Each IS_A statement corresponds to a single constant-valued instance or an array of constant-valued instances that we will create as part of the model we are defining. Thus, the user of `test` must supply an array of constants as the third argument. We will check that the instance supplied is subscripted on the set [1..n] and copy the corresponding values to the array p we create local to the instance of `test`.

WILL_BE statements can be used to pass complex objects (models) or arrays of objects. Both WILL_BE and IS_A statements can be passed arguments that are *more* refined than the type listed. If an object that is *less* refined than the type listed, the instance of parameterized model `test` will not be compiled. When a parameterized model type is specified with a WILL_BE statement, NO arguments should be given. We are only interested in the formal type of the argument, not how it was constructed.

19.5.2 THE WHERE LIST

We can write structural and equation constraints on the arguments in the WHERE list. Each statement is a WILL_BE_THE_SAME, a WILL_NOT_BE_THE_SAME, an equation written in terms of sets or discrete constants, or a FOR/CHECK statement surrounding a group of such statements. Until all the conditions in the WHERE list are satisfied, an object cannot be constructed using the parameterized definition. If the arguments given to a parameterized type in an IS_A or IS_REFINED_TO statement cannot possibly satisfy the conditions, the IS_A or IS_REFINED_TO statement is abandoned by the compiler.

We have not created a WILL_BE_ALIKE statement because formal type compatibility in ASCEND is not really a meaningful guarantee of object compatibility. Object compatibility is much more reliably guaranteed by checking conditions on the structure determining constants of a model instance.

19.5.3 THE ASSIGNMENT LIST

When we declare constant parameters with IS_A, we can in a later refinement of the parameterized model assign their values in the assignment list, thus removing them from the parameter list. If an array of constants is declared with IS_A, then we must assign values to ALL the array elements at the same time if we are going to remove them from the parameter list. If an array element is left out, the type which assigns some of the elements and any subsequent refinements of that type will not be compilable.

19.5.4 REFINING PARAMETERIZED TYPES

Because we wish to make the parameterized model lists represent all the parameters and conditions necessary to use a model of any type, we must repeat the parameters declared in the ancestral type when we make a refinement. If we did not repeat the parameters, the user would be forced to hunt up the (possibly long) chain of types that yield an interesting definition in order to know the list of parameters and conditions that must be satisfied in order to use a model. We repeat all the parameters of the type being refined before we add new ones. The only exception to this is that parameters defined with IS_A and then assigned in the `assignment_list` are not repeated because the user no longer needs to supply these values. A refinement of the model `test` given in Section 19.5.1 follows.

```
MODEL expanded_test (
   x WILL_BE real;
   p[1..n] IS_A integer_constant;
   q[0..2*n-1] WILL_BE better_widget;
   r[0..q[0].k] WILL_BE gizmo;
   ms WILL_BE set OF symbol_constant;
) WHERE (
   q[0].k >= 2;
   r[0..q[0].k].giz_part WILL_BE_THE_SAME;
) REFINES test(
   n :== 4;
);
```

In `expanded_test`, we see that the type of the array `q` is more refined than it was in `test`. We see that constants and sets from inside passed objects, such as `q[0].k`, can be used to set the sizes of subseqent array arguments. We see a structural constraint that all the `gizmo`s in the array `r` must have been constructed with the same `giz_part`. This condition probably indicates that the gizmo definition takes `giz_part` as a WILL_BE defined parameter.

19.6 MISCELLANY

19.6.1 VARIABLES FOR SOLVERS

solver_var

Solver_var is the base-type for all *computable* variables in the current ASCEND system. Any instances of an atom definition that refines solver_var are considered potential variables when constructing a problem for one of the solvers.

Solver_var 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.) In system.a4l we define the following parts with associated initial values for each:

Attributes:	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. All atoms which are refinements of solver_var will have these parts. The refining definitions may reassign the default values of the attributes.

The latest full definition of solver_var is always in the file system.a4l.

`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 the first refinement of solver_var and is also defined in system.a4l.

Kluges for MILPs

Also defined in system.a4l are the types for integer, binary, and semi-continuous variables.

`solver_semi, solver_integer, solver_binary`

We define basic refinements of solver_var to support solvers which are more than simply algebraic. Various mixed integer-linear program solvers can be fed solver_semi based atoms defining semi-continuous variables, solver_integer based atoms defining integer variables, and solver_binary based atoms defining binary variables.

Integers are relaxable.

All these types have associated boolean flags which indicate that either the variable is to be treated according to its restricted meaning or it is to be relaxed and treated as a normal continuous algebraic variable.

Kluges for ODEs

We have an alternate version of system.a4l called ivpsystem.a4l which adds extra flags to the definition of solver_var in order to support initial value problem (IVP) solvers (integrators). Integration in the ASCEND IV environment is explained in another chapter.

`ivpsystem.a4l`

Having ivpsystem.a4l is a temporary, but highly effective, way to keep people who want to use ASCEND only for algebraic purposes from having to pay for the IVP overhead. Algebraic users load system.a4l. Users who want both algebraic and IVP capability load ivpsystem.a4l instead of system.a4l. This method is temporary because part of the extended definition of ASCEND IV is that differential calculus constructs will be explicitly supported by the compiler. The calculus is not yet implemented, however.

19.6.2 SUPPORTED ATTRIBUTES

`(* 4+ *)` The solver_var, and in fact most objects in ASCEND IV, should have built-in support for (and thereby efficient storage of) quite a few more attributes than are defined above. These built-in attributes are not instances of any sort, merely values. The syntax for naming one of these supported attributes is:
object_name`.`$*supported_attribute_name.*

Supported attributes may have symbol, real, integer, or boolean values. Note that the `$` syntax is essentially the same as the derivative syntax for relations; derivatives are a supported attribute of relations. The supported attributes must be defined at the time the ASCEND compiler is built. The storage requirement for a supported boolean attribute is 1 bit rather than the 24 bytes required to store a run-time defined boolean flag. Similarly, the requirement for a supported real attribute is 4 or 8 bytes instead of 24 bytes.

19.6.3 SINGLE OPERAND REAL FUNCTIONS:

`exp()`	exponential (i.e., exp(x) = e^x)
`ln()`	log to the base e
`sin()`	sine. argument must be an angle.
`cos()`	cosine. argument must be an angle.
`tan()`	tangent. argument must be an angle.
`arcsin()`	inverse sine. return value is an angle.
`arccos()`	inverse cosine. return value is an angle.
`arctan()`	inverse tangent. return value is an angle.
`erf()`	error function (not available from Microsoft Windoze)
`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. This lnm function is parameterized by a constant a, which is typically set to about 1.e-8. lnm(x) is defined as follows:

ln(x) for x > a

(x-a)/a + ln(a) for x <= a.

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. This function and its first derivative are continuous. The second derivative contains a jump at a.

The lnm function can tolerate a negative argument while the ln function cannot. At present the value of a is controllable via the user interface of the ASCEND solvers.

Operand dimensionality must be correct.

The operands for an ASCEND function must be dimensionally consistent with the function in question. Most transcendental functions require dimensionless arguments. The trigonometric functions require arguments with dimensionality of plane angles, P. ASCEND functions return dimensionally correct results.

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

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

<u>Discontinuous functions:</u>

Discontinuous functions may destroy a Newton-based solution algorithm if used in defining a model equation. We strongly suggest considering alternative formulations of your equations.

`abs()` absolute value of argument. Any dimensionality is allowed in an abs() function.

19.6.4 LOGICAL FUNCTIONS

SATISFIED() (*4*) SATISFIED(relation_name,tolerance) returns TRUE if the relation named has a residual value less than the real value, tolerance, given. If the relation named is a logical relation, the tolerance should not be specified, since logical relations evaluate directly to TRUE or FALSE.

19.6.5 UNITS DEFINITIONS

As noted in 19.1.2, ASCEND will recognize conversion factors when it sees them as {units). These units are built up from the basic units, and new units can be defined by the user. Note that the assignment x:= 0.5 {100}; yields x == 50, and that there are no 'offset conversions,' e.g. F=9/5C+32. Please keep unit names to 20 characters or less as this makes life pretty for other users

One or more unit conversion factors can be defined with the UNITS keyword. A unit of measure, once defined, stays in the system until the system is shut down. A measuring unit cannot be defined differently without first shutting down the system, but duplicate or equivalent definitions are quietly ignored.

A UNITS declaration can occur in a file by itself, inside a model or inside an atom. UNITS definitions are parsed immediately, they will be processed even if a surrounding MODEL or ATOM definition is rejected. Because units and dimensionality are designed into the deepest levels of the system, a unit definition must be parsed before any atoms or relations use that definition. It is good design practice to keep customized unit definitions in separate files and REQUIRE those files at the beginning of any file that uses them. Unit definitions are made in the form, for example:

```
UNITS (* several unit definitions could be here. *)
   ohm = {kilogram*meter^2/second^3/ampere^2};
END UNITS;
```

The standard units library, measures.a4l, is documented in Chapter 20.