SIMSCRIPT III®

Reference Manual

CACI Products Company

SIMSCRIPT III Reference Manual, Version 1, contains parts from the book:

"SIMSCRIPT III"
by Stephen V. Rice, Ana K. Marjanski, Harry M. Markowitz, and Stephen M. Bailey
Manuscript in progress.

For product information or technical support contact:

CACI Products Company
1455 Frazee Road, Suite 700
San Diego, CA 92108
Phone: (619) 881-5806
Email: simscript@caci.com

Table of Contents

PREFACE 7

1 Introduction 9

2 Language Reference 11

2.01 MainModule 12
2.02 Subsystem 13
2.03 AccumulateTally 15
2.04 AddSubtract 17
2.05 BeforeAfter 18
2.06 BeginClass 21
2.07 Belongs 22
2.08 BreakTies 23
2.09 Call 24
2.10 Cancel 27
2.11 Close 29
2.12 Comma 30
2.13 Compute 31
2.14 CreateDestroy 35
2.15 Cycle 40
2.16 DefineConstant 42
2.17 DefineMethod 43
2.18 DefineRoutine 44
2.19 DefineRoutineArguments 46
2.20 DefineSet 48
2.21 DefineToMean 50
2.22 DefineVariable 51
2.23 Digit 54
2.24 Enter 55
2.25 Every 56
2.26 Expression 59
2.27 External 62
2.28 File 64
2.29 Find 68
2.30 For 70
2.31 GoTo 77
2.32 Has 79
2.33 Histogram 83
2.34 If 85
2.35 Implementation 88
2.36 Integer 90
2.37 InterruptResume 91
2.38 Jump 93
2.39 Label 94
2.40 Leave 97
2.41 Let 99
2.42 Letter 101
2.43 List 102
2.44 LogicalComparison 104
2.45 LogicalExpression 107
2.46 LogicalPhrase1 110
2.47 LogicalPhrase2 114

2.48 Loop 116
2.49 MethodsHeading 124
2.50 Mode 125
2.51 Move 126
2.52 Name 129
2.53 NameUnqualified 131
2.54 Normally 133
2.55 Number 135
2.56 Open 136
2.57 Owns 139
2.58 PermanentEntities 141
2.59 PreambleStatement 142
2.60 Print 143
2.61 Priority 146
2.62 Processes 147
2.63 ReadWrite 148
2.64 ReadWriteFormat 154
2.65 Release 164
2.66 Relinquish 167
2.67 Remove 168
2.68 Request 171
2.69 Reserve 172
2.70 Reset 177
2.71 Resources 178
2.72 Return 180
2.73 Routine 183
2.74 RoutineStatement 186
2.75 Schedule 187
2.76 Select 194
2.77 SignedNumber 197
2.78 Skip 198
2.79 SpecialSymbol 200
2.80 StartNew 201
2.81 StartSimulation 203
2.82 Stop 204
2.83 String 205
2.84 SubprogramLiteral 206
2.85 Substitute 207
2.86 SuppressResume 208
2.87 Suspend 209
2.88 TemporaryEntities 210
2.89 TheClass 211
2.90 TheSystem 212
2.91 Trace 213
2.92 Unit 214
2.93 Use 215
2.94 Variable 216
2.95 Wait 220
2.96 While 222
2.97 With 223

3 Library.m 225

3.01 Mode Conversion 226
Numeric Operations 229
3.02 Text Operations 234
3.03 Input/Output 236

3.04 Random-Number Generation 240
3.05 Simulation 244
3.06 Miscellaneous 249

PREFACE

This document contains information on CACI's new SIMSCRIPT III, Modular Object-Oriented Simulation Language, designed as a superset of the widely used SIMSCRIPT II.5 system for building high-fidelity simulation models.

CACI publishes a series of manuals that describe the SIMSCRIPT III Programming Language, SIMSCRIPT III Graphics and SIMSCRIPT III SimStudio. All documentation is available on SIMSCRIPT WEB site http://www.caciasl.com/products/simscript.cfm

- *SIMSCRIPT III Reference Manual* – this manual is a complete description of the SIMSCRIPT III programming language constructs in alphabetic order. Graphics constructs are described in the SIMSCRIPT III Graphics Manual.

- *SIMSCRIPT III Programming Manual* – A short description of the programming language and a set of programming examples.

- *SIMSCRIPT III User's Manual* – is a detailed description of the SIMSCRIPT III development environment: usage of SIMSCRIPT III Compiler and the symbolic debugger from the SIMSCRIPT Development studio - Simstudio, and from the Command-line interface.

- *SIMSCRIPT III Graphics Manual* — A detailed description of the GUI dialog boxes, presentation graphics and animation environment for SIMSCRIPT III

Since SIMSCRIPT III is a superset of SIMSCRIPT II.5, a series of manuals and text books for SIMSCRIPT II.5 language, Simulation Graphics, Development environment, Data Base connectivity, Combined Discrete-Continuous Simulation, can be used for additional information:

- *SIMSCRIPT II.5 Simulation Graphics User's Manual* — A detailed description of the presentation graphics and animation environment for SIMSCRIPT II.5

- *SIMSCRIPT II.5 Data Base Connectivity (SDBC) User's Manual* — A description of the SIMSCRIPT II.5 API for Data Base connectivity using ODBC

- *SIMSCRIPT II.5 Operating System Interface* — A description of the SIMSCRIPT II.5 APIs for Operating System Services

- *Introduction to Combined Discrete-Continuous Simulation using SIMSCRIPT II.5* — A description of SIMSCRIPT II.5 unique capability to model combined discrete-continuous simulations.

- *SIMSCRIPT II.5 Programming Language* — A description of the programming techniques used in SIMSCRIPT II.5.

• *SIMSCRIPT II.5 Reference Handbook* — A complete description of the SIMSCRIPT II.5 programming language, without graphics constructs.

• *Introduction to Simulation using SIMSCRIPT II.5* — A book: An introduction to simulation with several simple SIMSCRIPT II.5 examples.

• *Building Simulation Models with SIMSCRIPT II.5* —A book: An introduction to building simulation models with SIMSCRIPT II.5 with examples.

The SIMSCRIPT language and its implementations are proprietary program products of the CACI Products Company. Distribution, maintenance, and documentation of the SIMSCRIPT language and compilers are available exclusively from CACI.

Free Trial Offer

SIMSCRIPT III is available on a free trial basis. We provide everything needed for a complete evaluation on your computer. **There is no risk to you**.

Training Courses

Training courses in SIMSCRIPT III are scheduled on a recurring basis in the following locations:

San Diego, California
Washington, D.C.

On-site instruction is available. Contact CACI for details.

For information on free trials or training, please contact the following:

CACI Products Company
1455 Frazee Road, suite 700
San Diego, California 92108
Telephone: (619) 881-5806
www.caciasl.com

1 Introduction

The SIMSCRIPT III programming language is a superset of SIMSCRIPT II.5 with significant new features to support modular, object-oriented simulation programming.

It preserves existing world-view and the powerful data structures: entities, attributes and sets, process and event-oriented discrete simulation of SIMSCRIPT II.5, and adds the new, more elaborated data structures and concepts like classes, methods, objects, multiple inheritance and process-methods, to support object-view and object-oriented process and event discrete simulation. Object types are defined with the classes which can be instantiated, they may have methods which describe object behavior, and may contain special process-methods with time elapsing capabilities which can be scheduled for execution in defined instances of time. Both, world-view and object-view can exist in the same model, or a modeler may decide to use entirely object-view or a world-view only.

SIMSCRIPT III supports model decomposition into subsystems. Modules and packages are synonyms for subsystems. A model can consist only of main module (preamble and implementation), but larger models should be designed with modularity in mind, as a main module with a set of subsystems. This facilitates code reuse and model development in a team-work environment. Subsystem may contain public and private declarations and implementation. Public data and function/method declaration defines subsystem's interface with the system and other subsystems while private data and method declarations hide implementation details.

Modularity can be easily added to an existing SIMSCRIPT II.5 model, defining it as a main module (system) and adding new subordinate modules (subsystems/packages). SIMSCRIPT III compiler checks data scope of the subsystems and performs name resolution.

SIMSCRIPT III includes all standard language elements and can be used as a general-purpose object-oriented programming language with English-like syntax. In addition, it includes powerful support for building simulation models with interactive GUI, presentation graphics and animation. Building SIMSCRIPT III graphical models is explained in the SIMSCRIPT III Graphics Manual.

The SIMSCRIPT III models are developed inside "Simstudio", an Integrated Development Environment (IDE) which incorporates automatic project builder, syntax colored text editors and graphical editors for GUI elements: dialog boxes, menus, palettes, icons, graphs. Building SIMSCRIPT III projects using Simstudio is described in SIMSCRIPT III User's Manual.

Language statements and concepts are described in SIMSCRIPT III Programming Manual. It describes basic language elements and related enhancements like support for Latin character set, named constants, argument type checking, multiple-line comments,

and reference modes. It also contains description of classes, objects, multiple inheritance, object and class methods for support of object-oriented programming. Object-oriented simulation is facilitated by process-methods which can be used for process and event-based discrete simulation. Accumulate and tally statements are very convenient and effective tools for statistics collection.

SIMSCRIPT III example programs given in Programming Manual are rewritten from SIMSCRIPT II.5. Original programs are from the book: Building Simulation Models with SIMSCRIPT II.5. These examples illustrate use of classes, objects, inheritance, subsystems. They illustrate how to create simulations with process-methods and how to collect statistics on object attributes.

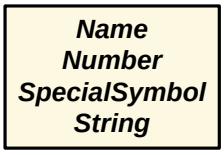
The lists the "system" routines, variables, and constants, which are defined by SIMSCRIPT III's standard, system **library.m** subsystem and are implicitly imported into every module, are given in Chapter 3 of this Reference Manual. Other system modules like GUI.m, SDBC.m, Continuous.m are imported on demand and described in specialized manuals.

This manual contains detailed description of all SIMSCRIPT III Language elements in alphabetic order. All language elements of SIMSCRIPT II.5 which are fully supported in SIMSCRIPT III are described, while deprecated statements and language elements are omitted.

2 Language Reference

Each section of this chapter describes an element of the SIMSCRIPT III language and includes a syntax diagram for the element at the beginning of the section. In a syntax diagram, arrows connect rectangles to indicate permitted sequences of language elements. A rectangle with *square* corners specifies the name of a required language element, or gives a vertical list of names from which one element must be chosen. A rectangle with *rounded* corners specifies a required keyword, phrase or symbol, or provides a vertical list of these from which one must be selected. For example:

The language element named ***Expression*** is required.



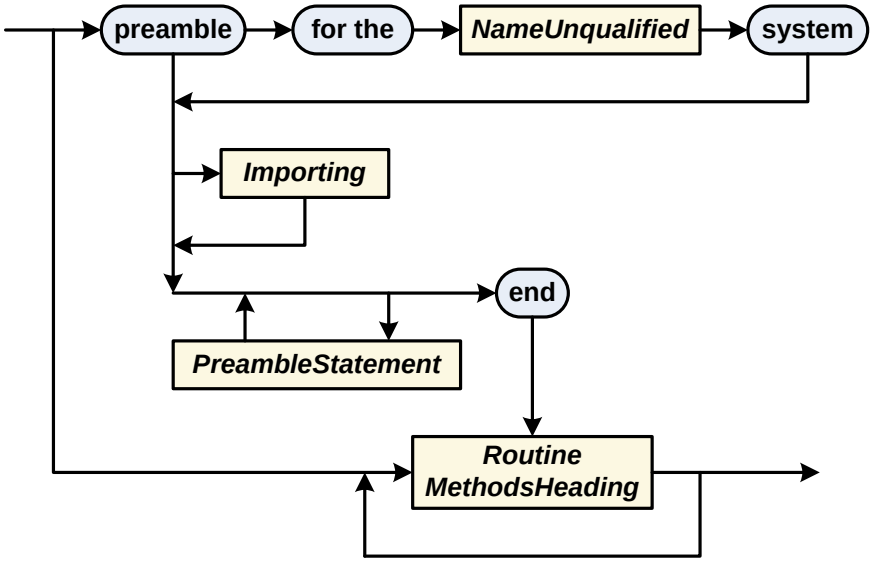
Either a ***Name***, ***Number***, ***SpecialSymbol***, or ***String*** must be provided.

The keyword **every** is required.

Either **thus**, **as follows**, or **like this** must be specified.

A SIMSCRIPT III program consists of a main module and zero or more subordinate modules called "subsystems." The first two sections of this chapter describe a ***MainModule*** and a ***Subsystem***. The remaining sections describe language elements in alphabetical order.

2.01 MainModule



A main module consists of an optional preamble followed by one or more routines and **methods** headings. One of the routines must be named **main**.

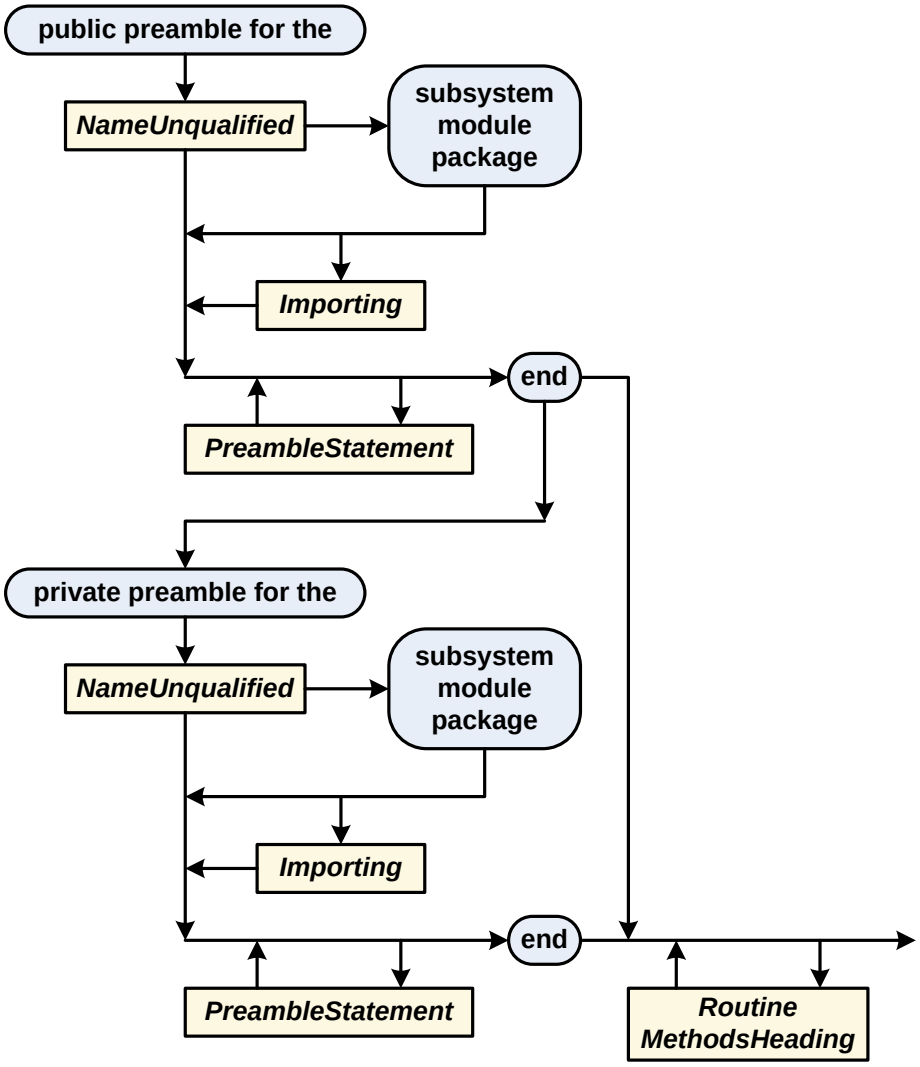
```
Preamble for the X system
            importing the A subsystem
end

methods for the X system
  main
  end

  Rout1
  end
```

Preamble contains definitions of data structures used in the program like: classes, entities, global variables, constants and sets. All statements in a preamble are non-executable. A main module can be given a name and can import subsystems.

SIMSCRIPT III compiler supports flexible source code organization so that main module may span multiple files. However, preamble or a routine may not span multiple files.

Program execution begins by executing each subsystem "initialize" routine once, in an indeterminate order, and then by executing the main module's "main" routine.

2.02 Subsystem



A subsystem begins with a public preamble and is followed by an optional private preamble and zero or more routines and **methods** headings. One of the routines may be named **initialize**. The declarations in the public preamble are visible to the private preamble and routines of the subsystem, and to every module that imports this subsystem. The declarations in the private preamble are visible only to the routines of the subsystem. The keywords **subsystem**, **module**, and **package** are synonymous.

A subsystem must be given a name and both its public and private preambles can import other subsystems.

Separate compilation of modules is supported. If a subsystem's private preamble or routines are modified, only the subsystem needs to be recompiled. However, each program that uses the subsystem must be re-linked.

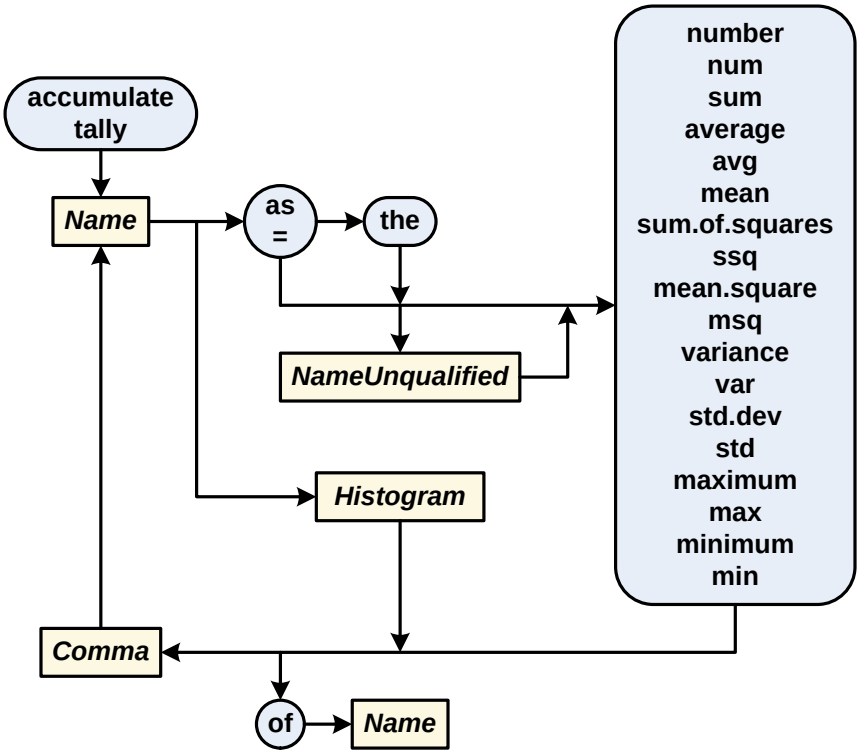
It is easier to develop and maintain a large program that has been divided into meaningful units called “modules.” Subsystems promote better source code organization and facilitate the reuse of code. The public preamble of a subsystem defines the interface to the subsystem, and the implementation is hidden in the private preamble and routines of the subsystem. A module may import any number of subsystems, and a subsystem may be imported by any number of modules.

A subsystem may be distributed as a source file containing only the public preamble, and one or more binary object files obtained by compiling the subsystem. The source file documents the subsystem interface and is read by the compiler when compiling a module that imports the subsystem. An executable program is built by linking the binary object files that were produced by compiling the main module and each of its subsystems.

SIMSCRIPT III compiler supports source code organization in which subsystem may span multiple files; however, a preamble or routine may not span multiple files.

Program execution begins by executing each subsystem "initialize" routine once, in an indeterminate order, and then by executing the main module's "main" routine.

2.03 AccumulateTally



An **accumulate** or **tally** statement specifies one or more statistics to collect on the values assigned to an attribute or global variable. The statistics are weighted by simulation time for an **accumulate** statement and are un-weighted for a **tally** statement. These statements may appear in a preamble.

The keyword **the** is optional for readability. The following are synonymous:

- **as** and **=**;
- **number** and **num**;
- **average**, **avg**, and **mean**;
- **sum.of.squares** and **ssq**;
- **mean.square** and **msq**;
- **variance** and **var**;
- **std.dev** and **std**;
- **maximum** and **max**;
- **minimum** and **min**.

Accumulate and tally cannot be declared for the same variable. The programmer must decide whether a variable is time-dependent or not, normally a simple task.

There is no difference between accumulating and tallying each of the following: number, maximum, minimum.

Accumulating a sum means the sum of each (value times the amount of time that value was held). The mean is the accumulated sum divided by the total elapsed time.

The collection of statistics may be reinitialized by executing a "reset" statement. The reset statement makes possible the preparation of reports on a cumulative or periodic basis. When both periodic and cumulative statistics are required, qualifiers can be specified. The qualifiers permit multiple sets of the same statistic to be gathered simultaneously, but the statistics can be reset at different times. These qualifiers allow some statistics to be reset while others are not. The names of qualifiers may not be qualified!

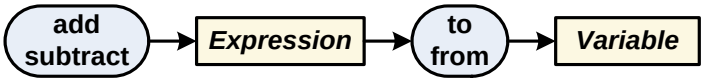
The variable upon which statistics are collected must be numeric (mode integer, integer2, integer4, alpha, double, real); it cannot be pointer, text, reference, or subprogram. This variable may have any dimensionality and may be monitored. It may be explicitly defined or implicitly defined, such as the n.set attribute. It may not be a function attribute, random variable, or a statistic collected by another accumulate/tally statement.

If the accumulate/tally statement appears inside a begin class block, the variable upon which statistics are collected must be an object attribute or class attribute defined by the class, or an object attribute inherited from a base class. If the variable is an object attribute, each statistic is defined as an object attribute, whereas if the variable is a class attribute, each statistic is defined as a class attribute. If the variable is defined in a public preamble and the accumulate/tally statement appears in the private preamble, the variable must be an object attribute; that is, private statistics cannot be collected on a public class attribute.

If the accumulate/tally statement appears outside a begin class block, the variable upon which statistics are collected must be defined in the same preamble. The variable may be defined as a global variable or as an attribute of a temporary entity, process notice, permanent entity, resource, compound entity, or the system. If the variable is a global variable or system attribute, then each statistic is correspondingly defined as a global variable or system attribute. If the variable is an attribute of an entity type, each statistic is defined as an attribute of that entity type.

Each statistic has the same dimensionality as the variable upon which it is collected, except each histogram, which has dimensionality one greater than the variable.

2.04 AddSubtract



An **add** statement adds the value of the ***Expression*** to the ***Variable***. A **subtract** statement subtracts the value of the ***Expression*** from the ***Variable***. These statements may be used in any routine. The keywords **to** and **from** are synonymous.

The statement, **add *Expression* to *Variable***, is interpreted as

```
let Variable = Variable + (Expression)
```

The statement, **subtract *Expression* from *Variable***, is interpreted as

```
let Variable = Variable – (Expression)
```

For example:

```
add M * N to Total              ' ' means Total = Total + (M * N)
subtract 100 from A(J)          ' ' means A(J) = A(J) – (100)
```

The addition and subtraction operators require operands of numeric mode: double, real, integer, integer4, integer2, or alpha. However, in an **add** statement, if the mode of ***Expression*** is text or alpha and the mode of ***Variable*** is text or alpha, then concatenation is performed instead of addition. In this case, ***Variable*** should be text, not alpha, to hold the result of the concatenation.

```
define T as a text variable
T = "abc"
add "def" to T  ' ' means T = T + ("def")
' ' T now contains "abcdef"
```

Beware of ***Variable*** having side effects since it is evaluated twice. For example, consider the following statement:

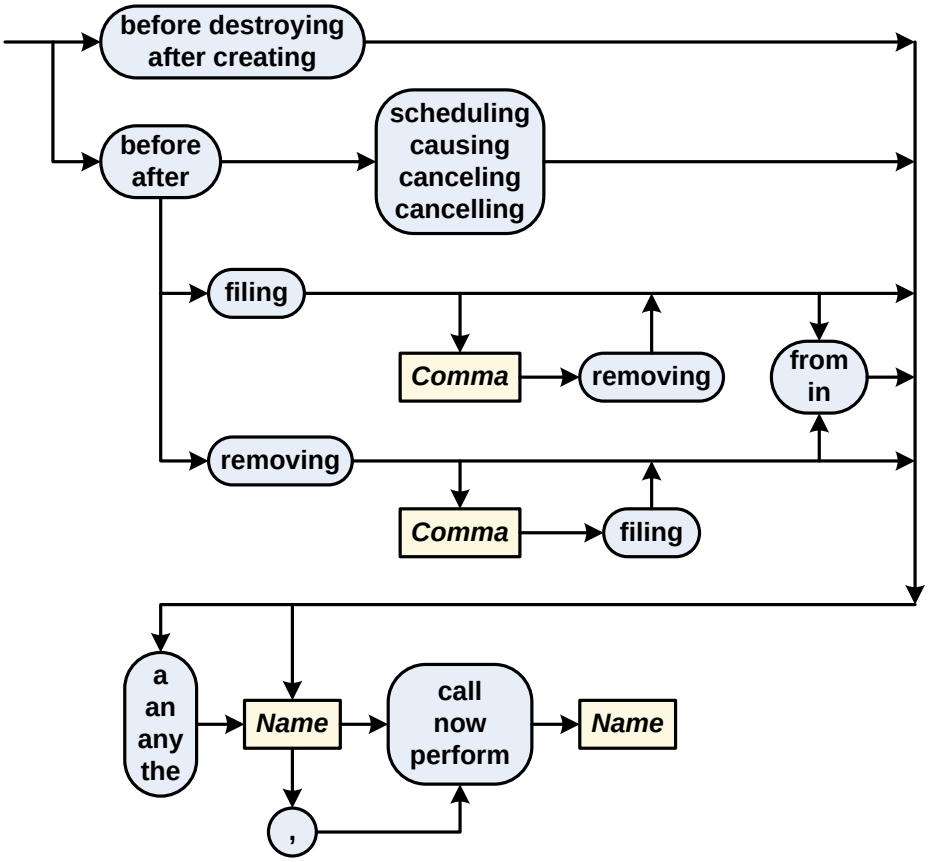
```
add 1 to Table(randi.f(1, 10, 1))
```

This statement is interpreted as

```
let Table(randi.f(1, 10, 1)) = Table(randi.f(1, 10, 1)) + 1
```

The **library.m** function **randi.f** will be called twice and return two different results.

2.05 BeforeAfter



A **before** or **after** statement specifies a routine to call immediately:

- after each object of the named class, or each temporary entity or process notice of the named entity type, is created by a **create** statement, or before each is destroyed by a **destroy** statement;
- before or after each **schedule** or **cancel** operation is performed for the named process method or process type;
- before or after each **file** and/or **remove** operation is performed on the named set.

Before and **after** statements may appear in a preamble. The keywords **a**, **an**, **any**, **from**, **in**, and **the** are optional for readability. The following are synonymous:

- **scheduling** and **causing**;
- **canceling** and **cancelling**;
- **call**, **now**, and **perform**.

Inside a "begin class" block, an "after creating" or "before destroying" statement names the class or an object method. Outside a "begin class" block, it names a temporary entity type or process type and a routine. The method or routine will be called automatically

after an object or entity is created and before an object or entity is destroyed. Whereas the method must have no explicit arguments (the reference value of the object is passed implicitly), the routine is given one argument which is the reference value of the entity.

Inside a "begin class" block, a "before/after scheduling/canceling" statement names an object process method and an object method, or a class process method and a class method. Outside a "begin class" block, it names a process type and a routine. The method or routine will be called automatically when a process method or process is scheduled or canceled. For "before/after scheduling," the method or routine accepts two given arguments; for "before/after canceling," it accepts one given argument. For both, the first argument is the reference value of the process notice representing the scheduled process method or process routine. The second argument for "before/after scheduling" is a double value representing the scheduled time of invocation.

Inside a "begin class" block, a "before/after filing/removing" statement names a set owned by an object and an object method, or a set owned by "the class" and a class method. Outside a "begin class" block, it names a set owned by a temporary entity, process notice, permanent entity, resource, compound entity, or "the system," and names a routine. The first argument to the method or routine identifies the member that is being filed or removed. This is a reference value of an object, temporary entity, or process notice, or the integer index of a permanent entity or resource. (The value is zero for a "remove first" or "remove last" statement.) If the set is owned by an object, "the class," or "the system," and the set is an array of sets, then the remaining arguments are set subscripts. If the set is owned by a temporary entity, process notice, permanent entity, or resource, then one remaining argument identifies the owner. If the set is owned by a compound entity, then two or more remaining arguments identify the owner. Note that no argument is passed to identify the preceding member in a "file after" statement or the succeeding member in a "file before" statement.

Inside a "begin class" block, the method to be called must have been defined by the class or inherited by the class. If defined by the class, the method need only appear in a "has ... method" phrase and need not be declared in a "define method" statement. Likewise, outside a "begin class" block, the routine to be called must have been defined by the module or imported by the module. If defined by the module, the routine need not be declared in a "define routine" statement. The number and modes of arguments of the method or routine are inferred by the compiler based on its appearance in a "before/after" statement. If the modes of arguments are specified by a "define method" or "define routine" statement, or by "define variable" statements within the method or routine, they must be consistent with the inferred modes.

The method or routine to be called is normally a subroutine with no yielded arguments. However, it may be a function, and if so, the function result is discarded. The method may also be a process method, and the routine may be a function attribute if it has the correct arguments.

The "before/after scheduling" method or routine is called for a "resume" statement. The "before/after canceling" method or routine is called for an "interrupt" statement.

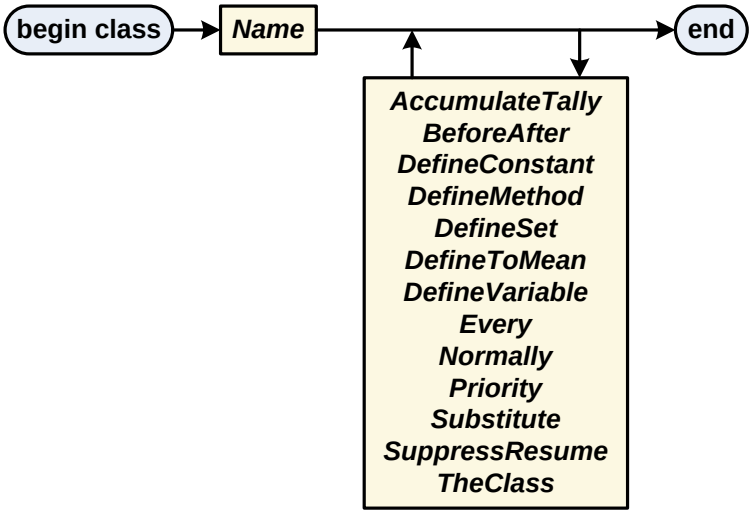
If both "after creating" and "before/after scheduling" are specified for a process type, the "after creating" routine is called before the "before/after scheduling" routine when a "schedule a" statement is executed for the process type.

When an object of a derived class is created, its "after creating" method is called after any "after creating" methods of the base classes. When an object of a derived class is destroyed, its "before destroying" method is called before any "before destroying" methods of the base classes.

The object methods specified in "before/after" statements for a base class can be overridden by a derived class.

"Before/after" methods and routines can be called directly like any other methods and routines, not just due to "before/after" actions.

2.06 BeginClass



A **begin class** block declares a class. This block may appear in a preamble.

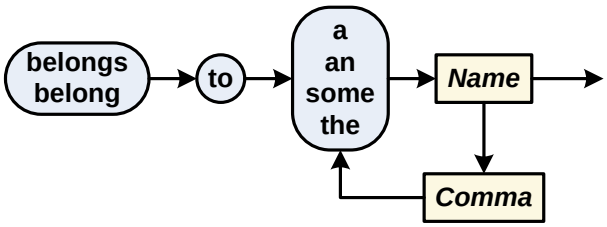
For each declared class, a reference mode is implicitly defined, its mode is "classname reference". Use of the reference mode may precede the declaration of the class.

A class is private to the main module if declared within the preamble of the main module. A class is private to a subsystem if declared only within the private preamble of the subsystem. A private class is visible only to the defining module. A class is public if it is declared within the public preamble of a subsystem.

Declaration of a class may be done partially, some parts can be private, some may be public. Private attributes, methods, sets, and base classes of the public class may be declared in the private preamble of the subsystem. Public attributes, methods, and sets are accessible to importing modules. Typically attributes of a public class are defined within the private preamble and methods defined in the public preamble provide the interface to the class.

Each class may access the attributes, methods, and sets of the other classes defined within the same module.

2.07 Belongs



A **belongs** phrase is part of an **every** statement. Inside a **begin class** block, it declares sets in which an object may be a member. Outside of a **begin class** block, it declares sets in which an entity may be a member.

The following are synonymous:

- **belongs** and **belong**;
- **a**, **an**, **some**, and **the**.

Inside a "begin class" block, a "belongs" phrase causes member attributes **p.*set_name***, **s.*set_name***, and **m.*set_name*** to be implicitly defined as 0-dimensional object attributes. The mode of **p.*set_name*** and **s.*set_name*** is the class reference mode.

Outside a "begin class" block, **p.*set_name***, **s.*set_name***, and **m.*set_name*** are implicitly defined as attributes of the temporary entity, process notice, permanent entity, or resource named in the "every" statement. If **p.*set_name*** and **s.*set_name*** are attributes of a temporary entity or process notice, they have the entity reference mode.

m.*set_name* keeps set membership information. It is zero when object instance or entity is not in a set. It is nonzero when the object or entity is in the named set. The nonzero value is an integer index or reference value identifying the owner of the set.

A set may have objects as members, temporary entities and/or process notices as members, or permanent entities and/or resources as members, but not a mixture of these.

A set named in a belongs phrase need not be mentioned in an owns phrase.

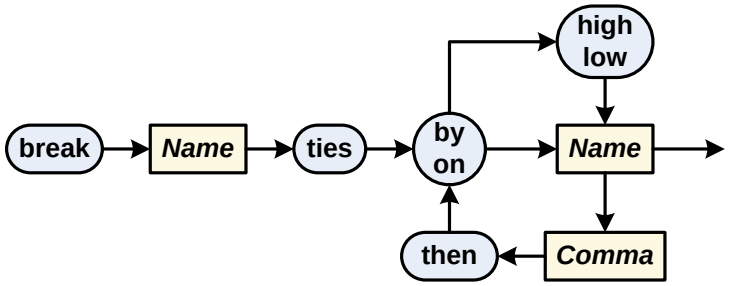
A set named in a belongs phrase defaults to fifo but this may be changed by a "define set" statement.

The name of a set of entities is global to the defining module, whereas the name of a set objects is local to the defining class.

A set of objects may contain objects of the defining class and objects of classes derived from the defining class. A derived class inherits the set attributes and the ability to be a member of each set defined by a base class.

An object or entity may belong to any number of sets.

2.08 BreakTies



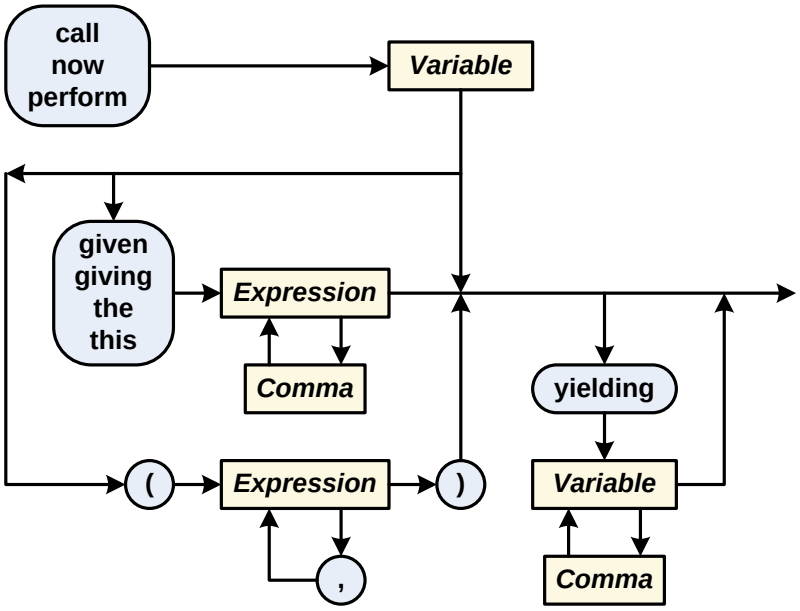
This statement refers to internal processes. Two or more invocations of a particular process routine may be scheduled for the same simulation time. A different ordering may be specified for an internal process type using a "break ties" statement. Only one "break ties" statement is permitted for each internal process type and it must follow the declaration of the process type in the same preamble. For the named process type, a **break ties** statement identifies one or more attributes of the process notice whose values will be used to determine the order of processes scheduled for the same simulation time. This statement should appear in a preamble. The keywords **by** and **on** are synonymous. If neither "high" nor "low" is specified, "high" is assumed. Ranking attributes cannot be a subprogram or reference variable.

A "schedule" statement places a new member into a ranked set according to the values of its ranking attributes: first time.a and then break ties attributes, if any. If those values need to be changed, the program should remove the member from the set (cancel), change the values, and then file the member back into the set (reschedule) so that it will be placed into the event set with correct rank ordering.

Several process methods can be scheduled for the same simulation time, and they will occur in the order in which they were scheduled, that is, first scheduled, first occurs. A "break ties" statement may not be specified for a process method, this means that this statement can not be used in a **begin class** block. A "shadow" process type can be used to break ties for process methods.

Break ties statement can not be used for a process type that is declared as "external."

2.09 Call



This statement invokes a routine, passing zero or more given arguments as inputs, and receiving zero or more yielded arguments as outputs. A routine may invoke another routine and may invoke itself recursively. The following are synonymous:

- **call**, **now**, and **perform**;
- **given**, **giving**, **the**, and **this**.

For example, suppose **Compute_Ellipse_Properties** is a subroutine with two given arguments, the length of the major and minor axes of an ellipse, and two yielded arguments, the area and circumference of the ellipse.

```
subroutine Compute_Ellipse_Properties
   given Major, Minor yielding Area, Circumference

   Area = pi.c * Major * Minor / 4
   Circumference = pi.c * sqrt.f((Major**2 + Minor**2) / 2)

end
```

The following statement invokes this subroutine. Upon entry to the subroutine, the values of **X** and **Y** are copied to **Major** and **Minor**, and upon return from the subroutine, the values of **Area** and **Circumference** are copied to **A** and **C**.

```
call Compute_Ellipse_Properties given X and Y yielding A and C
```

The **given** phrase may be replaced by a parenthesized list of given arguments. The above statement is equivalent to:

```
call Compute_Ellipse_Properties(X, Y) yielding A and C
```

Neither the **given** phrase nor the parenthesized list is specified when invoking a routine that has no given arguments. Likewise, the **yielding** phrase is omitted when invoking a routine that has no yielded arguments. For example:

```
call Write_Results                ' ' 0 given and 0 yielded arguments
call Check(N – M + 1)             ' ' 1 given and 0 yielded arguments
call Setup yielding First, Last   ' ' 0 given and 2 yielded arguments
```

The caller provides input values to a routine through given arguments and receives output values through yielded arguments. A variable specified as both a given argument and a yielded argument is used for both input and output. Upon entry to the routine, its value is provided to the routine. Upon exit from the routine, it is assigned a value provided by the routine. For example:

```
call Update given Count yielding Count
```

Suppose **Drive** is an object method of a class named **Vehicle**, and this method accepts two given arguments, the distance to travel and the average speed, and yields one argument, the duration of the trip. The following statement sends the **Vehicle** object identified by a reference variable named **Chevy** on a 200-mile trip with an average speed of 50 miles per hour:

```
call Drive(Chevy) given 200, 50 yielding Trip1_Duration
```

In the next example, we send the Chevy on a second trip, this time a 600-mile journey at an average speed of 60 miles per hour. The given arguments are expressed in parentheses.

```
call Drive(Chevy)(600, 60) yielding Trip2_Duration
```

Chevy, reference value, is passed as an implicit argument to an object method. Upon entry to the method, it is assigned to the implicitly-defined local reference variable which has the same name as the class. This reference value argument is not one of the method's given arguments. In this example, the value of **Chevy** is assigned to the implicitly-defined local reference variable named **Vehicle** upon entry to the **Drive** method.

Now suppose the **Drive** method is called within an object method of the **Vehicle** class. In this case, the reference value expression may be omitted and the implicitly-defined reference variable is implied. That is, these statements,

```
call Drive given 200, 50 yielding Trip1_Duration
call Drive(600, 60) yielding Trip2_Duration
```

are interpreted as:

```
call Drive(Vehicle) given 200, 50 yielding Trip1_Duration
call Drive(Vehicle)(600, 60) yielding Trip2_Duration
```

The caller must specify the correct number of given and yielded arguments. The modes of the caller's arguments must be compatible with the modes of the routine's arguments. A given ***Expression*** specified by the caller is assigned to the corresponding given argument within the routine. A yielded argument within the routine is assigned to the corresponding yielded ***Variable*** specified by the caller .

If a given or yielded argument is an array, only the array pointer is copied, not the entire array. If an array pointer is passed as a given argument, the elements of the array may be modified by the routine. Thus, a given array provides input to and may receive output from a routine.

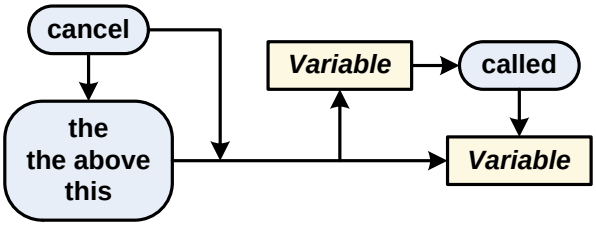
A ***Call*** statement may invoke any subroutine or a method, including process methods. It may also invoke the right implementation of a function, provided it is not a right-monitoring function; however, the function's return value is discarded. The call statement can not invoke process routine, which can only be scheduled.

A ***Call*** statement may indirectly invoke a non-method routine using a subprogram variable. No argument checking is performed by the compiler in this case. The following code indirectly calls **Compute_Ellipse_Properties**:

```
define Get_Properties as a subprogram variable
let Get_Properties = 'Compute_Ellipse_Properties'
call Get_Properties(X, Y) yielding A and C
```

Here, it is not possible to specify array subscripts after the name of the subprogram variable because a parenthesized list of expressions is assumed to contain given arguments to the routine. Hence, a subprogram variable must be scalar (i.e., 0-dimensional) to be used in a ***Call*** statement.

2.10 Cancel



This statement, which may be used in any routine, removes a process notice from the event set to cancel the pending execution of a process method or process routine. It has two forms. The keywords **the**, **the above**, and **this** are optional for readability.

1. **Cancel *Variable***. The process notice, whose reference value is in ***Variable***, is removed from the event set. The mode of ***Variable*** must be pointer or the reference mode of a process type. In the following example, **Voyage(Ship)** is an object process method, and the object attribute of the same name holds the reference value of the process notice:

   ```
   cancel Voyage(Ship)
   ```

2. **Cancel *Variable2* called *Variable***. As in Form 1, the process notice, whose reference value is in ***Variable***, is removed from the event set, and the mode of ***Variable*** must be pointer, or the reference mode of a process type. ***Variable2*** names a process method or process type is used for runtime error checking. ***Variable*** must identify a process notice associated with the named method or type. In the following example, **Rescue** holds the reference value of the process notice, which must be associated with the process method, **Voyage(Ship)**:

   ```
   cancel Voyage(Ship) called Rescue
   ```

The event set **ev.s** is an array of sets. Each process method and process type has a unique event set index. When a process notice is allocated, its **ipc.a** attribute is automatically initialized to the event set index of its process method or process type. When scheduled, the process notice is inserted into the event set at this index (see the ***Schedule*** statement). Upon removal, the number of elements in this set, is decremented by one, and zero is assigned to **m.ev.s(P)** to indicate that the process notice is no longer a member of the event set.

A process notice removed from the event set is not destroyed; the program may destroy it explicitly. It is an error to destroy a process notice that is a member of the event set; therefore, it must be removed from the event set before it is destroyed. For example:

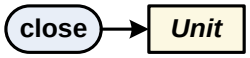
```
cancel Rescue          ' ' remove the process notice from the event set
destroy Rescue         ' ' destroy the process notice
```

It is an error to cancel a process notice that is not scheduled. Before canceling a process notice, the program can verify that it is a member of the event set. For example:

```
if Rescue is in ev.s
    cancel Rescue
always
```

A “before canceling” routine and an “after canceling” routine, if defined, are called automatically before and after each process notice is removed from the event set. These routines accept one argument, which is the reference value of the process notice. See ***BeforeAfter*** for more information.

2.11 Close



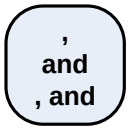
This statement, which may be used in any routine, closes the specified I/O unit. The file associated with the unit is closed and is disassociated from the unit. For example, the following statement closes unit 12:

```
close 12
```

The unit number must be in the range 1 to 99, but may not be one of the special units: 5 (standard input), 6 (standard output), 98 (standard error), and 99 (the buffer). It is an error to close a unit that is not open. If the current input unit is closed, unit 5 becomes the current input unit; if the current output unit is closed, unit 6 becomes the current output unit.

After a unit has been closed and disassociated from a file, it may be reopened and re-associated with the file or associated with a different file. When a program terminates, all open units are automatically closed, including the special units: standard input standard output, standard error, and the buffer.

2.12 Comma



This language element is used in many statements to separate consecutive elements in a sequence or list. The three choices are synonymous.

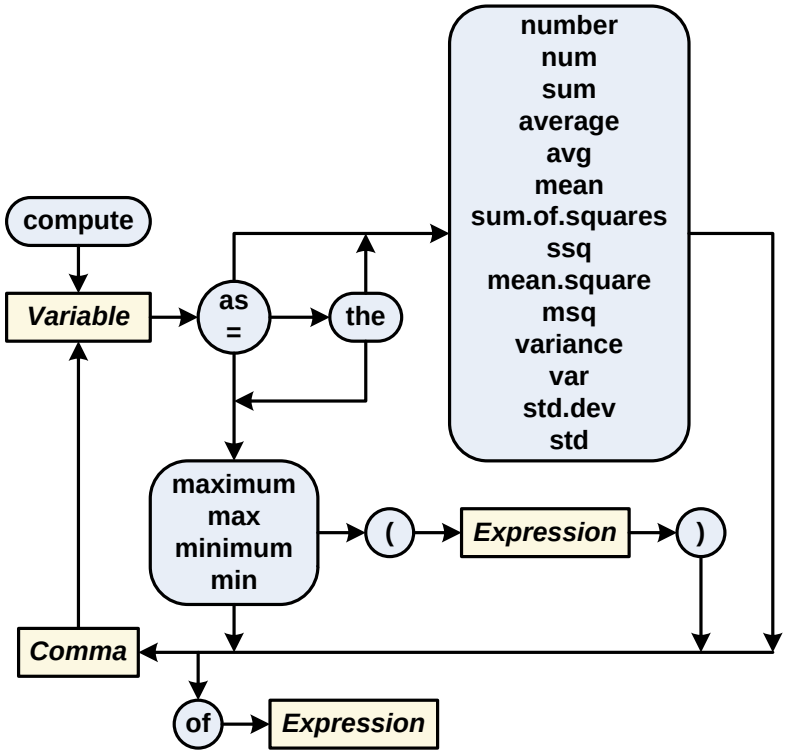
In some cases, use of a ***Comma*** is required, for example, to separate variable names in a ***DefineVariable*** statement. These statements are equivalent:

```
define X, Y, Z as double variables
define X, Y, and Z as double variables
define X and Y and Z as double variables
define X, and Y, Z as double variables
```

In other cases, a ***Comma*** is optional and is used to enhance the readability of a statement. These statements are equivalent:

```
define Count as an integer 1–dimensional saved array
define Count as an integer, 1–dimensional, saved array
```

2.13 Compute



This statement may be specified in the body of a loop. Each time it is executed, the ***Expression*** following the **of** keyword is evaluated. One or more statistics are computed based on the values obtained. Each statistic is assigned to a ***Variable*** upon termination of the loop.

The keyword **the** is optional for readability. The following are synonymous:

- **as** and **=**;
- **number** and **num**;
- **average**, **avg**, and **mean**;
- **sum.of.squares** and **ssq**;
- **mean.square** and **msq**;
- **variance** and **var**;
- **std.dev** and **std**;
- **maximum** and **max**;
- **minimum** and **min**.

In the following example, the average years of experience is computed for the sergeants in a platoon. Each time the ***Compute*** statement is executed, the expression **Experience(Soldier)** is evaluated. Upon termination of the loop, the average of these values is assigned to the variable named **Sgt_Experience**.

```
for each Soldier in Staff(Platoon)
with Rank(Soldier) = Sergeant
   compute Sgt_Experience = average of Experience(Soldier)
```

When loops are nested and end at the same location, the statistics are computed after termination of the outermost loop. In the following example, the average experience of all sergeants in the company is assigned to **Sgt_Experience**:

```
for each Platoon in Company
   for each Soldier in Staff(Platoon)
   with Rank(Soldier) = Sergeant
      compute Sgt_Experience = average of Experience(Soldier)
```

The above loop is equivalent to each of the following. Here the **loop** keyword marks the common end of the inner and outer loops.

```
for each Platoon in Company
   for each Soldier in Staff(Platoon)
   with Rank(Soldier) = Sergeant
   do
      compute Sgt_Experience = average of Experience(Soldier)
   loop

for each Platoon in Company
do
   also for each Soldier in Staff(Platoon)
   with Rank(Soldier) = Sergeant
   do
      compute Sgt_Experience = average of Experience(Soldier)
   loop
```

However, in the following example, the inner and outer loops have different endpoints. The average is assigned to **Sgt_Experience** each time the inner loop is terminated. This allows a separate average to be computed for each platoon.

```
for each Platoon in Company
do
   for each Soldier in Staff(Platoon)
   with Rank(Soldier) = Sergeant
      compute Sgt_Experience = average of Experience(Soldier)
   write Number(Platoon), Sgt_Experience as "The sergeants in platoon #", i *,
      " have an average of ", d(4,1), " years of experience", /
loop
```

Let *n* represent the number of times the ***Compute*** statement is executed, and let x_1, x_2, …, x_n denote the *n* values of the ***Expression***. The following statistics can be computed:

- *number* equals *n*
- *sum* equals $\sum_{i=1}^{n} x_i$
- *average* equals $\left(\frac{sum}{number} \right)$
- *sum.of.squares* equals $\sum_{i=1}^{n} x_i^2$
- *mean.square* equals $\left(\frac{sum.of\ .squares}{number} \right)$
- *variance* equals $\left(mean.square - average^2 \right)$
- *std.dev* (standard deviation) equals $\sqrt{variance}$

In the following example, we compute all seven of these statistics for the sergeants in a platoon. The statistics may appear in any order in the ***Compute*** statement.

```
for each Soldier in Staff(Platoon)
with Rank(Soldier) = Sergeant
   compute Number_of_Sergeants = number, Total_Experience = sum,
      Avg_Experience = average, SSQ_Experience = sum.of.squares,
      MSQ_Experience = mean.square, Var_Experience = variance, and
      SD_Experience = std.dev
         of Experience(Soldier)
```

If there are no sergeants in the platoon (i.e., $n = 0$), a value of zero is assigned to **Number_of_Sergeants**, **Total_Experience**, and **SSQ_Experience**. However, no value is assigned to the other variables because average, mean square, variance, and standard deviation are undefined.

In addition to these seven statistics, it is possible to compute the maximum and minimum of x_1, x_2, …, x_n. Here we find the maximum and minimum experience of sergeants in a platoon:

```
for each Soldier in Staff(Platoon)
with Rank(Soldier) = Sergeant
   compute Most_Experience = maximum and Least_Experience = minimum
      of Experience(Soldier)
```

If there are no sergeants in the platoon, no value is assigned to **Most_Experience** and **Least_Experience**.

A parenthesized expression may follow the **maximum** or **minimum** keyword, as in:

```
compute Variable = maximum(Expression2) of Expression
```

In this case, it is the value of ***Expression2*** when ***Expression*** is maximum that is assigned to ***Variable***. Normally ***Expression2*** names a loop control variable. With this form, we can obtain the reference values of the sergeant with the most experience and the sergeant with the least experience, rather than their number of years of experience.

```
for each Soldier in Staff(Platoon)
with Rank(Soldier) = Sergeant
    compute Most_Experienced = maximum(Soldier)
        and Least_Experienced = minimum(Soldier)
            of Experience(Soldier)
```

If two or more sergeants are tied for the most or least experience, the first sergeant encountered by the loop is the one that is identified. If there are no sergeants in the platoon, no value is assigned to **Most_Experienced** and **Least_Experienced**.

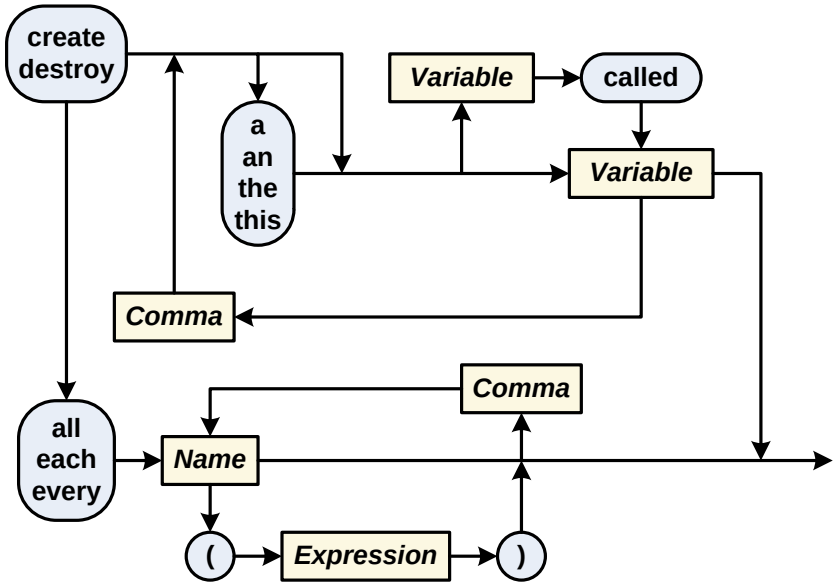
The mode of ***Variable*** and ***Expression*** must be numeric: double, real, integer, integer4, integer2, or alpha. However, if ***Expression2*** is specified, the mode of ***Variable*** and ***Expression2*** may be numeric, pointer, or reference.

The body of a loop may contain more than one ***Compute*** statement and specify conditional logic that causes one ***Compute*** statement to be executed more often than another. In the following example, we compute the average experience of sergeants, corporals, and privates, and the average experience of the entire platoon. **Sergeant**, **Corporal**, and **Private** are constants.

```
for each Soldier in Staff(Platoon)
do
    select case Rank(Soldier)
        case Sergeant
            compute Sgt_Experience = average of Experience(Soldier)
        case Corporal
            compute Cpl_Experience = average of Experience(Soldier)
        case Private
            compute Pvt_Experience = average of Experience(Soldier)
        default
    endselect
    compute Platoon_Experience = average of Experience(Soldier)
loop
```

If a loop is terminated by executing a ***GoTo*** statement that transfers control out of the loop, no values are assigned to the variables named in ***Compute*** statements.

2.14 CreateDestroy



This statement, which may be used in any routine, allocates or de-allocates storage for one or more objects, temporary entities, process notices, permanent entities, and/or resources. Upon allocation, each attribute of an object or entity is initialized to zero, except text attributes which are initialized to the null string (**""**). Attributes can be accessed after allocation and can not be accessed after de-allocation. The keywords **a**, **an**, **the**, and **this** are optional for readability. The keywords **all**, **each**, and **every** are synonymous.

This statement has seven forms:

1. **Create *Variable***. An object, temporary entity, or process notice is allocated and its reference value is assigned to ***Variable***. It is a new instance of the class, temporary entity type, or process type that is identified by the reference mode of ***Variable***. For example, suppose **Vehicle** is a class. The following **create** statement allocates a **Vehicle** type object and assigns its reference value to a variable named **Chevy**:

   ```
   define Chevy as a Vehicle reference variable
   create Chevy
   ```

   Normally, ***Variable*** has a reference mode. However, its mode may be a pointer if it is a local variable with the same name as a class, temporary entity type, or process type.

2. **Create *Variable2* called *Variable***. An object, temporary entity, or process notice is allocated and its reference value is assigned to ***Variable***. It is a new instance of the class, temporary entity type, or process type that is named by ***Variable2***. If ***Variable2*** names a class, the mode of ***Variable*** must be pointer, the reference mode of the named class, or the reference mode of a base class. If ***Variable2*** names a temporary entity type or process type, the mode of ***Variable*** must be pointer, or the reference mode of the named type. For example, the following **create** statement allocates a **Vehicle** object and assigns its reference value to a pointer variable named **Buick**:

```
define Buick as a pointer variable
create Vehicle called Buick
```

3. **Destroy *Variable***. The object, temporary entity, or process notice, whose reference value is in ***Variable***, is de-allocated. The mode of ***Variable*** must be pointer, or a reference mode. For example, suppose a variable named **Chevy** contains the reference value of a **Vehicle** object. The following statement de-allocates this object:

```
destroy Chevy
```

4. **Destroy *Variable2* called *Variable***. As in Form 3, the object, temporary entity, or process notice, whose reference value is in ***Variable***, is de-allocated, and the mode of ***Variable*** must be pointer or a reference mode. ***Variable2*** names a class, temporary entity type, process type, or process method, which is used for runtime error checking. If ***Variable2*** names a class, then ***Variable*** must identify an object of the named class or of a derived class. If ***Variable2*** names a temporary entity type, then ***Variable*** must identify a temporary entity of the named type. If ***Variable2*** names a process type or process method, then ***Variable*** must identify a process notice associated with the named type or method. In the following example, a variable named **Buick** identifies the instance to de-allocate. A runtime error occurs if this instance is neither an object of the **Vehicle** class nor an object of a class that is derived from **Vehicle**.

```
destroy Vehicle called Buick
```

5. **Create each *Name***. All entities of the named permanent entity type or resource type are allocated. The number of entities is given by the current value of the global variable **n.*Name***, which must be positive. For example, suppose **City** is a permanent entity type. The following sequence allocates 200 entities of this type:

```
let n.City = 200
create each City
```

Creating permanent entities and resources means allocating the arrays used to hold their attribute values. Suppose **Population** is an attribute of **City**. After the **create each** statement is executed, **Population** is an allocated array that is indexed

by an entity number ranging from 1 to 200. For example, the population of the fourth city is stored in **Population(4)**.

6. **Create each *Name*(*Expression*)**. All entities of the named permanent entity type or resource type are allocated. The number of entities is given by the value of ***Expression***, which must be positive. This value is implicitly assigned to the global variable **n.*Name***. The mode of ***Expression*** must be numeric, i.e., double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer. The sequence shown above for Form 5 is equivalently expressed by this single statement:

   ```
   create each City(200)
   ```

7. **Destroy each *Name***. All entities of the named permanent entity type or resource type are de-allocated. For example:

   ```
   destroy each City
   ```

   Destroying permanent entities and resources means de-allocating the arrays used to hold their attribute values.

Forms 1 and 2 may be combined within the same statement. For example:

```
define Chevy as a Vehicle reference variable
define Buick as a pointer variable
create a Chevy and a Vehicle called Buick
```

Likewise, Forms 3 and 4 may be combined within the same statement. For example:

```
destroy the Chevy and the Vehicle called Buick
```

If ***Variable*** contains zero in Forms 3 or 4, it is an error; otherwise, the identified instance is de-allocated and then zero is assigned to ***Variable***.

It is an error to destroy an object or entity that is a member of a set or is owner of a non-empty set.

When an object or temporary entity is no longer needed, an explicit **destroy** statement (Form 3 or 4) must be executed to reclaim its storage. It is important to retain its reference value so that its storage may be freed. When an object or entity is allocated by a **create** statement (Form 1 or 2), the reference value of the new instance is assigned to the named variable, which overwrites any existing reference value stored in the variable. Therefore, if the existing value has not been saved in another variable, and a **destroy** statement has not been executed using this value, then access to the instance is lost and the memory it occupies is unavailable to the program. This is known as a "memory leak."

Forms 1 and 2 may be used to allocate a process notice for a process type. Upon allocation, the **ipc.a** attribute of the process notice is automatically initialized to the event set index of the process type. This process notice may then be scheduled by a **schedule the** statement. Alternatively, the process notice may be allocated and scheduled in one step by a **schedule a** statement. Forms 1 and 2 may not be used to allocate a process notice associated with a process method, which must be allocated and scheduled at the same time by a **schedule a** statement. See ***Schedule*** for more information.

When a process routine or process method called by the timing routine has completed, the current process notice is implicitly destroyed. However, when a process is suspended, the process notice persists. If the program decides not to complete the suspended process, then an explicit **destroy** statement (Form 3 or 4) identifying the process notice must be executed to reclaim the storage used by the suspended process. If a process notice is currently scheduled, i.e., it is a member of the event set, a ***Cancel*** statement must be executed to remove the process notice from the event set before the **destroy** statement is executed.

An "after creating" routine, if defined, is called automatically for each object, temporary entity, and process notice created using Forms 1 or 2. This routine is called after the instance has been allocated and after its attributes have been initialized to zero or the null string. A "before destroying" routine, if defined, is called automatically for each object, temporary entity, and process notice destroyed using Forms 3 or 4. This routine is called before the instance has been de-allocated. "After creating" and "before destroying" routines may not be defined for permanent entity types and resource types, and so they are not applicable to Forms 5, 6, and 7. See ***BeforeAfter*** for more information.

Forms 5 and 6 may be combined within the same statement. The following sequence creates 200 **City** entities and 1000 **Taxi** entities:

```
let n.City = 200
create each City and Taxi(1000)
```

More than one permanent entity type and resource type may be specified in Form 7. For example:

```
destroy each City and Taxi
```

It is an error to execute a **create each** statement for a permanent entity type or resource type that is already allocated. A **destroy each** statement has no effect on an entity type that is unallocated.

After entities have been allocated by a **create each** statement, changing the value of **n.*Name*** does not change the number of entities. It is necessary to first de-allocate the existing entities and then execute a **create each** statement with the new value of **n.*Name***. For example:

```
' ' not enough taxis
destroy each Taxi
let n.Taxi = 2 * n.Taxi  ' ' double the number of taxis
create each Taxi
```

Creating compound entities means allocating the multi-dimensional arrays used to hold their attribute values. They are created implicitly after all constituent permanent entities and resources have been created. For example, suppose **Rate** is an attribute of a compound entity which has **City** and **Taxi** as its constituent entities:

```
every City, Taxi has a Rate
```

After the following statement is executed,

```
create each City(25) and Taxi(80)
```

a 25 × 80 array is implicitly allocated for each attribute of the compound entity. The rate charged by the third taxi in the second city is stored in **Rate(2, 3)**.

Destroying compound entities means de-allocating the multi-dimensional arrays used to hold their attribute values. They are destroyed implicitly once any of its constituent entities is destroyed. The following statement destroys all **Taxi** entities and all compound entities in which **Taxi** is a constituent entity. It does not destroy the **City** entities.

```
destroy each Taxi
```

2.15 Cycle

This statement may be specified in the body of a loop and terminates the current iteration of the loop, and the loop continues. The keywords **cycle** and **next** are synonymous.

For example, the following loop reads **N** positive values and stores them in an array named **Parameter**. For each zero or negative value that is entered, an error message is displayed and the value is ignored.

```
let J = 1

while J <= N
do
    write J as "Enter parameter ", i *, ": ", +
    read Value
    if Value <= 0
        write as "The value must be positive.  Please re–enter.", /
        cycle
    otherwise
    let Parameter(J) = Value
    add 1 to J
loop
```

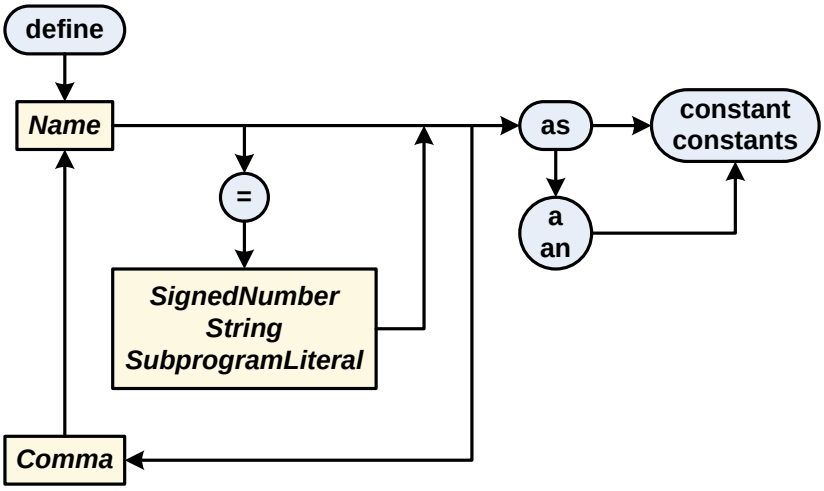
A ***Cycle*** statement behaves like a branch to a hidden label that immediately precedes the **loop** keyword. The above example can be rephrased as follows:

```
let J = 1

while J <= N
do
    write J as "Enter parameter ", i *, ": ", +
    read Value
    if Value <= 0
        write as "The value must be positive.  Please re–enter.", /
        go to Hidden_Label
    otherwise
    let Parameter(J) = Value
    add 1 to J
    'Hidden_Label'
loop
```

When used within the body of nested loops, a ***Cycle*** statement terminates the current iteration of the innermost loop. For example, the following loop writes the name of each soldier in each platoon and does some additional processing for sergeants. After the ***Cycle*** statement is executed, the innermost control variable (**Soldier**) is assigned its next value (the reference value stored in **s.Staff(Soldier)**).

```
for each Platoon in Company
    for each Soldier in Staff(Platoon)
    do
        write Name(Soldier) as t *, /
        if Rank(Soldier) <> Sergeant
            cycle  ' ' nothing more to do for this soldier
        otherwise
        ' ' additional processing for sergeants
        add 1 to Number_of_Sergeants
        …
    loop
```

2.16 DefineConstant



A **define constant** statement declares one or more named constants. This statement may appear in a preamble or routine. The keywords **a** and **an** are optional for readability. The keywords **constant** and **constants** are synonymous.

If the value of a named constant is unspecified, it is assigned the integer value that is one greater than the value of the preceding integer constant in the statement, or assigned a value of one if there is no preceding integer constant.

In the following example, the constants named **F**, **D**, **C**, **B**, and **A** represent letter grades and are assigned values zero through four:

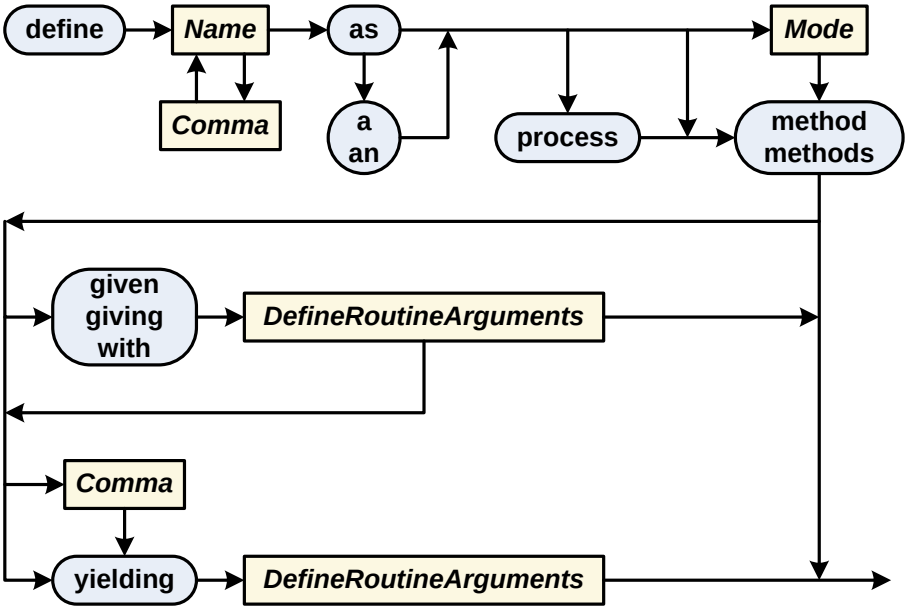
```
define F = 0, D, C, B, A as constants
```

Constants **Idle**, **Busy**, and **Terminated** are given values one to three:

```
define Idle, Busy, and Terminated as constants
```

Named constants declared in the preamble are global, i.e., they are accessible to every routine in the module. Named constants declared in a public preamble of a module are accessible to importing modules. Named constants declared in a routine are local, i.e., they are accessible only within the declaring routine.

2.17 DefineMethod



A **define method** statement describes the given and yielded arguments for one or more methods. If the methods are functions, the mode of the functions must also be specified and yielded arguments are not permitted. This statement may appear in a **begin class** block.

The keywords **a** and **an** are optional for readability. The following are synonymous:

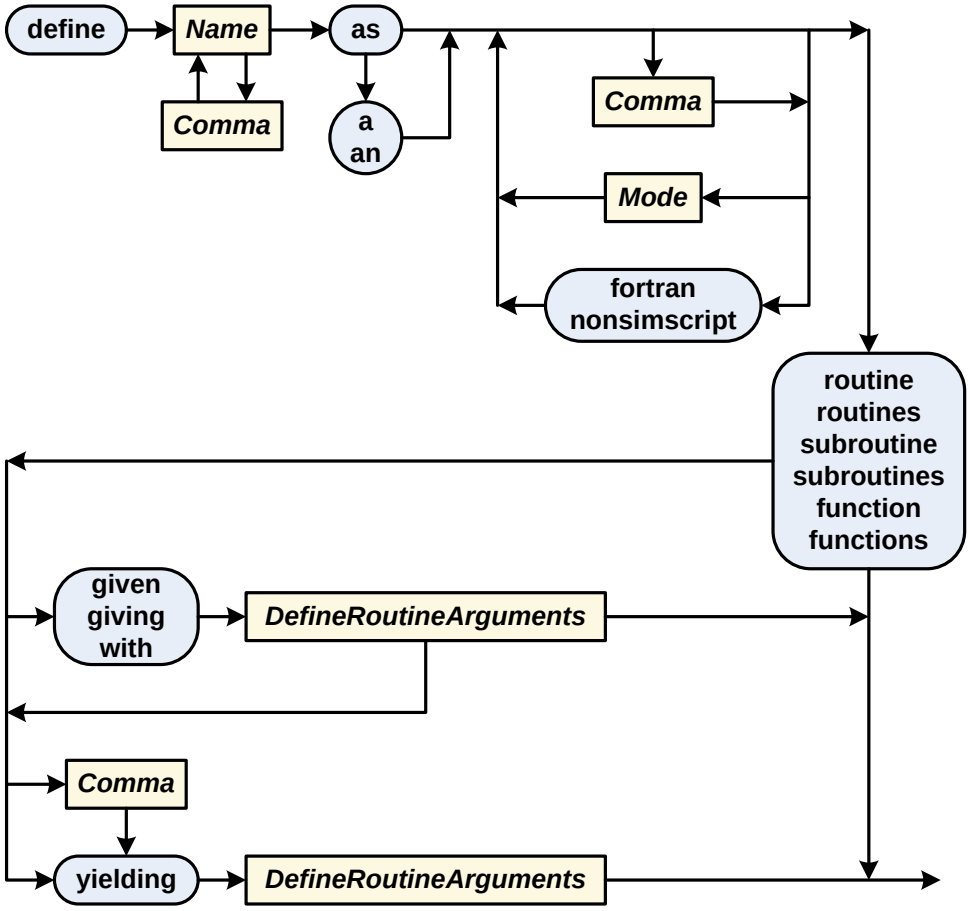
- **method** and **methods**;
- **given**, **giving**, and **with**.

This statement must be in the same begin class block as the "has ... method" phrase. If omitted, the method is assumed to be a subroutine with no arguments.

If a function result mode is not specified, the method is a subroutine.

A "define method" statement may be specified for an inherited overridden method. Covariant return mode and yielded arguments and covariant given arguments are supported.

2.18 DefineRoutine



A **define routine** statement describes the given and yielded arguments for one or more routines that are not methods. If the routines are functions, the mode of the functions may also be specified and yielded arguments are not permitted. This statement may appear in a preamble, but may not appear in a **begin class** block. A routine can be defined in only one "define routine" statement.

The keywords **a** and **an** are optional for readability. The following are synonymous:

- **routine**, **routines**, **subroutine**, **subroutines**, **function**, and **functions**;
- **given**, **giving**, and **with**.

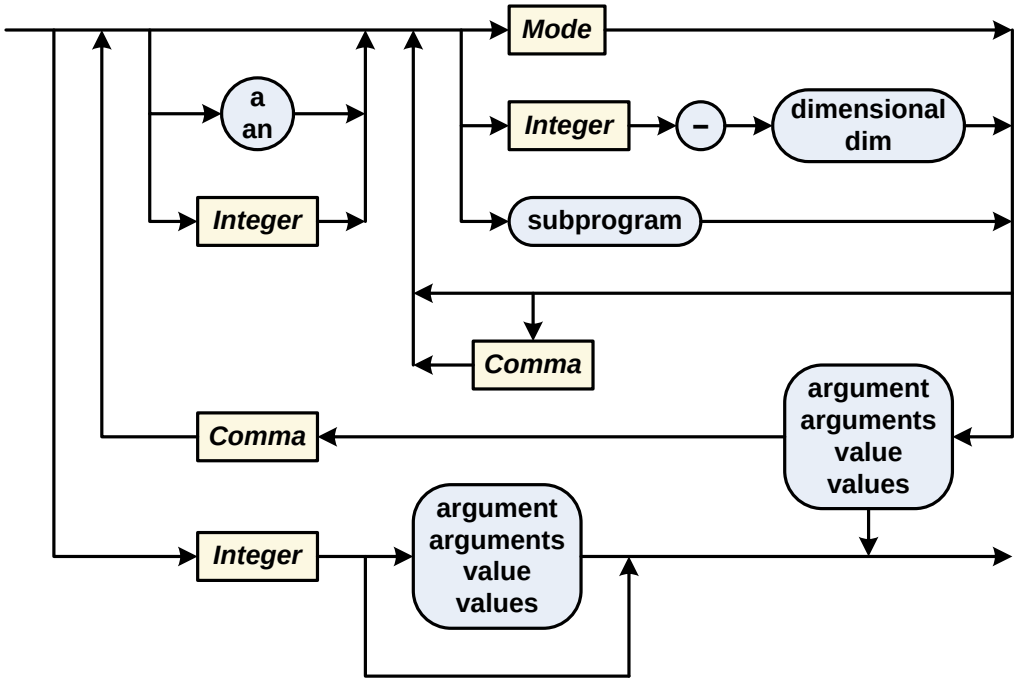
Each function must be declared in a preamble. A public subroutine must be declared in a public preamble; otherwise, subroutines need not be declared in a preamble, but it is recommended to declare them.

A function that returns an array must be declared as pointer.

If a mode of the routine is unspecified and the background mode is not undefined, then the routine is a function having the background mode. If the mode is unspecified and the background mode is undefined, the routine is a subroutine.

A subsystem routine is private unless it is declared in the subsystem's public preamble by a "define routine" statement. A private subroutine may optionally be declared by a "define routine" statement in the private preamble.

2.19 DefineRoutineArguments



This language element is used to specify the number, mode, and dimensionality of given and yielded arguments in a **define method** or **define routine** statement.

The keywords **a** and **an** are optional for readability. The following are synonymous:

- **dimensional** and **dim**;
- **argument**, **arguments**, **value**, and **values**.

It is not possible to define a routine with a variable number of arguments.

If the number of arguments is not specified, then no argument checking is done for calls of non-method routines. For methods, if the number of arguments is not specified, it is assumed to be zero.

If a routine is defined with only given arguments, it is assumed to have no yielded arguments. If a routine is defined with only yielded arguments, it is assumed to have no given arguments.

Given and yielded arguments can be arrays. A pointer to the array, and not the array elements, is passed. A given argument that is an array can have its elements changed by the routine.

No argument checking is performed for calls using a subprogram variable.

Arguments declared as real are treated as double.

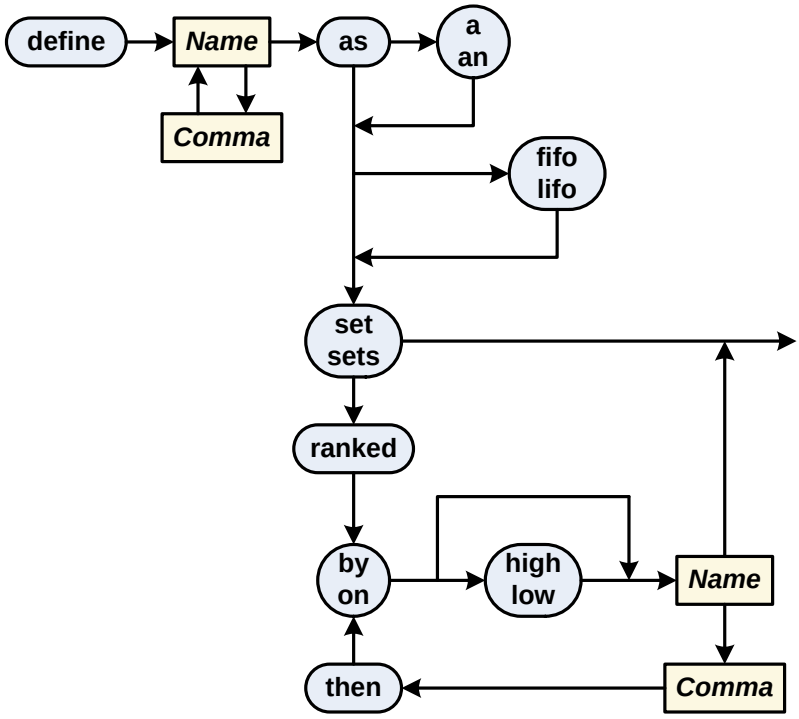
The order of arguments is important in the declaration.

In some cases, the mode of arguments can be inferred by the compiler, such as the mode of arguments to function attributes, monitoring routines, and before/after routines.

When the mode and dimensionality of a routine's arguments have been declared in a "define routine" or "define method" statement, it is not necessary to define the mode and dimensionality of the arguments within the routine implementation. If they are defined within the routine implementation, their definitions must agree with the definitions in the "define routine" or "define method" statement.

When overriding inherited methods, a class may define the reference modes of return values and yielded arguments to be subclasses of the original reference modes, and may define the reference modes of given arguments to be super-classes of the original reference modes.

2.20 DefineSet



A **define set** statement specifies the order of members in one or more sets to be **fifo** (first-in first-out, which is the default), **lifo** (last-in first-out), or **ranked** based on the values of one or more member attributes. The specified order determines the position of a new member added to the set by a **file** statement. For a **fifo** set, a **file** statement is treated as **file last**; for a **lifo** set, a **file** statement is interpreted as **file first**; and for a **ranked** set, a **file** statement inserts the new member in rank order. A **define set** statement may appear in a preamble.

The keywords **a** and **an** are optional for readability. The following are synonymous:

- **set** and **sets**;
- **by** and **on**.

A set is fifo if no "define set" statement is specified for it.

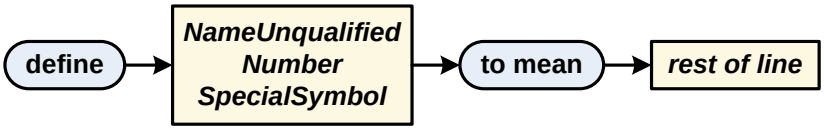
In a ranked set, "high" is assumed if neither "low" nor "high" is specified. A function attribute can be used for ranking. If two or more members are tied on all ranking attributes, they are filed in first-in first-out order.

A "define set" statement may be specified in a "begin class" block and must refer to a set named in a "belongs" phrase in the same block.

In a ranked set of objects, the ranking attributes may be defined or inherited by the member class. The ranking attributes can be any 0-dim object attributes and object function methods with no arguments.

If a set of entities has common members, then each member type must have each of the ranking attributes.

2.21 DefineToMean



A **define to mean** statement declares a source code substitution. Each occurrence of the unqualified name, number, or special symbol that follows in the source code will be replaced by the characters that appear after **to mean** on the current source line. This statement may appear in a preamble or routine.

If "define X to mean Y" is used, only standalone occurrences of X will be replaced. It will not replace "XRAY" with "YRAY".

The scope of a "define to mean" depends on where it was used. If it appears in a preamble it has global scope and applies after its definition in the preamble and then to every routine. If it appears in the routine it has local scope, i.e. it applies after its definition in a routine and only until the end of the routine. If it appears in the "begin class" block it has local class scope, it applies after its definition in a "begin class" block and only until the end of the block.

To change a substitution, "suppress substitution; define Word to mean NewThing; resume substitution".

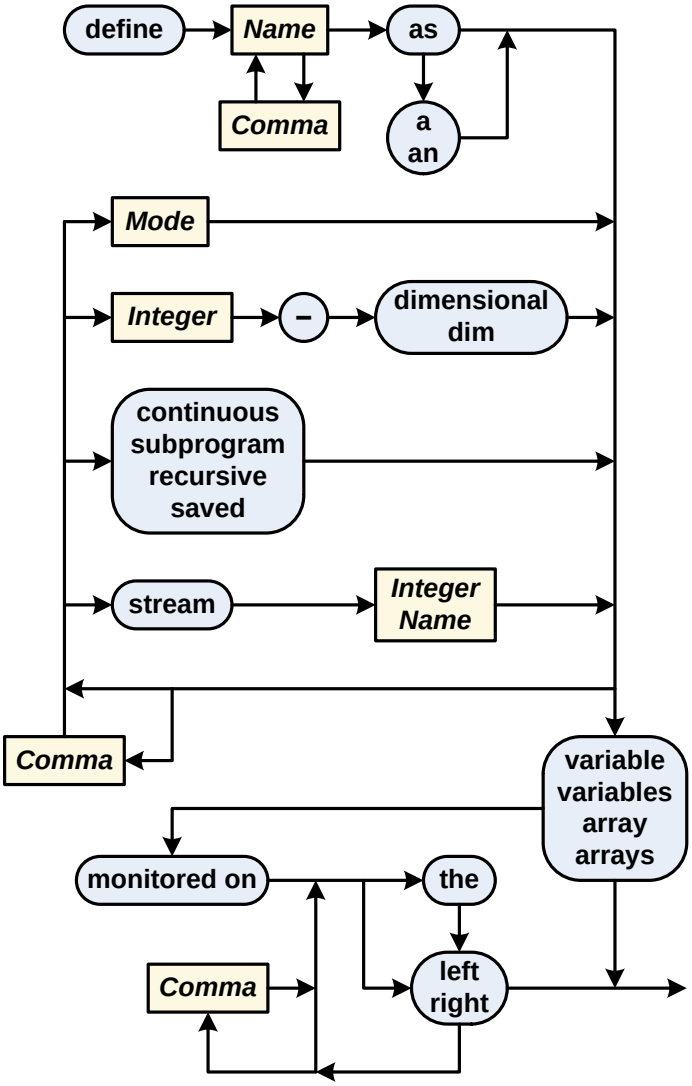
Substitutions can be used to change the vocabulary of the language. Care must be taken with statements of the form "define A to mean X" since "A" is used in the syntax of various statements.

A single word can be substituted by one or more statements.

Substitutions declared in a public preamble affect the interpretation of the public preamble source code but are not imported. Substitutions in effect at the end of the public preamble are in effect at the beginning of the private preamble, and those in effect at the end of the private preamble apply to the routines of the subsystem.

A "define to mean" can be used to define an equivalent unqualified name for a long qualified name.

2.22 DefineVariable



A **define variable** statement may be specified in a preamble to declare one or more attributes and global variables and in a routine to declare one or more local variables and arguments.

The keywords **a**, **an**, and **the** are optional for readability. The following are synonymous:

- **dimensional** and **dim**;
- **variable**, **variables**, **array**, and **arrays**.

Each variable is initialized to zero, except text variables which are assigned the null string **""**. A recursive local variable is initialized to zero for each call of the routine,

whereas a saved local variable is initialized to zero at the beginning of the program and retains its value from one call of the routine to the next. All recursive instantiations of a routine share the same group of saved local variables, but have separate recursive local variables. Arguments are recursive local variables and may not be declared as saved. The variable holding the pointer to a local array is treated as saved or recursive.

It is necessary to define the mode of a variable if it differs from the background mode or if the background mode is set to "undefined".

It is necessary to define the dimensionality of a variable if it differs from the background dimensionality.

It is necessary to define the type (recursive or saved) of a local variable if it differs from the background type.

An array is dynamic structure which elements can be accessed using indexes. They can be 1-dim, 2-dim, etc. All elements of an array have the same mode.

Each array has a pointer to the array as it is structured in memory. This pointer is accessed by simply using the name of the array, as in X, or X(*), or X(*,*), etc.

A variable should be defined by one define variable statement.

A global variable is defined in a preamble; a local variable is defined in a routine. The names of local variables within a routine have local scope and must be unique; however, the same names can be used for local variables in different routines and refer to different variables. A local variable can be defined with the same name as a global variable; however, the global variable is accessible using its qualified name.

The "define variable" statement may appear anywhere in a routine, and may even appear within a do...loop block.

A monitored variable has both a storage location and a function associated with it having the same name and mode, with left and/or right implementations. The function has as many integer arguments as the dimension of the monitored variable. A monitored variable defined as "real" is treated as "double." Local variables cannot be monitored. A function object method is defined for a monitored object attribute, and a function class method is defined for a monitored class attribute.

A subprogram variable holds the address of a non-method subroutine or right function, or contains zero (the initial value) to indicate that it does not refer to any routine. It may refer to routines in library.m. A 0-dim subprogram variable may be called using a Call statement.

A random variable uses stream 1 unless a different stream is specified in a "define variable" statement. A stream variable may be specified in the "define variable"

statement. It must be a 0-dim system/subsystem attribute, global variable, or class attribute, or integer function or class method with no arguments? A random variable declared as real is treated as double. An array of random variables can be declared, except in temporary entities. A random variable cannot be declared as monitored. A stream number can be given by a named constant.

An attribute of a temporary entity cannot be declared as an array. However, it may be declared as pointer, to which an array pointer may be assigned.

If a variable is not declared by a "define variable" statement, and there is no implicit definition of the background mode, the compiler will report an error. It is suggested that every variable be defined and the background mode be undefined. This can be achieved with "normally mode is undefined" statement in the preamble.

A "define variable" statement must follow "every" and "the system" statements whenever the "define variable" statement includes attributes named in either statement.

A "define variable" may be specified in a "begin class" block and must follow the declaration of an object (class) attribute in an "every" ("the class") statement.

An object attribute or class attribute can be declared as random variable.

When the mode and dimensionality of a routine's arguments have been declared in a "define routine" or "define method" statement, it is not necessary to define the mode and dimensionality of the arguments within the routine implementation. If they are defined within the routine implementation, their definitions must agree with the definitions in the "define routine" or "define method" statement.

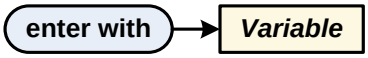
A 0-dim variable that has a reference mode is called a "reference variable." It can hold the "reference value" or address of an entity or object. A value of zero means it does not refer to any entity or object. Reference variables are automatically initialized to zero.

Except for local variables, any double variable can be declared as "continuous", including arrays and monitored variables.

2.23 Digit

This language element is a decimal digit. It may appear in an ***Integer***, ***Name***, ***Number***, or ***String***, and in a comment or keyword (e.g., **integer2**).

2.24 Enter



This language element is a special kind of assignment statement that may appear only with the left implementation of a function. A left implementation is invoked when a value is assigned to a function or to a variable or attribute that is monitored on the left. An ***Enter*** statement is used to set this assigned value.

For example, suppose **Duration** is an object method of a class named **Job**:

```
begin class Job

    every Job has a Duration method
    define Duration as a double method

end
```

If **Duration** has a left implementation, then a value can be assigned to it:

```
define Repair as a Job reference variable
create Repair

let Duration(Repair) = 3.75
```

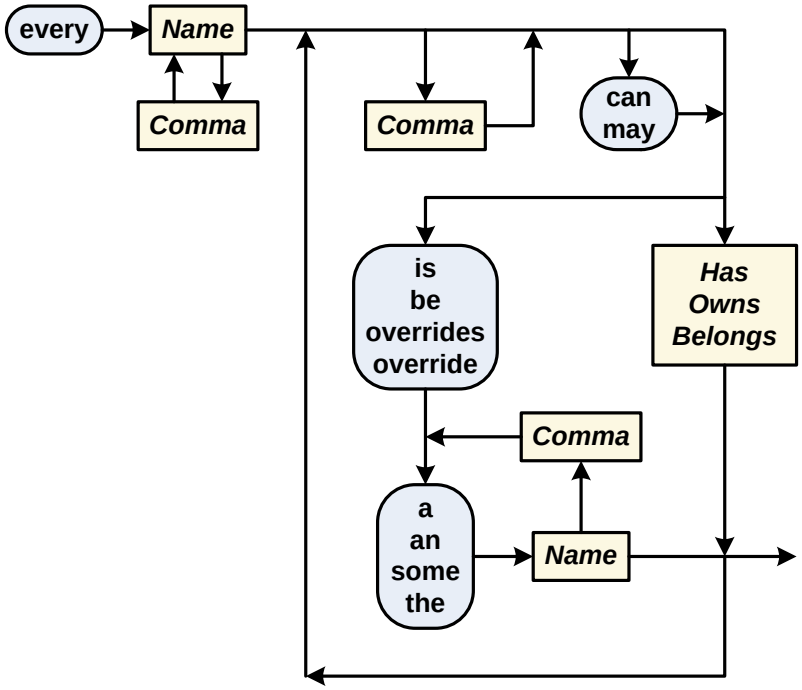
The left implementation shown below uses an ***Enter*** statement to set the assigned value, 3.75 to the local variable named **Hours**.

```
left method Job'Duration

    define Hours as a double variable
    enter with Hours  ' ' set assigned value
    …

end
```

Normally a left implementation specifies an ***Enter*** statement as the first executable statement of the routine; however, this is not a requirement. It may contain one or more ***Enter*** statements appearing anywhere within the routine. It may contain zero ***Enter*** statements if the value assigned to the function is to be ignored.

An ***Enter*** statement in a left monitoring function is typically followed by a **move from** statement which stores a value in the monitored variable or attribute. See page 126 for more information.

2.25 Every



An **every** statement appearing in a **begin class** block may specify the following for an object of the class:

- attributes and methods (***Has***);
- sets owned by the object (***Owns***);
- sets in which the object may be a member (***Belongs***);
- base classes (**is**);
- inherited attributes and methods overridden by the class (**overrides**).

An **every** statement appearing in a preamble, but not in a **begin class** block, declares an entity type, which may be a temporary entity, process, permanent entity, resource, or compound entity. This statement may specify the following for an entity of this type:

- attributes (***Has***);
- sets owned by the entity (***Owns***);
- sets in which the entity may be a member (***Belongs***).

The keywords **can** and **may** are optional for readability. The following are synonymous:

- **is** and **be**;
- **overrides** and **override**;
- **a**, **an**, **some**, and **the**.

For every entity type, and a class there is an implicit definition of global variable with same name as the entity type. Its mode is "entityname reference" for temporary entities; its mode is integer for permanent entities. Also implicit definition of n.entity global variable for permanent entities. "i.entity" is a global variable generated for process types.

A compound entity is indicated by two or more entity/class names specified after the "every" keyword. A compound entity may be declared only outside a "begin class" block. There can be no "belongs" phrases, but there may be "has" and "owns" phrases which describe attributes of and sets owned by the compound entity. The constituent entities of a compound entity must be defined in the same module as the compound entity.

An every statement cannot be specified privately for a public entity.

Base classes are specified by "is a" phrases within a "begin class" block. Inheritance conforms to the following rules:

- Each object of the derived class has its own copy of each object attribute of each base class.
- A method of the derived class may refer to each class attribute of each base class; however, there is only one copy of each class attribute in the program.
- Each object method of each base class may be invoked on behalf of an object of the derived class.
- Each class method of each base class may be invoked by a method of the derived class without qualification of the class method name.
- Each set of objects of a base class may contain objects of the derived class.
- Each object of the derived class owns each set that is owned by an object of a base class.
- A derived class cannot alter the definition of an inherited attribute, method, or set.
- Cyclic inheritance is not permitted, as in "every A is a B; every B is a C; every C is an A".

A derived class may override an inherited object method (or monitored object attribute or unmonitored numeric object attribute) and supply its own implementation(s) of the object method. Invoking the method using a base class reference variable that contains a derived class reference value invokes the derived class's implementation of the method. The derived class's implementation may invoke the base class's implementation to modify or extend its behavior. A derived class that overrides an inherited monitored object attribute may provide left, right, or both implementations of the monitoring method. A derived class may provide a left monitoring method to override an inherited unmonitored numeric attribute.

A derived class may override an inherited object process method. If the execution of the method is scheduled using a base class reference variable that contains a derived class reference value, the derived class's implementation is scheduled for execution. It may in turn call or schedule the execution of the base class's implementation.

All inherited programmer-defined object methods may be overridden. Inherited class methods cannot be overridden. It is not possible to override a private inherited object method.

Suppose class D is derived from base classes B and C, and that class A is a base class of both B and C. That is, "every D is a B and a C", "every B is an A", and "every C is an A". This is the well-known "diamond-shaped" inheritance. There is only one occurrence of A's object attributes in a D object. If both B and C override an object method named M inherited from A, then D must override M.

The name of an inherited attribute, method, or set in a derived class is the same as its name in the base class. However, if the name is inherited from more than one base class, it must be qualified by the name of the base class. If the derived class defines a name that it has inherited, then this definition is distinct from any inherited definitions of the name and the unqualified name refers to the derived class's definition.

If outside a "begin class" block, an "every" statement defines a temporary entity, process, permanent entity, resource, or compound entity based on the preceding heading. If no preceding heading, temporary entities is assumed.

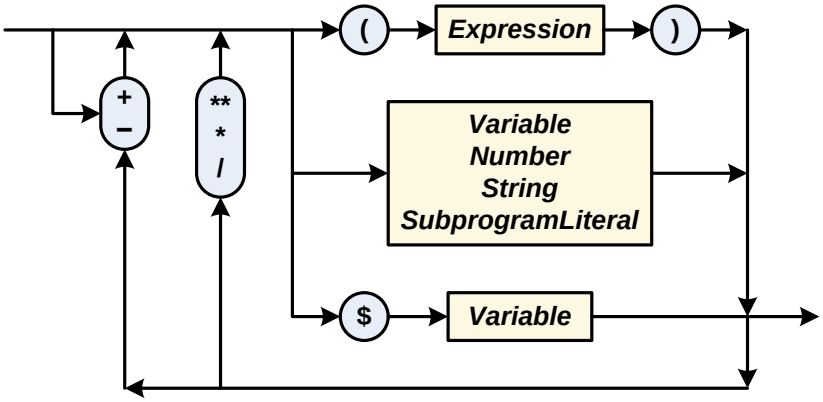
In single inheritance, a class is derived from one base class. In multiple inheritance, a class is derived from two or more base classes. If "every Z is a Y" and "every Y is a X", then it is optional to declare that "every Z is a X".

A derived class inherits the object attributes of each of its base classes. This means that an object of a derived class has a copy of each object attribute defined or inherited by its base classes. In addition, the derived class may define object attributes of its own. If the name is inherited from more than one base class, it must be qualified by the name of the base class. If the derived class defines a name that it has inherited, then this definition is distinct from any inherited definitions of the name and the unqualified name refers to the derived class's definition.

A derived class inherits the object methods of each of its base classes. This means that each object method defined or inherited by its base classes may be invoked on behalf of an object of the derived class. In addition, the derived class may define object methods of its own.

A derived class cannot alter the definition of an inherited object attribute or object method. A derived class may define an attribute or method having the same name as an inherited attribute or method, but it does not replace or change the inherited attribute or method. The result is that the derived class has two definitions of the name, one defined by the class and the other inherited from a base class. When a name has been inherited from two or more base classes, or has been defined by the derived class and inherited from one or more base classes, each inherited definition must be accessed using its qualified name.

2.26 Expression



This language element appears in many executable statements. It is evaluated when a statement is executed. The resulting value is then used in the statement.

If an ***Expression*** is a single ***Variable***, ***Number***, or ***String***, then its value is the current value of the named variable or attribute, the value returned by a function, or the value indicated by a constant. For example:

Cost	**' '** ***Variable***
16.25	**' '** ***Number***
"Yes"	**' '** ***String***

If two such elements are joined by an operator, the result of the operation becomes the value of the ***Expression***. The following operators are permitted:

Precedence Level	*Operator*	*Result Mode*	*Usage*
3	exponentiation ****	double	**X ** Y**
2	multiplication *	integer or double	**X * Y**
	real division /	double	**X / Y**
1	concatenation +	text	**X + Y**
	addition +	integer or double	**X + Y**
	subtraction –	integer or double	**X – Y**
	unary +	integer or double	**+X**
	unary –	integer or double	**–X**

The concatenation operator produces a text result by concatenating the values of one text or alpha operand with another. For example, if a text variable **T** is equal to **"State"**, then the value of **T + "s"** is **"States"**. The expression **T + "s"** is shorthand for **concat.f(T, "s")**.

If an integer variable **N** is equal to 3,

- the value of **2 ** N** is 8.0;
- the value of **N * N** is 9;
- the value of **7.5 / N** is 2.5;
- the value of **N + 7** is 10;
- the value of **0.5 – N** is –2.5;
- the value of **+N** is 3; and
- the value of **–N** is –3.

These arithmetic operators require operands of numeric mode: double, real, integer, integer4, integer2, or alpha. For the exponentiation and real division operators, the mode of the result is double. For the other operators, the mode of the result depends on the modes of the operands. If one or both of the operands are double or real, then the mode of the result is double; otherwise, the mode of the result is integer. If both operands of **+** are alpha, concatenation is performed, not addition.

Integer division and modulus operators are available in **library.m**. Given integers **X** and **Y**, **div.f(X, Y)** returns the integer quotient of **X** divided by **Y**, and **mod.f(X, Y)** returns the integer remainder. For example, **div.f(26, 6)** returns 4 and **mod.f(26, 6)** returns 2.

Multiple operators may be combined in an ***Expression***. For example:

```
Basic_Fare + 0.62 * Miles – Discount / 10
a * x ** 2 + b * x + c
```

Such expressions are evaluated according to the operator precedence rules. Exponentiation operators have the highest precedence and are evaluated first. Multiplication and real division operators are evaluated next, followed by the others. When operators have equal precedence, they are evaluated from left to right. The above statements are therefore evaluated as if they included the following parentheses, with the innermost parentheses evaluated first:

```
(Basic_Fare + (0.62 * Miles)) – (Discount / 10)
((a * (x ** 2)) + (b * x)) + c
```

These parentheses are optional and may be used to enhance readability. However, if the default order of evaluation is not desired, parentheses must be specified to indicate the desired order. For example:

```
(Basic_Fare + 0.62) * (Miles – Discount) / 10
(a * x) ** ((2 + b) * (x + c))
```

No two operators can appear consecutively. The expressions **A + (–B)** and **A – B** are valid, but **A + –B** is not. The two asterisks of the exponentiation operator **** must be consecutive with no intervening space.

An ***Expression*** may be a ***SubprogramLiteral*** representing a routine, and may be assigned to a subprogram variable which can be used to call the routine indirectly. To call a

function using this variable, it is necessary for the variable and function to have the same mode. Suppose **Area** is a double function that accepts two arguments. A subprogram variable named **Solve** can refer to this function:

```
define Solve as a double subprogram variable
define Result as a double variable
let Solve = 'Area'
let Result = $Solve(X, Y)
```

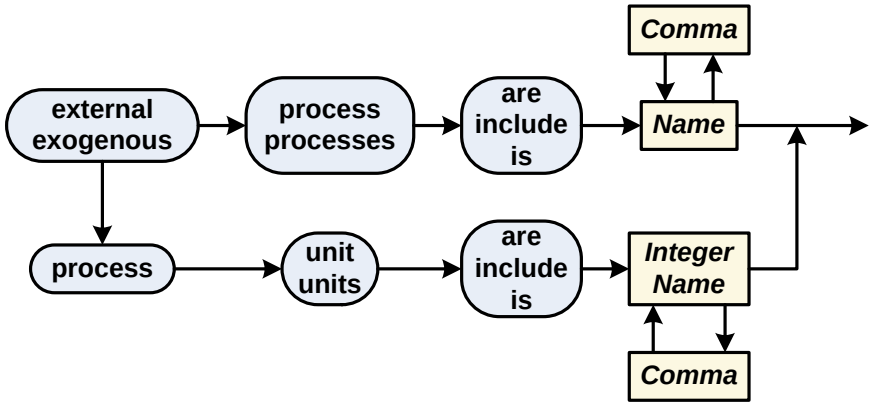
The expression **$Solve(X, Y)** is an indirect call of function **Area** with given arguments **X** and **Y**. The value returned by the function is the value of the expression.

Note that **Solve** must be a scalar (0-dimensional) subprogram variable in this context because the parenthesized expressions are interpreted as given arguments to the function and not as subscripts in a subprogram array. If the address of the **Area** function is stored in the third element of a subprogram array named **Measure**, this element must be assigned to a scalar subprogram variable which can be used to call the function.

```
define Measure as a 1–dimensional double subprogram array
let Measure(3) = 'Area'
…
let Solve = Measure(3)
write $Solve(X, Y) as "The solution is ", d(8, 2), /
```

No argument checking is performed by the compiler on the arguments passed in a subprogram variable call. The programmer must verify that the number and mode of arguments match the routine's argument definitions.

2.27 External



An **external** statement declares one or more process types to be external or declares one or more input units from which external processes are to be read during a simulation. This statement may appear in a preamble, but may not appear in a **begin class** block.

The following are synonymous:

- **external** and **exogenous**;
- **process** and **processes**;
- **unit** and **units**;
- **are**, **include**, and **is**.

A common validation technique used in simulation modeling is to exercise a model using event data derived from a record of events occurring in the system under study. This is termed **trace-driven simulation**. Alternatively, a collection of projected event times may be used to study the behavior of the modeled system.

To support this technique, Simscript provides a mechanism by which, rather than scheduling events using statements within a program, they may be scheduled directly from event times presented as an input data stream.

It is possible for processes to belong to one or both of two categories: internal or endogenous, and external or exogenous. Processes may be triggered from external input data by declaring them to be external processes. A process type cannot appear in more than one "external processes" statement.

For each process type declared as external, provision is made to create a process notice each time a data record containing the process type name appears on an external process unit. The process notice attribute named "eunit.a" contains the number of the external unit from which the external process was read. If a suspended process was external, its

eunit.a is changed automatically to zero so that when it is awakened, its eunit.a is zero indicating an internally scheduled resumption.

If no "external process units" statements are specified, the standard input unit is assumed to be a source of external process data. If "external process units" statements are specified and the standard input unit is intended to be one of them, then the standard input unit must be specified in one of the statements. External process units may not contain binary data. Any external process may be triggered by data read from any of the external process units. External process units may not be shared among modules.

External processes are read in chronological order (i.e., increasing time.a) from external process units. Records for different process types may be interspersed. Each record contains the name of a process type, the time at which it is to occur, and optional data which can be continued on subsequent lines. The data is terminated by an occurrence of the mark.v character (asterisk by default). The name and time are read by free-form read statements. Optional data may be in any programmer-defined character format. For each record, a process notice is created of the named type, time.a is set to the specified time, and eunit.a is set to the number of the external unit. The process notice is filed in the event set. Internally and externally generated process notices are filed together. They are distinguished by their eunit.a values.

The time at which an external process is to occur can be specified in three formats:

(i) a single floating-point value expressing an absolute time (e.g., 7.56)

(ii) three integer values specify the day, hour, and minute (e.g., 2 22 51); the value "0 0 0" represents the start of simulation; hours are numbered from 0 to 23 and minutes from 0 to 59 (affected by hours.v and minutes.v?)

(iii) a calendar day followed by hour and minute as integers (e.g., 1/15/97 5 35 or 1/15/1997 5 35)

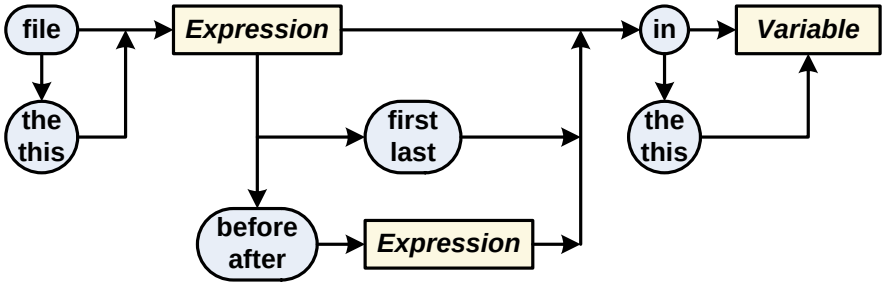
Before the calendar date can be used, origin.r must be called to established the origin date, before "start simulation" is executed.

Only one process notice is scheduled for initial invocation from each external process unit. Only after that process notice is removed from the event set and the process routine executed will the next process be read from the external unit and scheduled for initial invocation.

All process methods are internal and may not be scheduled externally; however, an external process may call or schedule a process method.

External units can be given by named constants.

2.28 File



This statement, which may be used in any routine, inserts an object or entity into a set. It has five forms. In each form, an ***Expression*** identifies an object or entity to be inserted into the set named by ***Variable***. The keywords **the** and **this** are optional for readability.

1. **File *Expression* in *Variable***. Each set has a defined ordering specified by a ***DefineSet*** statement (see page 48). The ordering is either first-in-first-out (FIFO), last-in-first-out (LIFO), or ranked according to the values of one or more member attributes (called "ranking attributes"). If no ***DefineSet*** statement is specified for the set, its ordering defaults to FIFO. This form of the ***File*** statement positions a member according to the defined ordering of the set. If the ordering is FIFO, the member is inserted last in the set. If the ordering is LIFO, the member is inserted first in the set. If the ordering is ranked, the member is positioned in rank order according to the values of its ranking attributes.

   For example, suppose **Job** is a class where each **Job** object has a **Priority** attribute and may belong to a set named **Queue**, i.e.,

   ```
   every Job has a Priority and belongs to a Queue
   ```

   Suppose **J** is a **Job** reference variable. Here we create a **Job** object and initialize its **Priority** to 7:

   ```
   create J
   let Priority(J) = 7
   ```

   Now consider the effect of the following statement:

   ```
   file J in Queue
   ```

If **Queue** is FIFO, then the **Job** object identified by **J** is inserted last in **Queue**. If **Queue** is LIFO, then the object is inserted first in **Queue**. If **Queue** is a ranked set with **Priority** as its sole ranking attribute, i.e.,

```
define Queue as a set ranked by high Priority
```

then the object is inserted in rank order. It is placed after all members that have **Priority >= 7**, and before all members with **Priority < 7**. Members with identical rank values (e.g., **Priority = 7**) are ranked on a first-in-first-out basis.

A member of a ranked set is placed into the set based on the values of its ranking attributes at the time of its insertion. If after insertion, the value of a member's ranking attribute is modified, the member is not automatically repositioned within the set based on the new value. In fact, it is an error to change the value of a ranking attribute because it breaks the ordering of the set. The correct procedure is first to remove the member from the set, change the attribute value, and then re-insert the member into the set. Upon re-insertion, the member is positioned correctly within the set using the new value. In our example, we must first remove the object identified by **J** before changing its **Priority**, and then place the object back into the **Queue**:

```
remove J from Queue
add 2 to Priority(J)
file J in Queue
```

2. **File *Expression* first in *Variable***. The object or entity identified by ***Expression*** is filed *first* in the set named by ***Variable***. For example:

```
file Tanker first in Awaiting(Tug)
```

3. **File *Expression* last in *Variable***. The object or entity identified by ***Expression*** is filed *last* in the set named by ***Variable***. For example:

```
file Tanker last in Awaiting(Tug)
```

4. **File *Expression* before *Expression2* in *Variable***. The object or entity identified by ***Expression*** is placed in the set named by ***Variable*** immediately *before* the member identified by ***Expression2***. In the following example, the object identified by **Tanker** immediately precedes the member identified by **Freighter** in the set, **Awaiting(Tug)**:

```
file Tanker before Freighter in Awaiting(Tug)
```

5. **File *Expression* after *Expression2* in *Variable***. The object or entity identified by ***Expression*** is placed in the set named by ***Variable*** immediately *after* the member identified by ***Expression2***. For example:

```
file Tanker after Freighter in Awaiting(Tug)
```

Any of the five forms may be used for FIFO and LIFO sets. However, only the first form is permitted for ranked sets; the other forms are disallowed because they break the rank ordering.

The set attributes of the new member, and of the set owner, are automatically updated when a ***File*** statement is executed. For example, when this statement is executed,

```
file Tanker first in Awaiting(Tug)
```

the following modifications are made to the set attributes:

```
' ' the new member has no predecessor because it is first in the set
let p.Awaiting(Tanker) = 0

' ' the successor of the new member is the member that was previously
' ' first in the set
let s.Awaiting(Tanker) = f.Awaiting(Tug)

' ' the "m." attribute is set to a nonzero value to indicate that the object or entity
' ' is a member of the set; the nonzero value identifies the owner of the set
let m.Awaiting(Tanker) = Tug

' ' the set has a new first member
let f.Awaiting(Tug) = Tanker

' ' if this is the only member of the set, then the set has a new last member
if l.Awaiting(Tug) = 0
    let l.Awaiting(Tug) = Tanker
always

' ' increment the number of members in the set
add 1 to n.Awaiting(Tug)
```

If the owner of the set can be identified by a single value, then that value is assigned to **m.*set_name***. Otherwise, a value of 1 is assigned to **m.*set_name***. The owner of a set can be identified by a single value if

- the set is 0-dimensional and owned by an object (the identification is the reference value of the object);
- the set is 1-dimensional and owned by a class or by the system or subsystem (the identification is the set subscript);
- the set is owned by a temporary entity or process notice (the identification is the reference value of the entity); or
- the set is owned by a permanent entity or resource (the identification is the entity number).

The object or entity identified by ***Expression*** must not already be a member of a set with the specified name, i.e., its **m.*set_name*** attribute must be zero before the ***File*** statement is executed. On the other hand, in Forms 4 and 5, ***Expression2*** must identify an object or entity that is currently a member of the set (its **m.*set_name*** attribute must be nonzero). These conditions may be verified before executing a ***File*** statement. For example:

```
if Tanker is not in Awaiting and Freighter is in Awaiting(Tug)
   file Tanker after Freighter in Awaiting(Tug)
always
```

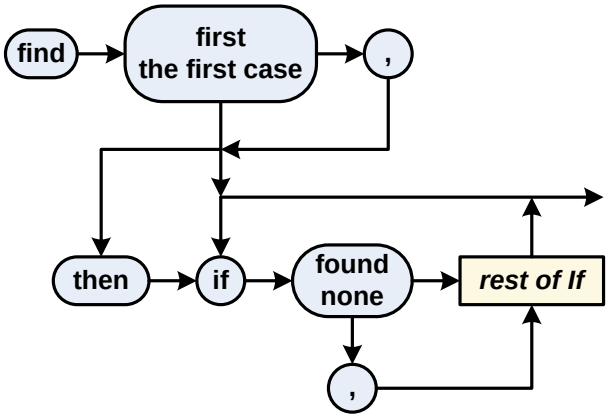
For a set of objects, the mode of ***Expression*** and ***Expression2*** must be integer, pointer, the reference mode of the member class, or the reference mode of a class that is derived from the member class.

For a set of temporary entities or process notices, the mode of ***Expression*** and ***Expression2*** must be integer, pointer, or the reference mode of the entity type.

For a set of permanent entities or resources, a member is identified by an entity number; therefore, the mode of ***Expression*** and ***Expression2*** must be numeric: double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer.

A "before filing" routine and an "after filing" routine, if defined, are called automatically before and after each object or entity is inserted into the set. The first argument to these routines is the value of ***Expression***. Additional arguments are supplied as needed to identify the owner of the set. The value of ***Expression2*** in Forms 4 and 5 is not passed to these routines. See ***BeforeAfter*** on page 18 for more information.

2.29 Find



The **find first** phrase may appear as the body of a loop. The first time the body of the loop is executed, the loop terminates with **found** equal to true and **none** equal to false. If the loop terminates without ever executing the body of the loop, then **found** is false and **none** is true. This condition may be tested by an ***If*** statement that begins with **if found** or **if none**, which must immediately follow the **find first** phrase with no intervening statements. The phrases **find first** and **find the first case** are synonymous. The commas are optional for readability.

Although the **find first** phrase may appear as the body of any loop, it is normally preceded by one or more ***For*** phrases and a ***With*** phrase. Here we check an array **X** to see if it contains any negative values. If it does, the loop terminates with the control variable **J** containing the index of the first negative value.

```
for J = 1 to dim.f(X)
with X(J) < 0
    find first

if found
    write X(J), J as "A negative value ", i *, " is stored at index ", i *, /
else
    write as "All elements of the array are positive or zero", /
always
```

This ***If*** statement may be written equivalently as:

```
if none
    write as "All elements of the array are positive or zero", /
else
    write X(J), J as "A negative value ", i *, " is stored at index ", i *, /
always
```

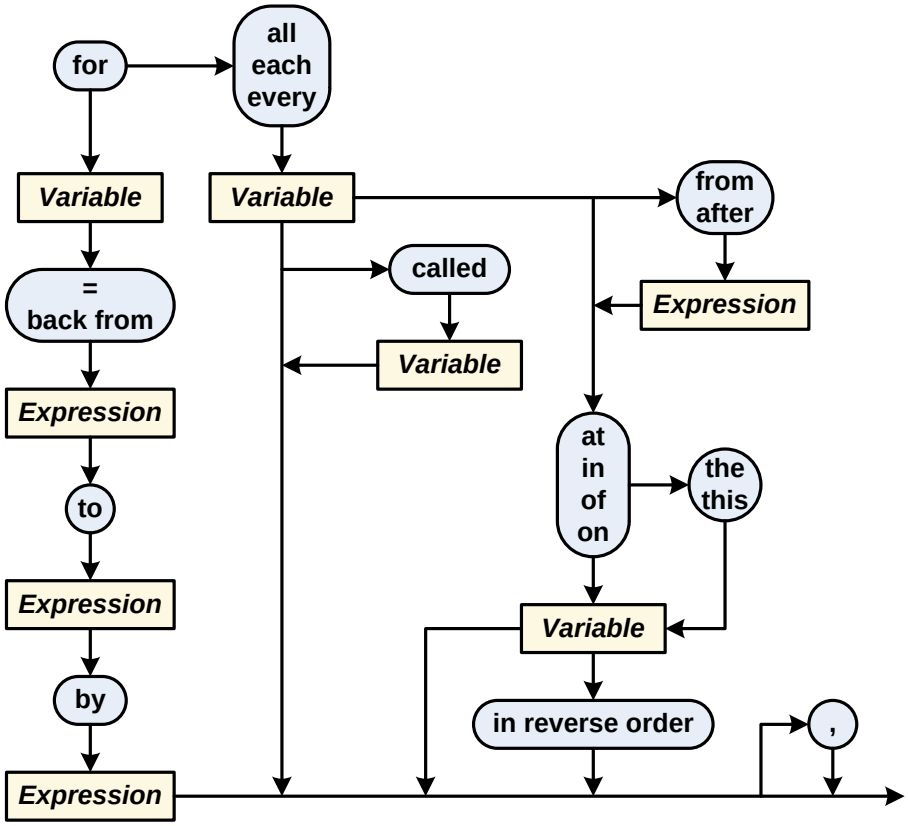
In the next example, we read the name of a sergeant and then search each platoon of the company to locate the sergeant. The loop terminates once the sergeant is found; the control variable **Soldier** contains the reference value of the sergeant, and the control variable **Platoon** refers to the sergeant's platoon.

```
read Sergeant_Name

for each Platoon in Company
    for each Soldier in Staff(Platoon)
    with Rank(Soldier) = Sergeant and Name(Soldier) = Sergeant_Name
        find the first case

if found
    write as "The sergeant has been located", /
    write Name(Soldier), Number(Platoon)
        as "Sergeant ", t *, " is in platoon #", i *, /
always
```

The ***If*** statement may be preceded by a **then** keyword when nested within another ***If*** statement. See ***If*** on page 85 for more information.

The ***If*** statement may be omitted. Upon termination of the loop, control passes to the statement that follows the loop.

2.30 For



This loop control phrase is part of a ***Loop*** statement and causes the body of the loop to be executed once for each value assigned to a control variable. The assigned values may range from a starting value to an ending value or may refer to consecutive members of a set.

The comma, and the keywords **the** and **this**, are optional for readability. The following are synonymous:

- **all**, **each**, and **every**;
- **at**, **in**, **of**, and **on**.

This phrase has 12 forms:

1. **for *Variable* = *Expression1* to *Expression2***. The body of the loop is executed once for each value of the control variable from ***Expression1*** to ***Expression2***. This form is equivalent to:

```
let Variable = Expression1

while Variable <= Expression2
do
    ' ' execute body of loop
    …
    add 1 to Variable
loop
```

   For example, the following phrase executes the body of the loop ten times, as the control variable **J** takes on the values 1, 2, …, 10.

```
for J = 1 to 10
```

2. **for *Variable* = *Expression1* to *Expression2* by *Expression3***. The body of the loop is executed once for each value of the control variable from ***Expression1*** to ***Expression2***, where the control variable is incremented by ***Expression3*** after each iteration. This form is equivalent to:

```
let Variable = Expression1

while Variable <= Expression2
do
    ' ' execute body of loop
    …
    add Expression3 to Variable
loop
```

   For example, the following phrase executes the body of the loop five times, as the control variable **J** takes on the values 1, 3, 5, 7, and 9.

```
for J = 1 to 10 by 2
```

   In the next example, a double variable **K** takes on the values –1.0, –0.5, 0.0, 0.5, and 1.0.

```
for K = –1 to 1 by 0.5
```

3. **for *Variable* back from *Expression1* to *Expression2***. The body of the loop is executed once for each value of the control variable from ***Expression1*** down to ***Expression2***. This form is equivalent to:

```
let Variable = Expression1

while Variable >= Expression2
do
   ' ' execute body of loop
   …
   subtract 1 from Variable
loop
```

For example, the following phrase executes the body of the loop ten times, as the control variable **J** takes on the values 10, 9, …, 2, 1.

```
for J back from 10 to 1
```

4. **for *Variable* back from *Expression1* to *Expression2* by *Expression3***. The body of the loop is executed once for each value of the control variable from ***Expression1*** down to ***Expression2***, where the control variable is decremented by ***Expression3*** after each iteration. This form is equivalent to:

```
let Variable = Expression1

while Variable >= Expression2
do
   ' ' execute body of loop
   …
   subtract Expression3 from Variable
loop
```

For example, the following phrase executes the body of the loop five times, as the control variable **J** takes on the values 10, 8, 6, 4, and 2.

```
for J back from 10 to 1 by 2
```

5. **for each *Variable***. The body of the loop is executed once for each entity number from 1 to the number of entities of the named permanent entity type or resource type. The control variable is the local or global variable with the same name as the entity type. This form is equivalent to the following Form 1 phrase:

```
for Variable = 1 to n.Variable
```

For example, if **City** is a permanent entity type, the phrase,

```
for each City
```

is equivalent to

```
for City = 1 to n.City
```

6. **for each *Variable* called *Variable2***. The body of the loop is executed once for each entity number from 1 to the number of entities of the permanent entity type or resource type specified by ***Variable***. The control variable is ***Variable2***. This form is equivalent to the following Form 1 phrase:

   ```
   for Variable2 = 1 to n.Variable
   ```

   For example, the phrase,

   ```
   for each City called C
   ```

   is equivalent to

   ```
   for C = 1 to n.City
   ```

7. **for each *Variable* in *Variable2***. The body of the loop is executed once for each member of the set identified by ***Variable2***, from the first member of the set to the last member. The reference value or entity number of each member is assigned to the control variable ***Variable***. This form is equivalent to the following code sequence, where ***Successor*** is an implicitly-defined local variable:

   ```
   let Variable = f.Variable2  ' ' start with the first member in the set

   while Variable <> 0  ' ' while we have not reached the end of the set
   do
      let Successor = s.set(Variable)
      ' ' execute body of loop
      …
      let Variable = Successor
   loop
   ```

   For example, the following phrase executes the body of the loop for each member of the set named **Fleet**. In each iteration, the control variable **Ship** refers to a different member of the set. The successor of the current member is stored in **s.Fleet(Ship)**.

   ```
   for each Ship in Fleet
   ```

8. **for each *Variable* from *Expression* in *Variable2***. This form is identical to Form 7 except that the body of the loop is executed starting with the member identified by ***Expression*** and continuing to the last member of the set. For example, the following phrase begins with the member identified by **Tanker**:

```
for each Ship from Tanker in Fleet
```

9. **for each *Variable* after *Expression* in *Variable2***. This form is identical to Form 7 except that the body of the loop is executed starting with the successor of the member identified by ***Expression*** and continuing to the last member of the set. For example, the following phrase begins with the ship that comes after the member identified by **Tanker**:

```
for each Ship after Tanker in Fleet
```

10. **for each *Variable* in *Variable2* in reverse order**. The body of the loop is executed once for each member of the set identified by ***Variable2***, from the last member of the set to the first member. The reference value or entity number of each member is assigned to the control variable ***Variable***. This form is equivalent to the following code sequence, where ***Predecessor*** is an implicitly-defined local variable:

```
let Variable = l.Variable2  ' ' start with the last member in the set

while Variable <> 0  ' ' while we have not reached the beginning of the set
do
    let Predecessor = p.set(Variable)
    ' ' execute body of loop
    …
    let Variable = Predecessor
loop
```

For example, the following phrase executes the body of the loop for each member of the set named **Fleet**, starting with the last member and moving backward through the set to the first member. In each iteration, the control variable **Ship** refers to a different member of the set. The predecessor of the current member is stored in **p.Fleet(Ship)**.

```
for each Ship in Fleet in reverse order
```

11. **for each *Variable* from *Expression* in *Variable2* in reverse order**. This form is identical to Form 10 except that the body of the loop is executed starting with the member identified by ***Expression*** and continuing backward to the first member of the set. For example, the following phrase begins with the member identified by **Tanker**:

```
for each Ship from Tanker in Fleet in reverse order
```

12. **for each *Variable* after *Expression* in *Variable2* in reverse order**. This form is identical to Form 10 except that the body of the loop is executed starting with the predecessor of the member identified by ***Expression*** and continuing backward to the first member of the set. For example, the following phrase begins with the ship that precedes the member identified by **Tanker**:

```
for each Ship after Tanker in Fleet in reverse order
```

It is possible that the body of the loop will not be executed at all. In the following example, if **First_Position** is greater than **Last_Position**, the body of the loop is never executed:

```
for Index = First_Position to Last_Position
```

Likewise, if the set named **Repair_Queue** is empty, then the body of this loop is never executed:

```
for each Car in Repair_Queue(Shop)
```

Normally the value of the control variable is accessed within the body of the loop. A value may be assigned to the control variable within the body of the loop, but this is not recommended.

The control variable retains its value after the loop has terminated. The mode of the control variable must be numeric (i.e., double, real, integer, integer4, integer2, or alpha) unless traversing a set of objects, temporary entities, or process notices. For a set of objects, the mode of the control variable must be the reference mode of the member class. In the above example, if **Repair_Queue** is a set of **Vehicle** objects (i.e., **every Vehicle belongs to a Repair_Queue**), then **Car** must be a **Vehicle** reference variable. For a set of temporary entities and/or process notices, the mode of the control variable must be integer, pointer, or the reference mode of a member type.

In Forms 1 through 4, the mode of each ***Expression*** must be numeric. Note that ***Expression2*** and ***Expression3*** are re-evaluated each iteration of the loop. The body of the loop may change the value of a variable used in these expressions, thereby changing the loop termination condition or control variable increment. However, such a loop is not recommended because it is complex and difficult to read. The value of ***Expression3*** may be zero or negative, but this is not recommended and may cause an infinite loop.

The ***Expression*** in the **from** phrase of Forms 8 and 11 must identify a member of the set or be zero; if it is zero, the body of the loop is never executed. The ***Expression*** in the **after** phrase of Forms 9 and 12 must identify a member of the set; however, if the first member is identified in Form 12, or the last member is identified in Form 9, then there is no predecessor or successor, and the body of the loop is never executed. For a set of objects, the mode of ***Expression*** must be the reference mode of the member class or the reference mode of a class that is derived from the member class; however, in Forms 8 and 11, the mode may also be integer or pointer. For a set of temporary entities and/or

process notices, the mode of ***Expression*** must be integer, pointer, or the reference mode of a member type. For a set of permanent entities or resources, the mode of ***Expression*** must be numeric; if it is double or real, it is implicitly rounded to integer.

If the control variable is already correctly initialized when a ***For*** phrase is encountered, its name may be specified as the starting ***Expression***. For example:

```
for Index = Index to Last_Position
…

for each Car from Car in Repair_Queue(Shop)
…
```

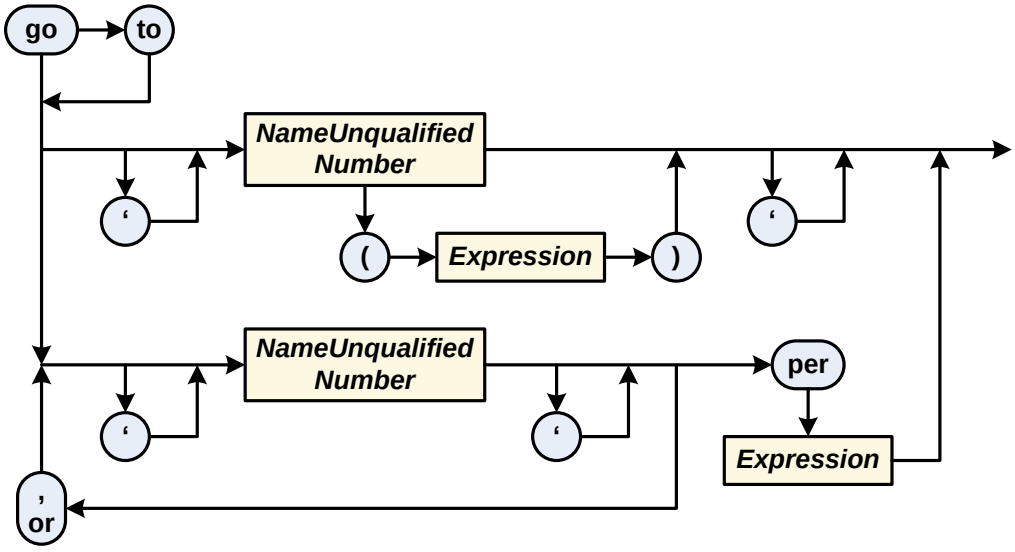
When traversing a set using Forms 7 to 12, the body of the loop may alter the membership of the set. The control variable identifies the current member of the set. When moving forward through the set (Forms 7 to 9), the body of the loop may remove any member from the set except the successor of the current member. When moving backward through the set (Forms 10 to 12), any member may be removed except the predecessor of the current member. It is possible to empty an entire set by removing the current member at each iteration. For example:

```
for each Car in Repair_Queue(Shop)
   remove Car from Repair_Queue(Shop)
```

Any member inserted after the successor of the current member when moving forward (Forms 7 to 9), or before the predecessor of the current member when moving backward (Forms 10 to 12), will be visited by the loop.

For phrases may be nested and qualified by ***While*** and ***With*** phrases. See ***Loop*** on page 116 for more information.

2.31 GoTo



This statement may be used in any routine and specifies an unconditional transfer of control to a label within the routine. It has three forms. The keyword **to** is optional for readability.

1. If an unsubscripted label is specified, control is transferred to the label. For example:

```
'Home'
…

go to Exit        ' ' jump to the label below
…

go Home           ' ' jump to the label above
…

'Exit'
…
```

2. If a subscripted label is specified, the subscript ***Expression*** is evaluated and control is transferred to the label with this subscript. In the following example, control is transferred to **Procedure(1)** if **N** is equal to 1, and to **Procedure(2)** if **N** is equal to 2:

```
go to Procedure(N)
…

'Procedure(1)'
…

'Procedure(2)'
…
```

3. If a list of two or more unsubscripted labels is specified, an ***Expression*** is evaluated. If its value is 1, control is transferred to the first label in the list; if its value is 2, control is transferred to the second label in the list; and so on. It is an error if its value is less than 1 or greater than the number of labels in the list. A label may appear more than once in the list. Labels are separated in the list by a comma or the keyword **or**, which are synonymous. In the following example, control is transferred to **Basic_Step** if **Indicator** is equal to 1 or 3, and to **Special_Case** if **Indicator** is equal to 2:

```
'Basic_Step'
…

'Special_Case'
…

go to Basic_Step, Special_Case or Basic_Step per Indicator
```

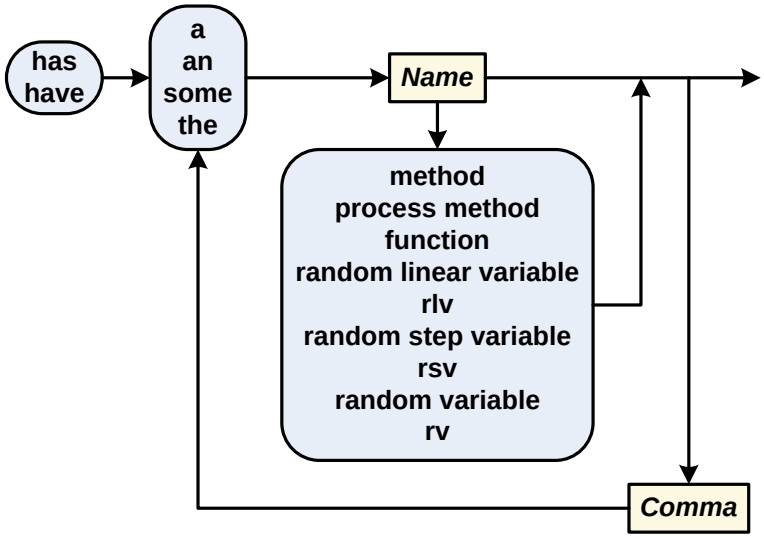
It is an error for a ***GoTo*** statement to refer to a label that does not exist. The ***Expression*** must have a numeric mode: double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer.

A label specified in a ***GoTo*** statement may optionally be enclosed in apostrophes. However, a space must separate the first apostrophe from the preceding keyword. For example:

```
go to'Generate'        ' ' invalid
go to 'Generate'       ' ' valid
```

The use of ***GoTo*** statements should be minimized. Overuse leads to unreadable "spaghetti code" with complicated control flows.

2.32 Has



This language element is part of an ***Every***, ***TheClass***, or ***TheSystem*** statement and specifies the names of attributes and methods. Inside a **begin class** block, it names object attributes and methods as part of an ***Every*** statement, or class attributes and methods as part of ***TheClass*** statement. Outside a **begin class** block, it names attributes of temporary entities, process notices, permanent entities, resources, or compound entities as part of an ***Every*** statement, or attributes of the system or subsystem as part of ***TheSystem*** statement.

The following are synonymous:

- **has** and **have**;
- **a**, **an**, **some**, and **the**;
- **random linear variable** and **rlv**;
- **random step variable** and **rsv**;
- **random variable** and **rv**.

After naming an object attribute or class attribute in a ***Has*** phrase, a ***DefineVariable*** statement follows within the **begin class** block and specifies the mode and dimensionality of the attribute. In the following example, object attributes named **Odometer** and **Trip_Meter**, and a class attribute named **Num_Vehicles**, are defined for a class named **Vehicle**.

```
begin class Vehicle

    every Vehicle has an Odometer and a Trip_Meter
    define Odometer as a real variable
    define Trip_Meter as a 1-dim real array

    the class has a Num_Vehicles
    define Num_Vehicles as an integer variable

end
```

Likewise, after naming an object method or class method in a ***Has*** phrase, a ***DefineMethod*** statement follows within the **begin class** block and defines the method arguments and return mode, if any. If a ***DefineMethod*** statement is not specified, the method is assumed to be a subroutine with no arguments. In the next example, **Travel** is an object process method given one argument, **Reset_Trip_Meter** is an object subroutine method with no arguments, **Count** is a class function method with no arguments, and **Direct** is a class process method given three arguments and yielding two arguments. The implementation of these methods must appear within the same module as the **begin class** block that defines them.

```
begin class Vehicle

    every Vehicle has a Travel process method and a Reset_Trip_Meter method
    define Travel as a process method given a double argument

    the class has a Count method and a Direct process method
    define Count as an integer method
    define Direct as a process method
        given a text argument and 2 real arguments
        yielding an integer argument and a double argument

end
```

Consider the following statements appearing outside a **begin class** block:

```
every T has an X
define X as an integer variable

every U, V has a Y
define Y as a double variable

the system has a Z
define Z as a 1-dim text array
```

In this case, **T** is the name of a temporary entity type, process type, permanent entity type, or resource type, and **X** is declared to be an integer attribute of each entity of type **T**. Also, **Y** is defined as a double attribute of a compound entity, where **U** and **V** are permanent entity types or resource types, and **Z** is declared as a 1-dimensional text attribute of the system. Note that ***DefineVariable*** statements follow ***Every*** and ***TheSystem*** statements within the preamble and specify the mode of the attributes. The

dimensionality may be specified for **Z** but not for **X** and **Y** because the number of subscripts is determined by the number of entity types appearing in the ***Every*** statement, i.e., **X** has one subscript and **Y** has two subscripts.

It is an error if no ***DefineVariable*** statement is provided that specifies the mode of an attribute unless a background mode has been established before the ***Has*** phrase by a ***Normally*** statement. In this case, the mode of the attribute is the background mode. Here both **Odometer** and **Trip_Meter** are defined as real:

```
begin class Vehicle

   normally mode is real
   every Vehicle has an Odometer and a Trip_Meter
   define Trip_Meter as a 1-dim array

end
```

Likewise, the background dimensionality is used if an attribute's dimensionality is unspecified. In this example, **Table** is defined as a 2-dimensional integer array:

```
normally mode is integer and dimension is 2
the subsystem has a Table
```

The above statements are equivalent to:

```
the subsystem has a Table
define Table as a 2-dim integer array
```

An object (class) attribute is implicitly defined and named for each object (class) process method. It is of mode pointer and used to hold a reference value of a process notice when scheduling the invocation of the process method. An integer class attribute named i.name is implicitly defined for each process method to hold the index of the corresponding process type in the event set.

Common attributes can be specified both publicly and privately within a single subsystem. They cannot be declared across modules. Attributes are treated as common attributes only if they have the same qualified name. Permanent entities cannot have common attributes. Names defined outside a "begin class" block are global and must be unique among global names (unless common attributes or sets are intended). Names defined inside a "begin class" block are local to the class and must be unique within the class. These names may also be defined at the global scope or local to another class, including within a base class. However, the definitions are independent and do not define common attributes.

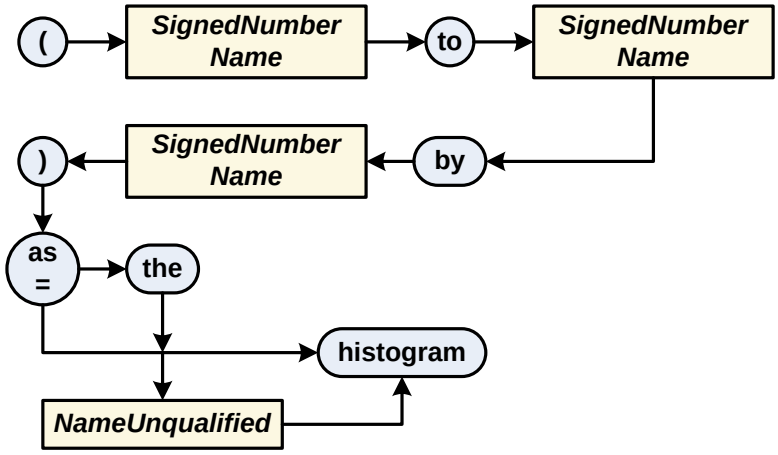
Function attributes - A temporary entity, permanent entity, or "the system" can have a function attribute. A function with the same name and mode as the attribute must be provided. This function must accept a number of given arguments that matches the attribute's dimensionality. There is no storage space allocated for function attributes.

Random attributes - A random variable is sampled using a table of possible numerical values and their associated probabilities. It selects a sample value by generating a random number (using random.f(1), unless an alternative stream is specified in a "define variable" statement) and matching it against the probability values. (See discussion on p. 317 of 1973 book.) Random step variables can be integer or double. Random linear variables must be double.

If the programmer requires a type of sampling other than step or linear, he must omit the words "step" and "linear" and declare a "random variable", and provide his own sampling function. istep.f(table, stream) and rstep.f(table, stream) return a value for an integer and real random step variable, respectively. lin.f(table, stream) returns a value for a random linear variable. Accessing a random variable on the right generates a random value. (It can also be used as the name of a set.) It can be accessed on the left only when reading it.

A random variable is read as follows. Pairs of free-form data values are read until a mark.v character appears. The first of each pair is assumed to be a probability. The second is assumed to be a sample value. A "random.e" entity is created for each pair. The probability value is assigned to the "prob.a" attribute of the entity. The sample value is assigned to the "ivalue.a" attribute if the random variable is integer, or to the "rvalue.a" attribute if the random variable is real or double. The entities are filed in a set having the same name as the random variable (so we can say "for each random.e in random_var"). Each "random.e" entity has a set attribute "s.random_var" which refers to the successor "random.e" in the set. "f.random_var" occupies the space declared for the random variable and refers to the first "random.e" in the set. Input probabilities can be read as cumulative or individual; if cumulative, the last probability must be 1.0; if individual, they must sum to 1.0. However, the probabilities are stored cumulatively in the "prob.a" attributes of "random.e" entities. If the last probablity read is 1.0, the probabilites are assumed to be cumulative; otherwise, they are treated as individual and summed and the last probability is set to 1.0. It is an error is any probability read is less than zero or greater than one. Instead of reading a random variable, a random variable "set" can be constructed by the program by creating "random.e" entities, settings their attributes, and filing them in the set ("file random.e in random_var"). "If random_var is empty" is interpreted as "if f.random_var = 0".

2.33 Histogram



This language element appears in an **accumulate** or **tally** statement and specifies a histogram to collect on the values assigned to an attribute or global variable. The histogram is weighted by simulation time for an **accumulate** statement and is unweighted for a **tally** statement.

The keyword **the** is optional for readability. The keyword **as** and the equal sign are synonymous.

(Min to Max by Inc) declares a histogram array with (((Max - Min) / Inc) + 1) elements. If a sample falls is less than (Min + Inc), element 1 of the array is used. If a sample is greater than or equal to (Min + Inc) but less than (Min + 2*Inc), element 2 is used. If the sample is greater than or equal to (Max - Inc) but less than Max, the second to the last element is used. If a sample is greater than or equal to Max, the last element is used. For tally, the element is incremented. For accumulate, the amount of time that the sample held its current value is added to the element. A histogram array is accessed like any other array; dim.f can be used to get the number of elements in the array. The histogram array subscript is the last (rightmost) subscript.

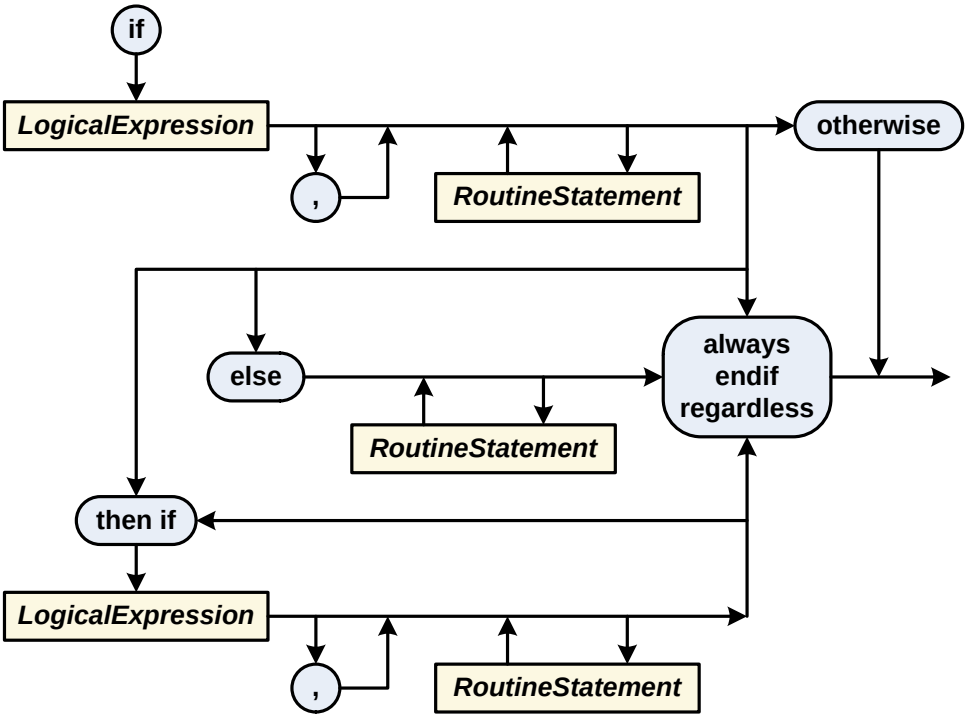
If the range specifications (Min, Max, and Inc) for a histogram are variables, they must be assigned meaningful values before the monitored variable is first referenced and should only be altered following a reset statement and before any subsequent reference to the monitored variable.

The histogram has dimensionality one greater than the variable. Because temporary entities and process notices cannot have array attributes, it is not possible to collect a histogram on an attribute of a temporary entity or process notice.

The Min, Max, and Inc values for a histogram must be numeric constants or 0-dimensional numeric variables. Inside a begin class block, the constants may be class

constants and the variables must be class attributes. Outside a begin class block, the constants may be global constants and the variables must be global variables or system attributes. The variables may be monitored and may not be function attributes or random variables. Min, Max, and Inc can be named constants.

2.34 If



This language element chooses a sequence of statements to execute depending on whether a ***LogicalExpression*** is true or false. It may be used in any routine. The comma after the ***LogicalExpression*** is optional for readability. The following are synonymous:

- **else** and **otherwise**;
- **always**, **endif**, and **regardless**.

For example:

```
if X > 0
    add 1 to J
    let A(J) = X
always
```

In this example, if the value of the ***LogicalExpression***, **X > 0**, is true, the sequence of statements following the ***LogicalExpression*** is executed; otherwise, this sequence is bypassed. In both cases, execution continues with the statement that follows the **always** keyword.

A second sequence of statements may be specified after an **else** keyword. If the ***LogicalExpression*** is true, only the first sequence of statements is executed. If the ***LogicalExpression*** is false, only the second sequence is executed. In both cases, execution continues with the statement that follows the **always** keyword.

```
if X > 0
   ' ' these statements are executed if X > 0
   add 1 to J
   let A(J) = X
else
   ' ' these statements are executed if X <= 0
   write as "Unexpected value", /
   list X
always
```

If the last executable statement in the first sequence is an unconditional transfer of control (i.e., a ***Cycle***, ***GoTo***, ***Jump***, ***Leave***, ***Return***, or ***Stop*** statement), then the first sequence is followed by an **else** or **always** keyword but not both. This keyword terminates the statement. By convention, the **otherwise** synonym of **else** is used in this context. For example:

```
if X > 0
   add 1 to J
   let A(J) = X
   return
otherwise  ' ' this marks the end of the if statement

' ' arrive here only when X <= 0
write as "Unexpected value", /
list X
```

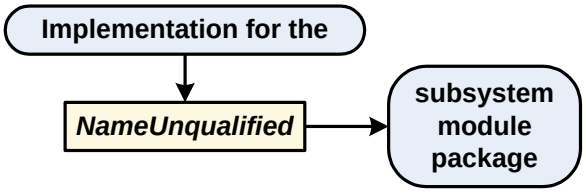
A sequence of statements within an ***If*** statement is any sequence of zero or more statements and may include "nested" ***If*** statements. For example:

```
if X > 0
   add 1 to J
   let A(J) = X
   if X > Limit
      add 1 to N
      let Outlier(N) = X
      if N = dim.f(Outlier)
         write as "Maximum number of outliers reached", /
      always
   always
always
```

A special form is permitted when nested ***If*** statements with no **else** keywords are terminated at the same location, as in the above example. Rather than specify a series of **always** keywords, one for each ***If*** statement, each nested ***If*** statement is preceded by **then** and a single **always** keyword terminates the entire group. Using this form, the above example can be rewritten as follows:

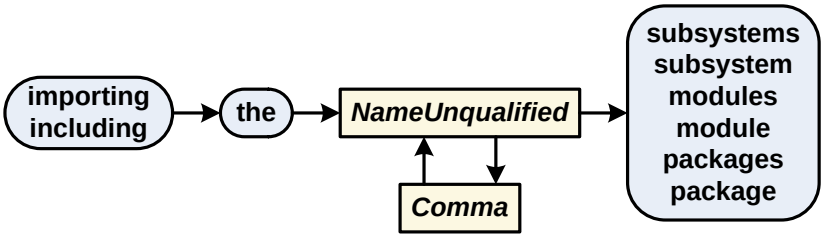
```
if X > 0
    add 1 to J
    let A(J) = X
    then if X > Limit
        add 1 to N
        let Outlier(N) = X
        then if N = dim.f(Outlier)
            write as "Maximum number of outliers reached", /
        always
```

2.35 Implementation



The purpose of this heading is to identify the implementation code for subsequent routines and methods as belonging to the given subsystem. This statement must appear as the first line of executable code in the file. (The heading should not be provided is if preceded by a public or private preamble heading in the same file.) The heading is used for subsystems only. When this heading is omitted, the implementation code within the file is assumed to be part of the "main" module.

Importing



An **importing** phrase specifies one or more subsystems to import. It is optionally appended to a preamble heading.

The following are synonymous:

- **importing** and **including**;
- **subsystems**, **subsystem**, **modules**, **module**, **packages**, and **package**.

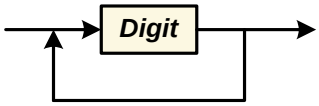
A module that imports a subsystem may refer to any name defined within the public preamble of the subsystem. However, if the name is ambiguous within the importing module, it must be qualified.

Importing a subsystem imports not only the names defined in the subsystem's public preamble, but automatically imports each subsystem imported by this public preamble. Subsystems imported by a private preamble are not automatically known/imported.

The compiler finds the file containing the public preamble for an imported subsystem based on the name of the subsystem. For platforms with case-sensitive filenames, the name of the subsystem in an "importing" phrase is case sensitive.

A module that imports a subsystem does not import, nor is affected by, the substitutions and settings (normally, suppress/resume) defined in the subsystem's public preamble.

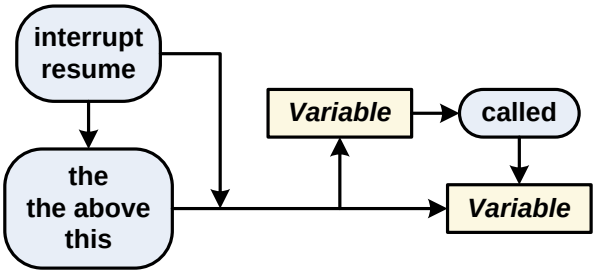
2.36 Integer



This language element is a sequence of one or more decimal digits representing a nonnegative integer. It appears in many contexts. Leading zeros are ignored. The following are examples:

0 5 007 21 102 20000

2.37 InterruptResume



An **interrupt** statement removes a process notice from the event set, thereby canceling the pending execution of a process method or process routine, and the amount of time until the execution would have occurred is saved in the **time.a** attribute of the process notice. A **resume** statement inserts the process notice back into the event set, scheduling the process method or process routine to be executed after the time indicated by **time.a** has elapsed. These statements may be used in any routine. The keywords **the**, **the above**, and **this** are optional for readability.

1. **Interrupt *Variable***. This form is equivalent to the following sequence of statements:

   cancel *Variable*
   subtract time.v from time.a(*Variable*)

2. **Interrupt *Variable2* called *Variable***. This form is equivalent to the following sequence of statements:

   cancel *Variable2* called *Variable*
   subtract time.v from time.a(*Variable*)

3. **Resume *Variable***. This form is equivalent to the following statement:

   schedule the *Variable* in time.a(*Variable*) units

4. **Resume *Variable2* called *Variable***. This form is equivalent to the following statement:

   schedule the *Variable2* called *Variable* in time.a(*Variable*) units

These statements are often used in conjunction with a ***Wait*** statement (see page 220). Suppose a machine is performing a task. A process executes a ***Wait*** statement to model the time taken to perform the task. The process is suspended and awakens once the task has completed. However, suppose the machine fails during the task and must be repaired before work can continue. This may be modeled by another process interrupting the task. When the repair is finished, the task may be resumed.

For example, suppose that at time 100, a machine begins a task that requires 20 time units to complete. The following statement suspends the current process and schedules it to awaken at time 120:

```
work 20 units  ' ' the machine is working
' ' arrive here when the machine has finished the task
```

Suppose **Task** is a variable that holds the reference value of the process notice of this suspended process, and suppose that the following statement is executed at time 105 because the machine has failed. The pending resumption of the process, scheduled for time 120, is canceled, and the number of time units remaining, 15, is saved in **time.a(Task)**.

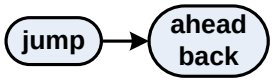
```
interrupt Task
```

Now suppose at time 155, the machine has been repaired and is ready to continue the task. The following statement schedules the process to be awakened in **time.a(Task)** = 15 units, i.e., at time 170, when the task has been completed, assuming no further interruptions.

```
resume Task
```

See the ***Schedule*** statement on page 187 and the ***Cancel*** statement on page 27 for more information.

2.38 Jump



This statement may be used in any routine and specifies an unconditional transfer of control to a **here** label within the routine. It has two forms:

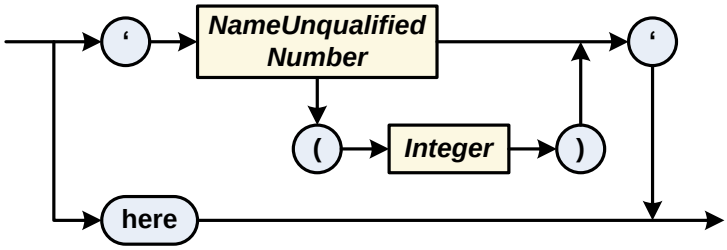
1. A **jump ahead** statement transfers control to the nearest **here** label that follows the statement. It is an error if no **here** label follows the statement. For example:

```
if Job_Status = Complete
    jump ahead
otherwise
…

here ' ' arrive here when Job_Status = Complete
call Write_Job_Summary
```

2. A **jump back** statement transfers control to the nearest **here** label that precedes the statement. It is an error if no **here** label precedes the statement. For example:

```
here ' ' arrive here when Job_Status = Incomplete
call Perform_Task
…

if Job_Status = Incomplete
    jump back
otherwise
```

2.39 Label



This language element identifies a location within a routine which may be the destination of an unconditional transfer of control. If it is a name or number enclosed in apostrophes, it may be the target of ***GoTo*** statements within the routine. If it is the **here** keyword, it may be the target of ***Jump*** statements.

For example:

```
' ' this statement transfers control to the label below
go to Handle_Error
…

'Handle_Error'
' ' take care of the error here
close unit 12
write as "Unexpected error condition", /
```

A label may be the target of zero, one, or more than one ***GoTo*** statements. It may precede or follow a ***GoTo*** statement that refers to it.

A label may be a ***Number***. In this example, **1.1**, **1.2**, and **2** are labels:

```
'1.1'     read X;  go to 2
'1.2'     read X and Y; add Y to X; go to 2
…

'2'       add X to Sum
```

A statement may be preceded by two or more labels, which may appear on the same line. In the following example, the statements, **go to Wrap_up** and **go to Finish**, transfer control to the same location.

```
'Wrap_up'  'Finish'
call Free_Arrays
…
```

The same name may appear in more than one label if each occurrence is followed by a different integer subscript. Only one subscript may be specified and it must be between 1 and 3000. These "subscripted labels" need not start with subscript 1 or be consecutively numbered, and they are not required to be placed in numerical sequence. They are the targets of subscripted ***GoTo*** statements. For example:

```
'Step(2)'  /~ arrive here if K = 2 ~/
…

'Step(5)'  /~ arrive here if K = 5 ~/
…

'Step(3)'  /~ arrive here if K = 3 ~/
…

go to Step(K)
```

In this example, if **K** has a value other than 2, 3, or 5 when the ***GoTo*** statement is executed, the program aborts with a runtime error because the destination does not exist. Having used **Step** as a subscripted label, it is an error to also use **Step** as an unsubscripted label within the routine. It is neither necessary nor permitted to execute a ***Reserve*** or ***Release*** statement for subscripted labels.

A **jump ahead** statement transfers control to the nearest **here** keyword that follows the statement. A **jump back** statement transfers control to the nearest **here** keyword that precedes the statement. A **here** label may be the target of zero, one, or more than one ***Jump*** statements, and several **here** labels may appear within a routine. For example:

```
here            ' ' Point A
…

jump back       ' ' go to Point A
…

jump ahead      ' ' go to Point B
…

here            ' ' Point B
…

jump ahead      ' ' go to Point C
…

jump back       ' ' go to Point B
…

here            ' ' Point C
…
```

Except for a **here** label, this language element must be enclosed in apostrophes. Each apostrophe may be separated from the label by one or more spaces. If a name or keyword immediately precedes the first apostrophe on the same line, or immediately follows the second apostrophe, the name or keyword must be separated from the apostrophe by at least one space.

A label is local to its routine. Two routines may use the same label name without conflict. It is not possible for a ***GoTo*** or ***Jump*** statement in one routine to transfer control to another routine. A label may have the same name as a variable without conflict.

A label may appear anywhere within a routine. However, placing labels within ***If***, ***Loop***, and ***Select*** statements is discouraged. A transfer of control from outside a loop to a label within the body of the loop may result in unpredictable behavior.

2.40 Leave

This statement may be specified in the body of a loop and terminates the loop.

For example, the following loop reads up to **N** positive values and stores them in an array named **Parameter**. If a zero or negative value is entered, the loop is terminated.

```
let J = 1

while J <= N
do
    write J as "Enter parameter ", i *, ": ", +
    read Value
    if Value <= 0
        leave
    otherwise
    let Parameter(J) = Value
    add 1 to J
loop
```

A ***Leave*** statement behaves like a branch to a hidden label that immediately follows the **loop** keyword. The above example can be rephrased as follows:

```
let J = 1

while J <= N
do
    write J as "Enter parameter ", i *, ": ", +
    read Value
    if Value <= 0
        go to Hidden_Label
    otherwise
    let Parameter(J) = Value
    add 1 to J
loop

'Hidden_Label'
```

If the **loop** keyword that follows the ***Leave*** statement marks the end of two or more nested loops, then the ***Leave*** statement terminates the outermost of these nested loops.

Example 1:

```
for I = 1 to N              ' ' outer loop
    for J = 1 to N          ' ' inner loop
    do
        …
        leave               ' ' terminates the outer loop
        …
    loop

' ' control is transferred here
```

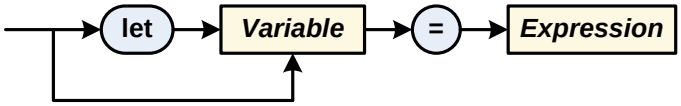
Example 2:

```
for I = 1 to N              ' ' outer loop
do
    for J = 1 to N          ' ' inner loop
    do
        …
        leave               ' ' terminates the inner loop
        …
    loop
    ' ' control is transferred here
loop
```

Example 3:

```
for I = 1 to N              ' ' outer loop
do
    also for J = 1 to N     ' ' inner loop
    do
        …
        leave               ' ' terminates the outer loop
        …
    loop

' ' control is transferred here
```

2.41 Let



When this statement is executed, the ***Expression*** on the right-hand side of the equal sign is evaluated and its value is assigned to the ***Variable*** on the left-hand side of the equal sign. This statement may be used in any routine. The keyword **let** is optional for readability.

The ***Expression*** on the right-hand side is the "source" of the assigned value and the ***Variable*** on the left-hand side is the "destination" of the assigned value. The source and destination must be compatible. The following assignments are permitted.

1. The modes of the source and destination can be any combination of numeric modes: double, real, integer, integer4, integer2, and alpha. These modes are listed in order from "largest" to "smallest." Loss of precision can occur if the mode of the source is "larger" than the mode of the destination. An implicit call of **library.m** function **int.f** rounds a double or real source to the nearest integer when assigning to an integer, integer4, integer2, or alpha destination. In the following example, a double source, **17.5**, is rounded to **18** and then assigned to the integer destination **N**:

   ```
   define N as an integer variable
   let N = 17.5
   ```

2. The modes of the source and destination can be any combination of text and alpha. **Library.m** function **ttoa.f** is called implicitly when assigning a text source to an alpha destination, and **atot.f** is called implicitly when assigning an alpha source to a text destination. In the following example, the alpha destination **A** is assigned the value **"H"**, which is the first character of the text source **T**:

   ```
   define A as an alpha variable
   define T as a text variable
   T = "Hello"
   A = T
   ```

3. The modes of the source and destination can be any combination of pointer, integer, and reference modes, except that if both are reference modes, they must be the same reference mode or the destination reference mode must be a base class of the source reference mode. In the following example, assume that class **Freighter** is derived from class **Ship**. A **Freighter** reference variable can be assigned to a **Ship** reference variable:

```
define S as a Ship reference variable
define F as a Freighter reference variable
create F
let S = F  ' ' allowed because every Freighter is a Ship
let F = S  ' ' not allowed
```

4. The source and destination can be any combination of pointer mode, integer mode, and array pointer, except that if both are array pointers, they must be compatible. A source array is compatible with a destination array if they have the same dimensionality and compatible modes, or if the mode of the destination array is pointer and its dimensionality is less than the dimensionality of the source array. For example:

```
define X and Y as 2–dim double arrays
define A as a 2–dim alpha array
define D as a 1–dim double array
define P as a 1–dim pointer array
define PTR as a pointer variable
reserve X as 20 by 30
Y = X          ' ' allowed
A = X          ' ' not allowed
D = X          ' ' not allowed
D = X(1)       ' ' allowed
P = X          ' ' allowed
PTR = X        ' ' allowed
```

5. If the destination is a subprogram variable, the source can be a subprogram variable, a subprogram literal, or **0** (zero). In the following example, assume that **Load** is the name of a subroutine:

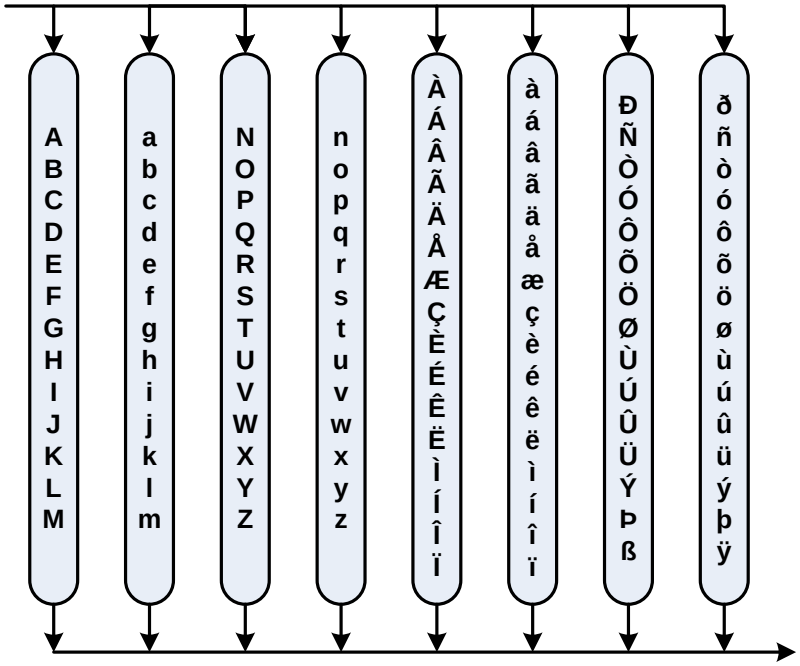
```
define Job as a subprogram variable
Job = 'Load'
perform Job  ' ' calls Load
```

Note that a variable can appear on the both the left-hand and right-hand side of the equal sign, and may appear more than once on the right-hand side. The following statement obtains the value of variable **X**, squares it, and assigns the result to **X**:

```
let X = X * X
```

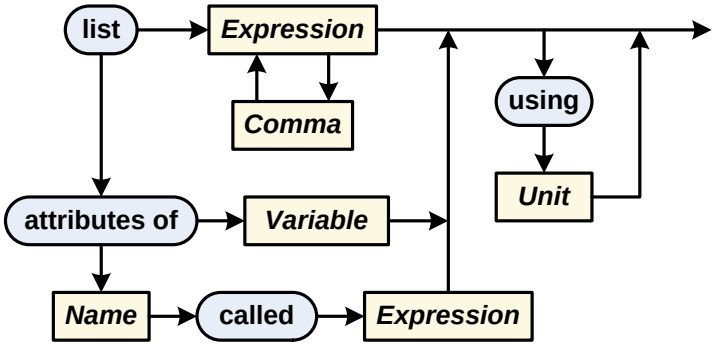
If **X** is monitored on the right, then the right implementation of function **X** is called twice, once for each occurrence of **X** on the right-hand side. If **X** is monitored on the left, then the left implementation of function **X** is called once with the assigned value.

2.42 Letter



This language element is an uppercase or lowercase alphabetic Latin1 character, which may appear in a ***Name*** or ***String***, and in a comment or keyword.

2.43 List



This statement, which may be used in any routine, writes the values of one or more program variables to an output unit using a standard format. It is helpful when debugging a program. It has three forms.

1. **List *Expression***. One or more expressions may be specified of any mode and the value of each expression is displayed. For example, if **J** and **K** are integer variables containing the values 23 and 17, respectively, then this statement,

   ```
   list J, K, J + K – 2
   ```

   writes the following lines to the current output unit:

   ```
   J =       23
   K =       17
   J + K – 2 =       38
   ```

   If the name of an array is specified (i.e., a variable with nonzero dimensionality), then each element of the array is displayed. For example, if **Vector** is a one-dimensional double array with four elements, then this statement,

   ```
   list Vector
   ```

   writes the value of each array element to the current output unit:

   ```
   Vector(1) =       4.5218540000
   Vector(2) =       2.8975000000
   Vector(3) =       1.2193333333
   Vector(4) =       9.0371250000
   ```

2. **List attributes of *Variable***. In this form, ***Variable*** must be 0-dimensional (scalar), have a reference mode, and contain the reference value of an object, temporary entity, or process notice; or ***Variable*** must be a 0-dimensional global or local integer variable with the same name as a permanent entity type or resource type,

and must contain an entity number. The attributes of the identified object or entity are displayed. For example, suppose **Point** is the name of a class with three object attributes named **X**, **Y**, and **Z**, and suppose that **Center** is a **Point** reference variable. This statement,

```
list attributes of Center
```

writes the reference value stored in **Center** followed by the value of each object attribute, **X(Center)**, **Y(Center)**, and **Z(Center)**:

```
ATTRIBUTES OF CENTER
CENTER = 003E5ED8 (hex)
X =        7.8546345000
Y =        1.0485044232
Z =        2.3530684841
```

3. **List attributes of *Name* called *Expression***. This form is the same as Form 2 except that ***Name*** specifies the class or entity type and ***Expression*** provides the reference value or entity number. If ***Name*** specifies a class, temporary entity type, or process type, then the mode of ***Expression*** must be reference, integer, or pointer. If ***Name*** specifies a permanent entity type or resource type, then the mode of ***Expression*** must be numeric, i.e., double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer. The following statement produces the same output as above:

```
list attributes of Point called Center
```

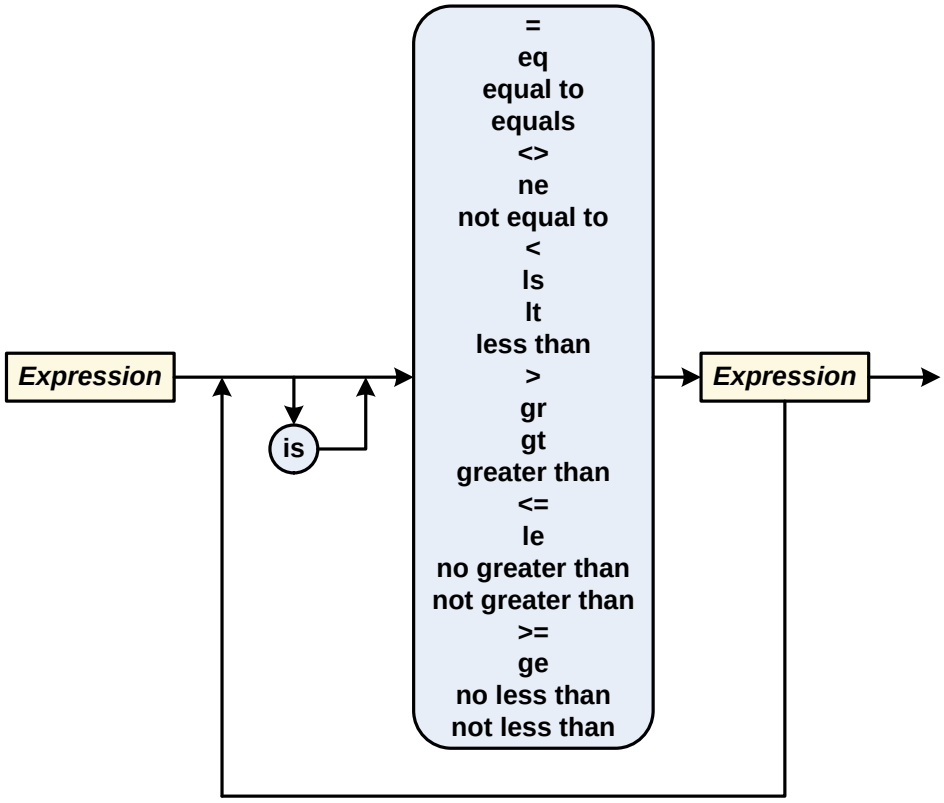
Suppose **Point_Set** is a set of **Point** objects. The following loop writes the attributes of each object in the set:

```
define P as a Point reference variable

for each P in Point_Set
   list attributes of P
```

If a ***Unit*** is specified, output is written to the indicated unit. For example, the following statement writes the value of **Waiting_Time** to unit 14:

```
list Waiting_Time using 14
```

2.44 LogicalComparison



This language element, which is part of a ***LogicalExpression***, compares the values of two or more expressions and produces a value of true or false.

There are six comparison operators: equals (**=**), not equals (**<>**), less than (**<**), greater than (**>**), less than or equals (**<=**), and greater than or equals (**>=**). The keyword **is** is optional for readability. The following are synonymous:

- **=**, **eq**, **equal to**, and **equals**;
- **<>**, **ne**, and **not equal to**;
- **<**, **ls**, **lt**, and **less than**;
- **>**, **gr**, **gt**, and **greater than**;
- **<=**, **le**, **no greater than**, and **not greater than**;
- **>=**, **ge**, **no less than**, and **not less than**.

The two operands of a comparison operator must be compatible. If the operands are assignment compatible according to the rules on page 99, then they may be compared using any of the comparison operators with the following exception: only the **=** and **<>** operators may be used if an operand is pointer mode, reference mode, an array pointer, or a subprogram variable.

If one operand is double (or real) and the other operand is integer (or integer4, integer2, or alpha), then the value of the integer operand is implicitly converted to double before it is compared with the double operand. If an integer comparison is desired, the double operand must be explicitly converted to integer. For example:

```
define X as a double variable
define N as an integer variable
X = 2.25
N = 2

' ' the following performs a double comparison and evaluates to false
if X = N  …  ' ' interpreted as:  if X = real.f(N)  …

' ' the following performs an integer comparison and evaluates to true
if int.f(X) = N  …
```

A comparison of text operands uses the Latin1 collating sequence and is case sensitive. If the value of one text operand has fewer characters than the value of the other text operand, and the shorter value is not the null string, then blanks are implicitly appended to the shorter value to match the length of the longer value. The null string compares less than all other text values. For example:

```
define T as a text variable
T = "abc"                  ' ' assign "abc" to T

if T = "abc"  …            ' ' evaluates to true

if T < "about"  …          ' ' evaluates to true ("c" < "o" in the collating sequence)

if T = "ABC"  …            ' ' evaluates to false (case–sensitive comparison)

if T = "abc       "  …     ' ' evaluates to true (blanks are appended to the value of T)

if T > "ab"  …             ' ' evaluates to true ("c" > blank in the collating sequence)

if "" < "      "  …        ' ' evaluates to true (blanks not appended to a null string)
```

If one operand is text and the other operand is alpha, the value of the alpha operand is implicitly converted to text before it is compared with the text operand. If an alpha comparison is desired, the text operand must be explicitly converted to alpha. For example:

```
define T as a text variable
define A as an alpha variable
T = "hello"
A = "h"

' ' the following performs a text comparison and evaluates to false
if T = A  …  ' ' interpreted as:  if T = atot.f(A)  …

' ' the following performs an alpha comparison and evaluates to true
if ttoa.f(T) = A  …
```

More than two expressions may be connected using more than one comparison operator. For example, the phrase,

```
if Min <= A(J) <= Max
```

is interpreted as:

```
if Min <= A(J) and A(J) <= Max
```

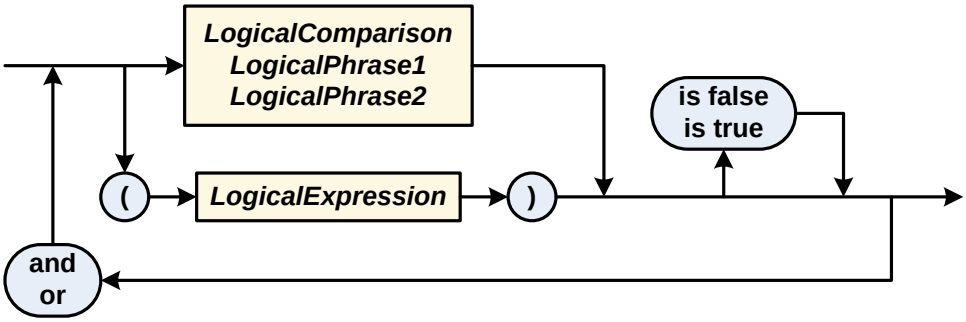
Many expressions may be so connected. For example, the phrase,

```
if N <> M > C / 2 = K <= (L – 1) ** 2
```

is equivalent to:

```
if (N <> M) and (M > C / 2) and (C / 2 = K) and (K <= (L – 1) ** 2)
```

2.45 LogicalExpression



This language element appears in an ***If***, ***While***, or ***With*** phrase and evaluates to true or false. Logical **and** and **or** operators may be used, and logical negation is specified by the **is false** phrase.

For example, the expression **X < Y** evaluates to true if the value of **X** is less than the value of **Y**. The expression may be enclosed in parentheses but it is not a requirement. The following are equivalent:

```
if X < Y  …

if (X < Y)  …
```

Two expressions may be operands of a logical **and** operator. The result is true if *both* operands are true. The following expression is true if **X** is less than **Y** *and* **Y** is less than **Z**:

```
if X < Y and Y < Z  …
```

A shorter form of this expression has an implied **and** operation:

```
if X < Y < Z  …
```

Two expressions may be operands of a logical **or** operator. The result is true if *one or both* operands are true. This is known as an "inclusive or." The following expression is true if the value of **N** is equal to zero or greater than 100:

```
if N = 0 or N > 100  …
```

"Short-circuit" evaluation is used for **and** and **or** operators. If the first operand of **and** is false, the result is false and the second operand is not evaluated. Likewise, if the first operand of **or** is true, the result is true and the second operand is not evaluated.

It is an error to access an attribute using a zero reference value. Because of short-circuit evaluation, the following expressions access attributes only when the **Tanker** reference variable holds a nonzero value:

```
if Tanker <> 0 and Current_Capacity(Tanker) <= Max_Capacity(Tanker)  …

if Tanker = 0 or Current_Capacity(Tanker) > Max_Capacity(Tanker)  …
```

The operands of consecutive **and** operators are evaluated from left to right. The first false operand terminates the evaluation with a result of false. All operands must evaluate to true for the result to be true. For example:

```
if I > 0 and J > 0 and K > 0 and Count <> 0 and Table(I, J, K) > Sum / Count  …
```

Likewise, the operands of consecutive **or** operators are evaluated from left to right. The first true operand terminates the evaluation with a result of true. All operands must evaluate to false for the result to be false. For example:

```
if I <= 0 or J <= 0 or K <= 0 or Count = 0 or Table(I, J, K) <= Sum / Count  …
```

In these examples, the values of **I**, **J**, and **K** are guaranteed to be positive before they are used as array subscripts, and division by zero is prevented.

When **and** and **or** operators appear together, the **and** operators are evaluated first. Thus, the expression,

```
if J > 1 and K = 0 or K = 1 and Queue(J – K) is not empty  …
```

is interpreted as:

```
if (J > 1 and K = 0) or (K = 1 and Queue(J – K) is not empty)  …
```

Parentheses are required to specify a different order of evaluation. For example, the **or** operator is evaluated before the **and** operators in the following expression:

```
if J > 1 and (K = 0 or K = 1) and Queue(J – K) is not empty  …
```

An **is false** phrase negates the preceding expression. For example, the expression,

```
if Demand <= Supply is false  …
```

is equivalent to:

```
if Demand > Supply  …
```

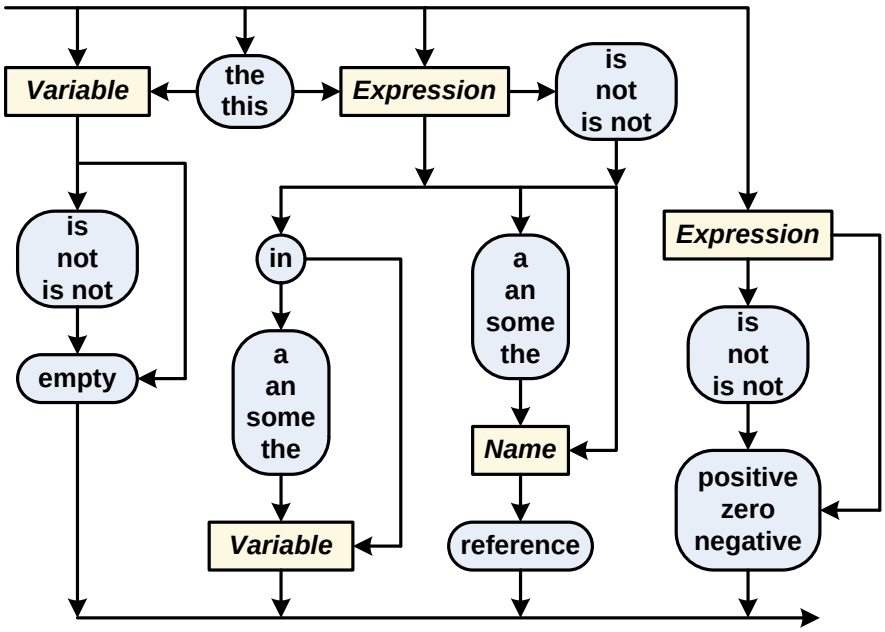
To negate an expression that contains one or more **and** and **or** operators, the expression must be enclosed in parentheses. For example:

```
if (Back_Orders > 0 and Inventory < Minimum_Level) is false  …
```

An **is true** phrase is optional for readability. These expressions are equivalent:

```
if X < Y  …

if X < Y is true  …
```

2.46 LogicalPhrase1



This language element, which is part of a ***LogicalExpression***, produces a value of true or false. It has four forms. The keyword **not** can be used in each form and negates the logical value. The keywords **a**, **an**, **is**, **some**, **the**, and **this** are optional for readability.

1. ***Variable*** **is empty**. ***Variable*** must name a set. This condition evaluates to true if the set has no members. This condition is evaluated by checking whether the pointer to the first member of the set, **f.*Variable***, is equal to zero. For example,

   ```
   if List is empty  …
   ```

   is interpreted as

   ```
   if f.List = 0  …
   ```

   It is an error to attempt to remove a member from an empty set. A check can be placed in the code to verify that the set is non-empty before executing a **remove** statement:

   ```
   if List is not empty
       remove first Item from List
   always
   ```

If an array of sets is named, then explicit or implicit subscripts are required. For example, if **Port** is a one-dimensional array of sets, then the following condition evaluates to true if the third set in the array is empty:

```
if Port(3) is empty  …
```

2. ***Expression* is in *Variable***. ***Expression*** must be nonzero and identify an object or entity. ***Variable*** must name a set in which the object or entity may be a member, and the set name may be followed by zero or one subscript. This condition evaluates to true if the identified object or entity is currently a member of the named set. For example, suppose **every Customer belongs to a Queue** and **every Teller owns a Customer'Queue**. Here we create a **Customer** object and two **Teller** objects and file the **Customer** object in the **Queue** owned by the first **Teller** object:

```
define C as a Customer reference variable
define T1 and T2 as Teller reference variables
create C, T1, T2
file C in Queue(T1)
```

When this **file** statement is executed, the reference value in **T1** is assigned to the set attribute **m.Queue(C)**. Now the following phrase, with no subscript after the set name, evaluates to true because **m.Queue(C)** is nonzero:

```
if C is in Queue  …
```

This phrase evaluates to true if the object identified by **C** is currently a member of *any* **Queue** set, regardless of who is the owner. A subscript may be specified after the set name for a more exacting test. This subscript identifies the owner of the set. In our example, the following phrase evaluates to true,

```
if C is in Queue(T1)  …
```

because **m.Queue(C) = T1**. However, the next phrase evaluates to false because the **Customer** object is not a member of the **Queue** owned by the second **Teller** object:

```
if C is in Queue(T2)  …
```

It is an error to file an object or entity into a set if it is already a member of some set with that name. A check can be placed in the code to verify this before executing a **file** statement. For example:

```
' ' make sure C is not in any Queue before filing it into a specific Queue
if C is not in Queue
    file C in Queue(T1)
always
```

It is an error to remove an object or entity from a set if it is not currently a member of that specific set. A check can be placed in the code to verify set membership before executing a **remove** statement:

```
' ' make sure C is in Queue(T1) before attempting to remove it
if C is in Queue(T1)
   remove C from Queue(T1)
always
```

The owner of a set can be identified by a single value if

- the set is 0-dimensional and owned by an object (the identification is the reference value of the object);
- the set is 1-dimensional and owned by a class or by the system or subsystem (the identification is the set subscript);
- the set is owned by a temporary entity or process notice (the identification is the reference value of the entity); or
- the set is owned by a permanent entity or resource (the identification is the entity number).

If the owner of a set cannot be identified by a single value, then the **m.*set_name*** attribute of an object or entity is assigned a value of 1 when the object or entity is filed into the set. In this case, the owner cannot be identified by a subscript following the set name and it is not possible to test for membership in a specific set.

3. ***Expression* is a *Name* reference**. This phrase is used to determine the type of the object or entity identified by ***Expression***. The mode of ***Expression*** must be integer, pointer, or any reference mode. ***Name*** must be the name of a class, temporary entity type, or process type. If ***Name*** specifies a class, this condition evaluates to true if the value of ***Expression*** is the reference value of an object of the specified class, or an object of a class that is derived from the specified class. If ***Name*** specifies a temporary entity type or process type, this condition evaluates to true if the value of ***Expression*** is the reference value of an entity of the specified type.

   One use of this phrase is to safely assign a reference value from an integer or pointer variable to a reference variable. For example, suppose **P** is a pointer variable which may contain the reference value of a **Customer** object. The following code safely assigns its value to a **Customer** reference variable named **C**:

```
if P is a Customer reference
   let C = P
always
```

   In SIMSCRIPT II.5 programs, the reference value of a temporary entity is stored in an integer or pointer variable. This phrase can be used to verify that the variable holds the reference value of a temporary entity before accessing one of

its attributes. For example, suppose **Duration** is an attribute of a temporary entity type named **Task**, and **P** is a pointer variable.

```
if P is not a Task reference
    destroy P
    create a Task called P
always

let Duration(P) = 2.5
```

4. Determining whether an ***Expression*** is positive, zero, or negative:

Expression **is positive**	is equivalent to	***Expression*** **> 0**
Expression **is not positive**	is equivalent to	***Expression*** **<= 0**
Expression **is zero**	is equivalent to	***Expression*** **= 0**
Expression **is not zero**	is equivalent to	***Expression*** **<> 0**
Expression **is negative**	is equivalent to	***Expression*** **< 0**
Expression **is not negative**	is equivalent to	***Expression*** **>= 0**

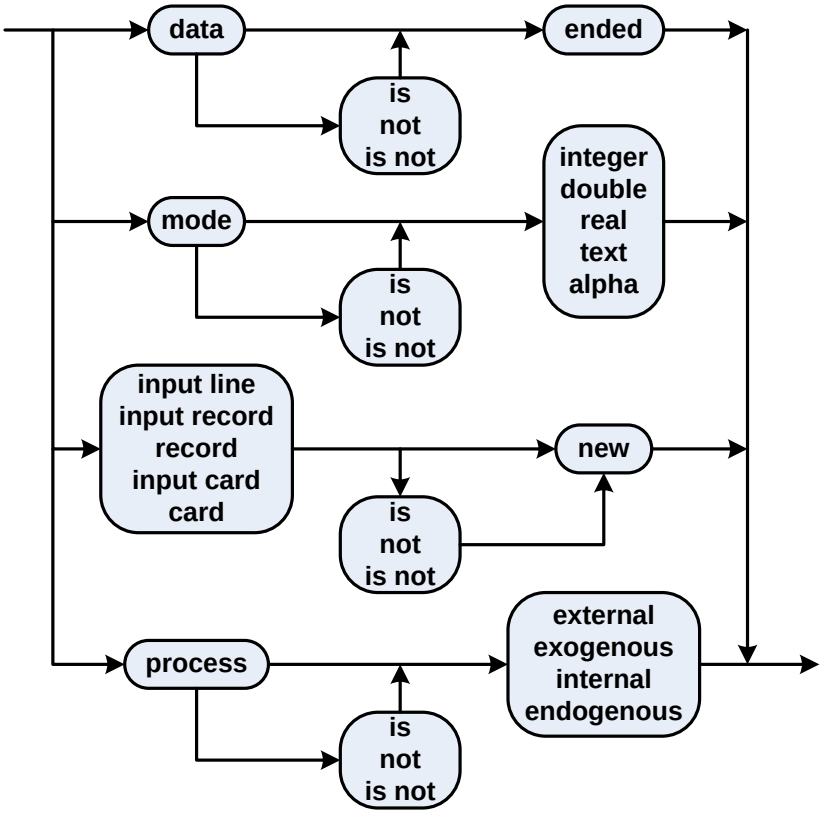
For example, the phrase,

```
if J is positive and Count(J) is not negative
```

is interpreted as

```
if J > 0 and Count(J) >= 0
```

2.47 LogicalPhrase2



This language element, which is part of a ***LogicalExpression***, produces a value of true or false. It has four forms. The keyword **not** can be used in each form and negates the logical value. The keyword **is** is optional for readability. The following are synonymous:

- **double** and **real**;
- **text** and **alpha**;
- **input line**, **input record**, **record**, **input card**, and **card**;
- **external** and **exogenous**;
- **internal** and **endogenous**.

1. **Data is ended**. This condition evaluates to true if there are no more values (i.e., non-blank characters) to be read from the current input unit. It indicates whether end-of-file has been reached when performing free-form read operations and may be used only for character (non-binary) input units. For example:

```
until data is ended  ' ' while not at end–of–file
do
    read Value
    …
loop
```

2. **Mode is …** This condition determines the mode of the next value to be read from the current input unit: **mode is integer** is true if the next value is an integer; **mode is double** is true if the next value is a real number; and **mode is text** is true otherwise, i.e., the next value is not integer or real, or there are no more values to be read (**data is ended**). This condition may be used only for character (non-binary) input units. The following loop reads integer values from the current input unit until a non-integer value is encountered or end-of-file is reached:

```
while mode is integer  '' while the next value is integer
do
   read Value
   …
loop
```

3. **Input line is new**. This condition evaluates to false if the next value to be read from the current input unit appears on the current input line. It evaluates to true if there is no current input line (no lines have been read from the current input unit or end-of-file has been reached), or if all remaining unread characters on the current input line are blank (hence, the next value to be read appears on a "new" input line). This condition may be used only for character (non-binary) input units. The following loop reads all remaining values on the current input line:

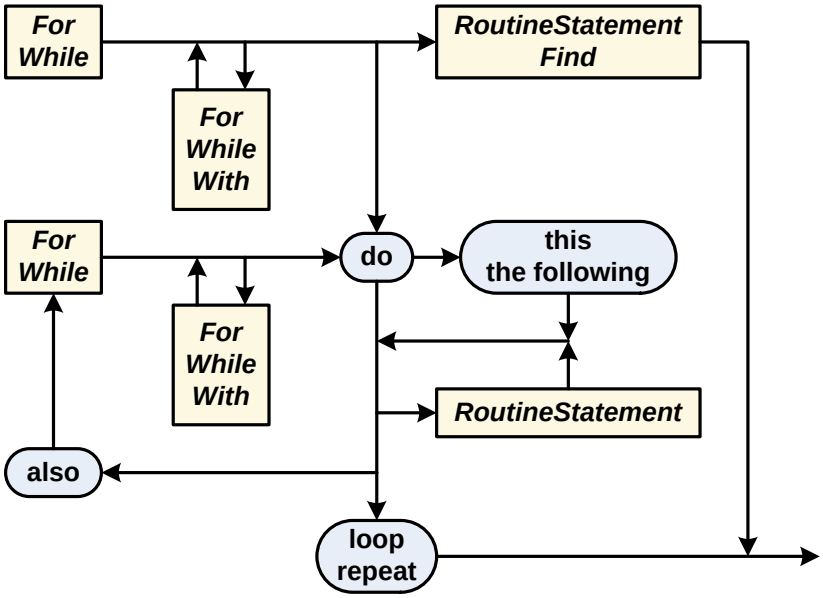
```
until input line is new  '' while the next value is on the current line
do
   read Value
   …
loop
```

   Suppose the next value to be read appears on a new input line (**input line is new** is true). Evaluating a **data is …** or **mode is …** condition advances the current input line to this new line (i.e., the new line becomes the current input line). This causes subsequent evaluation of **input line is new** to be false.

4. **Process is …** The condition, **process is external**, evaluates to true if the currently-executing process was scheduled by reading an external process record from an external process unit. Otherwise, this condition evaluates to false and **process is internal** evaluates to true. (An external process that is suspended is considered internal upon resumption because the resumption was scheduled internally.) This condition is typically used in the process routine for a process type that can be scheduled both externally and internally. The given arguments of the routine are assigned values upon entry to the routine if the process was initiated internally; otherwise, their values can be read from the external process record. For example:

```
process Target given X, Y  '' values assigned to X, Y if process is internal
   if process is external
      read X, Y  '' read values from the external process record
   always
```

2.48 Loop



This language element specifies a loop and may appear in any routine. It is a sequence of one or more ***For***, ***While***, and ***With*** loop control phrases, followed by the "body" of the loop, which is a single statement standing alone, or a sequence of zero or more statements between the keywords **do** and **loop**. The keywords **this** and **the following** are optional for readability. The keywords **loop** and **repeat** are synonymous.

The first loop control phrase must be a ***For*** or ***While*** phrase. A ***For*** phrase executes the body of the loop for each value assigned to a control variable. The loop in the following example initializes each element of array **A** to –1. The assignment statement is the body of the loop, which is executed once for each value of the control variable **J**, from 1 to the number of elements in the array.

```
for J = 1 to dim.f(A)
    let A(J) = –1
```

In the next example, the body of the loop is the sequence of statements enclosed within **do** and **loop**. It is executed for each member of the set named **Inventory**. The control variable **Product** refers to a different member of the set in each iteration. This loop reduces the price of each product by 20% and writes a message to indicate the new price.

```
for each Product in Inventory
do
    let Price(Product) = 0.8 * Price(Product)
    write Name(Product), Price(Product)
        as t *, " marked down to $", d(6, 2), /
loop
```

The ***For*** phrase has 12 different forms; we have illustrated two of the forms above. A ***For*** phrase represents the following logic. See ***For*** on page 70 for more information.

assign the initial value to the control variable

'Loop_Begin'
check the value of the control variable;
if the terminating condition has been reached, go to 'Loop_End'

execute the body of the loop

assign the next value to the control variable
go to 'Loop_Begin'

'Loop_End'

The first control phrase of a loop may be a ***While*** phrase, which executes the body of the loop until a terminating condition has been reached. If the **while** keyword is used in the phrase, the loop terminates when the specified ***LogicalExpression*** becomes false; if the **until** keyword is used, the loop terminates when the ***LogicalExpression*** becomes true. For example:

```
while Num_Requested <= Num_Available
do
   subtract Num_Requested from Num_Available
   read Num_Requested
loop
```

This loop may be equivalently expressed using the **until** keyword:

```
until Num_Requested > Num_Available
do
   subtract Num_Requested from Num_Available
   read Num_Requested
loop
```

An initial ***While*** phrase represents the following logic. See ***While*** on page 222 for more information.

'Loop_Begin'
if the terminating condition has been reached, go to 'Loop_End'

execute the body of the loop

go to 'Loop_Begin'

'Loop_End'

The initial ***For*** or ***While*** phrase may be qualified by a ***With*** phrase that follows it. The ***With*** phrase indicates which executions of the body of the loop are to take place and which are to be skipped or bypassed. It specifies a ***LogicalExpression*** that is evaluated

after the loop terminating condition has been tested (and the terminating condition has not been reached), but before the body of the loop is executed. If the **with** keyword is used in the phrase and the ***LogicalExpression*** is false, or if the **unless** keyword is used and the ***LogicalExpression*** is true, the body of the loop is bypassed for this iteration. However, the loop is not terminated and continues on.

A ***With*** phrase that follows a ***For*** phrase typically specifies a ***LogicalExpression*** that refers to the control variable named in the ***For*** phrase. In the following example, prices are marked down only on products whose actual sales are less than forecasted sales.

```
for each Product in Inventory
with Sales(Product) < Sales_Forecast(Product)
do
   let Price(Product) = 0.8 * Price(Product)
   …
loop
```

This loop may be equivalently expressed using the **unless** keyword:

```
for each Product in Inventory
unless Sales(Product) >= Sales_Forecast(Product)
do
   let Price(Product) = 0.8 * Price(Product)
   …
loop
```

This loop may be expressed in other, less succinct, ways. For example:

```
for each Product in Inventory
do
   if Sales(Product) < Sales_Forecast(Product)
      let Price(Product) = 0.8 * Price(Product)
      …
   always
loop

for each Product in Inventory
do
   if Sales(Product) >= Sales_Forecast(Product)
      cycle
   otherwise
   let Price(Product) = 0.8 * Price(Product)
   …
loop
```

The ***Cycle*** statement terminates the current iteration of the loop. See ***Cycle*** on page 40 for more information.

A ***While*** phrase appended to a ***For*** phrase does not indicate a "nested" loop (i.e., a loop within a loop). Instead it specifies a second terminating condition for the loop. In the

following example, each platoon receives one new soldier from the reserves. The loop terminates when there are no more platoons or no more soldiers in the reserves.

```
for each Platoon in Company
while Reserves are not empty
do
   remove first Soldier from Reserves
   file Soldier in Staff(Platoon)
loop
```

This loop is expressed equivalently by:

```
for each Platoon in Company
do
   if Reserves are empty
      leave
   otherwise
   remove first Soldier from Reserves
   file Soldier in Staff(Platoon)
loop
```

The ***Leave*** statement terminates the loop. See ***Leave*** on page 97 for more information.

A ***For-While*** combination represents the following logic:

assign the initial value to the control variable

'Loop_Begin'
if the For terminating condition has been reached, go to 'Loop_End'
if the While terminating condition has been reached, go to 'Loop_End'

execute the body of the loop

assign the next value to the control variable
go to 'Loop_Begin'

'Loop_End'

Note that the following example does indicate an inner loop nested within an outer loop and has an entirely different behavior. In this case, all of the soldiers in the reserves are assigned to the first platoon.

```
for each Platoon in Company
do
   while Reserves are not empty
   do
      remove first Soldier from Reserves
      file Soldier in Staff(Platoon)
   loop
loop
```

A special form is permitted when loops end at the same location. Rather than specify a series of **loop** keywords, one for each loop, each nested loop is preceded by **also** and a single **loop** keyword marks the end of all of the loops. Using this form, the above example can be rewritten as follows:

```
for each Platoon in Company
do
   also while Reserves are not empty
   do
      remove first Soldier from Reserves
      file Soldier in Staff(Platoon)
   loop
```

Now suppose there is a limit on the number of soldiers in a platoon. The following example illustrates a ***For-While-With*** combination. Each platoon receives one new soldier from the reserves, but only if the platoon is not already at its staffing limit. As before, the loop terminates when there are no more platoons or no more soldiers in the reserves.

```
for each Platoon in Company
while Reserves are not empty
with n.Staff(Platoon) < Staff_Limit
do
   remove first Soldier from Reserves
   file Soldier in Staff(Platoon)
loop
```

A ***For-While-With*** combination represents the following logic:

```
assign the initial value to the control variable

'Loop_Begin'
if the For terminating condition has been reached, go to 'Loop_End'
if the While terminating condition has been reached, go to 'Loop_End'
if the With condition indicates to skip the current iteration, go to 'After_Body'

execute the body of the loop

'After_Body'
assign the next value to the control variable
go to 'Loop_Begin'

'Loop_End'
```

If the ***With*** phrase precedes the ***While*** phrase (i.e., a ***For-With-While*** combination), the order of the second and third *if* statements is reversed.

The initial ***For*** or ***While*** phrase marks the beginning of an outer loop and each subsequent ***For*** phrase marks the beginning of an inner, nested loop. Here we sum the elements in a three-dimensional array named **X**:

```
let Sum = 0

for I = 1 to dim.f(X)
    for J = 1 to dim.f(X(I))
        for K = 1 to dim.f(X(I, J))
            add X(I, J, K) to Sum
```

Here we write the name of each soldier in each platoon:

```
for each Platoon in Company
    for each Soldier in Staff(Platoon)
        write Name(Soldier) as t *, /
```

Each ***For*** phrase may be qualified by a ***While*** phrase and a ***With*** phrase. Suppose we wish to write only the names of sergeants in platoons stationed locally:

```
for each Platoon in Company
with Location(Platoon) = Local
    for each Soldier in Staff(Platoon)
    with Rank(Soldier) = Sergeant
        write Name(Soldier) as t *, /
```

Note that a ***While*** phrase appended to a nested ***For*** phrase is evaluated for each iteration of the inner loop but terminates the execution of the outer loop. In the next example, the ***While*** phrase terminates the entire loop once an array of soldier names has been filled.

```
let Count = 0

for each Platoon in Company
    for each Soldier in Staff(Platoon)
    while Count < dim.f(List)
    do
        add 1 to Count
        let List(Count) = Name(Soldier)
    loop
```

If this example were written as either of the following, the ***While*** phrase terminates only the inner loop:

```
for each Platoon in Company
do
    for each Soldier in Staff(Platoon)
    while Count < dim.f(List)
    do
        …
    loop
loop
```

```
for each Platoon in Company
do
   also for each Soldier in Staff(Platoon)
   while Count < dim.f(List)
   do
      …
   loop
```

A ***For-While-With-For-While-With*** combination represents the following logic:

assign the initial value to the outer control variable

'Outer_Loop_Begin'
if the outer For terminating condition has been reached, go to 'Outer_Loop_End'
if the outer While terminating condition has been reached, go to 'Outer_Loop_End'
if the outer With condition indicates to skip the current iteration, go to 'Inner_Loop_End'

assign the initial value to the inner control variable

'Inner_Loop_Begin'
if the inner For terminating condition has been reached, go to 'Inner_Loop_End'
if the inner While terminating condition has been reached, go to 'Outer_Loop_End'
if the inner With condition indicates to skip the current iteration, go to 'After_Body'

execute the body of the loop

'After_Body'
assign the next value to the inner control variable
go to 'Inner_Loop_Begin'

'Inner_Loop_End'
assign the next value to the outer control variable
go to 'Outer_Loop_Begin'

'Outer_Loop_End'

A single ***For*** phrase may be qualified by more than one ***While*** phrase and by more than one ***With*** phrase. After the ***For*** terminating condition has been tested, the ***While*** and ***With*** conditions are evaluated in the order in which they appear.

Any ***RoutineStatement*** may appear within the body of a loop. ***Find***, ***Compute***, ***Cycle***, and ***Leave*** statements *must* appear within the body of a loop. When specified inside a loop, ***If***, ***Select***, ***Cycle***, and ***Leave*** statements must be enclosed within a **do** … **loop** block. A ***Find*** statement must stand alone as the only statement of the body of a loop, without **do** and **loop** keywords. See ***Find*** on page 68 for more information.

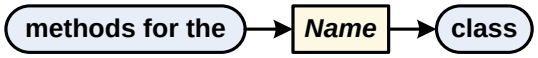
Each of the following definitional statements may appear in the body of a loop, provided they are enclosed within a **do** … **loop** block: ***DefineConstant***, ***DefineToMean***, ***DefineVariable***, ***Normally***, ***Substitute***, and ***SuppressResume***. The definitions and settings established by these statements affect the source code that follows within the routine, including source code beyond the end of the loop. A constant, variable, or substitution

defined within a loop may be accessed by the statements that follow the definition, inside and outside of the loop.

A ***Suspend*** or ***Wait*** statement may appear within the body of a loop and suspends execution of the routine. Upon resumption of the routine, the loop continues.

A ***GoTo*** or ***Jump*** statement may appear within the body of a loop and transfer control to a ***Label*** inside or outside of the loop. It is permitted, but ill defined, to transfer control from outside of a loop to a ***Label*** inside the loop.

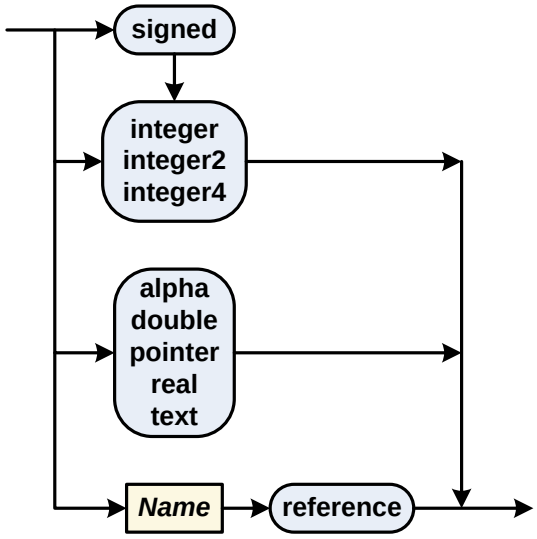
2.49 MethodsHeading



A **methods** heading identifies the class for each method implementation that follows in which the name of the method is not qualified by a class name. This heading may appear after the preamble(s); however, it may not appear within a routine.

A methods heading identifies the class of the method implementations with unqualified names that follow it, up to the next methods heading in the source file or to the end of the source file, whichever comes first. The scope of a methods heading does not span source files.

2.50 Mode



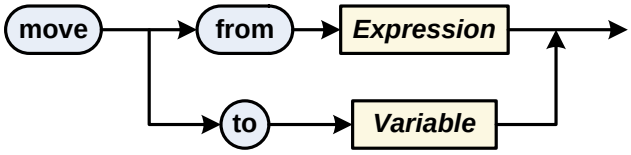
This language element specifies a data type for attributes and variables declared by a **define variable** statement and for routine arguments and functions declared by a **define method** or **define routine** statement. It also specifies the background mode in a **normally** statement.

Integers represent whole numbers. Real numbers can be whole numbers or have fractional parts.

An alpha variable holds one character. A text variable holds a sequence of zero or more characters.

For each declared class, temporary entity type, and process type, a reference mode is implicitly defined. Use of the reference mode may precede the declaration of the class, temporary entity type, or process type.

2.51 Move



This language element is a special kind of assignment statement that may appear only within a monitoring function. A **move from** statement is used in a left monitoring function to assign the value of the ***Expression*** (the "source") to the monitored variable (the "destination"). A **move to** statement is used in a right monitoring function to assign the value of the monitored variable (the "source") to the ***Variable*** (the "destination"). The source must be compatible with the destination according to the assignment compatibility rules on page 99.

Suppose a class named **Road** has a **Name** attribute and a monitored **Length** attribute:

```
begin class Road

    every Road has a Name and a Length
    define Name as a text variable
    define Length as a double variable monitored on the left and right

end
```

A left monitoring function is called each time a value is assigned to a variable or attribute that is monitored on the left. It can be used to validate the assigned value before it is stored and to change the units of measurement if desired. The following method is called whenever a value is assigned to the **Length** attribute of a **Road** object. It verifies that the assigned value is nonnegative, converts it from miles to kilometers, and stores the result in the attribute.

```
left method Road'Length

    define Miles as a double variable
    enter with Miles                    ' ' get the assigned value
    let Miles = max.f(0, Miles)         ' ' replace a negative value with zero
    move from Miles / 0.62137           ' ' store the value in kilometers

end
```

The following statements create a **Road** object and assign values to its attributes:

```
define MyStreet as a Road reference variable
create MyStreet
let Name(MyStreet) = "Jefferson Avenue"

' ' the following assignment invokes left method Road'Length
let Length(MyStreet) = 7.5  ' ' miles
```

When a value of 7.5 miles is assigned to the **Length** attribute, the left method is invoked and the **enter with** statement assigns 7.5 to the local variable named **Miles**. The **move from** statement stores in the **Length** attribute a value of 12.07 kilometers.

A right monitoring function is called each time the value is retrieved from a variable or attribute that is monitored on the right. The following method is called whenever the value is retrieved from the **Length** attribute of a **Road** object. The **move to** statement copies the stored value of the attribute to a local variable named **Kilometers**. This value is then converted to miles and returned.

```
right method Road'Length

   define Kilometers as a double variable
   move to Kilometers                ' ' get the stored value
   return with Kilometers * 0.62137  ' ' convert to miles

end
```

Although the value stored in **Length(MyStreet)** is 12.07, the right method returns 7.5. This method is invoked in this statement,

```
write Name(MyStreet), Length(MyStreet)
   as "The length of ", t *, " is ", d(4,1), " miles.", /
```

which produces the following output:

```
The length of Jefferson Avenue is  7.5 miles.
```

Note that a left monitoring function uses **enter with** and **move from** statements, whereas a right monitoring function uses **move to** and **return with** statements.

It is possible for a derived class to override the monitoring method for an inherited object attribute. Suppose class **Highway** is derived from class **Road** and overrides the **Length** attribute:

```
begin class Highway

   every Highway is a Road and overrides the Length

end
```

The **overrides** phrase allows the **Highway** class to provide a left method and/or a right method for **Length**. Suppose a **Highway** object has a minimum length of 10 miles. This can be enforced by the following left method:

```
left method Highway'Length

   define Miles as a double variable
   enter with Miles                  ' ' get the assigned value
   let Miles = max.f(10, Miles)      ' ' minimum length is 10 miles
   move from Miles / 0.62137         ' ' store the value in kilometers

end
```

However, this duplicates the logic in left method **Road'Length** for converting the value from miles to kilometers. In the following implementation, the inherited method is called to perform the conversion and store the result in the attribute:

```
left method Highway'Length

   define Miles as a double variable
   enter with Miles                  ' ' get the assigned value
   let Miles = max.f(10, Miles)      ' ' minimum length is 10 miles

   ' ' invoke the inherited left method to store the value in kilometers
   let Road'Length = Miles

end
```

Name qualification is required here. If the last statement were replaced by

```
let Length = Miles
```

it is interpreted as

```
let Highway'Length = Miles
```

which causes the left method **Highway'Length** to invoke itself in an infinite recursion.

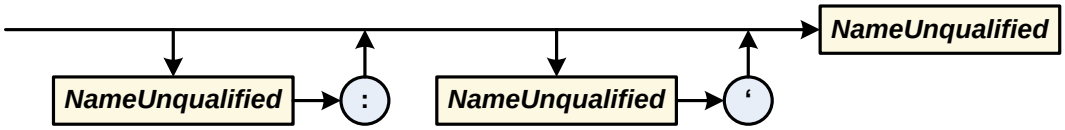
If **Length** were not monitored on the left by class **Road**, then the following statement in left method **Highway'Length**,

```
let Road'Length = Miles
```

is equivalent to

```
move from Miles
```

2.52 Name



This language element is a name that may be qualified by a module name and/or a class name. This element is used in many contexts.

The qualified name **M:X** identifies an **X** that is defined in module **M** to be one of the following:

- a class
- a routine that is not a method
- a global variable
- a global constant
- a temporary entity type or process type
- a permanent entity type or resource type
- an attribute of, or set owned by:
    - the system or subsystem
    - one or more temporary entity types and process types
    - a permanent entity type, resource type, or compound entity type
- a set of:
    - temporary entities and/or process notices
    - permanent entities or resources

The qualified name **X'Y** identifies a **Y** that is defined or inherited by class **X**, where **Y** is one of the following:

- an object attribute
- an object method
- a class attribute
- a class method
- a class constant
- a set of objects
- a set owned by an object or class

The fully-qualified name **M:X'Y** identifies a **Y** defined or inherited by class **X**, where class **X** is defined in module **M**.

Name qualification is required only when the unqualified name identifies more than one definition. The qualification indicates which definition to use. When it is not required, name qualification may be used for readability.

If a name imported from a subsystem is the same as a name defined by the importing module, or if the same name is imported from two or more subsystems, then it is necessary to qualify the imported name within the importing module.

If a name inherited from a base class is the same as a name defined by the derived class, or if the same name is inherited from two or more base classes, then it is necessary to qualify the inherited name within the derived class.

If **Harbor_Queue** is a set owned by the **Shipping** subsystem, its qualified name is **Shipping:Harbor_Queue**. If **Number_At_Sea** is an attribute of the **Freighter** class, which is defined within the **Shipping** subsystem, the fully-qualified name of the attribute is **Shipping:Freighter'Number_At_Sea**. Within the **Shipping** subsystem, it suffices to refer to **Harbor_Queue** and **Freighter'Number_At_Sea**. Within a method of the **Freighter** class (or within a method of a class derived from **Freighter**), **Number_At_Sea** may be unqualified.

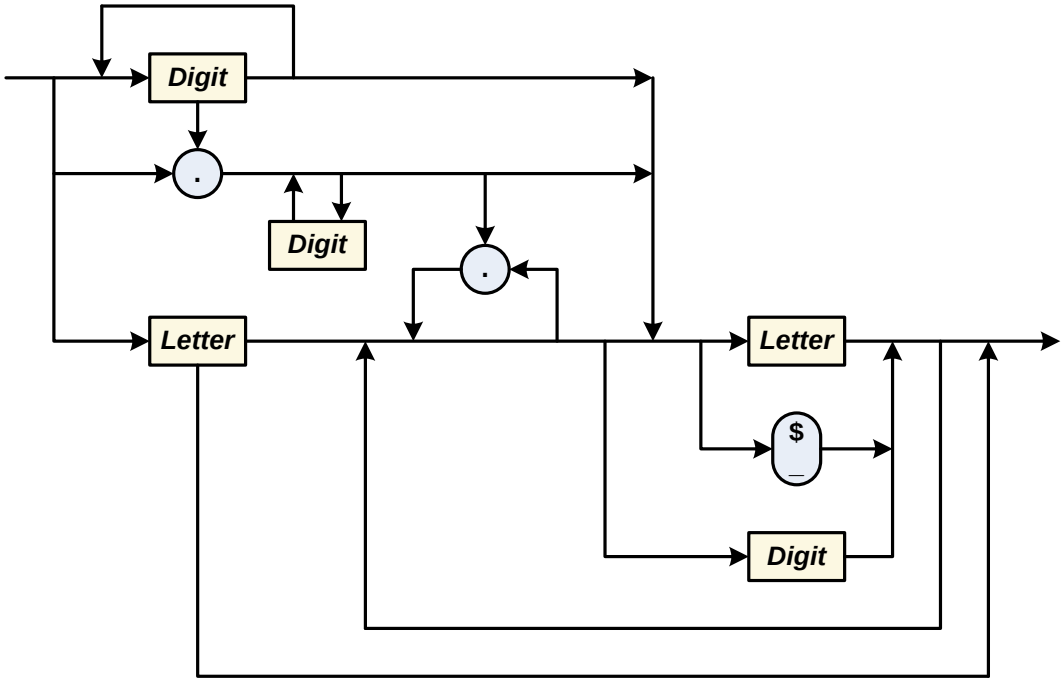
Spaces are not permitted before or after the colon and apostrophe characters used for name qualification. Periods at the end of a name are ignored. These names are equivalent:

Shipping:Freighter'Number_At_Sea
Shipping:Freighter'Number_At_Sea.
Shipping:Freighter'Number_At_Sea..
Shipping:Freighter'Number_At_Sea...

Names of the following are never qualified:

- modules
- local variables and arguments
- local constants
- labels
- accumulate/tally intervals
- substitutions

2.53 NameUnqualified



This language element is a simple name without qualification consisting of a sequence of letters, digits, periods, dollar signs, and underscores. The first character is normally a letter. However, it may be a digit or a period provided that the name is not all periods and is distinguishable from a ***Number***. Although these names are unusual, they are all valid:

u	**L**	**kV**	**Y1**	**T$**	**g_**	**.Z**	**.$**	**._**	**2w**	**5$**
PbZ	**v.S**	**R.2**	**d._**	**.$$**	**.B5**	**..4**	**78Q**	**9_9**	**1.2.3**	**4..1**

Names are typically chosen that are longer and more meaningful:

Ship	**Distance_in_km**	**dock.ID**	**Average.Cost**
capacity	**NumberOfSteps**	**TOTAL$**	**FAVORITE_CAFÉ**
Queue	**.max.queue.length**	**Table2**	**Straße_des_17**

Names are case insensitive; therefore, each of these refers to the same name:

FAVORITE_CAFÉ	**Favorite_Café**	**favorite_café**
FAVORITE_café	**favorite_CAFÉ**	**FaVORiTe_caFÉ**

Except for the keyword **and**, language keywords may be used as names:

set	**Mean**	**FILE**	**list**	**Print**	**array**	**HISTOGRAM**

To avoid conflicts with implicitly-defined names, do not choose names that begin with a letter followed by a period, or end with a period followed by a letter:

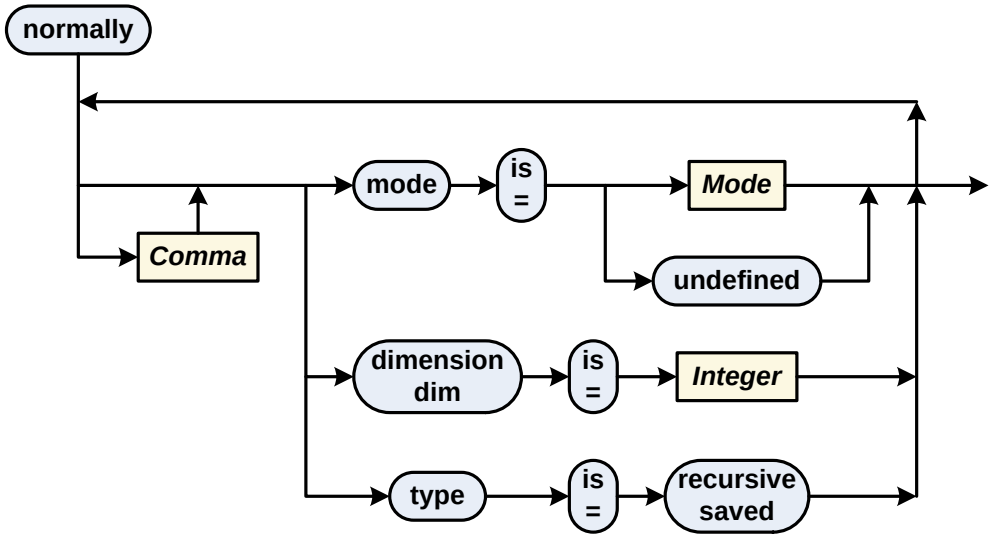
N.QUEUE i.Travel random.f pi.c

There is no limit to the length of a name. These are valid names:

periodic_sum_of_downtime_resulting_from_mechanical_failure
Average.Number.of.Unexpected.Customer.Arrivals.per.Week

A name may not be split across lines.

2.54 Normally



A **normally** statement sets the "background" or default mode, dimensionality, and/or the type of local variables (**recursive** or **saved**). This statement may appear in a preamble or routine.

The following are synonymous:

- **is** and **=**;
- **dimension** and **dim**.

A normally statement may have one, two, or three phrases in any order. Can phrases be duplicated in the same statement?

By default, tt the beginning of a preamble, normally mode is undefined, dimension is zero, type is recursive.

Setting the background mode means that all variables and attributes have the background mode unless otherwise specified.

The settings at the end of a routine do not carry over to other routines.

There can be more than one normally statement in a preamble and in a routine. Each normally statement sets one or more background conditions that hold until overridden.

"mode is undefined" will produce a diagnostic for every undefined variable. This is strongly recommended to guard against inadvertent background definition is misspelled variables.

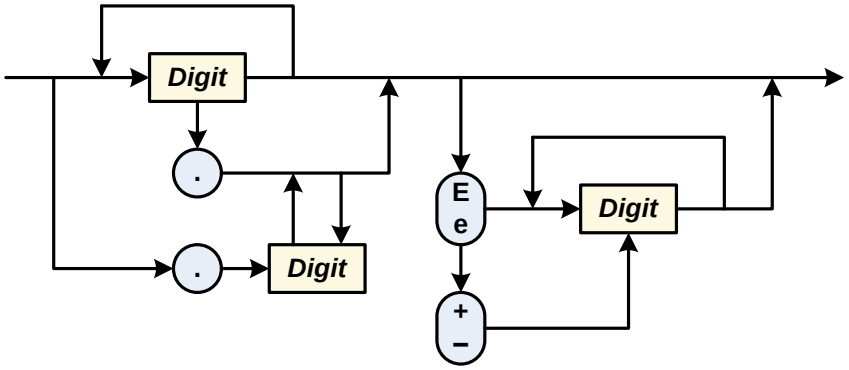
At the beginning of a begin class block or method, normally mode is undefined, dimension is 0 and type is recursive. These settings may be changed by a normally statement within a begin class block or method but the settings are no longer in effect when the end of the begin class block or method is reached.

A normally statement appearing in a subsystem's public preamble affects the interpretation of the subsystem's source code but does not affect importing modules.

"Normally type is ..." is not allowed in a "begin class" block.

At the beginning of every preamble, every "begin class" block, and every routine, the background settings are mode is undefined, dimension is 0, and type is recursive. There is no carryover of settings from a preamble to routines, or from a public preamble to a private preamble. "Normally type is" may not be specified in a preamble.

2.55 Number



This language element represents a nonnegative integer or double value. It appears in many contexts. The following are examples:

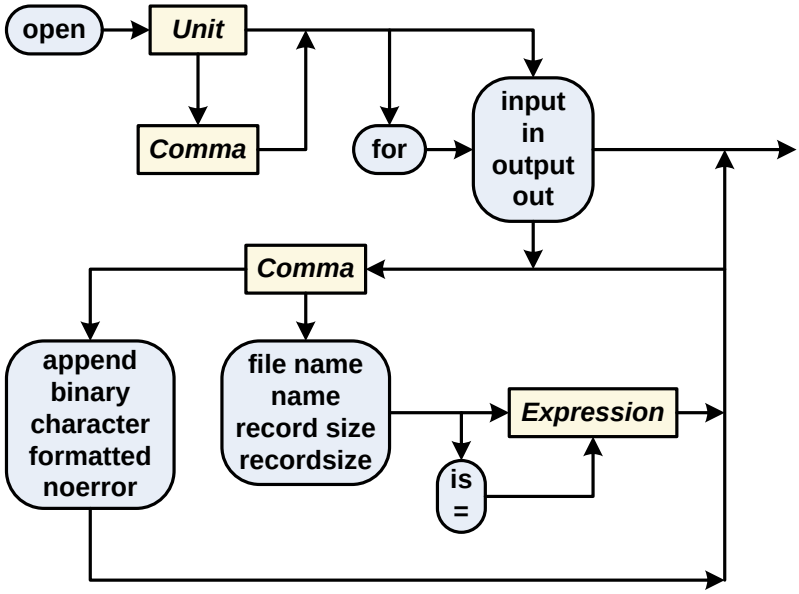
0	**0.0**	**.025**	**.5**	**0.5**
4	**04**	**4.**	**4.00**	**0004.00000**
1024		**1024.0**		**001024.**
4289.1750		**3173214.1**		**20000000**

A double value may be specified using scientific notation. In the exponent, the characters **E** and **e** are synonymous. If no sign is given, plus is assumed. Each of the following represents the value **8125**:

8125	**81250.0E–1**
81.250e+02	**0812500000e–5**
8.125E3	**.000000008125000e12**

Leading zeros are ignored. If no decimal point or exponent is specified, the mode of the ***Number*** is integer; otherwise, it is double. Thus, **8125** is integer but **8125.** and **8125e0** are double.

2.56 Open



This statement, which may be used in any routine, opens the specified I/O unit. A file is associated with the unit. The keywords **for**, **file**, and **is**, the equal sign, and the ***Comma*** after ***Unit***, are optional for readability. The following are synonymous:

- **input** and **in**;
- **output** and **out**;
- **character** and **formatted**;
- **record size** and **recordsize**.

This statement indicates whether the I/O unit will be used for input or output. It may be followed by a ***Use*** statement (see page 215) which designates the unit as the current input unit or current output unit, or the unit may be specified in the **using** phrase of an I/O statement. It is an error to use the unit for output if it has been opened for input or to use the unit for input if it has been opened for output. For example:

```
open 12 for input
use 12 for input

open Report_Unit for output
write Report_Title as t *, / using Report_Unit
```

The unit number specified in an ***Open*** statement must be in the range 1 to 99, but may not be one of the special units: 5 (standard input), 6 (standard output), 98 (standard error), and 99 (the buffer). At the beginning of program execution, these units are opened automatically. It is an error to open a unit that is already open.

Several options may be specified in any order after the **input** or **output** keyword. If **character** is specified, the associated file contains zero or more lines of varying length. Each line is a sequence of zero or more ASCII or Latin1 characters and is terminated by an end-of-line or "newline" character. The default record size for character files is 132, which means each line may contain up to 132 characters, not counting the end-of-line character. If a line is read or written that is longer than the record size, it is implicitly divided into multiple lines. A record size may be specified, for example:

```
open 12 for input, character, record size = 256
```

If **binary** is specified, the associated file contains zero or more fixed-length records of binary data. Each record contains the number of bytes indicated by the record size, which defaults to 128. For example:

```
open 20 for output, binary, record size = 1024
```

It is an error to specify both **character** and **binary**. If neither **character** nor **binary** is specified, **character** is assumed. The record size ***Expression*** must have a positive value and a numeric mode, i.e., double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer.

The name of the associated file may be specified by a text or alpha ***Expression***. For example:

```
open 3 for input, binary, name = "input.dat"          ' ' binary input file
open 4 for output, name = Report_File_Name            ' ' character output file
```

If the name of the file is unspecified, it defaults to "SIMU*nn*" where *nn* is the two-digit unit number. For example:

```
open 51 for output      ' ' unit 51 is associated with the file named "SIMU51"
open 2 for input        ' ' unit 2 is associated with the file named "SIMU02"
```

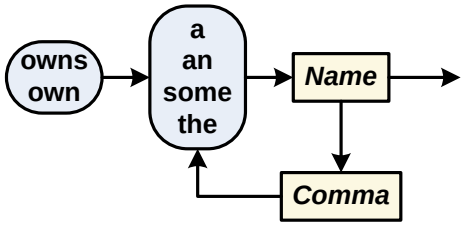
The first time a unit is used (i.e., specified in a ***Use*** statement or **using** phrase), the associated file is accessed. If the file cannot be accessed (e.g., an input file does not exist or an output file cannot be created), the program is terminated with a runtime error unless **noerror** is specified in the ***Open*** statement. If **noerror** is specified, a **library.m** variable, **ropenerr.v** for an input unit or **wopenerr.v** for an output unit, is set to zero if the file is accessible or set to a nonzero value if the file is inaccessible. For example:

```
write as "Enter input file name: ", +
read Input_File_Name

open 7 for input, noerror, name = Input_File_Name
use 7 for input             ' ' try to access the input file
if ropenerr.v = 0           ' ' the input file is accessible
   ' ' we can now read from the input file using unit 7
   …
else                        ' ' failure
   close 7                  ' ' disassociate unit 7 from the inaccessible file
   write Input_File_Name as t *, " cannot be accessed", /
always
```

Normally when opening an output file, if a file already exists with the specified name, that file is deleted and a new empty file is created. However, if **append** is specified in the ***Open*** statement, the existing file is retained and any data written to the output unit is appended to the file. For example:

```
open 1 for output, name = "qlength.txt", append
use 1 for output
write Queue_Length as d(5,1), /  ' ' this line is appended to "qlength.txt"
```

2.57 Owns



An **owns** phrase that appears in an **every** statement within a **begin class** block declares sets of objects and sets of entities that are owned by an object of the class. An **owns** phrase that appears in **the class** statement declares sets of objects and sets of entities that are owned by the class.

An **owns** phrase that appears in an **every** statement outside a **begin class** block declares sets of objects and sets of entities that are owned by a temporary entity, process notice, permanent entity, resource, or compound entity. An **owns** phrase that appears in **the system** or **the subsystem** statements declares sets of objects and sets of entities that are owned by the system or subsystem.

The following are synonymous:

- **owns** and **own**;
- **a**, **an**, **some**, and **the**.

The set has the background dimensionality for sets owns by "the system/subsystem/class" and sets owns by an object.

An entity or object of one type can own a set of entities or objects of that type.

A set of objects may be owned by an object, class, temporary entity, process notice, permanent entity, resource, compound entity, the system, or the subsystem.

An object or class may own a set of objects; a set of temporary entities and/or process notices; or a set of permanent entities and/or resources.

An owns phrase refers to a set named in a belongs phrase. If it is a set of objects of another class that is owned, the name of the set must be qualified by the name of the other class; however, the set is known by its unqualified name within the owner's class. The unqualified set name is prefixed by f., l., and n. for owner attributes. As a result, it is not possible to declare "every X owns a Y'QUEUE and a Z'QUEUE" because it would define two sets named X'QUEUE and two trios of set attributes, f.QUEUE, l.QUEUE,

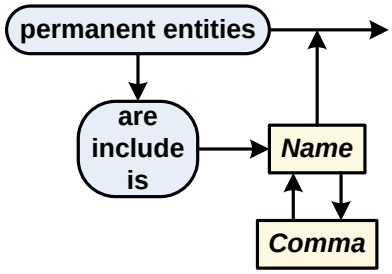
and n.QUEUE. It is, however, permitted to declare "every belongs to a QUEUE and owns a Y'QUEUE". In a preamble, X'QUEUE refers to the set of X objects, whereas in executable code, X'QUEUE refers to a set of Y objects owned by an X object.

The same problem occurs if a temporary entity, permanent entity, compound entity, or the system/subsystem wishes to own two or more sets named QUEUE (without defining an array of sets).

An object, the class, an entity, and the system/subsystem may own any number of sets and arrays of sets.

An object of a derived class inherits the ability (and needed set attributes) to own the sets owned by objects of its base classes.

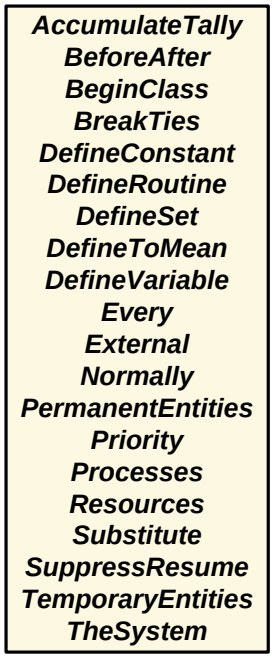
2.58 PermanentEntities



A **permanent entities** statement declares the names in the statement, and the entity types declared by subsequent **every** statements, as permanent entity types. This statement may appear in a preamble, but may not appear in a **begin class** block. The keywords **are**, **include**, and **is** are synonymous.

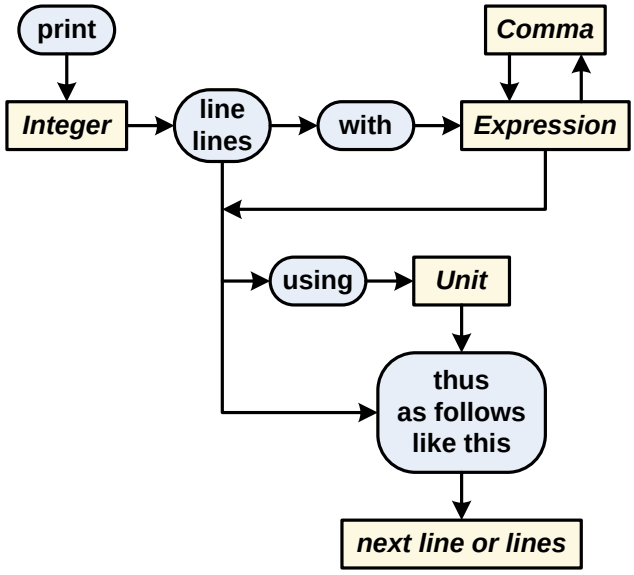
Global variable with same name as the entity type is implicitly defined, as well as global variable n.entity.

2.59 PreambleStatement



A preamble contains statements from this list.

2.60 Print



This statement, which may be used in any routine, writes one or more source code lines verbatim to an output unit, including the formatted values of zero or more expressions.

The following are synonymous:

- **line** and **lines**;
- **thus**, **as follows**, and **like this**.

The ***Integer*** specifies the number of source lines to write. These lines must immediately follow the line on which **thus** (or one of its synonyms) appears. For example, the statement,

```
print 2 lines thus
Simulation Output Report for Run # 37
   Duration of Run = 120.0 hours
```

writes these two lines verbatim to the current output unit:

```
Simulation Output Report for Run # 37
   Duration of Run = 120.0 hours
```

The values of expressions may be inserted into the output. Asterisks placed within the source lines indicate the location and format of these values. The following statement produces the same output as above, but obtains the run number and duration from variables **Run_Num** and **End_Time**, respectively.

```
print 2 lines with Run_Num, End_Time thus
Simulation Output Report for Run # **
   Duration of Run = ***.* hours
```

The value of the first ***Expression*** (from **Run_Num**) is placed in the first field (******), and the value of the second ***Expression*** (from **End_Time**) is formatted in the second field (*****.***). One field must be specified for each ***Expression***.

The ***Print*** statement is convenient for producing tabular reports. In the following example, each **Customer** object has attributes **Name** (text), **Priority** (integer), and **Arrival_Time** (double). We first write the title and column headings. Note that a blank source line writes a blank output line.

```
print 3 lines thus
                Customers in the Queue

Customer Name        Priority        Time of Arrival
```

Now we write one line for each customer in the queue.

```
for each Customer in the Queue
   print 1 line with Name(Customer), Priority(Customer),
   Arrival_Time(Customer) thus
   ************          **                *.**
```

The output from these statements may look like this:

```
                Customers in the Queue

Customer Name        Priority        Time of Arrival
   Robinson             10              143.57
   Smith                 8               97.01
   Jackson               8              147.92
   Williams              5              117.81
```

The value of an integer or double ***Expression*** is right-justified within a field. A period may appear among the asterisks and indicates the column of the decimal point. A double value is rounded to the least significant digit that is displayed. If the integer or double value is too large to fit in the field, blank columns to the left of the field are used.

The value of a text or alpha ***Expression*** is left-justified within a field. If the length of the text value exceeds the number of asterisks, the text value is truncated on the right.

A vertical bar (**|**) may be used in place of the first asterisk of a field to specify contiguous fields. For example, two contiguous three-column fields are specified by *****|****. The sequence ****||**** specifies a two-column field (******) followed by a one-column field (**|**) and a three-column field (**|****). Each vertical bar represents the first column of a new field.

A sequence of eight or more consecutive periods defines a field in which an integer or double value is written using scientific notation. For example, if **X** is a double variable containing the value 83026.75, then the statement,

```
print 2 lines with X thus
The value is .............
at the end of the simulation.
```

writes these lines:

```
The value is +8.30268E+004
at the end of the simulation.
```

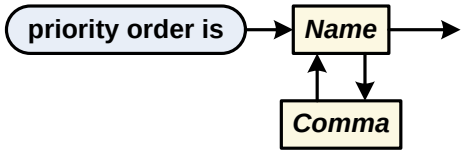
If the current output line is not a new line, then the first line written by a ***Print*** statement is appended to the current line; however, each subsequent line is written as a new line. For example, the following statements produce the same output as above.

```
write as "The value is "
print 2 lines with X thus
.............
at the end of the simulation.
```

If a ***Unit*** is specified, lines are written to the indicated unit. For example, the following statement writes a line to unit 18:

```
print 1 line with X, Y using 18 thus
X = *****.***       Y = *****.***
```

2.61 Priority



A **priority** statement specifies the order in which process methods and/or processes are to be executed when they are scheduled to occur at the same simulation time. This statement may appear in a preamble.

Process methods and processes not mentioned in a priority statement are given lower priority than those that are listed, and are ranked among themselves in the order in which they first appear in preamble declarations, with first appearance given higher priority.

Both internal and external processes may be specified. It is not possible to give priority to external process units.

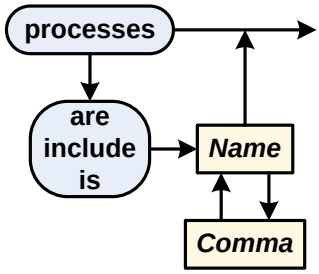
The first process method or process named is given the highest priority.

A priority statement must follow statements defining the process types, such as "processes" and "external processes".

A priority statement specified outside of a begin class block may refer to process methods.

A priority statement inside a "begin class" block specifies the priority order of the process methods of the class . A priority statement outside a "begin class" block may specify the priority order of process methods in different classes, and the priority order of processes.

2.62 Processes



A **processes** statement declares the names in the statement, and the entity types declared by subsequent **every** statements, as process types. This statement may appear in a preamble, but may not appear in a **begin class** block. The keywords **are**, **include**, and **is** are synonymous.

Global variable with same name as the entity type and i.entity global variable are implicitly defined.

For each declared process type, a reference mode is implicitly defined. Use of the reference mode may precede the declaration of the process type.

A "process notice" is a temporary entity that represents a process and has N special attributes (placed in the first N words):

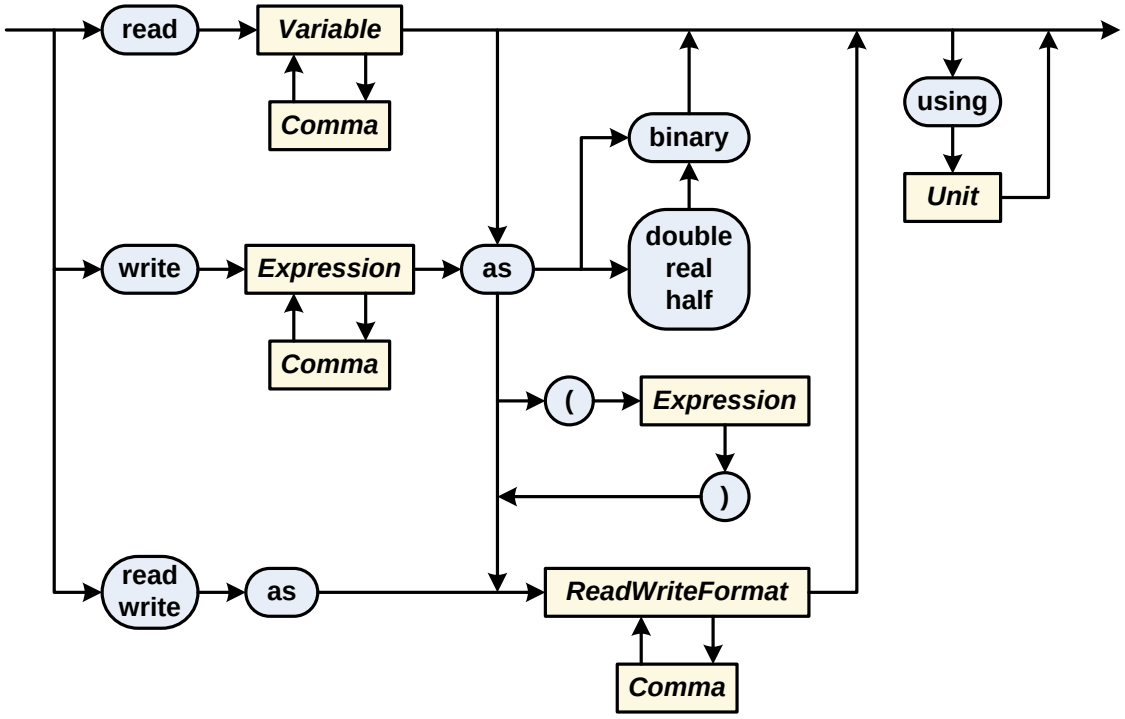
time.a: the simulated time at which the process is to be executed
eunit.a: indicates whether the processes was scheduled internally or externally;
if external, this attribute contains the number of the external unit;
if internal, value is zero
p.ev.s: predecessor in the event set when the process is scheduled
s.ev.s: successor in the event set when the process is scheduled
m.ev.s: nonzero when the process is scheduled

Also these attributes:
sta.a: integer equal to Passive (0), Active (1), Suspended (2), or Interrupted (3)
rsa.a: pointer to the recursive storage area
(destroyed automatically when the process notice is destroyed)
ipc.a: contains the value of i.process;
it is initialized when the process notice is created
f.rs.s: pointer to first qc.e for resources acquired by this process

Additional attributes are declared by "every" statements. Processes can own and belong to sets.

Process notices are created and destroyed like temporary entities.
Each process type must have a process routine.

2.63 ReadWrite



This statement, which may be used in any routine, reads values from an input unit and assigns each value to a ***Variable***, or writes the value of each ***Expression*** to an output unit. There are five forms of this statement: formatted read, formatted write, free-form read, binary read, and binary write.

1. *Formatted read*: **read *Variable* as *ReadWriteFormat***

   See ***ReadWriteFormat*** on page 154 for a description of this form.

2. *Formatted write*: **write *Expression* as *ReadWriteFormat***

   See ***ReadWriteFormat*** on page 154 for a description of this form.

3. *Free-form read*: **read *Variable***

   Free-form read statements may be used to read from character (non-binary) input units. All blank input characters are skipped and lines are read, as needed, until a non-blank character is found. The value that is read and assigned to ***Variable*** begins with this non-blank character. (If a non-blank character cannot be located, then the logical condition, **data is ended**, is true; see ***LogicalPhrase2*** on page 114 for more information.)

If the mode of ***Variable*** is integer, integer4, integer2, or pointer, then an integer value is read and assigned to ***Variable***. This value is expressed in the input as a sequence of one or more decimal digits with an optional leading sign (**+** or **–**). (In this case, **read *Variable*** is synonymous with **read *Variable* as i ***.)

If the mode of ***Variable*** is alpha, then a single non-blank character is read and assigned to ***Variable***. (In this case, **read *Variable*** is synonymous with **read *Variable* as a ***.)

If the mode of ***Variable*** is double or real, then a floating-point value is read and assigned to ***Variable***. This value is expressed in the input as a sequence of non-blank characters consisting of decimal digits and an optional leading sign. If a period appears in the sequence, it represents a decimal point. The value may be expressed in scientific notation, with an exponent following the least significant digit. The exponent is specified by **E** or **e**, an optional sign, and one or more digits. It may also be specified as a sign and one or more digits, without the **E** or **e**.

If the mode of ***Variable*** is text, then a text value is read and assigned to ***Variable***. This value is expressed in the input as a sequence of non-blank characters. (A formatted read statement must be used to read a text value that contains blanks.)

Suppose the input contains the following values on one or more lines, separated by spaces: **`–42 y 3.51 High`**

If **A** is integer, **B** is alpha, **C** is double, and **D** is text, then the following statement assigns –42 to **A**, **"y"** to **B**, 3.51 to **C**, and **"High"** to **D**:

```
read A, B, C, D
```

Because variables are processed from left to right, it is possible to read an integer value and use it as a subscript in the same statement:

```
read J, Vector(J)
```

If ***Variable*** is an array (or attribute of a permanent or compound entity), then specifying its name in a free-form read statement causes the entire array to be read. A free-form read is performed implicitly for each element of the array. If **X** is a 1-dimensional array, then

```
read X
```

is equivalent to

```
for J = 1 to dim.f(X)
   read X(J)
```

If **Y** is a 2-dimensional array, then

```
read Y
```

is equivalent to

```
for J = 1 to dim.f(Y)
    for K = 1 to dim.f(Y(J))
        read Y(J, K)
```

and

```
read Y(3)
```

is equivalent to

```
for J = 1 to dim.f(Y(3))
    read Y(3, J)
```

4. *Binary read*: **read *Variable* as binary**

Binary read statements are used to read data from binary input units.

A full-word signed integer value is read and assigned to ***Variable*** if **as binary** is specified and the mode of ***Variable*** is integer, integer4, integer2, alpha, pointer, or reference. (A loss of precision may occur if the mode of ***Variable*** is integer4, integer2, or alpha.) For example:

```
read Count as binary
```

A half-word unsigned integer value is read and assigned to ***Variable*** if **as half binary** is specified and the mode of ***Variable*** is numeric, i.e., double, real, integer, integer4, integer2, or alpha. (A loss of precision may occur if the mode of ***Variable*** is alpha.) For example:

```
read Count as half binary
```

A double-precision floating-point value is read and assigned to ***Variable*** if **as double binary** is specified and the mode of ***Variable*** is numeric. (A loss of precision may occur if the mode of ***Variable*** is not double.) For example:

```
read Mean as double binary
```

A single-precision floating-point value is read and assigned to ***Variable*** if **as real binary** is specified and the mode of ***Variable*** is numeric, or if **as binary** is specified and the mode of ***Variable*** is double or real. (A loss of precision may occur if the mode of ***Variable*** is not double or real.) For example:

```
read Mean as real binary
```

A text value is read and assigned to ***Variable*** if the mode of ***Variable*** is text. For example:

```
read Name as binary
```

5. *Binary write*: **write *Expression* as binary**

Binary write statements are used to write data to binary output units.

An ***Expression*** is written as a full-word signed integer value if **as binary** is specified and the mode of ***Expression*** is integer, integer4, integer2, alpha, pointer, or reference. For example:

```
write N + 1 as binary
```

An ***Expression*** is written as a half-word unsigned integer value if **as half binary** is specified and the mode of ***Expression*** is numeric. (A loss of precision may occur if the mode of ***Expression*** is not integer2 or alpha.) For example:

```
write N + 1 as half binary
```

An ***Expression*** is written as a double-precision floating-point value if **as double binary** is specified and the mode of ***Expression*** is numeric. For example:

```
write Mean as double binary
```

An ***Expression*** is written as a single-precision floating-point value if **as real binary** is specified and the mode of ***Expression*** is numeric, or if **as binary** is specified and the mode of ***Expression*** is double or real. (A loss of precision may occur if the mode of ***Expression*** is double.) For example:

```
write Mean as real binary
```

An ***Expression*** is written as a text value if the mode of ***Expression*** is text. For example:

```
write First_Name + " " + Last_Name as binary
```

If a ***Unit*** is specified, data is read from or written to the indicated unit; otherwise, data is read from the current input unit and written to the current output unit. For example:

```
read X, Y, Z using 12
write as "Simulation run completed", / using standard output
```

It is not permitted to write to an input unit or read from an output unit. However, "the buffer" (unit 99) is a special unit that may be used for both input and output. "The buffer" is a single line containing character (non-binary) data. The number of characters in the line is given by the **library.m** variable **buffer.v** and defaults to 132. Data may be alternately written to the line and then read from the line. This is useful for various data conversions. In the following example, a floating-point value and a text value are written to the buffer and then read back as text and integer values, respectively.

```
define Amount as a double variable
define ID_String, Dollar_Amount as text variables
define ID_Number as an integer variable

Amount = 23.785
ID_String = "5164"

' ' the following statement writes to the buffer: $23.79  5164
write Amount, ID_String as "$", d(5,2), "  ", t * using the buffer

' ' the next statement assigns "$23.79" to Dollar_Amount and 5164 to ID_Number
read Dollar_Amount, ID_Number using the buffer
```

Successive **write** statements append values to the single line of the buffer. The first **read** statement after writing to the buffer reads from the beginning of the line, and successive **read** statements read successive values from the line. The first **write** statement after reading from the buffer clears the buffer (stores all blanks) and writes to the beginning of the line. **Write as /** also clears the buffer and the next **write** statement writes to the beginning of the line.

An attribute declared as a random variable is read by a special free-form read operation. Suppose **Customer_Order** is a class with an object attribute named **Num_Units** and a class attribute named **Quantity**, where **Quantity** is declared to be an integer random step variable.

```
begin class Customer_Order

   every Customer_Order has a Num_Units
   the class has a Quantity random step variable
   define Num_Units, Quantity as integer variables

end
```

The number of units ordered by each customer is randomly generated according to the random variable. After creating a **Customer_Order** object, we generate a random value and assign it to the **Num_Units** attribute of the object:

```
define Order as a Customer_Order reference variable
create Order
Num_Units(Order) = Customer_Order'Quantity
```

However, the random variable must be read before it can be used. When a random variable is read, consecutive pairs of numbers are implicitly read by free-form read operations. The first number of a pair is a probability, which must be a floating-point value in the range 0.0 to 1.0. The second number is the value of the random variable which will occur with the specified probability. (This value must be integer if the mode of the random variable is integer; otherwise, it may be a floating-point value.) The sequence of pairs is terminated by the character stored in the **library.m** variable **mark.v**, which is an asterisk (*****) by default.

Suppose that a customer orders one unit 60% of the time, two units 30% of the time, and three units 10% of the time. The input data to represent this random variable consists of three pairs of numbers followed by an asterisk:

```
0.6  1  0.3  2  0.1  3  *
```

This data is read by the following statement:

```
read Customer_Order'Quantity  ' ' read random variable
```

The data may be read from any input unit. However, it is convenient to use the buffer:

```
write as "0.6  1  0.3  2  0.1  3  *" using the buffer
read Customer_Order'Quantity using the buffer
```

The sum of the probabilities must be equal to one (0.6 + 0.3 + 0.1 = 1.0). Alternatively, cumulative probabilities may be specified. In this case, each probability must be less than or equal to the next, and the last probability must be equal to one. The following data expresses cumulative probabilities and is equivalent to the data shown above:

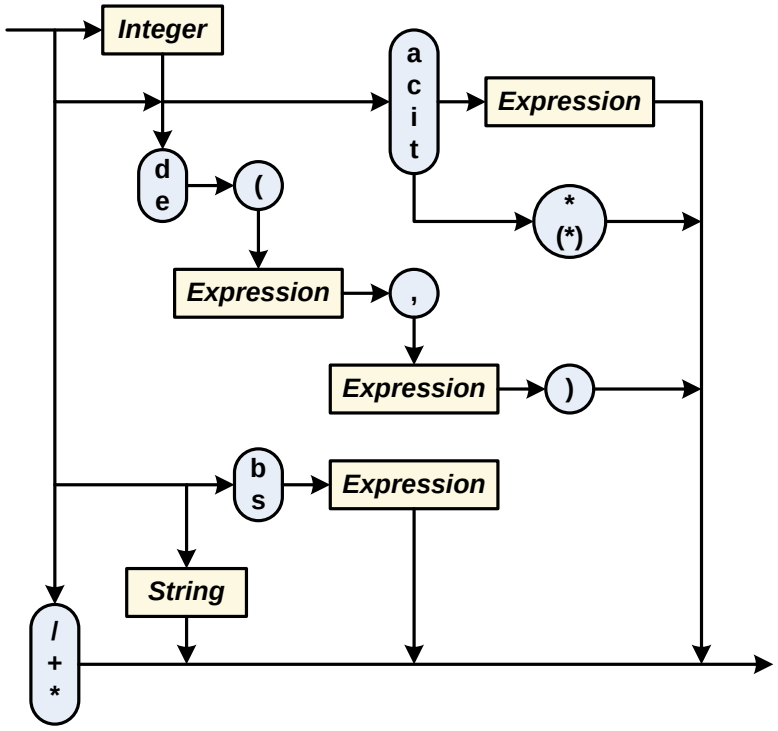
```
0.6  1  0.9  2  1.0  3  *
```

The above rules apply to attributes declared as **random step variable** or **random variable**. For an attribute declared as **random linear variable**, however, cumulative probabilities must be specified and the first probability must be equal to zero.

For an array of random variables (or a random attribute of a permanent or compound entity), it is necessary to read each element of the array explicitly. For example, the following loop initializes a 1-dimensional array of random variables named **R**. The abbreviated form, **read R**, is not permitted.

```
for J = 1 to dim.f(R)
   read R(J)
```

2.64 ReadWriteFormat



This language element specifies a format descriptor in a ***ReadWrite*** statement. There are six kinds of format descriptors: integer (**i**), floating-point (**d**, **e**), character (**t**, **a**, ***String***), hexadecimal (**c**), column (**b**, **s**), and line (***/***, **+**, *****).

1. *Integer descriptor*: **i**

   Read *Variable* as i *n*. An integer value is read from the next ***n*** columns of the current input line and is assigned to ***Variable***. The columns may contain only decimal digits, blanks, and an optional leading sign (**+** or **–**). Blanks are interpreted as zeros.

   Read *Variable* as i *. An integer value is read, starting with the next non-blank character read from the input unit, and is assigned to ***Variable***. (One or more lines are read as needed to locate this character. If end-of-file is reached without finding a non-blank character, then zero is assigned to ***Variable***.) The input value must be a sequence of one or more decimal digits, with an optional leading sign and no intervening spaces.

   Suppose **X** and **Y** are integer variables, and the next seven input columns contain **_–8_63_** (space, hyphen, eight, space, six, three, space). The statement,

```
read X, Y as i 5, i 2
```

assigns –806 to **X** and 30 to **Y**. Given the same input, the statement,

```
read X, Y as i *, i *
```

assigns –8 to **X** and 63 to **Y**.

Write *Expression* as i *n*. The integer value of ***Expression*** is written to the next ***n*** columns of the current output line. If fewer than ***n*** characters are needed to represent the value, the value is right-justified with leading blanks. If more than ***n*** characters are needed to represent the value, a series of ***n*** asterisks is written to indicate that the value cannot be represented in ***n*** columns.

Write *Expression* as i *. The integer value of ***Expression*** is written to the current output line using the minimum number of columns needed to represent the value.

Suppose **X** is equal to –806 and **Y** is equal to 30. The statement,

```
write X, Y, X – Y as i 3, i 3, i 6
```

writes `***_30_−836`, where each _ represents a space. The value –806 cannot be represented in three columns and so appears as three asterisks. The other values are right-justified. By contrast, the statement,

```
write X, Y, X – Y as i *, i *, i *
```

writes the three values as `−80630−836`, with no intervening spaces. Using the ***String*** descriptor, however, a space can be inserted between values. The statement,

```
write X, Y, X – Y as i *, " ", i *, " ", i *
```

writes the three values as `−806_30_−836`.

When using an integer descriptor, the mode of ***Variable*** and ***Expression*** is numeric (i.e., double, real, integer, integer4, integer2, or alpha), and the value of a double or real ***Expression*** is implicitly rounded. However, if the mode is pointer or reference, or ***Variable*** or ***Expression*** is an array pointer, then an integer address is read or written.

2. *Floating-point descriptors*: **d**, **e**

Read *Variable* as d(*n*, *m*). A floating-point value is read from the next ***n*** columns of the current input line and is assigned to ***Variable***. The columns may contain decimal digits, blanks, and an optional leading sign (**+** or **–**). Blanks are interpreted as zeros. If a period appears in the input, it represents a decimal point

and the value of ***m*** is disregarded; otherwise, a decimal point is implied before the ***m*** rightmost (least significant) digits, where $0 \le m \le n$.

An input value may also be expressed in scientific notation, with an exponent following the least significant digit. The exponent is specified by **E** or **e**, an optional sign, and one or more digits. It may also be specified as a sign and one or more digits, without the **E** or **e**.

Suppose **W** is a double variable, and the next ten input columns contain **␣␣␣␣−70254**, where each ␣ denotes a blank. The statement,

```
read W as d(10, 2)
```

assigns –702.54 to **W**. Note the implied decimal point before the two rightmost digits. Each of the following inputs assigns this same value to **W**:

```
␣−702.540␣      −.70254e+3      ␣−702540−1
−70.254E01      −0.70254e3      −702540E−1
−7.0254E␣2      −.7␣254+03      −702540␣−2
```

Read *Variable* as e(*n*, *m*). This form is synonymous with **Read *Variable* as d(*n*, *m*)**.

Write *Expression* as d(*n*, *m*). The floating-point value of ***Expression*** is written using standard notation to the next ***n*** columns of the current output line. The value is right-justified with leading blanks. ***m*** digits are displayed to the right of the decimal point, where $0 \le m < n$. The value is rounded to the least significant digit that is displayed. If the magnitude of the value is too large to be represented in ***n*** columns, the value is written using scientific notation as described in the next paragraph.

Write *Expression* as e(*n*, *m*). The floating-point value of ***Expression*** is written using scientific notation to the next ***n*** columns of the current output line. The value is right-justified with leading blanks. ***m*** digits are displayed to the right of the decimal point, where $0 \le m < n$. The value is rounded to the least significant digit that is displayed. It is necessary for ***n*** to be greater than (***m*** + 7) to allow room for all ***m*** digits, the sign, decimal point, and exponent.

Suppose **W** is equal to –702.54. The statement,

```
write W as d(10,2)
```

writes **␣␣␣−702.54**, where each ␣ represents a space. The statement,

```
write W as e(10,2)
```

writes **−7.03E+002**.

When using a floating-point descriptor, the mode of ***Variable*** and ***Expression*** must be numeric. The value assigned to an integer ***Variable*** is implicitly rounded.

3. *Character descriptors*: **t**, **a**, ***String***

Read *Variable* as t *n*. A text value is read, containing the sequence of characters found in the next ***n*** columns of the current input line, and is assigned to ***Variable***.

Read *Variable* as t *. A delimited text value is read from the input unit and is assigned to ***Variable***. First, a non-blank character is located in the input. One or more lines are read as needed to find it. This character serves as the delimiter and can be any non-blank character. Then a second occurrence of this delimiter is located in the input. Again, one or more lines are read as needed to find it. The text value that is read contains all of the characters between the two occurrences of the delimiter, excluding the delimiters and any end-of-line characters. If end-of-file is reached before locating the first delimiter, a null string (**""**) is assigned to ***Variable***. If the first delimiter is located but end-of-file is reached before finding the second delimiter, the second delimiter is implied at end-of-file.

Suppose **S** is a text variable, and the next 11 input columns contain **`_/John_Doe/`**, where each **_** denotes a blank. The statement,

```
read S as t 4
```

assigns the text value of length four, **" /Jo"**, to **S**. Given the same input, the statement,

```
read S as t *
```

assigns the text value, **"John Doe"**, to **S**. Here the slash character (**/**) is used as the delimiter.

Write *Expression* as t *n*. The value of ***Expression*** is written to the next ***n*** columns of the current output line. If fewer than ***n*** characters are needed to represent the value, the value is left-justified with trailing blanks. If more than ***n*** characters are needed to represent the value, only the first ***n*** characters of the value are written.

Write *Expression* as t *. The value of ***Expression*** is written to the output unit using the exact number of columns needed to represent the value. The number of columns is equal to the length of the value. If the value does not fit in its entirety on the current output line, the value is begun on the current output line and completed on one or more subsequent lines.

Suppose **S** is equal to **"John Doe"**. The statement,

```
write S as t 6
```

writes only the first six characters, **`John_D`**, whereas the statement,

```
write S as t 10
```

writes ten columns: **`John_Doe_ _`**. The statement,

```
write S as t *
```

writes **`John_Doe`** using the exact number of columns (eight).

When using a **t** descriptor, the mode of ***Variable*** and ***Expression*** must be text or alpha; however, an alpha ***Variable*** saves only the first character of the text value that is read.

Read *Variable* as a *n*. A character is read from the next column of the current input line and is assigned to ***Variable***. If ***n*** > 1, the next (***n*** – 1) columns are skipped.

Read *Variable* as a ***. The next non-blank character is located in the input and is assigned to ***Variable***. (One or more lines are read as needed to locate this character. If end-of-file is reached without finding a non-blank character, then a blank is assigned to ***Variable***.)

Suppose **T**, **U**, and **V** are alpha variables, and the next eight input columns contain **`John_Doe`**. The statement,

```
read T, U, V as a 1, a 3, a *
```

assigns **"J"** to **T**, **"o"** to **U** (**`hn`** is skipped in the input), and **"D"** to **V** (the preceding blank is skipped).

Write *Expression* as a *n*. The character given by ***Expression*** is written to the next column of the current output line. If ***n*** > 1, then (***n*** – 1) trailing blanks are written following the character. That is, the character is left-justified in ***n*** columns.

Write *Expression* as a ***. This form is synonymous with **Write *Expression* as a 1**.

Write as *String*. The ***String*** is written verbatim to the output unit. It is synonymous with **Write *String* as t *****.

Suppose **T** is equal to **"J"**. The statement,

```
write T as "The first initial is ", a 1, "."
```

writes the following: **`The_first_initial_is_J.`**

When using an **a** descriptor, the mode of ***Variable*** and ***Expression*** must be numeric or text. A numeric ***Expression*** specifies the Latin1 code of the character to be written. The value of a double or real ***Expression*** is implicitly rounded. Only the first character of a text ***Expression*** is written; however, if the text ***Expression*** is the null string (**""**), a blank is written.

4. *Hexadecimal descriptor*: **c**

Read *Variable* as c *n*. A hexadecimal value is read from the next ***n*** columns of the current input line and is assigned to ***Variable***. The columns may contain only hexadecimal digits (**0** to **9**, **A** to **F**, **a** to **f**), and blanks which are interpreted as zeros.

Read *Variable* as c ***. A hexadecimal value is read, starting with the next non-blank character read from the input unit, and is assigned to ***Variable***. (One or more lines are read as needed to locate this character. If end-of-file is reached without finding a non-blank character, then zero is assigned to ***Variable***.) The input value must be a sequence of one or more hexadecimal digits without intervening spaces.

Suppose **X** and **Y** are integer variables, and the next 12 input columns contain **_5FE___3ac9_**. The statement,

```
read X, Y as c 4, c *
```

assigns 1,534 (hex 05FE) to **X** and assigns 15,049 (hex 3AC9) to **Y** (the three preceding blanks are skipped).

Write *Expression* as c *n*. The hexadecimal representation of the value of ***Expression*** is written to the next ***n*** columns of the current output line. If fewer than ***n*** characters are needed to represent the value, the value is right-justified with leading zeros. If more than ***n*** characters are needed to represent the value, the ***n*** rightmost (least significant) hexadecimal digits are written.

Write *Expression* as c ***. The hexadecimal representation of the value of ***Expression*** is written to the current output line using the minimum number of columns needed to represent the value.

Suppose **X** is equal to 1,534 and **Y** is equal to 15,049. The statement,

```
write X, Y as "X=", c 8, ",Y=", c *
```

writes **X=000005FE,Y=3AC9**.

When using a hexadecimal descriptor, the mode of ***Variable*** and ***Expression*** may be numeric, pointer, or reference. It is also permitted for ***Variable*** and ***Expression*** to be an array pointer, and for ***Expression*** to be a subprogram variable.

5. *Column descriptors*: **b**, **s**

Read as b *n*. This form does not read any columns, but specifies that column number ***n*** is the next column to read.

Write as b *n*. This form does not write any columns, but specifies that column number ***n*** is the next column to write.

This statement reads a floating-point value starting at column 24 of the current input line:

```
read W as b 24, d(8,3)
```

This statement writes a text value starting at column 5 of the current output line:

```
write S as b 5, t *
```

Multiple **b** descriptors may be specified in any order and can move the current column backwards, allowing values to be read or written more than once. This statement reads two integer values (starting at columns 12 and 30) and then rereads the first as a text value:

```
read X, Y, S as b 12, i 8, b 30, i 6, b 12, t 8
```

Read as s *n*. This form does not read any columns, but skips ***n*** input columns. The next column to read is now ***n*** columns to the right.

Write as s *n*. This form does not write any columns, but skips ***n*** output columns. The next column to write is now ***n*** columns to the right. The characters in the skipped columns are unchanged. Each output line is implicitly initialized to all blanks; therefore, if no characters have been explicitly written to the skipped columns, they contain blanks by default.

This statement reads a floating-point value in columns 8 to 17, skips 12 columns, and then reads an integer value starting at column 30:

```
read W, X as b 8, d(10,4), s 12, i 4
```

This statement skips three columns, writes an integer value, skips seven more columns, and then writes a text value:

```
write X, S as s 3, i *, s 7, t *
```

6. *Line descriptors*: ***/***, **+**, *****

Read as */*. This form is equivalent to the statement, **start new input line**. It finishes the current input line; any unread characters on the current line are

skipped. It then reads the next input line. When reading from a character (non-binary) input unit, each tab character in the new line is automatically replaced by blanks, assuming tab stops at columns 9, 17, 25, 33, etc. Blanks are then automatically appended to the line so that its length is equal to the record size of the input unit. The number of input characters in the line is assigned to **rreclen.v**; the number is determined after tabs are replaced but before blanks are appended.

Note that if end-of-file is reached when trying to read the next line of input, the value of **eof.v** is consulted. If **eof.v** is equal to zero, a runtime error occurs. Otherwise, the program has set **eof.v** to a nonzero value (typically 1) and wishes to be notified when end-of-file has been reached. The value of **eof.v** is set to 2 to provide this notification. Each input unit has its own **eof.v** so it is necessary to use the unit (i.e., make it the current input unit) before assigning a value to **eof.v**. For example:

```
use 8 for input
eof.v = 1
read as /
if eof.v = 2
   write as "End–of–file reached!  No more input data!", /
always
```

The following statement reads an integer value from the current input line, starts a new input line, and reads a double value starting at column 1 of the new line:

```
read X, W as i 3, /, d(7,2)
```

Write as /. This form is equivalent to the statement, **start new output line**. It finishes the current output line and writes it to the output unit. For a character (non-binary) output unit, an end-of-line character is also written.

This statement writes the value of **X** and finishes the current line, skips a line (i.e., writes a blank line), and then writes the value of **Y** to a third line:

```
write X, Y as "X = ", i *, /, /, "Y = ", i *, /
```

Write as +. This form is equivalent to **write as /** except the line is written without the end-of-line character. It is useful when prompting for user input. The user's entry appears on the same line as the prompt. For example:

```
write as "Enter the number of servers: ", +
read Number_of_Servers
```

Write as *. This form is equivalent to the statement, **start new page**, and is ignored unless pagination is enabled (**lines.v** is greater than zero). **Write as /** is performed implicitly to finish the current output line and a form feed character (Latin1 decimal 12) is written before the next output line so that it will be the first line on a new page.

This statement writes a heading on the first line of a new page:

```
write as *, "Simulation Results", /
```

All format descriptors may be used only with character I/O units, except the **/** and **+** descriptors which may be used for character or binary I/O.

Note that ***n*** is an ***Expression*** and its mode must be numeric. The value of a double or real ***Expression*** is implicitly rounded. Because ***n*** need not be a constant, it is possible to determine a starting column or field width at runtime. In the following example, the variable **Col** indicates the starting column and the variable **Width** specifies the field width for displaying the text value in **S**:

```
write S as b Col, t Width
```

The value of ***n*** must be nonnegative. Except for the **b** descriptor, it is not an error for ***n*** to be zero; such a descriptor has no effect and nothing is read or written. However, in a **read** statement, a default value is assigned to the ***Variable*** associated with the descriptor: a null string (**""**) is assigned for a **t** descriptor, a blank (**" "**) is assigned for an **a** descriptor, and a zero is assigned otherwise.

The value of ***n*** must not exceed the record size of the I/O unit. It is permitted to read the blanks appended to the **rreclen.v** characters of an input line (see the discussion above regarding **read as /**), but it is an error to read column positions greater than the record size. However, it is not an error to write to a column position greater than the record size. In this case, **write as /** is performed implicitly to start a new output line and the write operation is performed at the beginning of the new line.

A positive integer constant may precede an **a**, **c**, **d**, **e**, **i**, or **t** descriptor and indicates the number of times to repeat the descriptor. This constant may not appear within parentheses and at least one blank must separate it from the descriptor letter. The statement,

```
read T, U, V, W, X, Y as a 1, a 1, a 1, d(9, 3), i 6, i 6
```

is equivalent to:

```
read T, U, V, W, X, Y as 3 a 1, 1 d(9, 3), 2 i 6
```

For readability, the asterisk in an **a**, **c**, **i**, or **t** descriptor, and the ***Expression*** given for ***n***, may be parenthesized. The statement,

```
write X, Y, S as i *, b 12, i 8, s 10, t *, /
```

is equivalent to:

```
write X, Y, S as i(*), b(12), i(8), s(10), t(*), /
```

Without the parentheses, the ***Expression*** given for ***n*** must be separated from the descriptor letter by at least one blank. That is, **i 8** is permitted, but not **i8**. However, it is permitted to specify **i*** with no intervening space.

A single parenthesized ***Expression*** may precede the entire list of format descriptors in a **read** or **write** statement and specifies the frequency of implicit **read as /** or **write as /** operations. However, the statement must be the sole statement of the body of a loop and must not be enclosed within **do** and **loop** keywords. The mode of ***Expression*** must be numeric, i.e., double, real, integer, integer4, integer2, or alpha; if it is double or real, it is implicitly rounded to integer. The value of ***Expression*** must be nonzero; if it is negative, the absolute value is used.

In the following example, five integer values are read per line from columns 1 to 20 and assigned to consecutive elements of an array named **Z**. The remaining columns on each line are ignored because a **read as /** operation is performed implicitly after every fifth value is read. If the number of elements in **Z** is not a multiple of five, then fewer than five values are read from the final line, and a **read as /** operation is performed implicitly after the last value is read.

```
for J = 1 to dim.f(Z)
   read Z(J) as (5) i 4
```

In the following loop, the array elements are displayed in three columns. A **write as /** operation is performed implicitly after every third value and after the last value.

```
for J = 1 to dim.f(Z)
   write J, Z(J) as (3) "Z(", i 2, ") = ", i 4, s 5
```

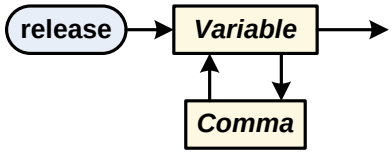
Suppose **Z** has 14 elements and the input is:

```
1281 781-902 4321332these columns are ignored
 761 -8717122314 902these columns are ignored
 374 911 512  21these columns are ignored
```

Then the output is:

```
Z( 1) = 1281     Z( 2) =  781     Z( 3) = -902
Z( 4) =  432     Z( 5) = 1332     Z( 6) =  761
Z( 7) =  -87     Z( 8) = 1712     Z( 9) = 2314
Z(10) =  902     Z(11) =  374     Z(12) =  911
Z(13) =  512     Z(14) =   21
```

2.65 Release



This statement, which may be used in any routine, de-allocates the storage for one or more arrays. The elements of these arrays are no longer accessible.

The following statements de-allocate the storage for one-dimensional arrays named **X** and **Y** and a two-dimensional array named **Table**:

```
release X
release Y
release Table
```

These statements may be combined into one:

```
release X, Y, and Table
```

Each ***Variable*** must have dimensionality greater than zero, or its mode must be integer or pointer. Given a ***Variable*** that contains zero, the statement has no effect. Given a ***Variable*** that contains a nonzero pointer to an allocated array, the array is de-allocated and zero is assigned to the ***Variable***. Given a ***Variable*** that contains some other nonzero value, it is an error.

A ***Variable*** may name a local or global array, an array attribute of an object or class, or an array attribute of the system or subsystem. It may also name an array of sets owned by an object or class, or owned by the system or subsystem. For example, suppose **Group** is a one-dimensional set owned by the system. The following statement,

```
release Group
```

de-allocates the storage for this array of sets. Each set in the array must be empty before executing this statement. Storage is de-allocated implicitly for three arrays of set attributes. The above statement is equivalent to:

```
release f.Group, l.Group, n.Group
```

A ***Variable*** may also name an attribute, or set owned by, a permanent entity or compound entity. However, it is recommended to use **destroy each** statements to de-allocate these arrays; see the ***CreateDestroy*** statement on page 35 for more information.

An array may be "partially" de-allocated. Suppose the **Table** array has been defined and allocated by the following statements:

```
define Table as a 2–dimensional double array
reserve Table as 3 by 4
```

Each of the three rows of the two-dimensional array contains four elements. The following statements de-allocate the second row and re-allocate it to have eight elements:

```
release Table(2)
reserve Table(2) as 8
```

When an allocated array is no longer needed, a ***Release*** statement must be executed to reclaim the storage used by the array. It is important to retain at least one pointer to each allocated array, so that its storage may be freed by a ***Release*** statement. When an array is allocated using a local recursive variable, the only pointer to the array is stored in this variable, yet this variable will be discarded upon return from the routine. Therefore, the array must be de-allocated before returning from the routine, or the array pointer must be copied to another location. For example:

```
subroutine Calculate given Size yielding Result

    ' ' define local array
    define Vector as a 1–dim double array
    reserve Vector as Size
    …

    ' ' free the array before returning
    release Vector
    return

end
```

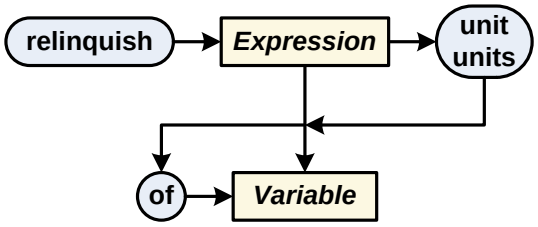
Instead of de-allocating the array in this example, the array pointer might be passed back to the caller as a yielded argument:

```
    let Result = Vector
    return
```

If an object has array attributes, it is necessary to explicitly de-allocate each array before the object is destroyed. To guarantee the de-allocation of these arrays, ***Release*** statements may be specified in a "before destroying" method, which is called automatically before an object is destroyed. See below for an example and ***BeforeAfter*** on page 18 for more information.

```
begin class Widget

    every Widget has an A, a B, a C, and a Cleanup method
    define A, B, and C as 1–dim double arrays
    before destroying a Widget, call Cleanup

end
…

method Widget'Cleanup

    ' ' free the arrays before destroying the object
    release A, B, and C

end
…

define W as a Widget reference variable
create W
reserve A(W), B(W), and C(W) as 30
…
destroy W  ' ' Cleanup is called implicitly
```

2.66 Relinquish



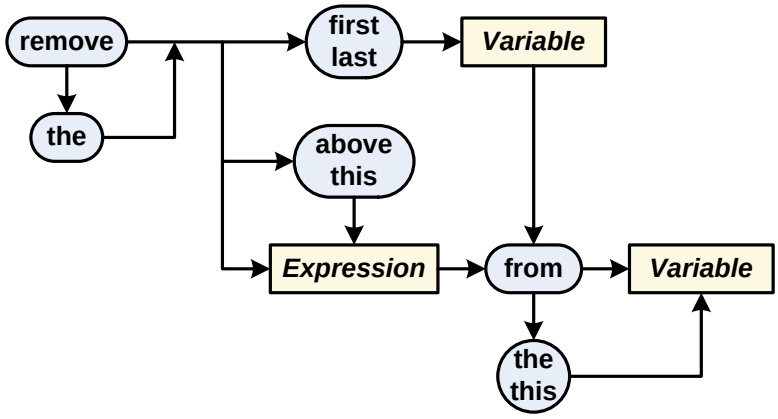
A **relinquish** statement frees one or more units of the specified resource. This statement may be used in any routine. The keywords **of**, **unit**, and **units** are optional for readability.

A process that has requested some units of a resource may relinquish some or all of them. The number of units of the resource being relinquished is added to the total quantity available. If any processes are queued awaiting the resource, they are scanned from the front of the queue. Each is reactivated with a corresponding reduction in the quantity of available units of resource, until one is found whose request cannot be satisfied.

The process relinquishing the resource continues execution at the statement immediately following the relinquish statement.

A positive integer number of units must be relinquished.

2.67 Remove



This statement, which may be used in any routine, removes an object or entity from a set. It has three forms. The keywords **above**, **the**, and **this** are optional for readability.

1. **Remove *Expression* from *Variable***. The *specific* object or entity identified by ***Expression*** is removed from the set named by ***Variable***, regardless of its position in the set. In the following example, the object identified by **Tanker** is removed from the set, **Awaiting(Tug)**:

   remove Tanker from Awaiting(Tug)

2. **Remove first *Variable2* from *Variable***. The *first* object or entity in the set named by ***Variable*** is removed and its reference value or entity number is assigned to ***Variable2***. In the following example, the first object in the set named **Awaiting(Tug)** is removed and its reference value is assigned to the variable named **Ship**:

   remove first Ship from Awaiting(Tug)

3. **Remove last *Variable2* from *Variable***. The *last* object or entity in the set named by ***Variable*** is removed and its reference value or entity number is assigned to ***Variable2***. For example:

   remove last Ship from Awaiting(Tug)

The set attributes of the removed object or entity, and the set attributes of the set owner, are automatically updated when a ***Remove*** statement is executed. For example, when this statement is executed,

```
remove first Ship from Awaiting(Tug)
```

the following modifications are made to the set attributes:

```
' ' the reference value of the first member is assigned to Ship and its "m."
' ' attribute is set to zero to indicate that it is no longer a member of the set
let Ship = f.Awaiting(Tug)
let m.Awaiting(Ship) = 0

' ' the set has a new first member
let f.Awaiting(Tug) = s.Awaiting(Ship)

' ' if the set is now empty, there is no first or last member
if f.Awaiting(Tug) = 0
    let l.Awaiting(Tug) = 0
always

' ' decrement the number of members in the set
subtract 1 from n.Awaiting(Tug)
```

In Form 1, it is an error if the object or entity identified by ***Expression*** is not a member of the specified set. However, the program may verify membership before executing the ***Remove*** statement. For example:

```
if Tanker is in Awaiting(Tug)
    remove Tanker from Awaiting(Tug)
always
```

It is an error to execute a ***Remove*** statement if the specified set is empty. It is good practice to verify beforehand that the set is non-empty. For example:

```
if Awaiting(Tug) is not empty
    remove first Ship from Awaiting(Tug)
always
```

An object or entity that has been removed from a set is not destroyed; however, the program may destroy it explicitly. It is an error to destroy an object or entity that is a member of a set; therefore, it must be removed from all sets before it is destroyed. For example:

```
remove Tanker from Awaiting(Tug)
destroy Tanker
```

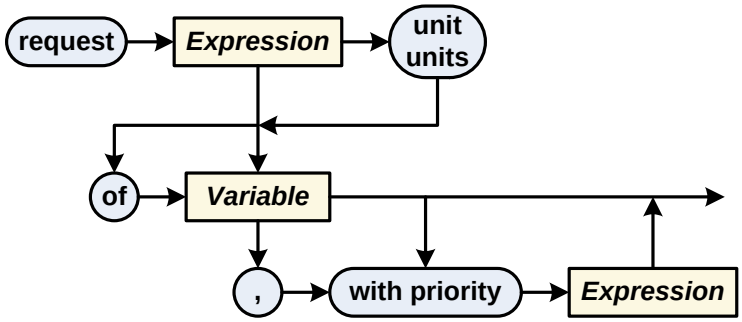
For a set of objects, the mode of ***Expression*** and ***Variable2*** must be integer, pointer, or the reference mode of the member class. The mode of ***Expression*** may also be the reference mode of a class that is derived from the member class. The mode of ***Variable2*** may also be the reference mode of a base class of the member class.

For a set of temporary entities or process notices, the mode of ***Expression*** and ***Variable2*** must be integer, pointer, or the reference mode of the entity type.

For a set of permanent entities or resources, a member is identified by an entity number; therefore, the mode of ***Expression*** and ***Variable2*** must be numeric: double, real, integer, integer4, integer2, or alpha. If the mode of ***Expression*** is double or real, it is implicitly rounded to integer.

A "before removing" routine and an "after removing" routine, if defined, are called automatically before and after each object or entity is removed from the set. The first argument to these routines is the value of ***Expression*** in Form 1, and is zero in Forms 2 and 3. Additional arguments are supplied as needed to identify the owner of the set. See ***BeforeAfter*** on page 18 for more information.

2.68 Request



A **request** statement acquires one or more units of the specified resource. This statement may be used in any routine. The keywords **of**, **unit**, and **units** are optional for readability.

If the requested quantity is available, it is given to the process, and the process continues execution at the statement following the request statement. If the requested quantity is not available, the process is filed in the queue of processes waiting for the particular resource and suspended awaiting availability of the request number of units. The queue is ranked on high priority, which may be positive, negative, or zero. If the "with priority" phrase is omitted, the priority is zero.

The resource is a permanent entity and must be subscripted explicitly or by an implicit subscript the variable with the same name as the resource; this variable is initialized to one at resource creation; or on some implementations, the implicit subscript is one

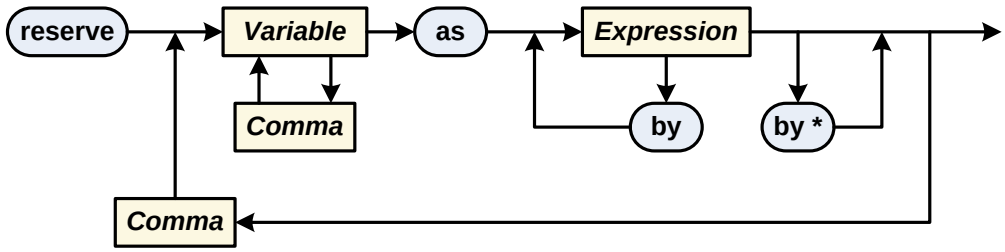
The request statement can only appear in a process routine. A request statement can be executed only if process.v is nonzero.

A positive integer number of units must be requested.

Resources are requested and owned by the process notice associated with a process method.

The "u.resource" attribute of the resource must be set to a nonzero value before any resource units can be requested.

2.69 Reserve



This statement, which may be used in any routine, allocates storage for one or more arrays. The number of elements in each array is specified. An array must be allocated before its elements are accessed.

The following statements define and allocate storage for one-dimensional arrays named **X** and **Y** and a two-dimensional array named **Table**:

```
define X as a 1–dimensional integer array
define Y as a 1–dimensional text array
define Table as a 2–dimensional double array

reserve X as 100
reserve Y as 100
reserve Table as 3 by 4
```

These ***Reserve*** statements may be combined into one:

```
reserve X and Y as 100, and Table as 3 by 4
```

When an array is allocated, each element is initialized to zero, except each element of a text array is initialized to the null string (**""**). In our example, each element of **X** is an integer variable initialized to zero; the first element is **X(1)** and the last element is **X(100)**. Each element of **Y** is a text variable initialized to the null string; the first element is **Y(1)** and the last element is **Y(100)**. Each element of **Table** is a double variable initialized to zero; there are 12 elements in all: **Table(1,1)**, **Table(1,2)**, **Table(1,3)**, **Table(1,4)**, **Table(2,1)**, **Table(2,2)**, **Table(2,3)**, **Table(2,4)**, **Table(3,1)**, **Table(3,2)**, **Table(3,3)**, and **Table(3,4)**. A two-dimensional array can be viewed as a one-dimensional array in which each element is a one-dimensional array. Thus, **Table(1)**, **Table(2)**, and **Table(3)** are each one-dimensional arrays containing four elements.

The **library.m** function **dim.f** can be used to obtain the number of elements in an array. In our example, **dim.f(X)** and **dim.f(Y)** are both equal to 100. **Dim.f(Table)** returns the number of elements in the first dimension of the **Table** array, which is 3. **Dim.f(Table(1))**, **dim.f(Table(2))**, and **dim.f(Table(3))** each return the number of elements in the second dimension, which is 4.

Each ***Expression*** indicates the number of elements in one dimension of the array. Its mode must be numeric (i.e., double, real, integer, integer4, integer2, or alpha), and its value must be positive. If it is double or real, it is implicitly rounded to integer.

Each ***Variable*** must have dimensionality greater than zero. It may name a local or global array, an array attribute of an object or class, or an array attribute of the system or subsystem. It may also name an array of sets owned by an object or class, or owned by the system or subsystem. For example, suppose **Group** is a one-dimensional set owned by the system. The following statement,

```
reserve Group as 20
```

allocates storage for this array of sets. Storage is allocated implicitly for three arrays of set attributes. The above statement is equivalent to:

```
reserve f.Group, l.Group, n.Group as 20
```

The number of elements in an array of sets may be obtained by calling **dim.f**. In our example, **dim.f(Group)** returns 20.

A ***Variable*** may also name an attribute, or set owned by, a permanent entity or compound entity. However, it is recommended to use **create each** statements to allocate these arrays; see the ***CreateDestroy*** statement on page 35 for more information.

When a one-dimensional array is allocated, the address of a block of contiguous elements is stored in the named array variable. When the name is subsequently used without subscripts in an ***Expression***, its value is this address, which we call an "array pointer." This value may be assigned to a variable of mode integer or pointer. For example, after the following statements are executed, variables **A** and **P** point to the same array:

```
define A as a 1–dimensional integer array
define P as a pointer variable

reserve A as 25
let P = A
```

The integer or pointer variable (**P** in our example) may not be subscripted directly. Its value must first be assigned to a one-dimensional variable, which may be subscripted:

```
define B as a 1–dimensional integer array
let B = P
write B(1) as "The value of the first element is ", i *, /
```

Although it is not possible to define an array attribute of a temporary entity or process notice, an integer or pointer attribute can be used to store an array pointer.

Normally, one ***Expression*** is supplied for each dimension of the ***Variable***. It is an error to specify more ***Expressions*** than dimensions. However, it is permitted to provide fewer ***Expressions*** than dimensions. In this case, the array is "partially" allocated.

If only one ***Expression*** is specified when allocating a two-dimensional array, then a one-dimensional array is allocated, where each element is an array pointer initialized to zero. Each element can point to a one-dimensional array representing one row of the two-dimensional array. These "row" arrays are allocated by subsequent ***Reserve*** statements to complete the allocation of the two-dimensional array. They are not required to have the same number of elements. We refer to a two-dimensional array with rows of unequal length as a "ragged array."

To illustrate, consider a square matrix W in which all entries above the main diagonal are zero. This is called a "lower triangular" matrix. For example:

$$\begin{bmatrix} W(1,1) & 0 & 0 & 0 \\ W(2,1) & W(2,2) & 0 & 0 \\ W(3,1) & W(3,2) & W(3,3) & 0 \\ W(4,1) & W(4,2) & W(4,3) & W(4,4) \end{bmatrix}$$

Rather than store the zero elements, a ragged array can be constructed that contains only the lower triangle. The first row contains one element, the second row contains two elements, the third row contains three elements, and the fourth row contains four elements.

Suppose **W** is an **N** × **N** lower triangular matrix. First we allocate the first dimension of the array:

```
define W as a 2–dimensional double array

reserve W as N
```

An optional **by *** phrase may be added for readability, to make clear that the two-dimensional array is only partially allocated:

```
reserve W as N by *
```

At this point, **W** points to a one-dimensional array of array pointers, where each array pointer, **W(1)**, **W(2)**, …, **W(N)**, is initialized to zero. **Dim.f(W)** returns **N**, the number of elements in the first dimension. We now allocate the second dimension, varying the number of elements in each row from 1 to **N**:

```
for Row = 1 to N
   reserve W(Row) as Row
```

Now **W(1)** points to an array of one element, **W(2)** points to an array of two elements, …, and **W(N)** points to an array of **N** elements. **Dim.f(W(1))** returns 1, **dim.f(W(2))** returns 2, …, and **dim.f(W(N))** returns **N**.

The elements of the array may be summed by the following loop:

```
let Sum = 0
for Row = 1 to N
    for Column = 1 to Row
        add W(Row, Column) to Sum
```

Note that it is an error to refer to **W(Row, Row + 1)** because this element does not exist. An element of an array is safely accessed by first verifying that the subscript values are in bounds. For example:

```
if 1 <= Row <= dim.f(W) and 1 <= Column <= dim.f(W(Row))
    ' ' Row and Column are in bounds; W(Row, Column) may be accessed
    …
always
```

All of the elements of an array are safely accessed by this loop:

```
for Row = 1 to dim.f(W)
    for Column = 1 to dim.f(W(Row))
    do
        ' ' access W(Row, Column)
        …
    loop
```

An array may have more than two dimensions. For example:

```
define Name as a 3–dimensional text array
reserve Name as 20 by 10 by N–1
let Name(I, J, K) = "Santa Monica"

define Q as a 4–dimensional real array
reserve Q as 8 by 4 by 2 by 8  ' ' allocates 8*4*2*8=512 elements
let Q(7,1,2,4) = 0.25
```

These arrays can be partially allocated as well:

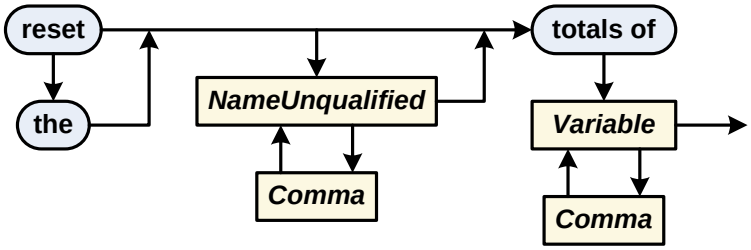
```
reserve Name as 20 by 10        ' ' still need to allocate 200 one–dimensional arrays

reserve Q as 8 by 4 by *        ' ' still need to allocate 32 two–dimensional arrays
```

When an allocated array is no longer needed, a ***Release*** statement must be executed to reclaim the storage used by the array (see page 164 for details). It is important to retain at least one pointer to each allocated array, so that its storage may be freed by a ***Release*** statement. When an array is allocated by a ***Reserve*** statement, a pointer to the newly-allocated array is assigned to the named variable, which overwrites any existing pointer

stored in the variable. Therefore, if the existing pointer has not been saved in another variable, and a ***Release*** statement has not been executed using this pointer, then access to the array is lost and the memory it occupies is unavailable to the program. This is known as a "memory leak." In the following example, without the assignment to **P**, access to the first array is lost when the second array is allocated:

```
define A as a 1–dimensional integer array
define P as a pointer variable

reserve A as 25          ' ' allocate first array
let P = A                ' ' save pointer to first array
reserve A as 40          ' ' allocate second array
```

2.70 Reset

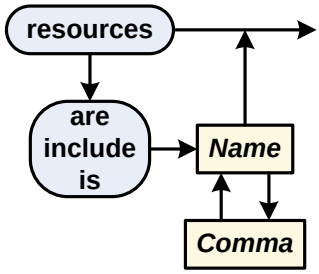


A **reset** statement initializes the collection of statistics on the values assigned to one or more attributes and global variables. This statement may be used in any routine. The keyword **the** is optional for readability.

The reset statement makes possible the preparation of reports on a cumulative or periodic basis. When both periodic and cumulative statistics are required, qualifiers can be specified. The qualifiers permit multiple sets of the same statistic to be gathered simultaneously, but the statistics can be reset at different times.

The appearance of one or more qualifiers in a reset statement specifies that only the indicated counters are to be reset. If no qualifiers are given in the reset statement, all counters associated with the variable(s) are initialized.

2.71 Resources



A **resources** statement declares the names in the statement, and the entity types declared by subsequent **every** statements, as resource types. This statement may appear in a preamble, but may not appear in a **begin class** block. The keywords **are**, **include**, and **is** are synonymous.

Global variable with same name as the entity type, and also n.resource global variable are implicitly defined.

A resource is a permanent entity with predefined attributes:

u.resource: specifies the integer number of units of this resource currently available
q.resource: set of processes currently waiting (queued) for this resource
n.q.resource: number of processes currently waiting for this resource
x.resource: set of processes currently using (executing with) this resource
n.x.resource: number of processes currently using this resource

Additional programmer-defined attributes must be specified by one or more "every" statements.

A "create each" statement is needed to create resources. The n.entity global variable contains the number of "resource types", whereas u.resource(i) contains the number of units of resource type "i". If only one type of resource is needed, then n.entity should be set to one.

Each unit of a resource is identical. A single queue waits for them. Each resource type has its own queue.

A "qc.e" entity is created for each "request" for a resource. It is these "qc.e" entities that are filed both in the set of resource associated with this process, rs.s(process), and also in either of the sets q.resource or x.resource, depending on whether the request has been satisfied. Each "qc.e" entity has the following attributes:

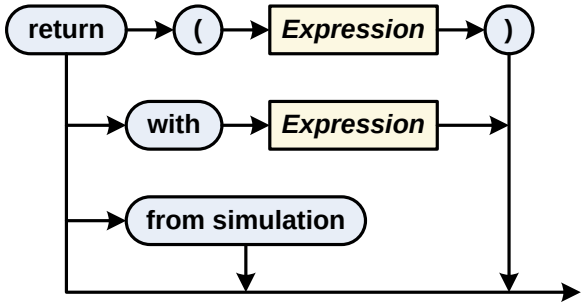
who.a: pointer to the process notice of the process that made the request
qty.a: integer number of resource units requested
pty.a: integer priority of request
p.rs.s: pointer to predecessor qc.e in rs.s(process)
s.rs.s: pointer to successor qc.e in rs.s(process)
p.q.resource/p.x.resource (equivalenced): pointer to predecessor qc.e waiting for or using the resource
s.q.resource/s.x.resource (equivalenced): pointer to successor qc.e waiting for or using the resource

The q.resource set is ranked by high pty.a.

If "u.resource" is explicitly incremented, will waiting processes be awakened?

Fractions of units may not be allocated.

2.72 Return



This statement terminates the execution of a routine and returns control to the calling routine. There are three forms of this statement.

1. A **return with** statement is executed by the right implementation of a function and provides the result of the function. The statements, **return with *Expression*** and **return (*Expression*)**, are synonymous. The ***Expression*** is evaluated and its value (the "source") is assigned to a hidden variable (the "destination") which has the same mode as the function. The value of the function call is taken from this variable and used by the calling routine. The source must be compatible with the destination according to the assignment compatibility rules on page 99.

   For example, the following double function named **Square** returns the square of its argument. When a routine calls **Square(2.5)**, control passes to the **Square** function with argument **X** equal to 2.5. The expression **X * X** is evaluated and control is returned back to the caller with the result of this expression. The value of **Square(2.5)** in the calling routine is 6.25.

```
function Square(X)

    return with X * X

end

…  ' ' in the calling routine
let Answer = Square(2.5)  ' ' 6.25 is assigned to Answer
```

2. A **return** statement is executed by a subroutine or by the left implementation of a function. No ***Expression*** may be specified. Control returns to the caller. Results of the subroutine, if any, are provided in yielded arguments. The following is a subroutine version of **Square**.

```
subroutine Square given X yielding X2

    let X2 = X * X
    return

end

…  ' ' in the calling routine
call Square given 2.5 yielding Answer  ' ' 6.25 is assigned to Answer
```

A **return** statement is implied at the end of a routine. This subroutine can therefore be rewritten as:

```
subroutine Square given X yielding X2

    let X2 = X * X

end  ' ' implied return occurs here
```

If the subroutine is a process method or process routine called by the timing routine during a simulation, then an explicit or implied **return** statement terminates the currently-executing process, destroys its process notice, and returns control to the timing routine. The subroutine's yielded values, if any, are discarded.

An explicit or implied **return** statement in the right implementation of a function is interpreted as **return with 0** (return zero), or for a text function, **return with ""** (return a null string).

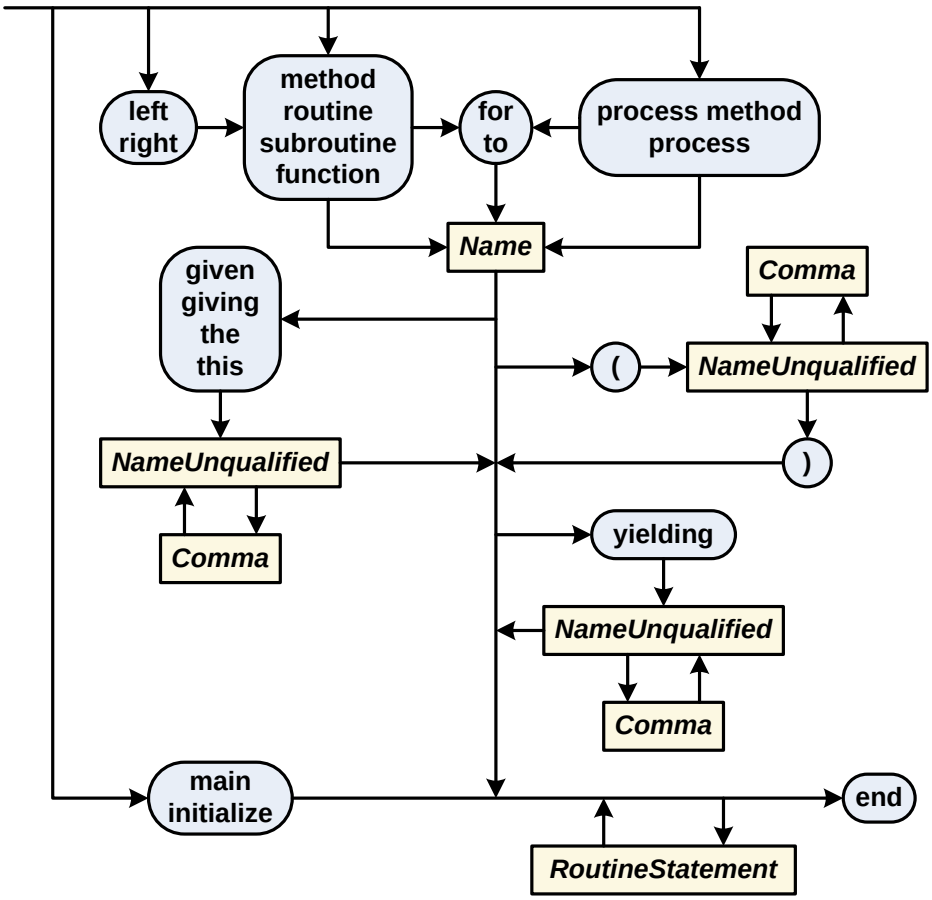
3. A **return from simulation** statement is executed by a subroutine during a simulation. The currently-executing process is terminated, its process notice is destroyed, and control passes to the statement that follows the **start simulation** statement. The subroutine's yielded values, if any, are discarded. For example:

```
process method Sim'Terminate

    return from simulation

end

…
schedule a Sim'Terminate at End_Time
start simulation
' ' arrive here when time.v = End_Time
```

If no simulation is running, a **return from simulation** statement acts as a **return** statement.

A routine may contain more than one ***Return*** statement and more than one form of ***Return*** statement.

2.73 Routine



A routine begins with a heading, which names the routine and its arguments, and ends with an **end** keyword. Between the heading and **end** are zero or more routine statements. A routine that is a method is specified as **method** or **process method**, and a process routine begins with the keyword **process**; otherwise, the routine is declared as **routine**, **subroutine**, or **function**. A left function must be declared as **left**, whereas a right function may optionally be declared as **right**. A main module must include a **main** routine, and each subsystem may have an **initialize** routine.

The keywords **for** and **to** are optional for readability. The following are synonymous:

- **routine**, **subroutine**, and **function**;
- **given**, **giving**, **the**, and **this**.

Cannot have "routine for/to". Must be "routine for/to for/to".

Explain difference between given and yielded arguments. Given arguments may appear in parentheses or after the "given" keyword. Given arguments as passed by value.

Yielded arguments are initialized to zero. A function cannot have yielded arguments but returns a function result value to the caller.

If the mode of arguments is unknown, it is assumed to be the background mode, provided it is not "undefined". The mode of arguments, if specified, must agree with the declaration of arguments in a "define routine/method" statement or inferred by the compiler. In some cases, the mode of arguments can be inferred by the compiler, such as the mode of arguments to function attributes, monitoring routines, and before/after routines. The mode of process routine arguments is determined by the modes of programmer-defined attributes in process notices.

Arguments are treated like local recursive variables.

A left and/or right implementation can be provided for each function. The "right" keyword is optional for a right implementation. The "left" keyword is required for a left implementation. If a variable is monitored on the right, a right-hand monitoring function must be supplied. If a variable is monitored on the left, a left-hand monitoring function must be supplied. A monitoring function has as many integer arguments as the dimension of the monitored variable. A function object method is defined for a monitored object attribute, and a function class method is defined for a monitored class attribute.

Process routines cannot be called. Their execution must be scheduled. Process routines cannot have yielded arguments and cannot be functions. A return statement (or reaching the "end") in a process routine returns control to the timing routine. A process routine can be suspended and thereby elapse simulation time. Ways in which the process routine can be suspended are: wait/work, request, suspend.

Upon entry to a process routine, time.v is set to the time.a of the process notice (i.e., the simulation clock has been updated), and the global variable with the same name as the process refers to the process notice. Attributes of the process notice are available via this global variable. Process.v also holds the reference value for a process notice, or is zero if no process is executing. A process notice is automatically destroyed when a process routine returns.

Upon entry to a process routine that is invoked for an external process, the current input unit is set to the external unit (its number is stored in eunit.a of the process notice). The routine may perform free-form or formatted read statements to read data from the external unit. (rcolumn.v is postioned at the last column of the time value.) It is not necessary to read all of the data; any data that is unread will be skipped up to the mark.v character. It is an error to read too much data, consuming the mark.v character and beyond. Upon returning from a process routine called for an external process, data on the external unit is read until a mark.v character is found, and the data following the mark.v character is read and used to schedule the next external process for this external unit. The current input unit is set to the standard input unit. Control is then passed to the timing routine.

Upon entry to a process routine that is invoked for an internal process, programmer-defined attributes are copied from the process notice to the given arguments of the routine. These arguments are not set for an external process and should be read from the external unit, which is the current input unit. It is an error to specify more given arguments than there are programmer-defined attributes. What is the mode of the given arguments differs from the mode of the process notice attributes? Note that upon resumption of a process routine, the given arguments contain the values they possessed when the process routine was suspended; so awakening a process routine does not copy process notice attributes to the given arguments of the routine.

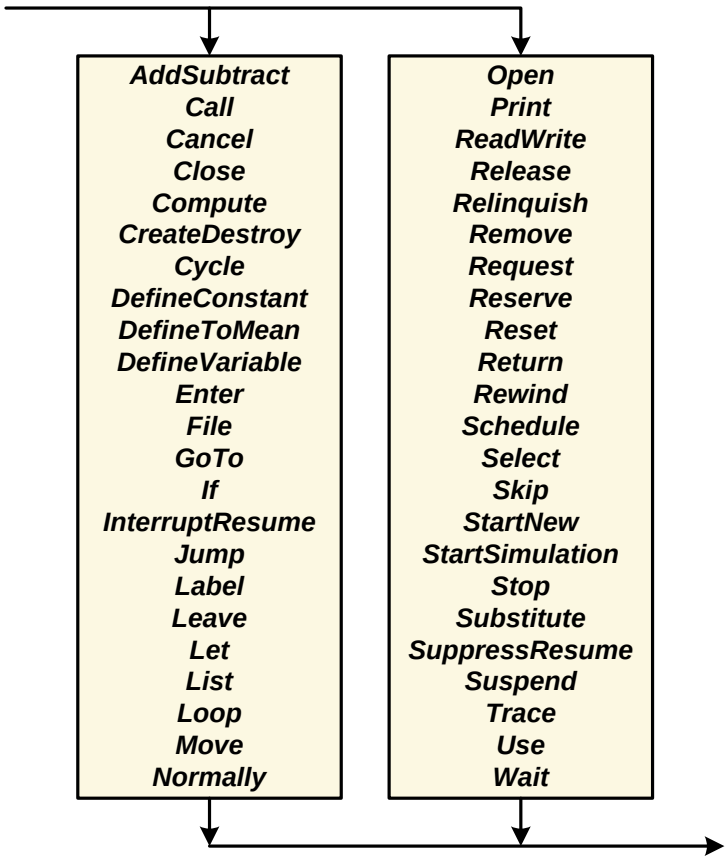
Program execution begins by executing each subsystem "initialize" routine once, in an indeterminate order, and then by executing the main module's "main" routine. An "initialize" routine can be used to initialize subsystem attributes, global variables, and class attributes defined by the subsystem.

Within the implementation of an object method, the reference value is stored in an implicitly-defined local reference variable with the same name as the class. This reference value is not defined within the implementation of a class method.

Method implementations for a class must appear within the module in which the class is defined.

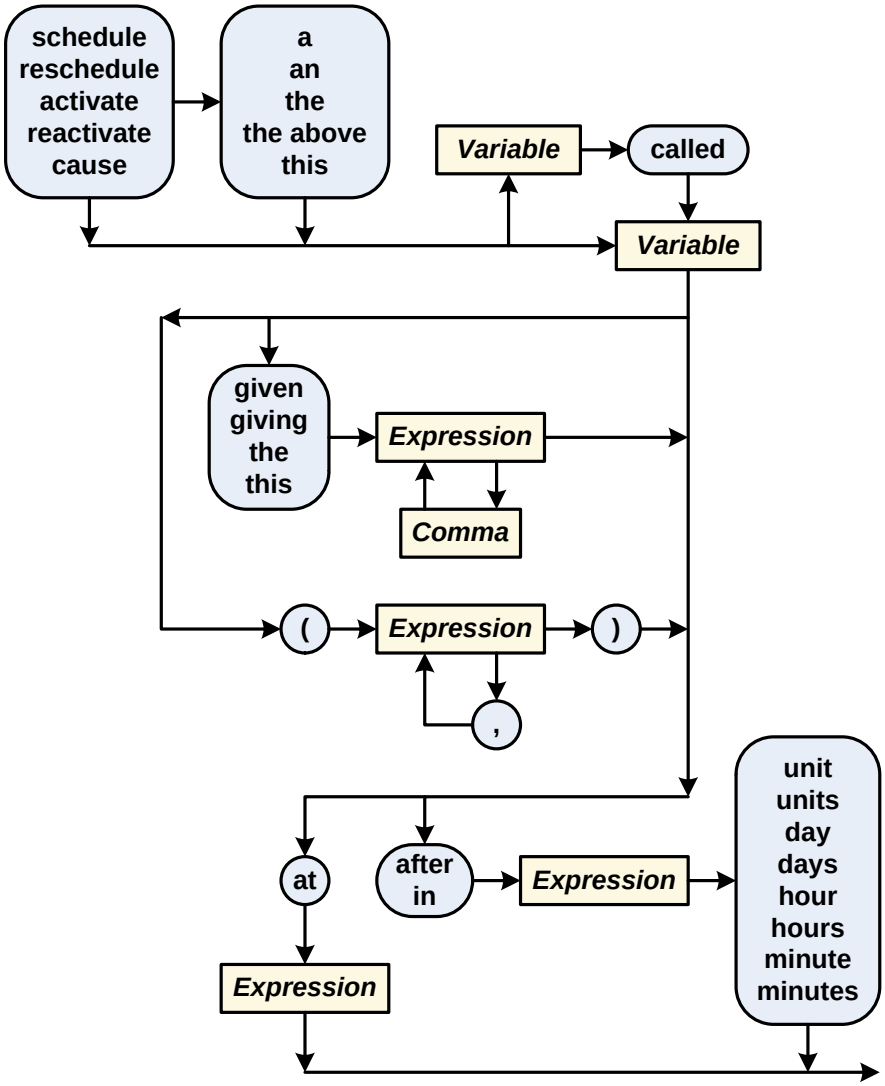
The given arguments to a process method must be specified in the method implementation and are initialized by process notice attributes for a scheduled invocation and by actual arguments for a direct invocation (i.e., a call).

2.74 RoutineStatement



A routine contains statements from this list.

2.75 Schedule



This statement, which may be used in any routine, inserts a new or existing process notice into the event set to schedule the execution of a process method or process routine. The following are synonymous:

- **schedule**, **reschedule**, **activate**, **reactivate**, and **cause**;
- **a** and **an**;
- **the**, **the above**, and **this**;
- **given**, **giving**, **the**, and **this**;
- **after** and **in**;
- **unit**, **units**, **day**, and **days**;
- **hour** and **hours**;
- **minute** and **minutes**.

This statement has four forms:

1. **Schedule a *Variable*** … A process notice is allocated and inserted into the event set to schedule the execution of a process method or process routine. The reference value of the new process notice is assigned to ***Variable***. To schedule a process method, ***Variable*** must name the process method, and the reference value is assigned to the attribute with the same name as the process method. To schedule a process routine, the mode of ***Variable*** must be the reference mode of the process type; however, its mode may be pointer or integer if it is a local variable with the same name as the process type.

2. **Schedule a *Variable2* called *Variable*** … A process notice is allocated and inserted into the event set to schedule the execution of a process method or process routine. The reference value of the new process notice is assigned to ***Variable***. To schedule a process method, ***Variable2*** must name the process method, and the mode of ***Variable*** must be pointer or integer. To schedule a process routine, ***Variable2*** must name the process type, and the mode of ***Variable*** must be pointer, integer, or the reference mode of the process type.

3. **Schedule the *Variable*** … An existing process notice, whose reference value is in ***Variable***, is inserted into the event set to schedule the execution of a process method or process routine. The mode of ***Variable*** must be pointer, integer, or the reference mode of a process type.

4. **Schedule the *Variable2* called *Variable*** … As in Form 3, an existing process notice, whose reference value is in ***Variable***, is inserted into the event set to schedule the execution of a process method or process routine, and the mode of ***Variable*** must be pointer, integer, or the reference mode of a process type. However, in addition, ***Variable2*** names a process method or process type, which is used for runtime error checking. ***Variable*** must identify a process notice associated with the named method or type.

A Form 1 or Form 2 statement is called a **schedule a** statement. A Form 3 or Form 4 statement is called a **schedule the** statement. If the **schedule** keyword is followed by a ***Variable*** with no intervening keyword, **the** is assumed. That is, **schedule *Variable*** is synonymous with **schedule the *Variable***.

A **schedule a** statement schedules the initial invocation of a process method or process routine. A **schedule the** statement schedules the initial invocation of a process method or process routine, or schedules the resumption of a suspended process method or process routine.

If a process method scheduled using Form 1 or 2 accepts one or more given arguments, then the correct number of ***Expressions*** must be specified after a **given** keyword, or within parentheses, to supply the argument values. The modes of these ***Expressions*** must be compatible with the modes of the process method's given arguments. The value of

each ***Expression*** (the "source") is assigned to the corresponding given argument within the process method (the "destination") upon entry to the method. The source must be compatible with the destination according to the assignment compatibility rules on page 99. If an ***Expression*** is an array, only the array pointer is copied, not the entire array. Any values yielded by the process method are discarded.

For example, suppose **Drive** is an object process method of a class named **Vehicle**, and this method accepts two given arguments, the distance to travel and the average speed, and yields one argument, the duration of the trip. The following statement schedules the execution of this process method to occur after three days of simulation time have elapsed. At that time, the **Vehicle** object identified by a reference variable named **Chevy** will commence a 200-mile trip at an average speed of 50 miles per hour.

```
schedule a Drive(Chevy) given 200, 50 in 3 days  '' Form 1 example
```

The given arguments may also be specified in parentheses:

```
schedule a Drive(Chevy)(200, 50) in 3 days  '' Form 1 example
```

A reference value is passed as an implicit argument to an object process method. Upon entry to the method, it is assigned to the implicitly-defined local reference variable which has the same name as the class. This reference value argument is not one of the method's given arguments. In this example, the value of **Chevy** is assigned to the implicitly-defined local reference variable named **Vehicle** upon entry to the **Drive** method.

In the **schedule a** statements shown above, the reference value of a newly-allocated process notice is stored in the object attribute named **Drive(Chevy)**. The mode of this attribute is pointer. The scheduled execution may be canceled by a ***Cancel*** statement that refers to this process notice. For example:

```
cancel the Drive(Chevy)
```

Execution of the process method may be rescheduled by a **schedule the** statement. However, given arguments may not be specified. Here we reschedule the 200-mile trip:

```
schedule the Drive(Chevy) in 7 days  '' Form 3 example
```

If the distance or average speed must be changed, then the process notice must be destroyed and then recreated using Form 1 or 2. For example:

```
cancel the Drive(Chevy)
destroy the Drive(Chevy)
schedule a Drive(Chevy) given 325, 55 in 7 days  '' Form 1 example
```

Any pointer or integer variable may be used to save the reference value of the process notice. For example:

```
define Trip as a pointer variable
schedule a Drive(Chevy) called Trip given 200, 50 in 3 days  ' ' Form 2 example
```

Here we access the process notice through the **Trip** variable:

```
cancel the Drive(Chevy) called Trip
schedule the Drive(Chevy) called Trip in 7 days  ' ' Form 4 example
```

Or simply:

```
cancel the Trip
schedule the Trip in 7 days  ' ' Form 3 example
```

Suppose the **Drive** method is scheduled within an object method of the **Vehicle** class. In this case, the reference value expression may be omitted and the implicitly-defined reference variable is implied. That is, these statements,

```
schedule a Drive given 200, 50 in 0 days
schedule a Drive(600, 60) in 12.5 days
```

are interpreted as:

```
schedule a Drive(Vehicle) given 200, 50 in 0 days
schedule a Drive(Vehicle)(600, 60) in 12.5 days
```

After executing a ***Schedule*** statement, a routine continues on without waiting for the scheduled process method to begin. When the process method completes, there is no one waiting to receive the values yielded by the method, and so they are discarded. For example:

```
schedule a Drive(Chevy) given 400, 50 in 0 days
' ' execution reaches here before the trip has begun
```

By executing a ***Call*** statement instead of a ***Schedule*** statement, a routine can invoke a process method immediately, wait for its completion, and obtain the yielded values. For example:

```
call Drive(Chevy) given 400, 50 yielding Drive_Duration
' ' execution reaches here after the trip has completed;
' ' Drive_Duration contains the duration of the trip
```

For process methods, as illustrated above, each ***Schedule*** statement has a corresponding ***Call*** statement. Because a derived class may override an inherited object process method, there may be more than one implementation of an object process method. A ***Schedule*** statement schedules for execution the same implementation invoked by the matching ***Call*** statement. That is, polymorphism is used in scheduling.

For example, suppose that the **All_Terrain_Vehicle** class is derived from the **Vehicle** class and overrides the **Drive** method. Then there are two implementations of the method, namely **Vehicle'Drive** and **All_Terrain_Vehicle'Drive**. (Presumably, the latter includes logic for "off-road" driving.) If **Chevy** contains the reference value of a **Vehicle** object, then **call Drive(Chevy)** invokes **Vehicle'Drive**, and **schedule a Drive(Chevy)** schedules the execution of **Vehicle'Drive**. On the other hand, if **Chevy** contains the reference value of an **All_Terrain_Vehicle** object, then **call Drive(Chevy)** invokes **All_Terrain_Vehicle'Drive**, and **schedule a Drive(Chevy)** schedules the execution of **All_Terrain_Vehicle'Drive**.

When scheduling a process routine using Form 1 or 2, one or more ***Expressions*** may be specified after a **given** keyword or within parentheses. Each ***Expression*** (the "source") is assigned to an explicitly-defined attribute of the process notice (the "destination"), in the order in which the attributes are defined in ***Every*** statements. The modes of these ***Expressions*** must be compatible with the modes of the attributes. Each source must be compatible with its destination according to the assignment compatibility rules on page 99. It is an error to specify more ***Expressions*** than attributes. If fewer ***Expressions*** are specified than attributes, the ***Expressions*** initialize the attributes that are defined first, and the remaining attributes are initialized to zero (or the null string for text attributes).

For example, suppose **Customer** is a process type with process notice attributes **Name** and **ID**:

```
processes
    every Customer has a Name and an ID
    define Name as a text variable
    define ID as an integer variable
```

After executing the following ***Schedule*** statement, a **Customer** process notice has been allocated and scheduled, and its reference value has been assigned to a reference variable named **NewCust**. The attributes of this process notice have been initialized as follows: **Name(NewCust) = "Johnson"** and **ID(NewCust) = 2415**.

```
define NewCust as a Customer reference variable
schedule a NewCust given "Johnson" and 2415 in 5 units
```

The above ***Schedule*** statement is equivalent to the following sequence of statements:

```
create a NewCust
let Name(NewCust) = "Johnson"
let ID(NewCust) = 2415
schedule the NewCust in 5 units
```

The statement,

```
schedule a NewCust given "Johnson" in 5 units
```

initializes the **Name** attribute to **"Johnson"** and the **ID** attribute to zero.

The statement,

```
schedule a NewCust in 5 units
```

initializes the **Name** attribute to the null string and the **ID** attribute to zero.

For a process type, the program may define attributes of the process notice using ***Every*** statements and may allocate process notices using ***Create*** statements or **schedule a** statements. However, for a process method, attributes of the process notice may not be defined and **schedule a** statements must be used to allocate process notices.

The **at** phrase specifies an "absolute" simulation time. The process method or process routine will be executed at the specified time. In the following example, the trip will commence when the simulation clock (i.e., the value of **time.v**) reaches 31.0:

```
schedule a Drive(Chevy) given 200, 50 at 31.0
```

The **in** phrase specifies a "relative" simulation time. The process method or process routine will be executed after the specified amount of time has elapsed. Here we schedule a customer arrival to occur when the simulation clock has advanced 8.5 time units. Thus, if the current value of **time.v** is 6.5, the arrival will occur when the simulation clock reaches 15.0.

```
schedule a NewCust given "Smith" in 8.5 units
```

By default, one unit of simulation time is interpreted as one day, each day consists of **hours.v** hours, and each hour consists of **minutes.v** minutes, where **hours.v** defaults to 24 and **minutes.v** defaults to 60. However, the program may interpret time units differently, and may change the values of **hours.v** and **minutes.v** as needed. For example, a value of 8 may be assigned to **hours.v** to simulate eight-hour workdays.

The following table shows how the time phrases determine the time at which the process method or process routine is scheduled for execution:

Time Phrase	*Scheduled Time of Execution*
schedule ... at *Expression*	***Expression***
schedule ... in *Expression* units	**time.v + *Expression***
schedule ... in *Expression* days	**time.v + *Expression***
schedule ... in *Expression* hours	**time.v + *Expression* / hours.v**
schedule ... in *Expression* minutes	**time.v + *Expression* / (hours.v * minutes.v)**

The ***Expression*** must have a numeric mode: double, real, integer, integer4, integer2, or alpha. The scheduled time of execution is assigned to the **time.a** attribute of the process notice. Time cannot go backwards; hence, the scheduled time must be greater than or equal to the current value of **time.v**.

The event set **ev.s** is an array of sets. Each process method and process type has a unique event set index. When a process notice is allocated, its **ipc.a** attribute is automatically

initialized to the event set index of its process method or process type. The process notice is inserted into the event set at this index. Thus, if **P** contains the reference value of a process notice, then this process notice is inserted into **ev.s(ipc.a(P))**. Upon insertion, the number of elements in this set, namely **n.ev.s(ipc.a(P))**, is incremented by one, and **m.ev.s(P)** is assigned a nonzero value (the event set index in **ipc.a(P)**) to indicate that the process notice is a member of the event set.

The process notices in **ev.s(i)** are ranked in increasing order of their **time.a** attributes. If two or more process notices in **ev.s(i)** have the same **time.a** value, they are ranked on a first-in-first-out basis. However, if index **i** is associated with a process type for which a ***BreakTies*** statement has been specified, then process notices in **ev.s(i)** with the same **time.a** value are ranked using the values of the ***BreakTies*** attributes. When a process notice is a member of the event set, it is an error to change the value of its **time.a** attribute, or the value of a ***BreakTies*** attribute, because it breaks the ordering of the set. See ***BreakTies*** on page 23 for more information.

It is an error to schedule a process notice that is already scheduled. Before scheduling a process notice, the program can verify that it is not already a member of the event set. For example:

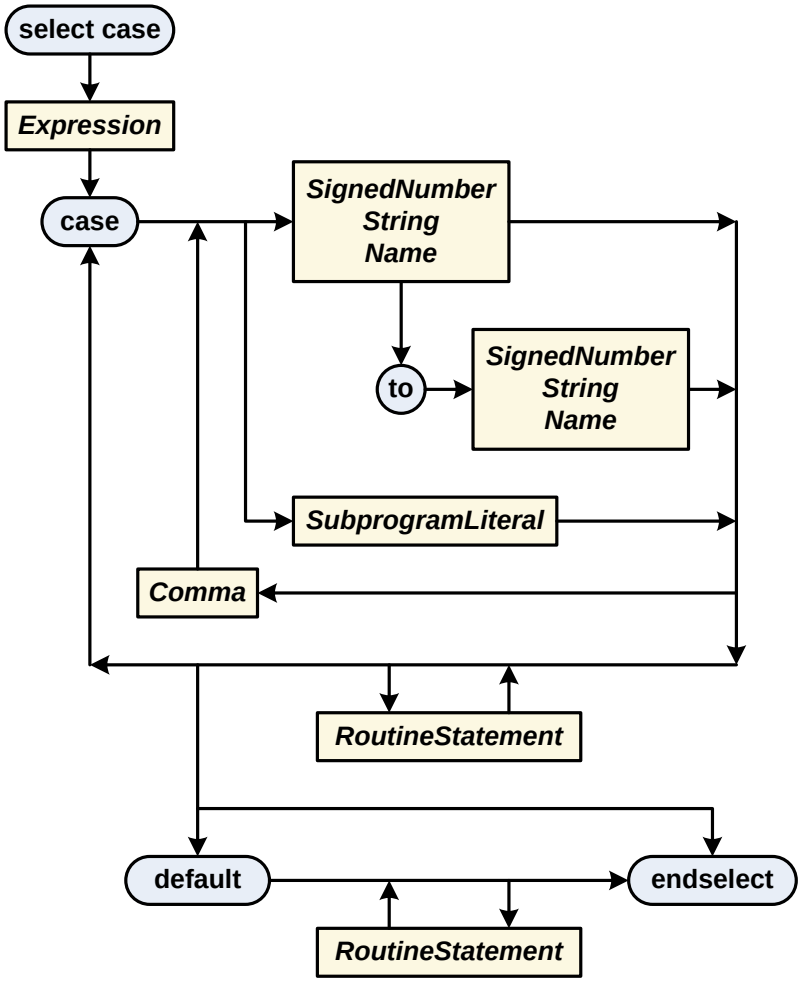
```
if the Drive(Chevy) is not in ev.s
   schedule the Drive(Chevy) in 2 hours
always
```

A "before scheduling" routine and an "after scheduling" routine, if defined, are called automatically before and after each process notice is inserted into the event set. The first argument to these routines is the reference value of the process notice. The second argument is the scheduled time. See ***BeforeAfter*** on page 18 for more information.

When scheduling a process routine using Form 1 or 2, the following steps are taken:

a. A process notice is allocated.
b. An "after creating" routine, if defined, is called.
c. The given ***Expressions***, if any, are assigned to consecutive attributes of the process notice.
d. A "before scheduling" routine, if defined, is called.
e. The scheduled time is assigned to the **time.a** attribute of the process notice.
f. The process notice is inserted into the event set.
g. An "after scheduling" routine, if defined, is called.

2.76 Select



This language element chooses a sequence of statements to execute depending on the value of an ***Expression***. It may be used in any routine. For example:

```
select case Letter_Grade
    case "A"
        Points = 4
    case "B"
        Points = 3
    case "C"
        Points = 2
    case "D"
        Points = 1
    case "F"
        Points = 0
endselect
```

In this example, if the value of an alpha variable named **Letter_Grade** is equal to **"A"**, then an integer variable named **Points** is assigned a value of 4; if **Letter_Grade** is equal to **"B"**, then **Points** is assigned a value of 3; and so on. If **Letter_Grade** is **"A"**, **"B"**, **"C"**, **"D"**, or **"F"**, a value is assigned to **Points** and execution continues with the statement that follows the **endselect** keyword. However, if **Letter_Grade** contains some other value, the program aborts with a runtime error because no default case was provided.

A range of values may be specified for a case. In the next example, we assign values to **Letter_Grade** and **Points** based on the value of **Score**. If **Score** is greater than or equal to 90 and less than or equal to 100, **"A"** is assigned to **Letter_Grade** and 4 is assigned to **Points**. The default case is provided and handles a failing grade; it will be reached if **Score** is less than 60 (or **Score** is greater than 100). The semicolons are optional and are used for readability.

```
select case Score
   case 90 to 100
      Letter_Grade = "A";  Points = 4
   case 80 to 90
      Letter_Grade = "B";  Points = 3
   case 70 to 80
      Letter_Grade = "C";  Points = 2
   case 60 to 70
      Letter_Grade = "D";  Points = 1
   default
      Letter_Grade = "F";  Points = 0
endselect
```

If the value of the ***Expression*** matches more than one case, the first matching case is selected. If **Score** is equal to 80 in this example, it matches two cases, 80 to 90 and 70 to 80; however, the 80 to 90 case is chosen because it appears first in the statement.

A case may specify any combination of individual values and ranges of values. If the value of the ***Expression*** matches any of the individual values or falls within any of the ranges, the case will be selected. Suppose **N** is an integer variable. In the following example, the first case is chosen if **N** is equal to 4 or 6, or is between 11 and 14, or between 21 and 24. The second case is selected if **N** is equal to 7, 8, 9, or 17. The third case is chosen for all other values of **N** between 1 and 25. Finally, the default case handles all values of **N** less than 1 and greater than 25.

```
select case N
   case 4, 6, 11 to 14, 21 to 24
      write as "This is the first case"
   case 7 to 9, 17
      write as "This is the second case"
   case 1 to 25
      write as "This is the third case"
   default
      write as "This is the default case"
endselect
```

All case values must be constants and may be named constants. For example:

```
define Machine_Status as an integer variable
define Out_of_Service, Available, and Busy as constants
define Message as a text variable
…

select case Machine_Status
   case Out_of_Service      let Message = "Needs repair"
   case Available           let Message = "Idle and ready"
   case Busy                let Message = "In use"
endselect
```

The sequence of statements specified for a case is any sequence of zero or more statements and may include "nested" ***Select*** statements. For example:

```
write as "Would you like to see a report? ", +
read User_Response

select case lower.f(User_Response)
   case "yes", "y"
      write as "Enter 1 for a detailed report or 2 for a summary: ", +
      read User_Choice
      select case User_Choice
         case 1   call Write_Detailed_Report
         case 2   call Write_Summary
         default   write as "Invalid entry", /
      endselect
   case "no", "n"
      ' ' nothing to do
   default
      write as "Unknown response", /
endselect
```

The ***Expression*** must be compatible with the constants specified for each case. The following rules apply:

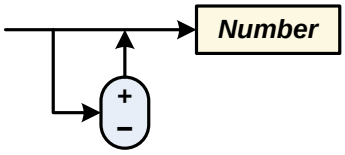
1. If the mode of the ***Expression*** is numeric (i.e., double, real, integer, integer4, integer2, or alpha), the mode of the constants must be numeric. However, text constants may be specified for an alpha ***Expression***.

2. If the mode of the ***Expression*** is text, the mode of the constants must be text or alpha.

3. If the mode of the ***Expression*** is pointer or reference, or the ***Expression*** is an array pointer, the only valid constant is 0 (zero).

4. If the ***Expression*** is a subprogram variable, each constant must be a ***SubprogramLiteral*** or 0 (zero).

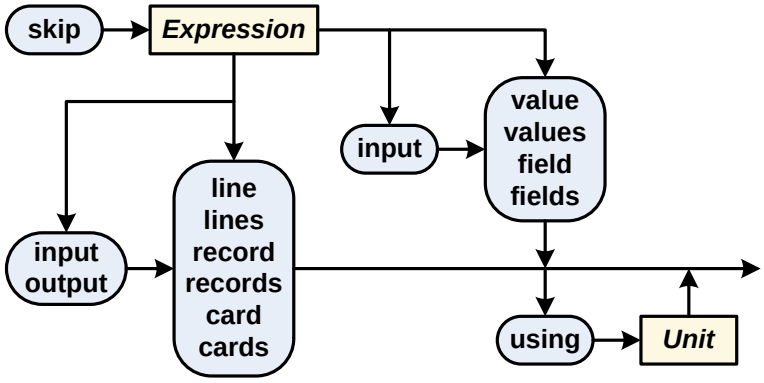
2.77 SignedNumber



This language element is a ***Number*** with an optional sign. If no sign is given, plus is assumed. For example:

50	**+50**
–.0094	**–9.4E–3**

2.78 Skip



This statement, which may be used in any routine, skips input values, input lines, or output lines. The following are synonymous:

- **value**, **values**, **field**, and **fields**;
- **line** and **lines**;
- **record**, **records**, **card**, and **cards**.

The following statements are equivalent. Each statement bypasses the next **N** values read from the current input unit, where a value is any sequence of one or more non-blank characters.

```
skip N values
skip N input values
```

If a ***Unit*** is specified, the values are bypassed on the indicated input unit, which must be a character (non-binary) input unit. For example:

```
skip N values using standard input
```

The following statements are equivalent. Each statement bypasses the remainder of the current line, and skips all of the next **(N – 1)** lines, on the current input unit.

```
skip N input lines
skip N input records
skip N records
```

Each of the above statements is equivalent to the following loop:

```
for J = 1 to N
    read as /
```

If a ***Unit*** is specified, the lines are bypassed on the indicated input unit. For example:

```
skip N input lines using 12
```

The following statements are equivalent. Each statement finishes the current line, and writes **(N – 1)** blank lines, on the current output unit.

```
skip N lines
skip N output lines
skip N output records
```

Each of the above statements is equivalent to the following loop:

```
for J = 1 to N
   write as /
```

If a ***Unit*** is specified, the lines are written to the indicated output unit. For example:

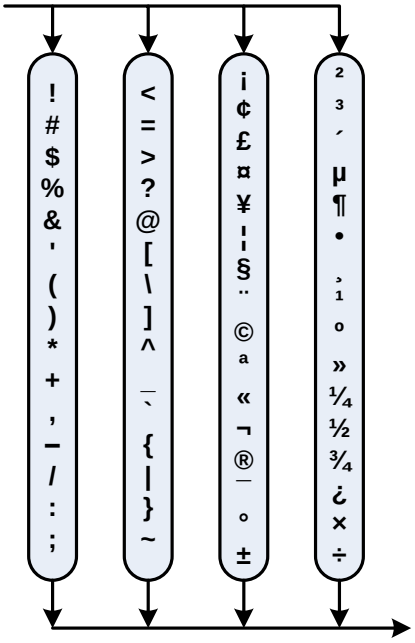
```
skip N lines using Report_Unit
```

If pagination is enabled (**lines.v** is greater than zero), then the number of blank lines written is limited to the number of lines needed to finish the current page.

In all cases, the mode of ***Expression*** must be numeric, i.e., double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer. If the value of ***Expression*** is zero, the statement has no effect. It is an error if the value of ***Expression*** is negative.

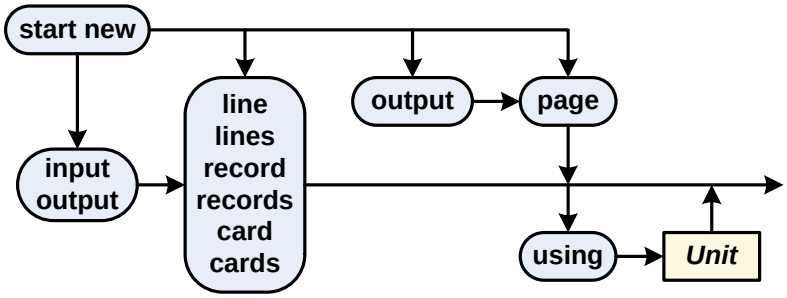
Skip 1 input line is equivalent to **start new input line**, and **skip 1 output line** is equivalent to **start new output line**. See ***StartNew*** on page 201 for more information. See ***ReadWriteFormat*** on page 154 for detailed information about the **/** format descriptor.

2.79 SpecialSymbol



This language element is a Latin1 special character. Any of these characters may appear in a ***String*** or comment, or may identify a substitution declared by a ***DefineToMean*** or ***Substitute*** statement.

2.80 StartNew



This statement, which may be used in any routine, starts a new input line, a new output line, or a new output page. The following are synonymous:

- **line** and **lines**;
- **record**, **records**, **card**, and **cards**.

The following statements are equivalent. Each statement starts a new line on the current input unit.

```
start new input line
start new input record
start new record
read as /
```

If a ***Unit*** is specified, a new line is started on the indicated input unit. For example:

```
start new input line using 16
```

The following statements are equivalent. Each statement starts a new line on the current output unit.

```
start new line
start new output line
start new output record
write as /
```

If a ***Unit*** is specified, a new line is started on the indicated output unit. For example:

```
start new line using standard output
```

The following statements are equivalent. Each statement starts a new page on the current output unit.

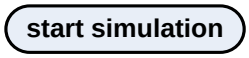
```
start new page
start new output page
write as *
```

If a ***Unit*** is specified, a new page is started on the indicated output unit. For example:

```
start new page using 8
```

See ***ReadWriteFormat*** on page 154 for detailed information about the **/** and ***** format descriptors.

2.81 StartSimulation



This statement calls the timing routine to run a simulation. It may appear in any routine but must be executed when a simulation is not running. Upon return from the timing routine, the statement that follows the **start simulation** statement is executed. For example:

```
start simulation  '' call the timing routine
'' arrive here upon return from the timing routine
```

The timing routine returns when there are no scheduled process methods or process routines (the event set is empty), or when a **return from simulation** statement is executed (see ***Return*** on page 180). At least one process method or process routine must be scheduled before executing a **start simulation** statement, or the timing routine will immediately return.

Upon return from the timing routine, a simulation is not running. However, if the timing routine returned with a non-empty event set, because a **return from simulation** statement was executed, the simulation may be considered to be "paused." Executing a **start simulation** statement calls the timing routine to continue the simulation.

2.82 Stop

This statement terminates the program. It may appear in any routine. For example:

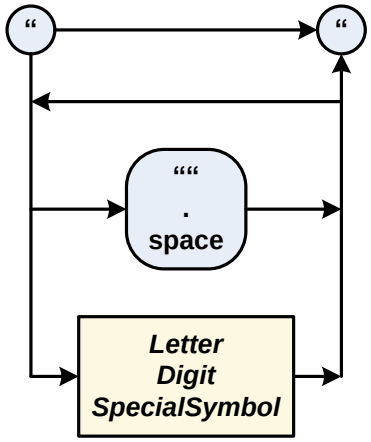
```
if Number_of_Runs <= 0
   write as "Invalid number of runs", /
   stop
otherwise
```

The execution of a program can terminate in four ways:

1. Executing a ***Stop*** statement.
2. Returning from the program's main routine.
3. Calling **exit.r**.
4. Performing an invalid operation, which aborts the program with a runtime error.

There may be more than one ***Stop*** statement in a program.

2.83 String



This language element is a sequence of zero or more characters enclosed in quotation marks. It may appear in an ***Expression*** or ***ReadWriteFormat***, or in a ***DefineConstant*** or ***Select*** statement.

A quotation mark within the sequence is specified by two consecutive quotation marks. If the sequence contains exactly one character, it is a constant of mode alpha; otherwise, its mode is text. The sequence containing no characters, **""**, is called the “null string.”

The following are examples of alpha constants:

```
"x"     "G"     "."     "4"     "&"     "è"     """"
```

The following are examples of text constants:

```
""          "California"     "San Diego, CA 92108"
"% UTILIZATION"     "Please enter ""Yes"" or ""No"":"
"Quale è il vostro caffè favorito?"
```

2.84 SubprogramLiteral

This language element represents the address of the named routine, which may be any routine except a method or process routine. This element may be assigned to a subprogram variable which can be used to call the routine indirectly. This element may appear in an ***Expression*** or in a ***DefineConstant*** or ***Select*** statement.

The routine name is enclosed in apostrophes. Each apostrophe may be separated from the routine name by one or more spaces. If a name or keyword immediately precedes the first apostrophe on the same line, or immediately follows the second apostrophe, the name or keyword must be separated from the apostrophe by at least one space.

If this language element names a function, it represents the right implementation of the function. It is not possible to represent and call indirectly the left implementation of a function.

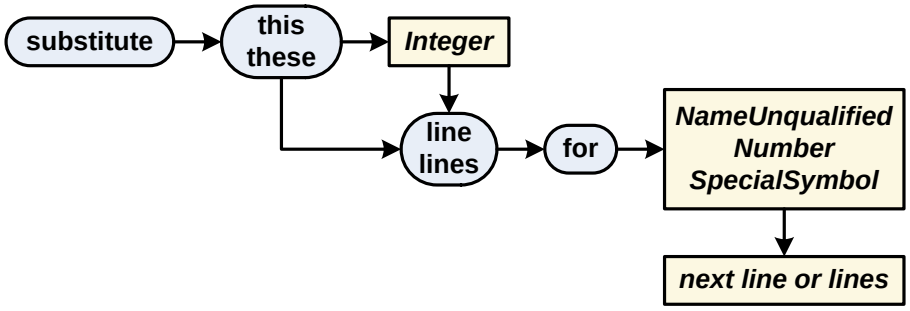
Suppose **Setup** is a subroutine given one argument and yielding one argument. Then **'Setup'** is a subprogram literal representing this subroutine which may be assigned to a subprogram variable:

```
define Init as a subprogram variable
Init = 'Setup'
call Init given Size yielding Count  ' ' calls Setup
```

Suppose **Inverse** is a double function imported from a module named **Geometry**, and that this function has a right implementation and accepts two given arguments. Then **'Geometry:Inverse'** is a subprogram literal representing this function. (Name qualification is not required if the unqualified name **Inverse** is unambiguous within the importing module.) This subprogram literal may be assigned to a double subprogram variable:

```
define Calculate as a double subprogram variable
define Result as a double variable
Calculate = 'Geometry:Inverse'
Result = $Calculate(X, Y)  ' ' calls Geometry:Inverse
```

2.85 Substitute



A **substitute** statement declares a source code substitution. Each occurrence of the unqualified name, number, or special symbol that follows in the source code will be replaced by the characters appearing on the next line or lines. This statement may appear in a preamble or routine.

The following are synonymous:

- **this** and **these**;
- **line** and **lines**.

The "substitute" statement is similar to a "define to mean" statement.

The scope of a "substitute". It applies after its definition in a preamble and then to every routine, or it applies after its definition in a routine and then only until the end of the routine. Also, it applies after its definition in a "begin class" block and then only until the end of the block.

To change a substitution, "suppress substitution; define Word to mean NewThing; resume substitution".

Substitution will not take place if the word is embedded in non-blank characters.

A "substitute" statement may appear within a begin class block and has effect only within that block.

Substitutions are not imported.

Substitutions in effect at the end of the public preamble are in effect at the beginning of the private preamble, and those in effect at the end of the private preamble apply to the routines of the subsystem.

2.86 SuppressResume

All source code substitutions defined by **define to mean** and **substitute** statements are not performed if they follow a **suppress substitution** statement; however, they are reinstated following a **resume substitution** statement. A warning message is issued by the compiler for each occurrence of an implicit subscript that appears after a **suppress implicit subscripts** statement; the warnings are discontinued following a **resume implicit subscripts** statement. These statements may appear in a preamble or routine.

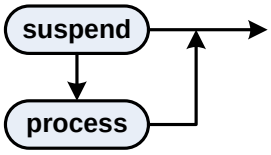
Substitutions will not be suppressed if on the same line following a suppress substitution statement. Substitutions will not be reinstated if on the same line following a resume substitution statement.

To suppress substitutions for a particular word, "suppress substitution; define Word to mean Word; resume substitution".

"Suppress" and "resume" statements may appear in a begin class block and have effect only within the block.

If "suppress" and "resume" statements are specified in a public preamble, it affects the interpretation of the public preamble source code, but does not affect importing modules.

2.87 Suspend



This statement suspends the execution of the currently-executing process and returns control to the timing routine. It may appear in any routine but must be executed when a simulation is running. The **process** keyword is optional for readability.

A **schedule the** statement referring to the process notice of the suspended process is used to schedule the resumption of the process (see ***Schedule*** on page 187). Hence, the reference value of the current process notice must be saved before executing a ***Suspend*** statement. Upon resumption of the process, execution begins with the statement that follows the ***Suspend*** statement. For example:

```
let P = process.v        ' ' save the reference value of the current process notice
suspend                  ' ' suspend execution of the current process
' ' arrive here upon resumption of the process
```

Another process schedules the suspended process to awaken now,

```
schedule the P in 0 units
```

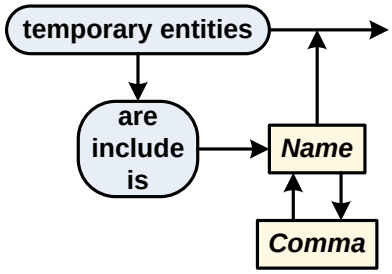
or to awaken at some future time:

```
schedule the P in 10 units
```

If a suspended process will not be awakened, then its process notice must be explicitly destroyed. For example:

```
destroy P
```

2.88 TemporaryEntities

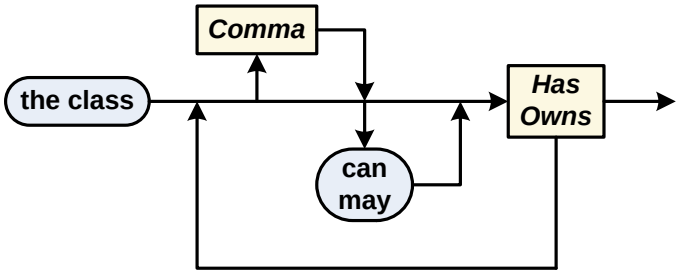


A **temporary entities** statement declares the names in the statement, and the entity types declared by subsequent **every** statements, as temporary entity types. This statement may appear in a preamble, but may not appear in a **begin class** block. The keywords **are**, **include**, and **is** are synonymous.

Global variable with same name as the entity type is implicitly defined.

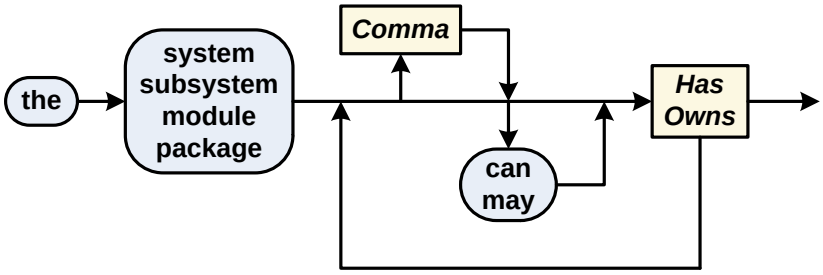
For each declared temporary entity, a reference mode is implicitly defined. Use of the reference mode may precede the declaration of the temporary entity.

2.89 TheClass



The class statements may appear in a **begin class** block to declare class attributes and class methods (***Has***), and sets owned by the class (***Owns***). The keywords **can** and **may** are optional for readability.

2.90 TheSystem



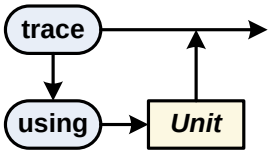
The system and **the subsystem** statements declare attributes of the module (***Has***) and sets owned by the module (***Owns***). **The system** statement may be specified in the preamble of a main module, whereas **the subsystem** statement may be used in a preamble of a subsystem. These statements may not appear in a **begin class** block.

The keywords **can** and **may** are optional for readability. The keywords **subsystem**, **module**, and **package** are synonymous.

"The system" can be thought of as an entity and it can have attributes and own sets. Attributes of "the system" are like global variables.

A preamble can have more than one "the system" statement.

2.91 Trace

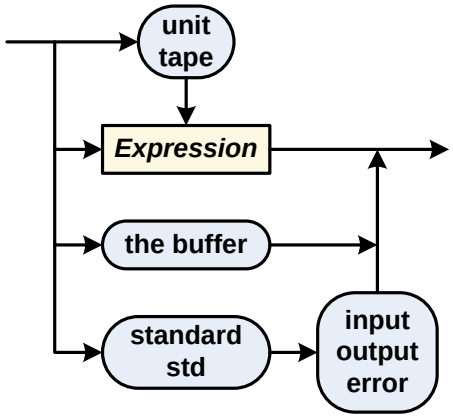


This statement, which may be used in any routine, writes information in a standard format about the current routine, the caller of the current routine, the caller of the caller, and so on. It is useful for debugging. If a ***Unit*** is specified, the information is written to the indicated output unit; otherwise, the information is written to the current output unit.

For example:

```
trace                          ' ' write to the current output unit

trace using 15                 ' ' write to unit 15

trace using standard error     ' ' write to the standard error unit
```

2.92 Unit



This language element, which appears in I/O statements, specifies the I/O unit to use. The keywords **unit** and **tape** are optional for readability. The keywords **standard** and **std** are synonymous.

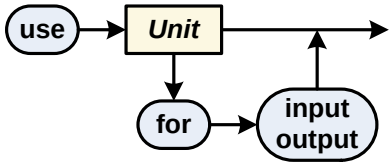
There are 99 I/O units, numbered from 1 to 99. Four of these units have special characteristics:

Unit Number	*Synonym*	*May Be Used For*
5	**standard input**	input only
6	**standard output**	output only
98	**standard error**	output only
99	**the buffer**	input and output

The number of the I/O unit is given by an ***Expression*** and must be in the range 1 to 99. If the unit number is 5, 6, 98, or 99, then a synonymous phrase may be used instead: **standard input**, **standard output**, **standard error**, or **the buffer**.

The mode of ***Expression*** must be numeric, i.e., double, real, integer, integer4, integer2, or alpha. If it is double or real, it is implicitly rounded to integer.

2.93 Use



This statement, which may be used in any routine, sets the current input unit or current output unit to the specified unit.

The unit number of the current input unit is stored in the **library.m** variable **read.v**. The following two statements are equivalent; each sets the current input unit to **N**.

```
use N for input
let read.v = N
```

The unit number of the current output unit is stored in the **library.m** variable **write.v**. The following two statements are equivalent; each sets the current output unit to **N**.

```
use N for output
let write.v = N
```

When a program begins executing, the current input unit is 5 (standard input) and the current output unit is 6 (standard output).

The **for** phrase may be omitted if ***Unit*** is one of these phrases: **standard input**, **standard output**, or **standard error**. **For input** is implied if ***Unit*** is **standard input**, and **for output** is implied if ***Unit*** is **standard output** or **standard error**. For example:

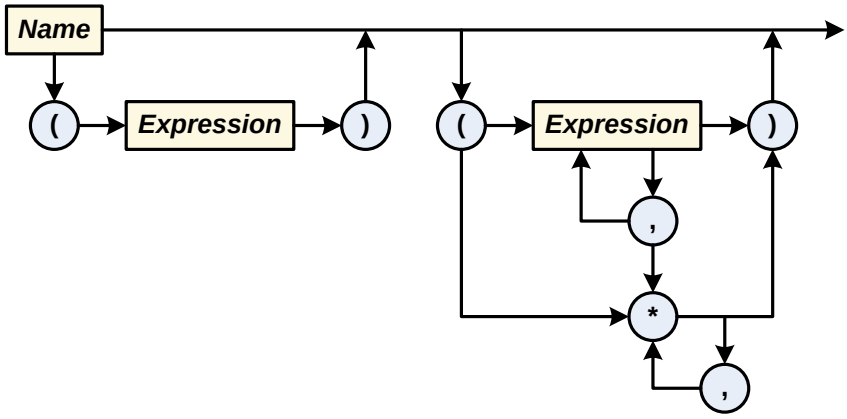
```
use standard input      ' ' same as: use standard input for input
use standard output     ' ' same as: use standard output for output
use standard error      ' ' same as: use standard error for output
```

If an unopened unit is specified in a ***Use*** statement or **using** phrase, the unit is implicitly opened. **Use N for input** implicitly executes **open N for input**, and **use N for output** implicitly performs **open N for output**. The unit contains character data and has the default record size (132) and default file name ("**SIMU***nn*" where *nn* is the two-digit unit number). See ***Open*** on page 136 for more information.

The record size of unit 99 (the buffer) is taken from the **library.m** variable **buffer.v** the first time this unit is used. The default value is 132. To specify a different record size, set the value of **buffer.v** before the first use of the buffer. For example:

```
let buffer.v = 1000
use the buffer for output
```

2.94 Variable



This language element is a name with optional subscripts. It appears in many executable statements and identifies a variable, attribute, routine, set, or constant. The rules for subscripts depend on what is named.

1. *Object attribute, object method, or set owned by an object*. The name is followed by a parenthesized reference value expression identifying the object. The name must have been defined or inherited by the class indicated by the reference mode of the expression. The object must be an instance of this class or of a derived class. If the parenthesized expression is omitted within an object method of a class, the implicitly-defined local reference variable with the same name as the class is used to identify the object. After the name and parenthesized expression, a separate parenthesized list specifies array subscript expressions (if the attribute or set is an array) or supplies given arguments to an object method (if the method has given arguments).

   For example, suppose **Vehicle** is a class, **Odometer** is a scalar (0-dimensional) attribute of a **Vehicle** object, and **Tire_Pressure** is a one-dimensional array attribute of a **Vehicle** object. If **MyCar** is a **Vehicle** reference variable, then **Odometer(MyCar)** refers to the scalar attribute of the object identified by **MyCar**, and **Tire_Pressure(MyCar)(2)** refers to the second element of the array attribute. The reference value expression may be omitted within an object method of the **Vehicle** class. In this case, **Odometer** and **Tire_Pressure(2)** are interpreted as **Odometer(Vehicle)** and **Tire_Pressure(Vehicle)(2)**, and is convenient shorthand for accessing attributes of the object for which the method was invoked.

2. *Attribute of, or set owned by, a temporary entity or process notice*. The name is followed by a parenthesized expression identifying the temporary entity or process notice. The mode of this expression must be pointer, integer, or the reference mode of the temporary entity type or process type. Unless the name refers to a common attribute of, or set owned by, two or more temporary entity

types and/or process types, the parenthesized expression may be omitted. In this case, the variable with the same name as the temporary entity type or process type is used to identify the temporary entity or process notice. This variable is either an explicitly-defined local variable, or if none, is the global variable implicitly defined for this type.

For example, suppose **Ship** is a temporary entity type and **Location** is one of its attributes. If **Tanker** is a **Ship** reference variable, then **Location(Tanker)** refers to the location of the temporary entity identified by **Tanker**. If **Location** is not a common attribute, the parenthesized expression may be omitted. In this case, **Location** is interpreted as **Location(Ship)**.

3. *Attribute of, or set owned by, a permanent entity or resource*. The name is followed by a parenthesized array subscript expression identifying the permanent entity or resource. If this parenthesized expression is omitted, the variable with the same name as the permanent entity type or resource type is used to identify the permanent entity or resource. This variable is either an explicitly-defined local variable, or if none, is the global variable implicitly defined for this type.

   For example, suppose **Server** is a permanent entity type and **Throughput** is one of its attributes. Then **Throughput(3)** refers to the throughput of the third **Server** entity. The parenthesized expression may be omitted; in this case, **Throughput** is interpreted as **Throughput(Server)**.

4. *Attribute of, or set owned by, a compound entity*. The name is followed by a parenthesized list of $n \geq 2$ array subscript expressions, where *n* is the number of constituent entity types composing the compound entity type. If this parenthesized list is omitted or contains fewer than *n* expressions, the missing subscripts are taken from the variables with the same name as the unrepresented constituent entity types. These variables are either explicitly-defined local variables or are the global variables implicitly defined for the constituent types.

   For example, suppose **Mechanic** and **Machine_Type** are resource types, and **Experience** is an attribute of the compound entity type composed of **Mechanic** and **Machine_Type** (in that order). That is, **every Mechanic and Machine_Type have an Experience**. Then **Experience(4, 2)** indicates the amount of experience that the fourth mechanic has working on machines of the second type. One or more subscripts may be omitted; **Experience** is interpreted as **Experience(Mechanic, Machine_Type)**, and **Experience(Chief)** is interpreted as **Experience(Chief, Machine_Type)**.

5. *Constant*. Subscripts may not be specified for a named constant.

6. For all other arrays, a parenthesized list of array subscript expressions may follow the array name. For all other routines, a parenthesized list of given arguments may follow the routine name if the routine has given arguments. For a monitored

array, the expressions in the parenthesized list are both array subscripts and given arguments to a monitoring function.

It is an error to access elements of an array that has not first been allocated storage by executing a ***Reserve*** statement. For an *n*-dimensional array, it is an error to specify more than *n* array subscripts. Exactly *n* array subscript expressions must be specified to access an element of an allocated array. However, an array pointer may be accessed by specifying fewer than *n* array subscript expressions.

Suppose **X** is a three-dimensional array with 10 elements in the first dimension, 5 elements in the second dimension, and 8 elements in the third dimension. This array may be allocated by the following ***Reserve*** statement:

```
reserve X as 10 by 5 by 8
```

Specifying **X** by itself refers to the entire three-dimensional array containing 10 × 5 × 8 = 400 elements. Because multi-dimensional arrays in SIMSCRIPT III are implemented as arrays of arrays, **X(I)** may be specified and identifies a two-dimensional array of 40 elements, and **X(I, J)** identifies a one-dimensional array of 8 elements. Three subscript expressions, as in **X(I, J, K)**, are required to access an individual element of a three-dimensional array.

For readability, one or more asterisks may be specified as placeholders for missing subscript expressions. **X(I, J, *)** is equivalent to **X(I, J)**. **X(I, *, *)** and **X(I, *)** are equivalent to **X(I)**. **X(*, *, *)**, **X(*, *)**, and **X(*)** are equivalent to **X**. The asterisks make clear that an array pointer, not an array element, is being accessed. Their use is optional except in the following case. Because variables are used implicitly in place of missing subscript expressions when accessing an attribute of a permanent entity, resource, or compound entity, asterisks are required in this case to access the attribute's array pointers.

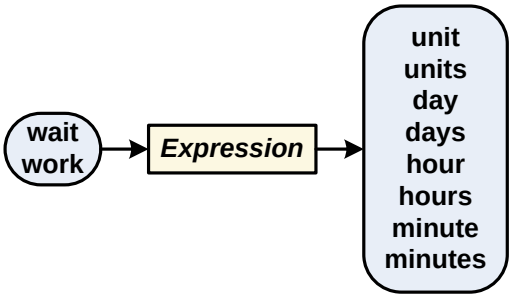
A subscript expression can have any numeric mode: double, real, integer, integer4, integer2, and alpha. A double or real expression is implicitly rounded to integer. The first element of an array is stored at subscript 1; therefore, an integer subscript must not be less than one. The integer subscript must not be greater than the number of elements in the array. In our example, **dim.f(X)** returns the number of elements in the first dimension of **X**. Therefore, the variable **I** must satisfy the following logical condition: **1 <= I <= dim.f(X)** to be used as a valid subscript of the first dimension. The number of elements in the second dimension is given by **dim.f(X(I))**; therefore, the variable **J** must satisfy: **1 <= J <= dim.f(X(I))**. Likewise, the number of elements in the third dimension is given by **dim.f(X(I, J))** and the variable **K** must satisfy: **1 <= K <= dim.f(X(I, J))**. Note that **dim.f(X(N))** may not be equal to **dim.f(X(N + 1))** if **X** has been allocated as a "ragged array." See the ***Reserve*** statement on page 172 for details.

The compiler will issue a warning message for each implicit subscript if implicit subscripts are "suppressed." See the ***SuppressResume*** statement on page 208 for information.

The value of a given argument expression (the "source") is copied to the corresponding routine argument (the "destination"). The source must be compatible with the destination according to the assignment compatibility rules on page 99. If a given argument expression is an array pointer, only the pointer value is copied, not the entire array.

If the name specified for this language element is undefined, the compiler checks the background mode. If the background mode is undefined (i.e., **normally mode is undefined**), then the compiler issues an error message. Otherwise, the compiler implicitly defines the name to be a local variable with the background mode and background type (recursive or saved). However, the background dimensionality is not applied. Instead the dimensionality of the variable is determined based on the number of subscripts that follow the first appearance of the name in the routine.

2.95 Wait



This statement suspends the execution of the currently-executing process and schedules its resumption after the specified amount of time has elapsed. This statement may appear in any routine but must be executed when a simulation is running. The ***Expression*** must have a numeric mode (i.e., double, real, integer, integer4, integer2, or alpha) and a nonnegative value. Upon resumption of the process, execution begins with the statement that follows the ***Wait*** statement.

The following are synonymous:

- **wait** and **work**;
- **unit**, **units**, **day**, and **days**;
- **hour** and **hours**;
- **minute** and **minutes**.

For example:

```
work 12 units  ' ' suspend execution
' ' arrive here after 12 time units have elapsed
```

The **library.m** global variable, **process.v**, contains the reference value of the process notice for the currently-executing process. The above statement is equivalent to the following sequence of statements:

```
schedule the process.v in 12 units
suspend
```

Likewise, the statement,

```
wait Delay hours
```

is equivalent to the following sequence of statements:

```
schedule the process.v in Delay hours
suspend
```

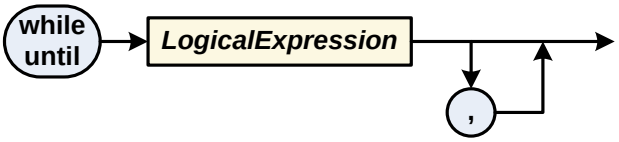
However, a “before scheduling” routine and an “after scheduling” routine, if defined, are not called for a ***Wait*** statement. See ***Schedule*** on page 187 and ***Suspend*** on page 209 for more information.

A ***Cancel*** or ***Interrupt*** statement referring to the process notice is used to cancel the scheduled resumption of the suspended process (see ***Cancel*** on page 27 and ***InterruptResume*** on page 91). The reference value of the process notice must therefore be saved before executing the ***Wait*** statement. For example:

```
let P = process.v  ' ' save the reference value of the current process notice
wait 6 days
```

Another process may then execute **cancel P** or **interrupt P**.

2.96 While



This loop control phrase is part of a ***Loop*** statement and specifies a terminating condition for the loop. It has two forms. The comma is optional for readability.

1. **while *LogicalExpression***. The loop terminates when the ***LogicalExpression*** is false.

2. **until *LogicalExpression***. The loop terminates when the ***LogicalExpression*** is true.

For example:

```
while Num > 0
do
   add Num to Total
   read Num
loop
```

Or equivalently:

```
until Num <= 0
do
   add Num to Total
   read Num
loop
```

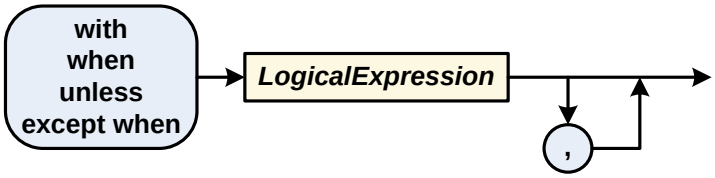
This logic may be written equivalently, but less elegantly, using ***GoTo*** statements:

```
'L1'  if Num <= 0  go to L2  otherwise
add Num to Total
read Num
go to L1
'L2'
```

Note that if **Num** is initially zero or negative, the body of the loop is never executed.

A ***While*** phrase may be qualified by a ***With*** phrase and may itself qualify a ***For*** phrase. See ***Loop*** on page 116 for more information.

2.97 With



This loop control phrase is part of a ***Loop*** statement and specifies a condition for skipping or bypassing the current iteration of the loop. The comma is optional for readability. The following are synonymous:

- **with** and **when**;
- **unless** and **except when**.

This phrase has two forms:

1. **with *LogicalExpression***. The current iteration of the loop is skipped when the ***LogicalExpression*** is false.

2. **unless *LogicalExpression***. The current iteration of the loop is skipped when the ***LogicalExpression*** is true.

In the following example, all elements of a square matrix are summed except the elements on the main diagonal.

```
let N = dim.f(Matrix)
let Sum = 0

for I = 1 to N
    for J = 1 to N
    with I <> J
        add Matrix(I, J) to Sum
```

This loop may be written equivalently as:

```
for I = 1 to N
    for J = 1 to N
    unless I = J
        add Matrix(I, J) to Sum
```

This loop may be written equivalently, but less succinctly, using an ***If*** statement:

```
for I = 1 to N
    for J = 1 to N
    do
        if I <> J
            add Matrix(I, J) to Sum
        always
    loop
```

A ***With*** phrase qualifies a ***For*** or ***While*** phrase. See ***Loop*** on page 116 for more information.

3 Library.m

Library.m is a special module that is implicitly imported by every preamble. This module defines routines, variables, and constants which are accessible to every module. These definitions may be accessed without qualification (for example, **time.v**) or with qualification (for example, **library.m:time.v**). The **library.m** definitions are described in the sections of this chapter:

- ***3.01 Mode Conversion***
- ***3.02 Numeric Operations***
- ***3.03 Text Operations***
- ***3.04 Input/Output***
- ***3.05 Random-Number Generation***
- ***3.06 Simulation***
- ***3.07 Miscellaneous***

3.01 Mode Conversion

---

atot.f (*alpha_arg*)

A text function that returns a text value of length one containing ***alpha_arg*** as its only character. For example, **atot.f("B")** converts an alpha **"B"** to a text **"B"**.

---

int.f (*double_arg*)

An integer function that returns the value obtained by rounding ***double_arg*** to the nearest integer. If the argument is positive, the rounded value is computed by adding 0.5 to the argument and truncating the result. If the argument is negative, the value is obtained by subtracting 0.5 from the argument and truncating. For example, **int.f(3.5)** returns 4 and **int.f(–3.5)** returns –4.

---

itoa.f (*integer_arg*)

An alpha function that returns the character representation of ***integer_arg***. The argument must be in the range 0 to 9. The return value is in the range **"0"** to **"9"**.

---

itot.f (*integer_arg*)

A text function that returns the text representation of ***integer_arg***. For example, **itot.f(100)** returns **"100"** and **itot.f(–5)** returns **"–5"**.

---

real.f (*integer_arg*)

A double function that returns the floating-point representation of ***integer_arg***. For example, **real.f(3)** returns 3.0.

---

rtot.f (*double_arg, total_width, fractional_width, use_exponential*)

A text function that returns the floating-point representation of ***double_arg***. The total width in places of the resulting text string is given as the second argument. That is followed by the number of places to the right the decimal and followed by a flag to use exponential notation (0 or 1). For example, **rtot.f(3.0008, 5, 3, 0)** returns **3.001**.

---

trunc.f (*double_arg*)

An integer function that returns the value obtained by truncating ***double_arg*** to remove its fractional part. For example, **trunc.f(3.5)** returns 3 and **trunc.f(–3.5)** returns –3.

---

ttoa.f (*text_arg*)

An alpha function that returns the first character of ***text_arg*** or returns a blank if ***text_arg*** is the null string. For example, **ttoa.f("yes")** returns **"y"** and **ttoa.f("")** returns **" "**.

---

ttoi.f (*text_arg*)
Converts the text representation of an integer to its integer value and returns it. If the text cannot be converted, zero is returned.

---

ttor.f (*text_arg*)
Converts the text representation of a floating point number to its double value and returns it. If the text cannot be converted, zero is returned.

Numeric Operations

---

abs.f (*numeric_arg*)

A function that returns the absolute value of an integer or double argument. If the argument is integer, the function returns an integer result. If the argument is double, the function returns a double result. For example, **abs.f(–5)** returns 5 and **abs.f(12.3)** returns 12.3.

---

and.f (*integer_arg1, integer_arg2*)

An integer function that returns the value obtained by performing a bitwise AND of ***integer_arg1*** and ***integer_arg2***. For example, **and.f(23, 51)** returns 19 because the bitwise AND of binary 010111 (23) and binary 110011 (51) is binary 010011 (19).

---

arccos.f (*double_arg*)

A double function that returns the arc cosine of ***double_arg*** in radians. The argument must be in the range –1 to +1. The return value is in the range zero to π.

---

arcsin.f (*double_arg*)

A double function that returns the arc sine of ***double_arg*** in radians. The argument must be in the range –1 to +1. The return value is in the range $-\frac{\pi}{2}$ to $+\frac{\pi}{2}$.

---

arctan.f (*double_argY, double_argX*)

A double function that returns the arc tangent of **(*double_argY / double_argX*)** in radians. Either argument may be zero but not both. If ***double_argY*** is positive, the return value is in the range zero to π. If ***double_argY*** is negative, the return value is in the range $-\pi$ to zero. If ***double_argY*** is zero and ***double_argX*** is positive, the return value is zero. If ***double_argY*** is zero and ***double_argX*** is negative, the return value is π.

---

cos.f (*double_arg*)

A double function that returns the cosine of ***double_arg***. The argument is specified in radians. The return value is in the range –1 to +1.

---

dim.f (*array_arg*)

An integer function that returns the number of elements in ***array_arg***. The argument is normally an array pointer. However, if the argument names an array of sets, then the **f.*set*** array pointer is implicitly passed in its place. If the argument is zero, then zero is returned.

---

div.f (*integer_arg1, integer_arg2*)

An integer function that returns the truncated result of **(*integer_arg1* / *integer_arg2*)**. ***Integer_arg2*** must be nonzero. For example, **div.f(17, 5)** returns 3 and **div.f(–12, 8)** returns –1.

---

exp.c

A double constant equal to the value of *e*, 2.718281828459045.

---

exp.f (*double_arg*)

A double function that returns the value of e^x where ***double_arg*** is the exponent.

---

frac.f (*double_arg*)

A double function that returns the fractional part of ***double_arg***. It is computed by subtracting the truncated value of the argument from the original value. If the argument is positive, the return value is positive. If the argument is negative, the return value is negative. For example, **frac.f(3.45)** returns 0.45 and **frac.f(–3.45)** returns –0.45.

---

inf.c

An integer constant equal to the largest integer value. On 32-bit computers, this value is $2^{31}-1 = 2{,}147{,}483{,}647$. The smallest integer value is **–inf.c–1**.

---

log.e.f (*double_arg*)

A double function that returns the natural logarithm (i.e., the base *e* logarithm) of ***double_arg***. The argument must be positive.

---

log.10.f (*double_arg*)

A double function that returns the base 10 logarithm of ***double_arg***. The argument must be positive.

---

max.f (*numeric_arg1, numeric_arg2, …*)

A function that returns the maximum value of two or more integer or double arguments. If every argument is integer, the function returns an integer result; otherwise, the function returns a double result.

---

min.f (*numeric_arg1, numeric_arg2, …*)

A function that returns the minimum value of two or more integer or double arguments. If every argument is integer, the function returns an integer result; otherwise, the function returns a double result.

---

mod.f (*numeric_arg1, numeric_arg2*)

A function that computes ***numeric_arg1*** divided by ***numeric_arg2*** and returns the remainder. If both arguments are integer, the function returns an integer result; otherwise, the function returns a double result. ***Numeric_arg2*** must be nonzero. If ***numeric_arg1*** is positive, the return value is positive. If ***numeric_arg1*** is negative, the return value is negative. For example, **mod.f(14.5, 3)** returns 2.5 and **mod.f(–14.5, 3)** returns –2.5.

---

or.f (*integer_arg1, integer_arg2*)

An integer function that returns the value obtained by performing a bitwise inclusive OR of ***integer_arg1*** and ***integer_arg2***. For example, **or.f(23, 51)** returns 55 because the bitwise inclusive OR of binary 010111 (23) and binary 110011 (51) is binary 110111 (55).

---

pi.c

A double constant equal to the value of π, 3.141592653589793.

---

radian.c

A double constant equal to the number of degrees per radian, which is $\frac{180}{\pi}$ or 57.29577951308232.

---

rinf.c

A double constant equal to the largest real value. On 32-bit computers, this value is approximately 3.4×10^{38}; however, a double value may be as large as 10^{308}. The smallest real value is **–rinf.c**.

---

shl.f (*integer_arg1, integer_arg2*)

An integer function that returns the value of ***integer_arg1*** shifted left by ***integer_arg2*** bit positions. For example, **shl.f(23, 2)** returns 92 because binary 00010111 (23) shifted left two positions is binary 01011100 (92). The value of ***integer_arg1*** is returned if ***integer_arg2*** is zero. The result is undefined if ***integer_arg2*** is negative.

---

shr.f (*integer_arg1, integer_arg2*)

An integer function that returns the value of ***integer_arg1*** shifted right by ***integer_arg2*** bit positions. For example, **shr.f(23, 2)** returns 5 because binary 010111 (23) shifted right two positions is binary 000101 (5). An arithmetic shift is performed with the sign bit copied to the most significant bit positions. The value of ***integer_arg1*** is returned if ***integer_arg2*** is zero. The result is undefined if ***integer_arg2*** is negative.

---

sign.f (*double_arg*)

An integer function that returns the sign of ***double_arg***: +1 if the argument is positive, –1 if the argument is negative, and zero if the argument is zero.

---

sin.f (*double_arg*)

A double function that returns the sine of ***double_arg***. The argument is specified in radians. The return value is in the range –1 to +1.

---

sqrt.f (*double_arg*)

A double function that returns the square root of ***double_arg***. The argument must be nonnegative.

---

tan.f (*double_arg*)

A double function that returns the tangent of ***double_arg***. The argument is specified in radians.

---

xor.f (*integer_arg1, integer_arg2*)

An integer function that returns the value obtained by performing a bitwise exclusive OR of ***integer_arg1*** and ***integer_arg2***. For example, **xor.f(23, 51)** returns 36 because the bitwise exclusive OR of binary 010111 (23) and binary 110011 (51) is binary 100100 (36).

3.02 Text Operations

---

concat.f (*text_arg1, text_arg2, …*)

A text function that returns the concatenation of two or more text arguments. For example, **concat.f("Phi", "ladelp", "hia")** returns **"Philadelphia"**.

---

fixed.f (*text_arg, integer_arg*)

A text function that returns the value obtained after appending space characters to, or removing trailing characters from, the value of ***text_arg*** to make its length equal the value of ***integer_arg***. For example, **fixed.f("abcd", 2)** returns **"ab"** and **fixed.f("abcd", 5)** returns **"abcd "**. ***Integer_arg*** must be nonnegative; if it is zero, a null string is returned.

---

length.f (*text_arg*)

An integer function that returns the number of characters in ***text_arg***. For example, **length.f("Chicago")** returns 7 and **length.f("")** returns zero.

---

lower.f (*text_arg*)

A text function that returns the value of ***text_arg*** with each uppercase letter converted to lowercase. All other characters are unchanged. For example, **lower.f("Chicago")** returns **"chicago"** and **lower.f("CAFÉ")** returns **"café"**.

---

match.f (*text_arg1, text_arg2, integer_arg*)

An integer function that returns the position of the first occurrence of ***text_arg2*** in ***text_arg1*** excluding the first ***integer_arg*** characters of ***text_arg1***, or returns zero if there is no such occurrence. Zero is returned if ***text_arg1*** or ***text_arg2*** is the null string. ***Integer_arg*** must be nonnegative. For example, **match.f("Philadelphia", "hi", 2)** returns 10 and **match.f("Chicago", "hi", 2)** returns zero.

---

repeat.f (*text_arg, integer_arg*)

A text function that returns the concatenation of ***integer_arg*** copies of ***text_arg***. For example, **repeat.f("AB", 3)** returns **"ABABAB"**. ***Integer_arg*** must be nonnegative. A null string is returned if ***text_arg*** is a null string or ***integer_arg*** is zero.

---

substr.f (*text_arg, integer_arg1, integer_arg2*)

A text function that returns a substring of ***text_arg*** when called as a right function, or modifies a substring of ***text_arg*** when called as a left function. The substring begins with the character at position ***integer_arg1*** and continues until the substring is ***integer_arg2*** characters long or until the end of ***text_arg*** is reached. (The first character of ***text_arg*** is at position 1.) For example, the statement,

```
T = substr.f("Philadelphia", 6, 5)
```

assigns **"delph"** to **T**. When called as a left function, the text value assigned to the function replaces the specified substring of ***text_arg***, which must be an unmonitored text variable. The following assignment changes the value of **T** from **"delph"** to **"delta"**:

```
substr.f(T, 4, 2) = "ta"
```

If the value assigned to the substring is not the same length as the substring, then space characters are appended to, or trailing characters are removed from, the assigned value. ***Integer_arg1*** must be positive and ***integer_arg2*** must be nonnegative. If ***integer_arg1*** is greater than the length of ***text_arg***, or ***integer_arg2*** is zero, then a null string is returned when **substr.f** is called as a right function, and no modification is made to ***text_arg*** when **substr.f** is called as a left function.

---

trim.f (*text_arg, integer_arg*)

A text function that returns the value obtained by removing leading and/or trailing blanks, if any, from the value of ***text_arg***. If ***integer_arg*** is zero, leading *and* trailing blanks are removed; if ***integer_arg*** is negative, only leading blanks are removed; and if ***integer_arg*** is positive, only trailing blanks are removed. If ***text_arg*** is the null string or contains all blanks, then a null string is returned. For example, **trim.f(" Hello ", 0)** returns **"Hello"**.

---

upper.f (*text_arg*)

A text function that returns the value of ***text_arg*** with each lowercase letter converted to uppercase. All other characters are unchanged. For example, **upper.f("Chicago")** returns **"CHICAGO"** and **upper.f("café")** returns **"CAFÉ"**.

3.03 Input/Output

---

buffer.v

An integer variable that specifies the length of "the buffer" when the first **use the buffer** statement is executed. Its default value is 132.

---

efield.f

An integer function that returns the ending column number of the next value to be read by a free-form **read** statement using the current input unit, or returns zero if there are no more input values.

---

eof.v

An integer variable that specifies the action to take when an attempt is made to read data from the current input unit beyond the end of file. If the value of the variable is zero (which is the default), the program is terminated with a runtime error. However, if the value of the variable is nonzero (typically the program sets it to 1), the variable is assigned a value of 2 to indicate that end-of-file has been reached. Each input unit has its own copy of this variable.

---

heading.v

A subprogram variable that specifies a routine to be called for each new page written to the current output unit when pagination is enabled (**lines.v** is greater than zero), or contains zero (which is the default) if no routine is to be called. The routine typically writes a page heading but may perform other tasks. Each output unit has its own copy of this variable.

---

line.v

An integer variable that contains the number of the current line for the current output unit. It is initialized to 1. If pagination is enabled (**lines.v** is greater than zero), then the first line of each page is number 1. Each output unit has its own copy of this variable.

---

lines.v

An integer variable that enables pagination for the current output unit if containing a positive value indicating the maximum number of lines per page, or disables pagination if zero (which is the default) or negative. Each output unit has its own copy of this variable.

---

mark.v

An alpha variable that specifies the character that marks the end of input data describing an external process or random variable. Its default value is **"*"** (asterisk).

---

out.f (*integer_arg*)

An alpha function that returns (when called as a right function), or modifies (when called as a left function), the specified character of the current output line. ***Integer_arg*** is the column number of the character, which must be between 1 and the record size. For example, the statement, **A = out.f(4)**, assigns the character in column four to the variable **A**. The statement, **out.f(4) = "s"**, changes the character in column four to **"s"**. This function may not be used if the current output unit has been opened for writing binary data.

---

page.v

An integer variable that contains the number of the current page for the current output unit. It is initialized to 1 and is incremented for each new page when pagination is enabled (**lines.v** is greater than zero). Each output unit has its own copy of this variable.

---

pagecol.v

An integer variable that specifies for the current output unit, a positive starting column number at which the word "Page," followed by the current page number, will be written as the first line of each page (preceding lines written by a **heading.v** routine) when pagination is enabled (**lines.v** is greater than zero); or the variable is zero (which is the default) or negative to disable this feature. Each output unit has its own copy of this variable.

---

rcolumn.v

An integer variable that contains the column number of the last character read from the current input line, or zero if no character has been read. Each input unit has its own copy of this variable.

---

read.v

An integer variable that contains the unit number of the current input unit. Its initial value is 5 because unit 5 (standard input) is the current input unit when a program begins execution. The assignment, **read.v = N**, changes the current input unit and has the same effect as the statement, **use N for input**.

---

record.v (*integer_arg*)

An integer function that returns the number of lines read from, or written to, the specified I/O unit. ***Integer_arg*** must be a valid unit number.

---

ropenerr.v

An integer variable that equals 1 to indicate that an error occurred when opening the file associated with the current input unit, or equals zero if no error occurred. If the ***Open*** statement for the unit specifies the **noerror** keyword, then the program can check the value of this variable after a **use** statement to determine whether an error occurred when opening the file; otherwise, such an error causes the program to terminate. Each input unit has its own copy of this variable.

---

rreclen.v

An integer variable that contains the number of characters read in the current input line, excluding the end-of-line character. Each input unit has its own copy of this variable.

---

rrecord.v

An integer variable that contains the number of lines read from the current input unit. Each input unit has its own copy of this variable.

---

sfield.f

An integer function that returns the starting column number of the next value to be read by a free-form **read** statement using the current input unit, or returns zero if there are no more input values.

---

wcolumn.v

An integer variable that contains the column number of the last character written to the current output line, or zero if no character has been written. Each output unit has its own copy of this variable.

---

wopenerr.v

An integer variable that equals 1 to indicate that an error occurred when opening the file associated with the current output unit, or equals zero if no error occurred. If the ***Open*** statement for the unit specifies the **noerror** keyword, then the program can check the value of this variable after a **use** statement to determine whether an error occurred when opening the file; otherwise, such an error causes the program to terminate. Each output unit has its own copy of this variable.

---

wrecord.v

An integer variable that contains the number of lines written to the current output unit. Each output unit has its own copy of this variable.

---

write.v

An integer variable that contains the unit number of the current output unit. Its initial value is 6 because unit 6 (standard output) is the current output unit when a program begins execution. The assignment, **write.v = N**, changes the current output unit and has the same effect as the statement, **use N for output**.

3.04 Random-Number Generation

---

beta.f (*double_arg1, double_arg2, integer_arg*)

A double function that returns a random number in the range zero to one from the beta distribution having shape parameters α_1 equal to ***double_arg1*** and α_2 equal to ***double_arg2***, and mean μ equal to $\frac{\alpha_1}{\alpha_1 + \alpha_2}$, where $\alpha_1 > 0$ and $\alpha_2 > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

binomial.f (*integer_arg1, double_arg, integer_arg2*)

An integer function that returns a random number in the range zero to *n* from the binomial distribution having parameters *n* equal to ***integer_arg1*** and *p* equal to ***double_arg***, and mean μ equal to *np*, where $n > 0$ and $p > 0$. The return value represents a random number of successes in *n* independent trials where *p* is the probability of success for each trial. ***Integer_arg2*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

If *n* equals 1, the binomial distribution is the same as the Bernoulli distribution.

---

erlang.f (*double_arg, integer_arg1, integer_arg2*)

A double function that returns a nonnegative random number from the Erlang distribution having mean μ equal to ***double_arg***, shape parameter α equal to ***integer_arg1***, and scale parameter β equal to $\frac{\mu}{\alpha}$, where $\mu > 0$ and $\alpha > 0$. ***Integer_arg2*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

exponential.f (*double_arg, integer_arg*)

A double function that returns a nonnegative random number from the exponential distribution having mean μ equal to ***double_arg***, where $\mu > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

gamma.f (*double_arg1, double_arg2, integer_arg*)

A double function that returns a nonnegative random number from the gamma distribution having mean μ equal to ***double_arg1***, shape parameter α equal to ***double_arg2***, and scale parameter β equal to $\frac{\mu}{\alpha}$, where $\mu > 0$ and $\alpha > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

If α equals 1, the gamma distribution is the same as the exponential distribution. If α is an integer, the gamma distribution is the same as the Erlang distribution. If μ is an integer and α equals $\frac{\mu}{2}$, the gamma distribution is the same as the chi-square distribution with μ degrees of freedom.

---

log.normal.f (*double_arg1, double_arg2, integer_arg*)

A double function that returns a nonnegative random number from the lognormal distribution having mean μ equal to ***double_arg1*** and standard deviation σ equal to ***double_arg2***, where $\mu > 0$ and $\sigma > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

normal.f (*double_arg1, double_arg2, integer_arg*)

A double function that returns a random number from the normal distribution having mean μ equal to ***double_arg1*** and standard deviation σ equal to ***double_arg2***, where $\sigma > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

poisson.f (*double_arg, integer_arg*)

An integer function that returns a nonnegative random number from the Poisson distribution having mean μ equal to ***double_arg***, where $\mu > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

randi.f (*integer_arg1, integer_arg2, integer_arg3*)

An integer function that returns a random number in the range *m* to *n* from the discrete uniform distribution having parameters *m* equal to ***integer_arg1*** and *n* equal to ***integer_arg2***, and mean μ equal to $\frac{m+n}{2}$, where $m \le n$. ***Integer_arg3*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

random.f (*integer_arg*)

A double function that returns a uniform random number in the range 0 to 1. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate equal to **1 – random.f(–*integer_arg*)**.

---

seed.v

A one-dimensional integer array that contains the current seed value for each random number stream. A stream number is used as an index into the array. The number of array elements returned by **dim.f(seed.v)** is the number of streams and is initially 10; however, the program may **release** the array and **reserve** it to change the number of streams.

---

triang.f (*double_arg1, double_arg2, double_arg3, integer_arg*)

A double function that returns a random number in the range *m* to *n* from the triangular distribution having parameters *m* equal to ***double_arg1***, peak *k* (the mode) equal to ***double_arg2***, and *n* equal to ***double_arg3***, and mean μ equal to $\frac{m+k+n}{3}$, where $m \le k \le n$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

uniform.f (*double_arg1, double_arg2, integer_arg*)

A double function that returns a random number in the range *m* to *n* from the continuous uniform distribution having parameters *m* equal to ***double_arg1*** and *n* equal to ***double_arg2***, and mean μ equal to $\frac{m+n}{2}$, where $m \le n$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

---

weibull.f (*double_arg1, double_arg2, integer_arg*)

A double function that returns a nonnegative random number from the Weibull distribution having shape parameter α equal to ***double_arg1*** and scale parameter β equal to ***double_arg2***, where $\alpha > 0$ and $\beta > 0$. ***Integer_arg*** must specify a random number stream between 1 and **dim.f(seed.v)**, or a negative stream number to generate the antithetic variate.

If α equals 1, the Weibull distribution is the same as the exponential distribution. If α equals 2, the Weibull distribution is the same as the Rayleigh distribution.

3.05 Simulation

---

between.v

A subprogram variable that specifies a routine to be called by the timing routine before each process method or process routine is executed, or contains zero (which is the default) if none is to be called. The process notice is removed from the event set (**ev.s**), and the simulation time (**time.v**) and event set index (**event.v**) are updated, before this routine is called; however, the pointer to the process notice (**process.v**) is not yet assigned.

---

clearevents.r

This routine is a utility that can be called to remove and destroy all remaining process notices in the event set(s).

---

date.f (*integer_arg1, integer_arg2, integer_arg3*)

An integer function that returns the number of days from the origin date (established by a prior call of **origin.r**) to the specified date, where month m equals ***integer_arg1***, day d equals ***integer_arg2***, and year y equals ***integer_arg3***. The arguments must satisfy $1 \le m \le 12$, $1 \le d \le 31$, and $y \ge 100$.

---

day.f (*double_arg*)

An integer function that returns the day of the month in the range 1 to 31 for the date that is ***double_arg*** days after the origin date (established by a prior call of **origin.r**). The argument must be nonnegative.

---

ev.s

A one-dimensional array of sets called the "event set." Each process method and process type in the program is assigned a unique index into this array. A smaller index value gives higher priority to the process method or process type. The set at an index contains a process notice for each scheduled invocation of the process method or process type associated with the index. The process notices are ranked within the set by increasing

time of occurrence (**time.a**). The number of elements in this array is contained in **events.v**.

---

event.v

An integer variable that contains the event set index, in the range 1 to **events.v**, of the current process method or process type during a simulation.

---

events.v

An integer variable that contains the largest event set index, which is equal to the total number of process methods and process types defined by the program.

---

f.ev.s

A one-dimensional pointer array that contains in each element the reference value of the process notice for the most imminent invocation (smallest **time.a**) of a process method or process type, or is zero if there are no scheduled invocations. The number of elements in this array is contained in **events.v**.

---

hour.f (*double_arg*)

An integer function that returns the hour part, in the range 0 to **hours.v–1**, of the number of days specified by ***double_arg***, which must be nonnegative.

---

hours.v

A double variable that specifies the number of hours per day. Its default value is 24.0.

---

l.ev.s

A one-dimensional pointer array that contains in each element the reference value of the process notice for the least imminent invocation (largest **time.a**) of a process method or process type, or is zero if there are no scheduled invocations. The number of elements in this array is contained in **events.v**.

---

minute.f (*double_arg*)

An integer function that returns the minute part, in the range 0 to **minutes.v–1**, of the number of days specified by ***double_arg***, which must be nonnegative.

---

minutes.v

A double variable that specifies the number of minutes per hour. Its default value is 60.0.

---

month.f (*double_arg*)

An integer function that returns the month in the range 1 to 12 for the date that is ***double_arg*** days after the origin date (established by a prior call of **origin.r**). The argument must be nonnegative.

---

n.ev.s (*integer_arg*)

An integer function that returns the number of process notices in **ev.s(*integer_arg*)**. The argument must be in the range 1 to **events.v**.

---

nday.f (*double_arg*)

An integer function that returns the day part of the number of days specified by ***double_arg***, which must be nonnegative.

---

origin.r (*integer_arg1, integer_arg2, integer_arg3*)

A subroutine that establishes the specified date as the origin, where month m equals ***integer_arg1***, day d equals ***integer_arg2***, and year y equals ***integer_arg3***. The arguments must satisfy $1 \le m \le 12$, $1 \le d \le 31$, and $y \ge 100$.

---

process.v

A pointer variable that contains the reference value of the process notice for the current process method or process routine during a simulation, or zero if no process method or process routine is active.

---

time.v

A double variable that contains the current simulation time. Its initial value is zero, which corresponds to the start of the day of origin.

---

weekday.f (*double_arg*)

An integer function that returns the weekday, in the range 1 to 7 representing Sunday through Saturday, for the date that is ***double_arg*** days after the origin date. If no origin date has been established by a prior call of **origin.r**, the origin is assumed to be a Sunday. The argument must be nonnegative.

---

year.f (*double_arg*)

An integer function that returns the year for the date that is ***double_arg*** days after the origin date (established by a prior call of **origin.r**). The argument must be nonnegative.

3.06 Miscellaneous

---

batchtrace.v

An integer variable that specifies the action to take when a runtime error occurs. The debugger is invoked unless the value of the variable is 1 or 2. If the value is 1, a traceback is written to a file named "simerr.trc" and **snap.r** is called. If the value is 2, the program exits without a traceback or **snap.r** invocation. The default value is zero, which invokes the debugger.

---

date.r yielding *text_arg1*, *text_arg2*

A subroutine that returns the current date in the form **MM/DD/YYYY** in ***text_arg1*** and the current time in the form **HH:MM:SS** in ***text_arg2***.

---

err.message.f

A left function that can be assigned a text string in the event of an error detected while running the program. The error message will be printed and the debugger will be invoked.

---

exit.r (*integer_arg*)

A subroutine that terminates the program with an exit status of ***integer_arg***.

---

high.f

Returns the upper index boundary of an array. "DIM.F" will be returned unless the array is reserved using the **reserve** statement in conjunction with **to** keyword. I.e. "**reserve ARR(*) as –10 to 10"**

---

low.f

Returns the lower index boundary of an array. "1" will be returned unless the array is reserved using the **reserve** statement in conjunction with **to** keyword. I.e. "**reserve ARR(*) as –10 to 10"**

---

parm.v

A one-dimensional text array that contains the command-line arguments given to the program when it was invoked. **Dim.f(parm.v)** is the number of command-line arguments and is zero if no arguments were provided.

---

snap.r

A subroutine that may be provided by the program which is invoked when a runtime error occurs and the value of **batchtrace.v** is 1. The subroutine may write to the file named "simerr.trc" by writing to the current output unit.

---

wordsize.f

Returns 64 for 64-bit simscript and 32 for 32-bit simscript
