1.1 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.

ASCEND is beginning its 4th generation. Some features we will describe are not yet implemented and these are clearly marked (* 4+ *) as such. Any feature not so marked 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. We will describe, starting in Section 1.1.2, 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.

1.1.1 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 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.
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 1.5.2. 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, to group terms in complex arithmetic, logical, or set expressions.
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}. 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 number 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.

: Colon. A separator used in various ways, principally to set the name of a 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 1.4.

; Semicolon. The separator of statements.

1.1.2 BASIC ELEMENTS

Boolean value

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.

User interface tip:

The ASCEND user interface programmers have found it very convenient, however, to allow 1/0, Y/N, and other obvious boolean conventions as interactive input when assigning boolean values. We are lazy.

Integer value

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
```

Typically, 2147483647.

```
MAX_INTEGER
```

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* 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
```

Normally MAX_REAL is about 1.79E+308.

```
MAX_REAL
```

while the following are not:

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

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.

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).

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*.

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 the file compiler/units_input as shown in Section 1.5.5. 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.

Units

A unit expression must be enclosed in curly brackets {}. When a real number is used in a mathematical expression in ASCEND, it must have a set of units expressed with it. If it does not, ASCEND assumes the number is dimensionless, which may not be the intent of the modeler. An example is shown 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 denominator (should be written `{m/K/kg_mole}`)
`{m^3.5}` power 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 who?'
```

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 without crashing.

Sets values

Sets 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 1.2.2.

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) 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) it 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. 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 should 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

Arrays can be 'sparse' or jagged (* 4+ *). 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;
END;
```

`flow` is an array with six elements spread over three rows. At present, we only have sparse arrays of *relations* implemented, due to historical, rather than technical, reasons. Sparse arrays of models and variables should be implemented soon, since their not being allowed is really just a bug.

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;
```

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 it.

Label:

One can label statements which define 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;
```

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

`relation_`modelname_linenumber

where *modelname* and *linenumber* are that of the name of the model and the line number in the model file on which the relation statement ends, respectively. An example is

`relation_mixture_14`

for the unlabeled relation in the model `mixture` ending on line `14` of the file containing the mixture definition.

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]
```

1.1.3 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 definition of the attributes (parts) and attribute default values that an object will have if it is created using the type definition.

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*. 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;                    1
i:== 2;
j:== 3;
```

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

Instances which are either explicitly or implicitly type incompatible cannot be merged. This will be discussed further in Section 1.3.

Instantiation

Creating an simulation based on a type definition is a 3 phase process called compiling (or instantiation). When an instantiation cannot be completed because some structural parameter (a symbol_constant, real_constant, integer_constant, or set) does not have a value there will be PENDING statements. The user interface will warn that something is wrong.

In phase 1 all statements that create instance structure and assign constant values are executed. This phase theoretically requires an infinite number of passes through the structural statements of the 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 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.

Phase 3 executes the variable defaulting statements made in the declarative section of each model IF AND ONLY IF there are no pending statements from phase 1 anywhere in the simulation.

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

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

1.2 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.

Examples of universal type definitions are

```
UNIVERSAL MODEL methane REFINES
generic_component_model;

UNIVERSAL CONSTANT circle_constant REFINES
real_constant :== 3.14159;

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.

1.2.1 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 1.3 and Section 1.4.

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 couldn't 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, components
:== ['c5','c6']);
     (* additional flowsheet information *)
END flowsheet;
```

In this example, the column model takes the first argument, ntrays, by reference. That is `c.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. The second argument is taken by value, `['c5','c6']`, and assigned to the components part that was declared with the IS_A in the parameter list. There is only one name for this set, `c.components`. Note that in the flowsheet model there is no part that is a set of symbol_constant. The :== in the flowsheet definition of ct is an argument list mnemonic which reminds us that this argument is being passed by value rather than by reference. It serves other purposes as well, which will be explained in a separate

section devoted to the ins and outs of using parameterized types and the construction of reusable libraries.

1.2.2 SETS

Arrays in ASCEND, as already discussed in Section 1.1.2, are defined over sets. A set is simply an instance with a set value.

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(list)

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. 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].

INTERSECTION()

INTERSECTION(list of set expressions). Find the intersection (and) of the sets listed. The syntax is

*****

```
INTERSECTION(list_of_sets)
```

A*B is shorthand for INTERSECTION(A,B)

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

Set difference:

-

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

```
first_set - second_set
```

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

CARD(set)

Cardinality. Returns an integer 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.

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)];
```

1.2.3 CONSTANTS

ASCEND supports real, integer, boolean and character 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 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.

1.2.4 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 whose base type is real, the dimensions will be left as undefined. Any variable which is later declared to be one of these types will be given *wild card* dimensionality (represented in the interactive display by an asterisk (*)). The system will deduce the dimensionality from its use in the relationships in which it appears or in the declaring of default values for it, if possible.

solver_var is a special case of ATOM and we will say much more about it in Section 1.5.1.

```
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 parametric values can be included in equations without treating them as variables.

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.

symbol

4+ Symbol valued variable quantity. At present, symbols, like symbol_constants, can only be assigned once. This restriction is a holdover from ASCEND III that should go away soon, as it is ideal to have manipulable character strings in any language.

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 12) 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 an equation to state that the LHS expression is to equal the RHS expression at the solution for the model.

Tabular assignments

(* 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

```
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;

(* 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 DIMEN-
SIONLESS. *)
(* 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;
```

END tabular_ex;

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

1.2.5 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 1.5.2. 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.

Unary Operators:

-, *function*()

–

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

***function*()**

unary real valued functions. The unary real functions we support are given in section 1.5.2.

Real functions of lists:

SUM(list) 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[i]
     |i IN 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(list) 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(list) maximum value on list of arguments

MIN(list) minimum value on list of arguments

1.2.6 DERIVATIVES IN RELATIONS (* 4+ *)

1.2.7 EXTERNAL RELATIONS

1.2.8 CONDITIONAL RELATIONS (* 4+ *)

1.2.9 LOGICAL RELATIONS (*4+*)

Logical expression

(The following is proposed but not implemented at this time.) 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)
```

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 syntax or the semantics of logical relations very hard yet.

1.2.10 EXPLANATIONS (* 4+ *)

1.3 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 FOR/CREATE and the EXPLANATIONS statements. The procedural compound statements allowed only in methods are the FOR/DO and the IF statements. Compound statements end with "`END;`", and they can be nested.

CASE statements coming eventually

(*4+*) WHEN/CASE and SELECT/CASE are also compound statements. They will handle conditional equations and conditional compilation, respectively, in a later version of ASCEND IV.

Type declarations are not 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.

`IS_A`

Constructor. Calls for one or more named instances to be compiled of the type specified.

(* 4+ *) May be followed by COMPLETE to indicate that the instance created is to be type-locked after it is compiled.

`ALIASES (* 4+ *)`

Part renaming statement. Establishes another name in the scope where the alias statement appears for an instance at the same scope or in a child instance.

`WILL_BE (* 4+ *)`

Forward declaration statement. Promises that a part with the given name and type will be constructed by an as yet unknown IS_A statement.

IS (* 4+ *) Global object renaming statement. Establishes another name in the scope where the IS statement occurs for an already existing object that occurs somewhere in the universe of simulations.

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.

ARE_ALIKE Refinement clique constructor. Causes a group of instances to always be of the same explicit type. Refining one of them causes a refinement of all the others.

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).

(* 4+ *) May be followed by the word COMPLETE to type-lock the instance after the reconstruction is finished.

ARE_CONNECTED (*4+*) Creates an array of particular equations that connect two locked instances. This has the same mathematical effect as merging the instances, but does not invalidate other objects that may have been derived on the two locked instances.

FOR/CREATE Indexed execution of other declarative statements. Required for creating arrays of relations over indexed variables and sparse arrays of other objects.

Reminder: In the following statement descriptions, we show keywords in capital letters. These words must appear in capital letters as shown in ASCEND statements. We show optional parts to a statement enclosed in double angle brackets (« ») and user supplied names in lower-case *italic* letters. (Remember that ASCEND treats the underscore (_) as a letter). The user may substitute any name desired for these names. We use names that describe the kind of name the user should use.

Operators in detail:

IS_A statement This statement has the syntax

list_of_instance_names IS_A *model_name*;

or

list_of_instance_names IS_A COMPLETE *model_name*;

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).

If a name is used twice in IS_A statements at the same scope ASCEND will complain and execute only the first IS_A statement encountered. Duplicate naming is a serious error. Labels on relations share the same name space as other objects.

Several examples of IS_A appear throughout this chapter, e.g. page 1.

(* 4+ *) IS_A COMPLETE tells the compiler that the model should be constructed immediately and locked so that no reconstruction is possible. This means that no further constant assignments, set assignments, refinements, or merges are to be allowed within the model, so if the model definition is incomplete the result will not be interactively reparable. Instances that have been locked can be used to determine the structure of other instances, and they can be stored in a particularly efficient manner.

ALIASES (* 4+ *) This statement has the syntax

list_of_instances ALIASES *instance_name*;

We use this statement to point at an already existing instance. For example, say we want a flash tank model to have a variable T, the temperature of the vapor-liquid 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. Since ASCEND uses an infinite pass compiler, this is not usually a problem.

WILL_BE (* 4+ *) *list_of_instances* WILL_BE *model_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. The secondary 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.

IS (* 4+ *) One of:

list_of_instances IS *simulation_name*;

list_of_instances IS *simulation_name OF model_name*;

list_of_instances IS COMPLETE *simulation_name*;

list_of_instances IS COMPLETE *simulation_name OF model_name*;

Here *simulation_name* is the full name of any object in the universe of already constructed objects. The *simulation_name* may be a qualified identifier, i.e. the full path name to any part of a global object. A local name is established for the global object. The OF *model_name* is optionally allowed to permit type checking of the found object. The COMPLETE modifier says that once found the object is to be *locked*. This lock may have unexpected l side-effects, so care should be taken when applying it.

(* 4++ *) We expect the IS syntax to be eventually extended to include objects in other instances of the ASCEND environment running anywhere on the network or in any other network object that can supply equivalent functionality.

IS_REFINED_TO This statement has the syntax

list_of_instances IS_REFINED_TO *model_name*;

or

(4+ *) list_of_instances* IS_REFINED_TO COMPLETE *model_name*;

We use this statement to change the type of each of the instances listed to the type *model_name*. The modeler has to have defined each member on the list of instances. The *model_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;
```

(* 4+ *) IS_REFINED_TO COMPLETE tells the compiler that the model should be reconstructed immediately and locked so that no further reconstruction is possible. This means that no further constant assignments, set assignments, refinements, or merges are to be allowed within the model, so if the refined model definition is incomplete the result will not be interactively reparable. Instances that have been locked can be used to determine the structure of other instances, and they can be stored in a particularly efficient manner.

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.

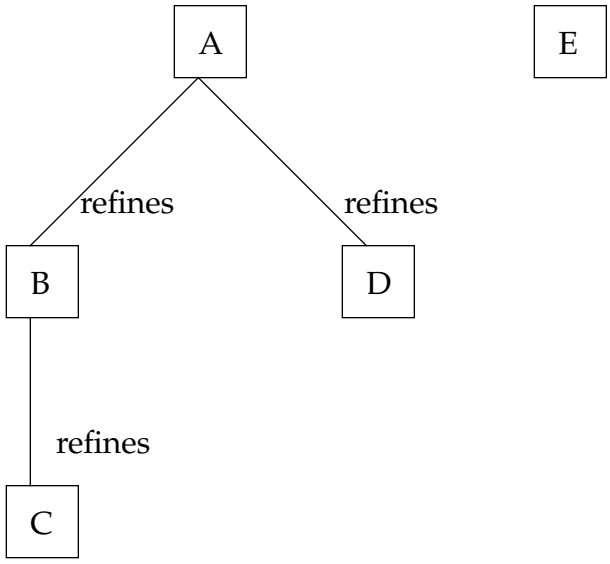

Figure 1. Diagram of the model type hierarchy A,B,C,D,E

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.

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 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 assign common storage locations, etc. It may be thought of as a partial merge. There are important consequences of modeling with such a partial merge.

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

Another purpose is the propagation of type through a model. 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.

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 explicit types but incompatible *implicit* types. This can lead to unexpected problems later and makes the ARE_ALIKE instruction a candidate for disappearing in a future release of ASCEND IV.

ARE_CONNECTED

(*4+*)

name: pair_of_instances ARE_CONNECTED;

Causes an array of equalities (named `name[]`) to be written between the variables of two identically typed, locked instances. This can end up being an enormous array and we take that into account internally since all the equations are essentially of the form `a.x = b.x;` If *name* is omitted, we will make one up for you, just as with other relations. The subscript of the relation array will be `[1..n]` where *n* is the number of variables that occurs in each of the objects.

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;
```

This statement must be in the non-procedural part of the model definition only. Every statement in the list should have at least one occurrence of the index variable, else the same statement will be produced multiple times. An example is

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

At present IS_A, ALIASES, and WILL_BE are not allowed within FOR statements because we have not yet implemented sparse arrays of general objects. This is a known bug. Eventually these operators must be allowed for indexed sparse array instances, as illustrated in ARRAYS CAN BE JAGGED on page 9.

1.4 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.

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 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.

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 on page 20) 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;
```

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

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

is an empty loop, while

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

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;
```

or

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

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.

SWITCH Vicente'?

CALL External calls?

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;
```

1.5 MISCELLANY

1.5.1 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.lib 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.lib.

generic_real

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

Kluges for MILPs

Also defined in system.lib 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 algebraic variable.

Kluges for ODEs

We have an alternate version of system.lib called ivpsystem.lib 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.lib

Having ivpsystem.lib 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.lib. Users who want both algebraic and IVP capability load ivpsystem.lib instead of system.lib. 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. Calculus is not yet implemented, however.

1.5.2 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.

1.5.3 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
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);

Discontinuous functions: Discontinuous functions may destroy a Newton-based solution algorithm if used in defining a model equation:

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

1.5.4 LOGICAL FUNCTIONS

SATISFIED()

(*4+*)

Vicente' needs to fill in this section with whatever logical functions are needed.

1.5.5 UNITS

The following section defines the dimensions and units and all the attendant conversion factors. Note that all conversions are simply multiplicative. This information is from the file compiler/units_input in the ASCEND source code.

Note that the units_input file can easily have additional units defined to suit the needs of local users. We are always on the lookout for new and interesting units, so if you have some send them in.

```
#                 Units input file
#                 by Tom Epperly
#                 Version: $Revision: 1.11 $
#                 Date last modified: $Date: 1995/02/19 23:03:44 $
#                 Copyright(C) 1990 Thomas Guthrie Epperly
#
# This is a file defining the conversion factors ASCEND will recognize when
# it sees them as {units). Note that the assignment x:= 0.5 {100}; yields
# x == 50, and that there are no 'offset conversions,' e.g. F=9/5C+32;
# Added money which isn't really time 3-94 BAA
# Expanded, including some of Karl's units, constants. 4-94 BAA
# Updated with supplementary SI dimensions and less ambiguous mole dim. jz/baa
#
# Please keep unit names to 20 characters or less as this makes life pretty
#
```

1.5.5.1 DEFINE THE SYSTEM UNITS IN SI MKS system

```
define kilogram   M;   # internal mass unit SI
define mole       Q;   # internal quantity unit SI
```

```
define second     T;   # internal time unit SI
define meter      L;   # internal length unit SI
define Kelvin     TMP; # internal temperature unit SI
define currency   C;   # internal currency unit
define ampere     E;   # internal electric current unit SI suggested
define candela    LUM; # internal luminous intensity unit SI
define radian     P;   # internal plane angle unit SI suggested
define steradian  S;   # internal solid angle unit SI suggested
#
```

1.5.5.2 distance

```
#
pc = 3.08374e+16*meter;
parsec = pc;
kpc = 1000*pc;
Mpc = 1e6*pc;
km = meter*1000;
m = meter;
dm = meter/10;
cm = meter/100;
mm = meter/1000;
um = meter/1000000;
nm = 1.e-9*meter;
kilometer = km;
centimeter = cm;
millimeter = mm;
micron=um;
nanometer = nm;
angstrom = m/1e10;
fermi = m/1e15;
#
mi = 1609.344*meter;
yd = 0.914412*meter;
ft = 0.304804*meter;
inch = 0.0254*meter;
mile = mi;
yard = yd;
feet = ft;
foot = ft;
in = inch;
#
```

1.5.5.3 mass

```
#
metton = kilogram *1000;
mton = kilogram *1000;
kg = kilogram;
g = kilogram/1000;
gram = g;
mg = g/1000;
milligram = mg;
```

```
ug= kilogram*1e-9;
microgram = ug;
ng=kilogram*1e-12;
nanogram=ng;
pg=kilogram*1e-15;
picogram=pg;
#
amu = 1.661e-27*kilogram;
lbm = 4.535924e-1*kilogram;
ton = lbm*2000;
oz = 0.028349525*kilogram;
slug = 14.5939*kilogram;
#
```

1.5.5.4 time

```
#
yr = 31557600*second;
wk = 604800*second;
dy = 86400*second;
hr = 3600*second;
min = 60*second;
sec = second;
s = second;
ms = second/1000;
us = second/1e6;
ns = second/1e9;
ps = second/1e12;
year = yr;
week = wk;
day = dy;
hour = hr;
minute = min;
millisecond = ms;
microsecond = us;
nanosecond = ns;
picosecond = ps;
#
```

1.5.5.5 molecular quantities

```
#
kg_mole=1000*mole;
g_mole = mole;
gm_mole = mole;
kmol = 1000*mole;
mol = mole;
mmol = mole/1000;
millimole=mmol;
umol = mole/1e6;
micromole=umol;
lb_mole = 4.535924e+2*mole;
#
```

1.5.5.6 temperature

```
#
K = Kelvin;
R = 5*Kelvin/9;
Rankine = R;
#
```

1.5.5.7 money

```
#
dollar = currency;
US = currency;
USdollar=currency;
CR = currency;
credits=currency;
#
```

1.5.5.8 reciprocal time (frequency)

```
#
rev = 1.0;
cycle = rev;
rpm = rev/minute;
rps = rev/second;
hertz = cycle/second;
Hz = hertz;
#
```

1.5.5.9 area

```
#
ha = meter^2*10000;
hectare=ha;
acre= meter^2*4046.856;
#
```

1.5.5.10 volume

```
#
l = meter^3/1000;
liter = l;
ml = liter/1000;
ul = liter/1e6;
milliliter = ml;
microliter = ul;
#
hogshead=2.384809e-1*meter^3;
cuft = 0.02831698*meter^3;
impgal = 4.52837e-3*meter^3;
gal = 3.785412e-3*meter^3;
barrel = 42.0*gal;
gallon = gal;
quart = gal/4;
pint = gal/8;
cup = gal/16;
floz = gal/128;
#
```

1.5.5.11 force

```
#
N = kilogram*meter/second^2;
newton = N;
dyne = N*1.0e-5;
pn=N*1e-9;
picoNewton=pn;
#
lbf = N*4.448221;
#
```

1.5.5.12 pressure

```
#
Pa = kilogram/meter/second^2;
MPa = 1.0e+6*Pa;
bar =1.0e+5*Pa;
kPa = 1000*Pa;
pascal = Pa;
#
atm = Pa*101325.0;
mmHg = 133.322*Pa;
torr = 133.322*Pa;
psia = 6894.733*Pa;
psi = psia;
ftH2O = 2989*Pa;
#
```

1.5.5.13 energy

```
#
J = kilogram*meter^2/second^2;
joule = J;
MJ = J * 1000000;
kJ = J * 1000;
mJ=J*1.0e-3;
uJ=J*1.0e-6;
nJ=J*1.0e-9;
milliJoule=mJ;
microJoule=uJ;
nanoJoule=nJ;
erg = J*1.0e-7;
#
BTU = 1055.056*J;
pCu = BTU * 1.8;
cal = J*4.18393;
calorie = cal;
kcal=1000*calorie;
Cal=1000*calorie;
#
```

1.5.5.14 power

```
#
W = J/second;
```

```
EW = 1.0e+18*W;
PW = 1.0e+15*W;
TW = 1.0e+12*W;
GW = 1.0e+9*W;
MW = 1.0e+6*W;
kW = 1000*W;
mW = W/1000;
uW = W/1000000;
nW = W/1e9;
pW = W/1e12;
fW = W/1e15;
aW = W/1e18;
terawatt = TW;
gigawatt = GW;
megawatt = MW;
kilowatt = kW;
watt = W;
milliwatt = mW;
microwatt = uW;
nanowatt = nW;
picowatt = pW;
femtowatt = fW;
attowatt = aW;
# aWW= 1*EW;
#
hp= 7.456998e+2*W;
#
```

1.5.5.15 absolute viscosity

```
#
poise = Pa*s;
cP = poise/100;
#
```

1.5.5.16 electric charge

```
#
coulomb=ampere*second;
C = coulomb;
coul = coulomb;
mC = 0.001*C;
uC = 1e-6*C;
nC = 1e-9*C;
pC = 1e-12*C;
#
```

1.5.5.17 misc. electro-magnetic fun

```
#
V = kilogram*meter^2/second^3/ampere;
F = ampere^2*second^4/kilogram/meter^2;
ohm = kilogram*meter^2/second^3/ampere^2;
mho = ampere^2*second^3/kilogram/meter^2;
S = mho;
```

```
siemens = S;
A=ampere;
amp = ampere;
volt = V;
farad= F;
mA= A/1000;
uA= A/1000000;
kV= 1000*V;
MV= 1e6*V;
mV= V/1000;
mF = 0.001*F;
uF = 1e-6*F;
nF = 1e-9*F;
pF = 1e-12*F;
kohm = 1000*ohm;
Mohm = 1e6*ohm;
kS = 1000*S;
mS = 0.001*S;
uS = 1e-6*S;
Wb = V*second;
weber = Wb;
tesla = Wb/m^2;
gauss = 1e-4*tesla;
H = Wb/A;
henry = H;
mH = 0.001*H;
uH = 1e-6*H;
#
```

1.5.5.18 numeric constants of some interest

```
# to set a variable or constant to these, the code is (in the declarations)
# ATOM constant REFINES real; END constant;
# MODEL gizmo;
# x IS_A constant;
# x := 1 {PI};
# ...
molecule = 1.0;
PI=3.141592653589793;                   # Circumference/Diameter ratio
EULER_C = 0.57721566490153286;          # euler gamma
GOLDEN_C = 1.618033988749894;           # golden ratio
HBAR = 1.055e-34*J*second;              # Reduced Planck's constant
PLANCK_C = 2*PI*HBAR;                   # Planck's constant
LIGHT_C = 2.99793e8 * meter/second;     # Speed of light in vacuum
MU0 = 4e-7*PI*kg*m/(C*C);               # Permeability of free space
EPSILON0 = 1/LIGHT_C/LIGHT_C/MU0;       # Permittivity of free space
BOLTZMAN_C = 1.3805e-23 * J/K;          # Boltzman's constant
AVOGADRO_C = 6.023e23 *molecule/mole;   # Avogadro's number of molecules
GRAVITY_C = 6.673e-11 * N*m*m/(kg*kg); # Newtons gravitational constant
GAS_C = BOLTZMAN_C*AVOGADRO_C;          # Gas constant
INFINITY=1.0e38;                        # damn big number;
#
```

```
eCHARGE = 1.602e-19*C;                  # Charge of an electron
EARTH_G = 9.80665 * m/(s*s);           # Earth's gravitational field, somewhere
eMASS = 9.1095e-31*kilogram;            # Electron rest mass, I suppose
pMASS = 1.67265e-27*kilogram;           # Proton mass
#
```

1.5.5.19 constant based conversions

```
#
eV = eCHARGE * V;
keV = 1000*eV;
MeV = 1e6*eV;
GeV = 1e9*eV;
TeV = 1e12*eV;
PeV = 1e15*eV;
EeV = 1e18*eV;
#
lyr = LIGHT_C * yr;                     # Light-year
#
oersted = gauss/MU0;
#
```

1.5.5.20 subtly dimensionless measures

```
#
rad = radian;
srad = steradian;
deg = radian*1.74532925199433e-2;
degrees = deg;
grad = 0.9*deg;
arcmin = degrees/60.0;
arcsec = arcmin/60.0;
#
```

1.5.5.21 light quantities

```
#
cd = candela;
lm = candela*steradian;
lumen = lm;
lx = lm/meter^2;
lux= lx;
#
```

1.5.5.22 misc. rates

```
#
gpm = gallon/minute;
#
```

1.5.5.23 time variant conversions

```
#
MINIMUMWAGE = 4.75*US/hr;
SPEEDLIMIT = 65*mi/hr;
#
# conversions we'd like to see, but probably won't
# milliHelen = beauty/ship;
# guy = quart;
```

```
# there's at least 4 guys for every gal around here.
```

1.6 GRAMMAR

The grammar presented here is a stripped down BNF description of the ASCEND language. It includes some features for which we have syntax but no semantics as yet. Therefore, the authoritative definition of the language is the informal one given above.

Mismatches between the informal description and the performance of ASCEND IV software are thus bugs. The implemented ASCEND language is merely an instance of a type that exists in the collective head of the developers and users of ASCEND and Art Westerberg.

1.6.1 FLEX DEFINITIONS

I need a volunteer to take scanner.l and ascend.y and strip them down to a text file that looks reasonable.

1.6.2 YACC DEFINITIONS