```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TIME (AIRPLANE) minutes
     relinquish 1 RUNWAY
end " process AIRPL

 process AIRP
      call T
      work
      requ
```

Reference Handbook

CACI Products Company

For product information or technical support contact:

In the US and Pacific Rim:

CACI Products Company
3333 North Torrey Pines Court
La Jolla, California 92037
Phone: (619) 824.5200
Fax: (619) 457.1184

In Europe:

CACI Products Division
Coliseum Business Centre
Riverside Way, Camberley
Surrey GU15 3YL UK
Phone: +44 (0) 1276.671.671
Fax: +44 (0) 1276.670.677

Table of Contents

List of Figures

Preface

SIMSCRIPT II.5 is a powerful, free-form, English-like, general-purpose simulation programming language. It supports the application of software engineering principles, such as structured programming and modularity, which impart orderliness and manageability to simulation models.

SIMSCRIPT II.5 is a fully documented language:

- *SIMSCRIPT II.5 Programming Language* illustrates usage of the language constructs without interactive graphics.
- *Building Simulation Models with SIMSCRIPT II.5* is oriented toward real applications of model building, and features the case study approach used successfully in the short course given regularly by CACI Products Company.
- *SIMGRAPHICS II User's Manual for SIMSCRIPT II.5* describes graphical editor and language statements and data structures for creating interactive graphical models with SIMSCRIPT II.5. These texts are available through CACI.

This handbook is intended for use by experienced SIMSCRIPT users who have specific questions about syntax or usage of individual non-graphical statements. The alphabetic organization by keyword supports this usage. For more general questions of modeling or strategy, the reader is referred to the other books mentioned above.

Free Trial Offer

SIMSCRIPT II.5 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 II.5 are scheduled on a recurring basis in the following locations:

La Jolla, California
Washington, D.C.
London, United Kingdom

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

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

In the U.S. and Pacific Rim:

CACI Products Company
3333 North Torrey Pines Court
La Jolla, California 92037
(619) 824.5200
Fax: (619) 457.1184

In the UK and Europe:

CACI Products Division
Coliseum Business Centre
Riverside Way, Camberley
Surrey GU15 3YL UK
+44 (0) 1276.671.671
Fax: +44 (0) 0276.670.677

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

In the U.S. and Pacific Rim:

CACI Products Company
3333 N. Torrey Pines Court
La Jolla, CA 92037
(619) 824.5200
Fax: (619) 457.1184

In the UK and Europe:

CACI Products Division
Coliseum Business Centre
Riverside Way
Camberley
Surrey
GU15 3YL UK
+44 (0) 1276.671.671
Fax: +44 (0) 0276.670.677

Formal Syntax Employed In This Publication

A. Description of Syntax

This publication employs a formal notation for describing the syntax of SIMSCRIPT II.5 statements. Where necessary, the notation is supplemented by comments that describe synonyms for keywords, semantic constraints, and other considerations for the programmer. View the comments as an integral part of the syntax wherever they are included.

The following conventions are used in the formal notation:

1. All of the following characters are to be used exactly as shown, except where appearing as a superscript (see 4, below):

General Term	Characters
Uppercase letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Digits	0123456789
Punctuation	' " . ,
Special characters	( ) + - * / $: < > = ¬
Blanks, one or more wherever indicated	

2. All of the following characters are used by the notation to refer to primitives and metavariables (defined in paragraph B below), except where appearing as a superscript (see 4 below):

General Term	Characters
Lowercase italics	*abcdefghijklmnopqrstuvwxyz*

3. When a list of words or expressions appear in brackets, i.e., [ ], a choice *may* be made from the options indicated. When a list of words or expressions appears in braces, i.e., { }, a choice *must* be made from the options indicated.

4. Superscripted text immediately following a right-hand bracket or brace denotes optional repetition of the material, or choice of material, enclosed in that set of brackets or braces. When the material to be repeated consists of a single keyword, primitive or metavariable, the superscripted text immediately follows. The superscripted text itself denotes the connective that must appear between repetitions of the material, as follows:

`c`	A comma, the word AND, or a comma followed by the word AND.
`c then`	Any of the above, followed by the word THEN.
`by`	The word BY.

`{and / or}`	Either of the two words AND and OR; the choice depends on the logic that the programmer wishes to express.
`or`	The word OR. A comma can be used instead of OR.
`i`	No connective; simply repeat the material.

B. Primitives and Metavariables Referred to in Syntax

B.1 Primitives

integer — A sequence of digits delimited by blanks, special characters, or the end of a record; a numeral denoting a whole number.

name — A sequence of letters, digits, and periods containing at least one letter, delimited by blanks, special characters, or the end of a record. In some cases, a programmer can use any name not already defined in the preamble. Most of the time, however, a name refers to one of the following:

- *attribute* (sometimes further specified):
  - *entity attribute* (sometimes further specified):
  - *permanent entity attribute*
  - *unsubscripted system attribute*
- *entity* (sometimes further specified):
  - *permanent entity*
  - *temporary entity*
- *event*
- *label*
- *process*
- *resource*
- *routine*
- *set*
- *variable* (sometimes further specified):
  - *array*
  - *global variable* (sometimes further specified):
  - *unsubscripted global variable*
  - *local variable*
  - *pointer variable*
  - *text variable*

statement — A series of words (q.v.) that can be generated by the formal syntax notation employed in Part II of this publication for any of the entries referred to as "statements". In this regard, a statement can only be defined by its inclusion in a list of allowable SIMSCRIPT II.5 statements.

string — Any sequence of characters, delimited by the end of a record.

word — Any single character, or a sequence of two or more characters not containing a special character, delimited by blanks (unless the word is a single special character) or the end of a record.

B.2 Metavariables

Metavariables are best defined by the portions of this publication that list and/or describe them.

c — A comma, the word `and`, or a comma followed by the word `and`.

for phrase — A string defined by the syntax for any of the following:

- `for ... each` (*class*)
- `for ... to` (*index*) phrase
- `for ... of` (*set*) phrase

logical expression — Any of the entries described in table 1. Also read the paragraph entitled Logical Expressions in Part I of this publication.

quantity — A value (q.v.) which is of integer or real mode.

relational operator — Any of the entries described in table 2.

selection phrase — A string defined by the syntax for either of the following:

- `unless` phrase
- `with` phrase

set attribute — Any of the letters defined in table 14.

set routine — Any of the mnemonic abbreviations defined in table 15.

stat kywd — Any of the statistical keywords described in tables 9 and 13.

termination phrase — A string defined by the syntax for either of the following:

- `until` phrase
- `while` phrase

unconditional transfer A string defined by the syntax for the following:

- `cycle` statement
- `go to` statement
- `go to ... per` statement
- `jump` statement
- `leave` statement
- `return` statement
- `stop` statement

value A constant, a variable, or any expression that evaluates to an integer, a real number, or a text string.

PART I
GENERAL REFERENCE

1. General Reference

1.1 Attributes

Each entity in a SIMSCRIPT II.5 simulation is composed of a collection of memory cells called *attributes*. The values define characteristics of the entity. All entities of the same entity class have the same set of attributes, but the values of these attributes differ and are set by the program. Attributes can have real or integer values (as declared in **`normally`** or **`define ... variable`** statements). Each attribute of each entity must have a different name, unless two or more attributes are placed in the same relative location in two or more entities ("common attributes"). An attribute is similar to a global variable in that it pertains to the global environment and is initialized to zero. However, attribute values can be packed, while values of global variables cannot. Entity classes and their associated attributes are declared in **`every`** statements, and system attributes are declared in **`the system`** statements.

1.1.1 Function Attributes

A function attribute is one whose value is computed by a function routine. Consequently, a routine must be written with the same name as the attribute and with as many arguments as the attribute has subscripts; that is, no arguments for an unsubscripted system attribute, one argument for a temporary or permanent entity, two arguments for a two-dimensional system attribute, and so on. Because function attributes designate routines, no storage in entity records is allocated for the values. Function attributes can be used, for example, to perform complex calculations and to monitor and print.

1.2 Constants

1.2.1 Numeric Constants

A constant is a quantity that remains unchanged during program execution, and refers to a literal value. Integer and real (decimal) numbers, signed or unsigned, are permitted. When equivalent representations of a number exist, they all have the same value; for example, +7.5, 7.5, and 07.5000 all represent the same number. Some integer constants are:

```
l576        -53         +2000
```

and sample real numbers are:

```
0.25        -0.0756     +2.68       351.9       9.
```

Note: **`9.`** is a real value. Constants in scientific notation (e.g., 1.56E04) are not allowed in programs, although input data may be read in this form.

1.2.2 Subprogram Literals

A subprogram literal is formed by enclosing the name of a subprogram (subroutine or right-hand function) in single apostrophe characters. This represents the address of that routine, which can then be stored in a subprogram variable for later indirect call. Examples are:

```
'cos.f'    'process'
```

1.2.3 Text Literals

A literal text constant is represented by a string of characters enclosed within two quotation (") marks; a quotation mark within a literal is represented by two successive quotation marks. The length of a text literal is limited only to the length of the input line. Because text literals are compiled directly into programs, they cannot be altered. Examples of literals are:

```
"word"
"a very long text literal string"
"Say  ""hi"""
"0123456789"
```

The null string is a string of length zero that contains no characters. It is represented by two quotation marks ("").

A text literal is stored as a pointer to an area of memory where one or more computer words contain the represented characters.

1.2.4 Alpha Literals

A literal alpha constant is similar to a text constant, except that a smaller number of characters may be represented. On most implementations, alpha constants represent only a single character. The SIMSCRIPT II.5 compiler distinguishes between text and alpha constants by context. A character literal is an alpha constant if it is assigned to a variable of mode ALPHA, as follows:

```
define ALPHA as an alpha variable
define STRING as a 1-dim alpha array
.
.
let ALPHA = "k"
let STRING(1) = """"
let STRING(2) = "x"
let STRING(3) = """"
```

For those implementations limited to one character per alpha constant, alpha values are stored right-justified within a computer word, padded on the left with zeros. Otherwise, alpha values are stored left-justified, padded on the right with blanks.

1.3 Arithmetic Expressions

An arithmetic expression is written as a string of variables, constants, functions, arithmetic operators, and parentheses. The simplest expression consists of a single constant or variable, and simple expressions can be combined with arithmetic operators to form compound expressions. Terms can be enclosed within parentheses to indicate a desired order of evaluation. Unless otherwise stated, an expression can include any of the following:

- System- and programmer-defined constants
- System variables
- Subscripted and unsubscripted local variables
- Subscripted and unsubscripted global variables
- Attributes and function attributes
- System- and programmer-defined function references
- Subprogram variables
- Subscripted and unsubscripted monitored variables.

1.3.1 Arithmetic Operators

All arithmetic operators must be expressed explicitly (e.g., no implied multiplication), and no two operators can appear consecutively. Because the exponentiation operator is treated as a single unit, blanks cannot appear between the two asterisks. The arithmetic operators are:

+	Addition	-	Subtraction
*	Multiplication	/	Division
**	Exponentiation		

1.3.2 Hierarchy of Operations

When computing the value of a complex expression, the following hierarchy specifies the order in which the different operations are performed relative to each other:

**	Exponentiation
* and /	Multiplication and division
+ and -	Addition and subtraction

Expressions are generally evaluated from left to right. However, operands may or may not be accessed in this order, depending on the implementation. As each expression is evaluated, exponentiation is performed before multiplication and division, which are performed before addition and subtraction. For example, the formula:

```
ax2 + bx + c
```

can be expressed as:

```
a  * x**2 + b  * x + c
```

SIMSCRIPT II.5 computes the value of this expression by:

1. Squaring **x**.
2. Multiplying x^2 by **a**.
3. Multiplying **x** by **b**.
4. Adding the two products together.
5. Adding **c** to the result.

Step 3 may be done before step 1 or before step 2 by some language implementations.

1.3.3 Parentheses

The standard hierarchy of operations may be altered by enclosing terms within parentheses. SIMSCRIPT II.5 performs all operations within parentheses before completing the remaining evaluation. For example, the system computes the value of the expression **k/(m + n)** by:

1. Adding **m** and **n**.
2. Dividing **k** by this sum.

When pairs of parentheses are embedded within other pairs, terms within the innermost pair are evaluated first. For example, SIMSCRIPT II.5 computes the value of the expression **(((a + b)*25.3)-c**2)/d** by:

1. Adding **a** and **b**.
2. Multiplying this sum by **25.3**.
3. Squaring **c**.
4. Subtracting the square of **c** from the product in step 2.
5. Dividing the difference by **d**.

SIMSCRIPT II.5 processes parenthesized expressions from left to right and removes all parentheses before computing a final value. In removing parentheses, the system may obtain values of subscripted variables from storage and place these values in temporary variables, or it may save only the subscript. Compound expressions containing parentheses are simplified. Whenever an expression enclosed within parentheses contains more than one term, the hierarchy of operations determines the order of evaluation. Subscripted variables and functions with argument lists are evaluated as if they were parenthesized expressions.

1.3.4 Mixed Mode Expressions

The result of evaluating simple arithmetic expressions involving both integer and real variables is as follows.

if:		then:				
a	b	a + b	a - b	a*b	a/b	a**b
integer	integer	integer	integer	integer	real	real
integer	real	real	real	real	real	real
real	integer	real	real	real	real	real
real	real	real	real	real	real	real

Compound expressions are evaluated as a sequence of simple expressions that follow the above rules. For example, if **a, b, c, x,** and **y** are all integer, **a*b/(x*y)** is real, but **a*(x+y*(b+c))** is integer.

Constants in simple expressions will be converted to the appropriate mode by the compiler to avoid run-time conversion.

1.4 Logical Expressions

Logical expressions are required in **if ... else ... always** constructs and in **with, unless, while** and **until** phrases. Table 1 lists the available logical expressions. A simple logical expression is formed by joining two arithmetic expressions with a relational operator. Relational operators are listed in table 2. A logical expression is true if the relationship expressed is true. It is false if the relationship is not true. During program execution, current values of variables in the arithmetic expressions determine whether a logical expression is true or false. A logical expression can optionally include the word **is** for clarity (e.g., **x is equal to y**). Simple logical expressions can be combined with relational operators, or with the **and** and **or** logical operators, to form compound logical expressions. Some examples follow:

```
x = 0
```

True if the value of variable **x** is zero.

```
x + (a**2) + (b**2) > a * b
```

True if the value of expression **x + (a**2) + (b**2)** is greater than the value of **a * b**.

```
(x(1)**2 / y(1)**2) LS (max + factor)
```

True if the value of expression **x(1)**2 / y(1)**2** is less than the value of **max + factor**.

```
x < = SAMPLE < = y = LIMIT
```

True if all conditions are true: **SAMPLE** is greater than or equal to **x**; **SAMPLE** is less than or equal to **y**; and **y** equals **LIMIT**.

```
NO.RUNWAYS(AIRPORT) is > = 5
```

True if the value of attribute **NO.RUNWAYS** is greater than or equal to 5; the word IS is optional.

```
CHARACTER equals (ALPHABET(I)) is false
```

True if the value of variable **CHARACTER** does not equal the **I**th value of array **ALPHABET**.

```
mode is alpha and card is not new
```

True if both logical expressions, **mode is alpha** and **card is not new**, are true.

```
(FARE(PATRON) ls LOW is true) or DESTINATION(PATRON)= "sfo" is
false
```

True if either, or both, logical expressions are true: value of attribute **FARE** is less than the value of **low** and/or the value of **DESTINATION** does not equal the alphanumeric literal **sfo**. Logical expressions can be optionally enclosed in parentheses.

```
this FLIGHT is not in some WAITING.LINE
```

True if the entity whose index is the value of **FLIGHT** is not in a set of the class **WAITING.LINE.**

Table 1. Logical Expressions

Any of the following logical expressions can be followed by the words **`is true`** or **`is false`**. Logical expressions can also be joined by **`and`** and **`or`**, and enclosed in parentheses, to express the logic desired. Relational operators (referred to in the first example) are listed in table 2.

value [is] *relational_operator value*

value [is] [not] { positive / negative / zero }

mode [is] [not] { integer / integer / alpha }

data [is] [not] ended

card [is] [not] new

page [is] [not] first

[the / this] *SET* is [not] empty

[the / this] *ENTITY* is [not] in [a / an / the / some] *SET*

{ event / process } is [not] { endogenous / exogenous / internal / external }

Table 2: Relational Operators

Mathematical Symbol	Permitted Forms
=	= eq equals equal to
≠	< > ne not equal to
<	< lt ls less than
>	> gt gr greater than
≤	< = le no greater than not greater than
≥	> = ge no less than not less than

1.4.1 Property Comparisons

Some logical expressionss employ nonarithmetic comparisons to test for a particular property. For example, an arithmetic expression can be compared with the names **`positive`**, **`negative`**, or **`zero`**. Similarly, a set can be **`empty`** or **`not-empty`**. In these cases, a true or false condition will exist.

1.4.2 Arithmetic Relational Conditions

Table 2 lists the mathematical symbol for each relational operator and the corresponding forms permitted in SIMSCRIPT II.5. Unless the special characters are used, each relational operator must be separated from the arithmetic expressions on either side by parentheses, or by at least one blank column.

1.4.3 Compound Relational Expressions

Simple relational comparisons may be cascaded to indicate that multiple comparisons are to take place. Consider the expression:

```
a = b > c and q/r = d = c
```

For this expression to be true, it must be the case that:

```
a = b and b > c and q/r = d and d = c
```

1.4.4 Mixed Mode Comparisons

If the modes of the arithmetic constituents of a logical expression differ, all integer expressions are converted to real before evaluating the logical expression as true or false.

1.4.5 IS TRUE and IS FALSE Phrases

A logical expression can be followed optionally by an **`is true`** or **`is false`** phrase. The is **`true`** phrase is used for clarity, but an **`is false`** phrase negates the logical expression and is used to maintain a desired program logic. When an **`is true`** or **`is false`** phrase appears in a compound logical expression, the phrase always applies to the logical expression immediately preceding it. If this logical expression is compound, it must be enclosed within parentheses, or the phrase will apply only to the immediately adjacent expression.

1.4.6 AND and OR Logical Operators

Simple logical expressions can also be combined with the logical operators **`and`** and **`or`** to form compound logical expressions, following the rules:

1. If two logical expressions are combined with an **`and`** logical operator, both logical expressions must be true for the compound logical expression to be true.
2. If two logical expressions are combined with an **`or`** logical operator, the compound logical expression is true if either, or both, logical expressions are true.

When more than two simple logical expressions appear in an unparenthesized compound logical expression, the logical operator **`and`** is evaluated first. Proceeding from left to right, successive logical expressions are used as operands of **`and`** operators, and these evaluated expressions are then operands of **`or`** operators. Parentheses can be used, however, to indicate a specific order of evaluation. For example:

```
a = b or c < d and e > f
```

is logically equivalent to:

```
a = b or (c < d and e > f)
```

but could be changed with parentheses to:

```
(a = b or c < d) and e > f
```

1.5 Labels

A statement label identifies a transfer point for a **go to** statement. Labels are always local. Because a label always refers to a statement in the routine containing that label, the same label can appear in different routines. In addition, different labels can identify the same statement; they are then called *equivalent* labels. Of course, transfers cannot be made between routines by means of a **go to** statement.

A label can be any combination of letters, digits, and periods enclosed in apostrophes. Some sample labels are:

```
'writeoutputcard'          'region.2'
'write.output.card'        'transfer'
'999'                      'error.13'
'...error'
```

1.5.1 Subscripted Labels

A label can be subscripted by placing a single positive integer in parentheses immediately after the label name. Subscripted labels permit labels to be added or deleted without modifying the **go to** statements that transfer control to them. They are convenient, for example, in programs containing **go to** statements that direct control to segments in a frequently modified program. When the system executes a **go to** statement having a subscripted label, control is transferred to the statement preceded by the same label and subscript equal to the integer value of the expression.

Subscripts need not start with the number 1, and subscripted labels need not be in consecutive order in a program. Some examples of subscripted labels are:

```
'place(3)'    'input.card(32)'    'x(1)'
```

1.6 Modes

A variable can have **integer, real, alpha**, or **text** mode. On some implementations, double-precision and single-precision floating point modes are recognized as **double** and **real** modes, respectively.

An **integer** variable represents a whole number or an entity pointer. A **real** (or **double**) variable represents a number that may have fractional values. **Alpha** and **text** variables represent character strings. The system assumes that the mode of a variable is **real** (**double** on some implementations) unless otherwise specified in **normally** or **define ... variable** statements. Subscripted variables can have any mode, but all elements of an array have the same mode. System attributes, and permanent and temporary entities, can also have any mode. Except for set pointers, which are automatically defined as integer, the mode of all attributes is declared by the programmer. The use of a variable should conform with its declared mode. Table 3 describes data conversion as performed automatically by SIMSCRIPT II.5 when reading input data.

1.6.1 Text Mode

The value of a text variable is a variable-length character string, that is, a sequence of alphanumeric characters. An alphanumeric character can be a letter, a digit, or any of the special characters for the particular implementation of SIMSCRIPT II.5. A text variable may contain from 0 to 32,000 characters. The way in which the characters are represented varies among the implementations.

Individual characters of a text string may be accessed by using the **substr.f** system function, which creates (or puts) a substring from (into) the text string. During input, output, and substring manipulation, if the source and destination strings are of different lengths the source field is left-justified, with excess characters truncated from the right, and blanks substituted for missing characters.

Text mode variables may not be mixed with other modes in expressions or **let** statements, except when using a mode conversion function.

Examples of possible values of text variables are:

```
X
0123456789
Dr. Mary P. Smith
****$2,347.02*
```

Table 3. Input Data Conversions

Data Entered	Variable Defined As:	SIMSCRIPT II.5 Action:
Integer	**`Integer`**	Value stored in variable as integer.
Integer	**`Real`**	Value converted to real and stored in variable.
Integer	**`Alpha`**	Value stored as fixed-length string.
Integer	**`Text`**	Value stored as variable-length string.
Real	**`Integer`**	Program terminates with error condition.
Real	**`Real`**	Value stored in variable as real.
Real	**`Alpha`**	Value stored as fixed-length string.
Real	**`Text`**	Value stored as variable-length string.
Character	**`Integer`**	Program terminates with error condition.
Character	**`Real`**	Program terminates with error condition.
Character	**`Alpha`**	Value stored as fixed-length string.
Character	**`Text`**	Value stored as variable-length string.
Note: Character data are acceptable as both alpha and text variable data.		

1.6.2 Alpha Mode

Alpha variables represent fixed-length character strings, with the length constant for a given implementation (typically from 1 to 10 characters). Strings that are too long to be represented by an alpha variable are truncated on the right. Possible values for alpha variables are similar to those for text variables, except for the fixed-length restriction.

Alpha variables are treated as integer, except for input and output and conversion to or from text values.

1.6.3 Mixed Numeric Modes

SIMSCRIPT II.5 permits mixed numeric mode. That is, integer and real variables may be combined in the same statement. The result of evaluating arithmetic expressions involving mixed numeric modes was described above (see Mixed Mode Expressions). Briefly, the system evaluates compound expressions as a series of simple expressions. Thus, if a compound expression having all integer variables contains only additions, subtractions, and multiplications, the expression will yield an integer value; and if a compound expression having all integer variables contains division or exponentiation, the expression will yield a real value. The library function can divide two integers and yield a truncated integer value.

Integer-to-real conversion converts the integer number to a real number with the same value (e.g., `-36` becomes `-36.0`, and `75` becomes `75.0`). The system rounds real values to

integers by adding 0.5 to the value, depending on whether the value is positive or negative, and truncating the result. Thus, if a real value is `-1.50`, the rounded integer value would be `-2`.

In **`let, add`**, and **`subtract`** statements, when the mode of the expression differs from that of the variable, SIMSCRIPT II.5 converts the expression as above to the mode of the variable before changing the value of the variable. When arithmetic expressions in a logical expression yield different modes, the integer expressions are converted to real before the system determines whether the logical expression is true or false.

1.6.4 Functions for Conversion

SIMSCRIPT II.5 provides seven library functions for data conversion:

1. **`Atot.f`** converts an alpha value to a text string.
2. **`Int.f`** converts a real expression to a rounded integer.
3. **`Itoa.f`** converts the first *n* digits of an integer expression to an alpha value, where *n* is the implementation-specific limit on the number of characters per alpha variable.
4. **`Itot.f`** converts an integer expression to a text value.
5. **`Real.f`** converts an integer expression to a real value.
6. **`Trunc.f`** converts a real expression to a truncated integer.
7. **`Ttoa.f`** converts the first *n* characters of a text string to an alpha value, where *n* is the implementation-specific limit on the number of characters per alpha variable.

1.7 Names

Attributes, variables, and routines must be named, as must entity classes, event classes, and set classes. A name can be any combination of letters, digits, and periods that contains at least one letter or two or more nonterminal periods. SIMSCRIPT II.5 disregards all periods written at the end of names and numbers. Names can have terminal periods, but the system deletes them during compilation. Thus, the names **`flight...`** and **`flight..`** become **`flight`** when a program is compiled. Some sample names are:

```
33A33                   table
region.1                ...89
first.class.seats       consumed.fuel
x                       stockholder
```

1.8 System

The tables that follow describe the automatically-generated attributes, routines, and variables, along with system constants, library functions, and system routines and variables. These system names cannot be used as programmer-defined names because they have predefined meanings. In forming system names, the following conventions are used:

a. Automatically-generated attributes, routines, and variables (listed in table 4) either begin with a letter followed by a period, or end with a period followed by the letter **A**.

b. Constants (table 5) end with a period followed by the letter **c**.

c. Functions (table 6) end with a period followed by the letter **f**.

d. Routines (table 7) end with a period followed by the letter **r**.

e. Variables (table 8) end with a period followed by the letter **v**.

Table 4. Automatically Generated Attributes, Routines, and Variables

Generated for	Generated Elements	Name	Definition
Accumulated and tallied variables	Routine	**R.variable**	A left-hand monitoring routine that accumulates or tallies data.
Entities	Variables	**entity**	Global variable having the entity class name.
		N.entity	Number of entities of the class (permanent entities only).
	Routines	**C.entity**	To reserve storage for permanent entities (i.e. to create them).
		D.entity	Called when destroying a temporary entity to check for set membership error.
		L.entity	Called to list the values of entity attributes.
Event notices	Variables	**event**	Global variable having the event notice name.
		I.event	Global variable holding the subscript for this event class in the event set.
	Routine	**C.event**	File events, whose priorities are declared in **break ... ties** statements, in the proper event set.
		D.event	Called when destroying event notice to check for set membership error.
		L.event	Called to list values of event notice attributes.

Table 4. Automatically Generated Attributes, Routines, and Variables (Continued)

Generated for	Generated Elements	Name	Definition
Event notice records	Attributes	**time.a**	Time event is to occur.
		eunit.a	Equals 0 for an endogenous event; equals input unit number (≠ 0) for an exogenous event.
		p.ev.s	Pointer to predecessor event in the event set.
		s.ev.s	Pointer to successor event in the event set.
		m.ev.s	Set to 1 if the event is in the set; set to 0 if the event is not in the set.
Processes	Routines	**C.process**	File processes, whose priorities are declared in **break ... ties** statements, in the proper event set.
		D.process	Called when destroying process notice to check for set membership error.
Process notices	Attributes	**time.a**	Next scheduled entry time for process.
		p.ev.s	Pointer to predecessor process in event set.
		s.ev.s	Pointer to successor process in event set.
		m.ev.s	Set to < > 0 if the process is in the set; set to 0 if the process is not in the set.
		sta.a	State of process: **3: interrupted** **2: suspended** **1: active** **0: passive**
		ipc.a	The value of **I.process** for this process class.
		f.rs.s	Set of resources owned.

Table 4. Automatically Generated Attributes, Routines, and Variables (Continued)

Generated for	Generated Elements	Name	Definition
Random variables	Attributes of **random.e**	**prob.a**	Probability values.
		ivalue.a	Sample value containing an integer value.
		rvalue.a	Sample value containing a real value.
		S.variable	Pointer to successor.
Resources	Sets	**Q.resource**	Set of processes using this resource.
		X.resource	Set of processes using this resource.
	Attributes	**U.resource**	Capacity in resource units.
Sets	Attributes of owner entities	**F.set**	Pointer to first entity in set.
		L.set	Pointer to last entity in set.
		N.set	Number of entities currently in set.
	Attributes of member entities	**P.set**	Pointer to predecessor entity in set.
		S.set	Pointer to successor entity in set.
		M.set	Equals < > 0 if the entity is in the set; equals 0 if the entity is not in the set.
	Routines	**T.set**	Files entity first or ranked highest in set.
		U.set	Files entity last in set.
		V.set	Files entity before specified entity in set.
		W.set	Files entity after specified entity in set.
		X.set	Removes first entity from set.
		Y.set	Removes last entity from set.
		Z.set	Removes specific entity from set.

Table 5. System Constants

Constant	Mode	Description
exp.c	Real	e; 2.718281828
inf.c	Integer	Largest integer value that can be stored.
pi.c	Real	π; 3.14159265
radian.c	Real	57.29577 degrees/radian (57.29577 = 180/π)
rinf.c	Real	Largest real value that can be stored.

Note: In table 6 below, the abbreviations **a, p, q**, and **t** refer to alpha expressions, pointer variables, quantities (numeric expressions), and text expressions, respectively.

Table 6. System Functions

Function Mnemonic	Arguments	Function Mode	Description
abs.f	q	Mode of q	Returns the absolute value of the expression.
and.f	q_1, q_2	Integer	Returns the logical product of q_1 and q_2. (q_1, q_2 = integers)
arccos.f	q	Real	Computes the arc cosine of a real expression; $-1 \leq q \leq 1$
arcsin.f	q	Real	Computes the arc sine of a real expression; $-1 \leq q \leq 1$
arctan.f	q_1, q_2	Real	Computes the arc tangent of q_2/q_1; $q_1, q_2 \neq (0.0)$
atot.f	a	Text	Converts an alpha expression to a text value.

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
beta.f	q_1, q_2, q_3	Real	Returns a random sample from a beta distribution: q_1 = power of x, real; $q_1 > 0$ q_2 = power of (1-x), real; $q_2 > 0$ q_3 = random number stream, integer
binomial.f	q_1, q_2, q_3	Integer	Returns a random sample from a binomial distribution: q_1 = number of trials, integer q_2 = probability of success, real q_3 = random number stream, integer
concat.f	t_1,t_2	Text	Concatenates two text values to produce a text value containing the characters of each.
cos.f	q	Real	Computes the cosine of a real expression given in radians.
date.f	q_1, q_2, q_3	Integer	Converts a calendar date to cumulative simulation time: q_1 = month, integer q_2 = day, integer q_3 = year, integer
day.f	q	Integer	Converts simulation time to the day portion; q = cumulative simulation time, real.
dim.f	p	Integer	Returns the number of elements pointed to by the pointer variable p, in the dimension of the array p.

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
div.f	q_1, q_2	Integer	Returns the truncated value of q_1/q_2: q_1 = dividend, integer q_2 = divisor, integer; $q_2 \neq 0$
efield.f	None	Integer	Returns the ending column of the next data field to be read by a statement.
erlang.f	q_1, q_2, q_3	Real	Returns a sample value from an Erlang distribution: q_1 = mean, real q_2 = k, integer q_3 = random number stream, integer
exp.f	q	Real	Computes exp.c to the q^{th} power; q must be real.
exponential.f	q_1, q_2	Real	Returns a random sample from an exponential distribution: q_1 = mean, real q_2 = random number stream, integer
fixed.f	t, q	Text	Returns a copy of t which is either space-padded or truncated so that its length is q. t = text q = integer
frac.f	q	Real	Returns the fractional portion of a real expression.

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
gamma.f	q_1, q_2, q_3	Real	Returns a random sample from a gamma distribution: q_1 = mean, real q_2 = k, real q_3 = random number stream, integer
hour.f	q	Integer	Converts event time to the hour portion; q = cumulative event time, real.
int.f	q	Integer	Returns the rounded integer portion of a real expression.
istep.f	p, q	Integer	Returns a random sample from a look-up table without interpolation. p = variable that points to look-up table q = random number stream integer interpolation
itoa.f	q	Alpha	Converts an integer expression to an alpha value, left-adjusted in a blank field.
itot.f	q	Text	Converts an integer expression to a text value.
length.f	t	Text	Returns the length of a text variable in characters.
log.e.f	q	Real	Computes the natural logarithm of a real expression; q > 0
log.normal.f	q_1, q_2, q_3	Real	Returns a random sample from a log-normal distribution: q_1 = mean, real q_2 = standard deviation, real q_3 = random number stream, integer

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
log.10.f	q	Real	Computes $\log_{10}$ of a real expression; q > 0.
lower.f	t	Text	Converts letters in a text string to lower case.
match.f	t_1, t_2, q	Integer	Returns the location of a text substring within a text string, or 0 if not found. t_1 = source, text t_2 = pattern to be matched, text q = number of characters of source to be skipped, integer
max.f	q_1, q_2, ..., q_n	Real, if any q_i real; if not, integer	Returns the value of largest q_i
min.f	q_1, q_2, ..., q_n	Real, if any q_i real; if not, integer	Returns the value of smallest q_i
minute.f	q	Integer	Converts event time to the minute portion; q = cumulative event time, real.
mod.f	q_1, q_2	Real if either q_i real; if not, integer	Computes a remainder as: q_1 - trunc.f(q_1/q_2) * q_2; $q_2 \neq 0$
month.f	q	Integer	Converts simulation time to month portion; q = cumulative simulation time, real.
nday.f	q	Integer	Converts event time to the day portion; q = cumulative event time, real.

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
normal.f	q_1, q_2, q_3	Real	Returns a random sample from a normal distribution: q_1 = mean, real q_2 = standard deviation, real Returns a random sample q_3 = random number stream, integer
out.f	q	Alpha	Sets or returns the alpha value of the qth character in the current output buffer; q must yield an integer value; $q \geq 0$; both right- and left-hand function.
or.f	q_1, q_2	Integer	Returns the logical sum of q_1 and q_2.
poissson.f	q_1, q_2	Integer	Returns a random sample from a Poisson distribution: q_1 = mean, real q_2 = random number stream, integer
randi.f	q_1, q_2, q_3	Integer	Returns a random sample uniformly distributed between a range of values: q_1 = beginning value, integer q_2 = ending, value, integer q_3 = random number stream, integer
random.f	q	Real	Returns a pseudorandom number between zero and one; q = random number stream, integer.
real.f	q	Real	Converts an integer expression to a real value.

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
repeat.f	t, q	Text	Returns a text value which is the concatenation of q copies of t. t = text q = non-negative integer
sfield.f	None	Integer	Returns the starting column of the next data field to be read by a read (Free-Form) statement.
shl.f	q_1, q_2	Integer	Returns the value of q_1 shifted left q_2 bit positions.
shr.f	q_1, q_2	Integer	Returns the value of q_1 shifted right q_2 bit positions.
sign.f	q	Integer	Indicates the sign of a real expression: 1 if q > 0 0 if q = 0 -1 if q < 0
sin.f	q	Real	Computes the sine of a real expression given in radians.
sqrt.f	q	Real	Computes the square root of a real expression; a real expression; q ≥ 0
substr.f	t_1, t_2, t_3	Text	Sets or returns a substring of a text value; both left- and right-hand function; in the left-hand usage, t_1 must be an unmonitored variable: t_1 = string, text t_2 = position, integer t_3 = length, integer
tan.f	q	Real	Computes the tangent of a real expression given in radians.

Table 6. System Functions (Continued)

Function Mnemonic	Arguments	Function Mode	Description
triang.f	q_1, q_2, q_3, q_4	Double	Returns a random sample from a triangular distribution. q_1 = minimum, double q_2 = mean, double q_3 = maximum, double q_4 = random number stream, integer
trim.f	t, q	Text	Returns a copy of t which has leading and/or trailing blanks removed. If q < 0, leading blanks are removed. If q > 0, trailing blanks are removed. To remove both leading and trailing blanks, use q= 0. t = text q = integer
trunc.f	q	Integer	Returns the truncated integer value of a real expression.
ttoa.f	t	Alpha	Converts first characters of text expression to alpha, subject to the limit of the implementation.
uniform.f	q_1, q_2, q_3	Real	Returns a uniformly distributed random sample between a range of values: q_1 = beginning value, real q_2 = ending value, real q_3 = random number stream, integer.
upper.f	t	Text	Converts letters in a text string to upper case.
weekday.f	q	Integer	Converts event time to the weekday portion; q = cumulative event time, real.

Table 6. System Functions (Continued)

weibull.f	q_1, q_2, q_3	Real	Returns a sample value from a Weibull distribution: q_1 = shape parameter, real q_2 = scale parameter, real q_3 = random number stream, integer.
xor.f	q_1, q_2	Integer	Returns the logical difference of q_1 and q_2.
year.f	q	Integer	Converts simulation time to the year portion: q = cumulative simulation time, real.

Table 7. System Routines

Routine	Arguments	Description
date.r	Date Time	Returns the current date and time as text: Date: mm/dd/yyyy mm = month dd = day yyyy = year Time: hh:mm:ss hh = hour mm = minute ss = second
exit.r	q	Terminates program execution passing the exit status to the command level. q = integer
origin.r	m, d, y	Establishes an origin time when the calendar format is used: m = month, integer d = day, integer y = year, integer
snap.r	None	User-supplied routine called by SIMSCRIPT II.5 when an execution error is detected.

Table 8. System Variables

Variable	Description	Default Value
batchtrace.v	Variable that controls what happens when a run-time error occurs: 0 SimDebug is invoked to allow interactive debugging. 1 SimDebug is not invoked. SIMSCRIPT shows the run-time error in a message box and then writes the traceback (including global variables) to the file `simerr.trc` in the current directory. 2 SimDebug is not invoked and SIMSCRIPT does not write a traceback.	0
between.v	Subprogram variable called before each event is executed.	0
buffer.v	The length of the internal buffer.	132
eof.v	End-of-file code; zero denotes that an end-of-file marker is an error; `1` indicates return control with `eof.v` set to `2` when end-of-file is encountered; one for each input unit[3].	
event.v	Code representing the event class to occur next.	0
events.v	The number of event classes.	0
ev.s	Event set; dimension is contained in `events.v`.	0
f.ev.s	Array containing the first-in-set pointers for the event set, `ev.s` (note that `n.ev.s` is not defined).	0
heading.v	A subprogram variable tested by the system for each new page.	0
hours.v	Number of hours per simulated day.	24
line.v	Number of the current output line[3].	0
lines.v	Number of lines per page[3].	55
mark.v	Termination character required on external event records and on the input for random variables.	*
minutes.v	Number of minutes per simulated hour.	60
page.v	Number of the current page[1].	
pagecol.v	If 0, column number in which the word `page` and the value of `page.v` are to be printed on the output listing.	0

Table 8. System Variables (Continued)

Variable	Description	Default Value
parm.v	Contains command line arguments passed to the program when it was invoked.	
process.v	If not zero, a pointer to the process notice of the currently executing process.	0
rcolumn.v	Pointer to the last column read in the input buffer[3].	0
ropenerr.v	Indicates error occured on the current input unit.	0
read.v	Number of the current input unit.[1]	0
record.v	The number of records read from the current input unit, or written on the current output unit; one for each input and output unit.	0
rreclen.v	Contains the length of the input record (number of characters) of the current input unit.	0
rrecord.v	The number of records read from the current input unit.	0
seed.v	Array containing initial random numbers.[1]	0
time.v	Current simulated time.	0
wcolumn.v	The number of records written to the current output unit.	0[2]
wopenerr.v	Indicates an error occured on the current output unit.	0
wrecord.v	The number of records written in the current output buffer[3].	0
write.v	Number of the current output unit[1].	0

Notes:
1. Default differs for the various implementations of SIMSCRIPT II.5.
2. Some implementations set `rcolumn.v` to `-1` before the corresponding input unit is first used.
3. A separate value is maintained for each unit. Only the currently used value is accessible to the program.

1.9 Variables

A variable is a name representing a memory location whose value can change during the execution of a program. At the start of program execution, the values of all variables are automatically initialized to zero, except for text variables, which are initialized to the null string. The SIMSCRIPT II.5 system includes dummy, global, local, monitored and subprogram variables, each of which is described below:

1.9.1 Dummy Variables

Dummy variables, which can appear in conjunction with **`accumulate`** and **`tally`** statements, are artificial variables or attributes, in that their values are not accessible to the program. If a program does not require that the values of tallied or accumulated variables be accessible, **`accumulate`** and **`tally`** computations can be performed on them without having their values stored. In programs with many statistical variables, a significant amount of storage can be saved by using dummy variables. Dummy attributes are declared in **`every`** or **`the system`** statements, while dummy global variables are declared in **`define ... variable`** statements.

1.9.2 Global Variables

A global variable, which must be declared in a **`define ... variable`** statement in the preamble, has a common meaning throughout a program. Whenever the name of a global variable appears in a statement, the system references the same storage location regardless of the routine in which the variable appears. Names of global variables can be temporarily defined as local in subroutines by using their names in **`define ... variable`** statements within the respective routines.

Side Effects. Unwanted side effects can occur when global variables are used in routines that interact with other routines. For example, there may be unexpected consequences when using functions in expressions in **`for ... to (`*index*`)`** phrases that are evaluated before each iteration, and in complex logical expressions where all sub-expressions are not always evaluated. These side effects can frequently be eliminated by using local rather than global variables.

1.9.3 Local Variables

A local variable applies only to the routine in which the variable appears. Consequently, the same name can be used for local variables in different routines, with the name referring to a different value in each routine. A local variable occupies storage only during the time between entry and return from the routine in which the variable appears. A local variable occupies storage for the duration of the run.

SIMSCRIPT II.5 assumes that variables not defined in the preamble are local variables. **`Normally`** statements in routines can specify general characteristics of local variables, and **`define ... variable`** statements can declare any exceptions. Neither statement is

required in a routine if local variables are to have the characteristics of the last **normally** statement in the preamble.

The system automatically assigns storage locations to unsubscripted (non-text) local variables and initializes them to zero when a routine is called. The storage is returned to the system when control returns to the calling program.

Text variables require two storage areas. One, in a fixed location, is a pointer to the dynamic storage area that contains the actual characters represented. When a routine is called, unsubscripted text variables are automatically initialized to the null string. When control returns to the calling routine, the dynamic storage area associated with any unsubscripted text variable is released.

SIMSCRIPT II.5 does not automatically assign storage to subscripted local variables except for the base pointer. Arrays must appear in **reserve** statements before they can be used, and when an array is reserved, the elements are automatically initialized to zero. The storage associated with an array should be released before control is returned to the calling program.

A subscripted local variable that is not defined in a **define ... variable** statement within a routine has the dimensionality implied by its first use. For example, the statement **let x(i) = 0** declares that **x** is a one-dimensional array, regardless of background dimensionality. Of course, when the program is executed, a **reserve** statement incorporating the appropriate dimensionality must be processed before accessing an element of the array.

1.9.4 Monitored Variables

A monitored variable is a subscripted or unsubscripted variable, or an attribute whose values are checked or used by a monitoring routine. A monitored variable, therefore, has a storage location and a routine associated with it, both of which have the same name. Whenever the value of a monitored variable is accessed, the corresponding routine is automatically executed. A monitored variable must be declared in a **define ... variable** statement as being monitored on the left, on the right, or on both the left and the right. The words **left** and **right** refer to the left and right side of an equal sign in a **let** statement. **Tally** and **accumulate** statements automatically use the monitoring feature.

Monitoring Routines. If a variable is monitored on the right, a right-hand monitoring routine must be written and, if monitored on the left, a left-hand routine must be provided. Normally, the task of a right-hand routine is to return a value to the calling program. In monitoring operations, an **enter with** statement is used in a left-hand routine to transmit a value to the left-hand routine. Also pertaining to monitoring routines is the **move** statement, one form of which is used in a right-hand routine and another in a left-hand routine. Depending on the form used, the **move** statement transfers the value of a monitored variable to a named variable in order to use that variable in computations, or it assigns a value to the monitored variable. A monitoring routine cannot be called with a **call** statement. It is executed only when the statement containing the monitored variable is executed.

Subscripted Monitored Variables. Because a monitored variable represents both a storage location and a routine, a name such as **scan(i,j)** is both a subscripted variable and a call on a routine with arguments **i** and **j**. All data references are to **scan(i,j)** as if it were a typical subscripted variable. The monitoring routine must have as many arguments as there are subscripts. During execution, arguments are automatically converted to integer (like a subscripted variable), but transmitted to the monitoring routine (like a routine).

1.9.5 Subprogram Variables

A subprogram variable has the address of a routine as its value and enables that routine to be called indirectly. The statement:

```
let SUBR = 'sqrt.f'
```

where **SUBR** is a subprogram variable, stores the address of the library function **sqrt.f** in the variable **SUBR**. The subprogram variable can subsequently be used in a **call** statement. Subprogram variables can be global or local, saved or recursive, and subscripted or unsubscripted. They are initialized to zero when a program begins execution or when routines containing them (as recursive variables) are called. During compilation, the number for a routine called with a subprogram variable is not compared with the declaration in the **define ... routine** statement.

Subprogram Arrays. Subprogram arrays can be defined, and routine names can be stored in the elements, but they cannot be used to call routines. The programmer must distinguish between the direct and indirect use of a subprogram array. If **x** is defined as a subprogram array in a **define ... variable** statement, the statement **release x** releases the storage allocated to array **x**, but the statement **x(2)** releases the routine whose name is stored in **x(2)**. Thus, when reference is made to all elements of a subprogram array, the subprogram array itself is the object of a statement. When a particular element of a subprogram array appears, it is an indirect reference to a routine.

Calling Functions. Subprogram variables can call functions as follows.

The subprogram variable must be declared in a **define ... variable** statement. The mode of the variable is either explicitly defined or declared by the current background mode. All functions called with the subprogram variable must be of the mode declared for that variable.

An indirect function call is indicated by placing a dollar sign (**$**) before the subprogram variable. For example, if **a** has been declared as a subprogram variable, **let x = a** assigns the name of a routine to variable **x**, but **let x = $a** computes a value by executing the function whose name is stored in **a** and stores the computed value in **x**.

When arguments follow a subprogram variable, the arguments apply to the function called indirectly, not to the subprogram variable itself.

PART II.
LANGUAGE REFERENCE

2. Language Reference

2.1 ACCUMULATE/TALLY Statement

The **accumulate/tally** statement computes statistical quantities and prepares histograms for time-dependent variables. It specifies automatic data collection and analysis.

```
{accumulate} {      {as [the][name] stat kywd                                    }  }c
{  tally   } {name  {(quantity to quantity by quantity) as the [name] histogram  }  }
```

```
of {unsubscripted global variable  }
   {entity attribute               }
   {unsubscripted system attribute }
```

Keyword	Synonym
as	=

EXAMPLES:

`accumulate AVG.STAT as the mean of STATUS`

STATUS becomes a left-hand monitored global variable. Each time its value changes, the monitoring routine computes **STATUS * (time.v - A.1)** and stores it in **A.3**. **AVG.STAT** is a function that returns **(A.3 + STATUS * (time.v - A.1)) / (time.v - A.2)** at any time during the simulation. **A.1**, **A.2**, and **A.3** are generated system attributes representing T_L (the beginning of the observation period), T_0 (the time of the last change in the variable), and $\sum x_i * t_i$ (where **X** is a sample value of **STATUS**), respectively.

`accumulate NUM.CARS as the num and MAX.CARS as the max of NO.PARKED CARS`

Assuming **NO.PARKED.CARS** is an attribute of an entity, it becomes a left-hand monitored attribute. **NUM.CARS** and **MAX.CARS** become attributes of the entity class(es) associated with **NO.PARKED.CARS**. The monitoring routine will increment **NUM.CARS** each time the value of **NO.PARKED.CARS** changes. It also keeps track of the maximum value of **NO.PARKED.CARS** in **MAX.CARS** (no other attributes are generated). Of course, the number of samples and maximum of **NO.PARKED.CARS** is kept for each particular entity created in the class(es) for which **NO.PARKED.CARS** is an attribute.

`accumulate TOTAL.MAX as the max, MAX.CARS as the weekly max, and CAR.GRAPH (1000 to 10000 by 50) as the histogram of NO.PARKED.CARS`

Assuming **NO.PARKED.CARS** is an attribute of the permanent entity class **AIRPORT**, both **TOTAL.MAX** and **MAX.CARS** are attributes used to keep track of the maximum value of **no.parked.cars** for each entity created. By appropriate use of **reset** (for example,

for each AIRPORT, reset weekly totals of NO.PARKED.CARS(AIRPORT)) in an event that occurs each time a week of simulated time has elapsed, **MAX.CARS** at any time will hold the maximum value of **NO.PARKED.CARS** for that week, while the **TOTAL.MAX** attribute will hold the overall maximum for each entity. An array named **CAR.GRAPH** is reserved when the **AIRPORT** entities are created to hold histogram values. It will be two-dimensional, the first dimension being **N.AIRPORT** and the second (10000-1000) / 50 + 1 = 181. Each time the value of **NO.PARKED.CARS** changes, the monitoring routine adds the *amount of time* (**time.v - T_L**) that this attribute has held this value to the running sum in the appropriate element of the histogram array. For example, if the value of **NO.PARKED.CARS(2)** has been 2010 for three hours (**time.v - T_L = 3, time.v** in hours) and then changes to 2032, three will be added at the time to the value of **CAR.GRAPH**(2,21) by the left-hand monitoring routine.

`tally` AVERAGE.LIST `as the mean of` LIST

Assuming that **LIST** is a one-dimensional array, the system generates a left-hand monitoring routine named **LIST**, with the subscript as an argument. This routine is called whenever a value is assigned to **LIST**. The function counts the number of times **LIST** changes value and accumulates sum and number. The system also generates global arrays **A.1** and **A.2**, each having as many elements as **LIST**, in which to keep the sum and number for each element of **LIST** in order to compute the means. The system also generates a function named **AVERAGE.LIST** that computes a mean from **A.1** and **A.2** whenever the function is referenced in the program.

`tally` NUM.POP `as the num and` MAX.POP `as the max of` POPULATION

Assuming that **POPULATION** is an attribute of permanent entity class **CITY**, the system generates a left-hand monitoring routine named **POPULATION** with the index number of the entity as an argument. This routine uses the **NUM.POP** attribute of the entity **CITY** to tally the number of changes to **POPULATION** for each entity. **MAX.POP** is an attribute which contains the maximum population assigned to each city as computed in the monitoring routine.

`tally` VAR.POP `as the weekly variance,` TOTAL.VAR `as the cumulative variance and` GRAPH(0 TO 1000 BY 100) `as the histogram of` POPULATION

Assuming that **POPULATION** is an attribute of permanent entity class **CITY**, the system generates a left-hand monitoring routine named **POPULATION** with the index number of the entity as an argument. The system also generates attributes **A.1**, **A.2**, **A.3**, **A.4**, **A.5**, and **A.6**, each having **N.CITY** elements, to accumulate the number, sum, and sum of squares for each entity for both the **weekly** and **cumulative** variances. The system also generates function variances named **VAR.POP** and **TOTAL.VAR** which compute the variance whenever the function is referenced. **VAR.POP** uses attributes **A.1**, **A.2**, and **A.3**, while **TOTAL.VAR** uses attributes **A.4**, **A.5**, and **A.6**. The system also defines a two-dimensional array named **GRAPH** to hold histogram values for a range of values from `0` to `1000`, in intervals of `100`, indicating the number of times

sample values fall within each interval. The array will be reserved when the **CITY** entities are created and will be dimensioned **N.CITY** by **(1000-0) / 100 + 1**.

The **accumulate** statement specifies data collection variables, routines to compute statistical quantities, and arrays to collect histograms for time-dependent variables in a global environment. The statistical keywords used to specify the desired statistics appear in table 9. SIMSCRIPT II.5 permits **accumulate** statements for each preamble-defined global variable or attribute, (called the *accumulated variable* in this description). Whenever the value of the accumulated variable changes, computations are made to compile the statistics required in the **accumulate** statement(s). These computations include time, i.e., the collected observations are weighted by the length of time they have retained their values. The results are used in time-series analyses.

The **accumulate** statement can appear only in the preamble. Because the principal results from simulation models are statistical measurements, this statement relieves the programmer of the necessity of writing the required routines. **Tally** and **accumulate** statements cannot be declared for the same variable.

For each **accumulate** statement, the system generates attributes and routines. A left-hand routine that accumulates data is always generated for each accumulated variable. This monitoring routine has the same name as the accumulated variable. Generated global variables and attributes are named **A.1, A.2**, ..., **A.N**, and the number of other routines and attributes generated depends on the statistics requested (see table 10). Counters are assigned as follows:

If the accumulated variable is a global variable, system attribute, or attribute of a permanent entity, the system automatically reserves as many variables for statistical counters as there are elements of the accumulated variable. For example, a global variable will only require single counters, while a permanent entity will use arrays of counters.

If the accumulated variable is an attribute of a temporary entity, each entry record is assigned statistical accumulation attributes.

Table 9. Statistical Keywords for Accumulate/Tally Statement

Statistic	Accumulate Computation	Tally Computation
NUMBER	N = The number of changes in X	N = The number of samples of X
SUM	Σ (X(time.v - T_L))	Σ X
MEAN	$\frac{\text{Sum}}{(\text{time.v} - T_o)}$	$\frac{\text{Sum}}{N}$
SUM.OF.SQUARES	Σ (X^2(time.v - T_L))	Σ X^2
MEAN.SQUARE	$\frac{\text{Sum.of.squares}}{(\text{time.v} - T_o)}$	$\frac{\text{Sum.of.squares}}{N}$
VARIANCE	Mean.square - Mean^2	Mean.square - Mean^2
STD.DEV	$\sqrt{\text{VARIANCE}}$	$\sqrt{\text{VARIANCE}}$
MAXIMUM	M = Maximum (X) for all X	M = Maximum (X) for all X
MINIMUM	m = Minimum (X) for all X	m = Minimum (X) for all X

Notes:

time.v = current simulated time.

T_L = simulated time at which variable was set to its current value.

T_o = simulated time of last reset for this variable.

X = sample value of variable before change occurs.

Table 10. Attributes & Functions for Accumulate/Tally Statement

Statistical Keyword	System Action
NUMBER	The named data collection variable is an attribute of the observed variable.
SUM[12]	A function having the same name as the named data collection variable is generated. An attribute with an arbitrary name is also generated.
MEAN[1]	A function having the same name as the named data collection variable is generated. A **SUM** attribute is generated whether or not requested.
SUM. OF SQUARES[1]	A function having the same name as the named data collection variable is generated. An attribute with an arbitrary name is also generated.
MEAN.SQUARE	A function having the same name as the named data collection variable is generated. An **SSQ** attribute is generated whether requested or not.
VARIANCE[1]	A function having the same name as the named data collection variable is generated. **SSQ** and **SUM** attributes are generated whether requested or not.
STD.DEV[1]	A function having the same name as the named data collection variable is generated. **SSQ** and **SUM** attributes are generated whether requested or not.
MAXIMUM	The named data collection variable is an attribute of the observed variable. A **NUM** attribute is generated whether requested or not.
MIMIMUM	The named data collection variable is an attribute of the observed variable. A **NUM** attribute is generated whether requested or not.

Notes:

1. For **accumulate** statements, attributes are generated for T_L and T_0. These values are not directly available to the user, as they are arbitrarily named, e.g., **A.1** and **A.2**.
2. For **tally** statements, the attribute has the same name as the named data collection variable and no function is generated.

A programmer-defined qualifying name may optionally be supplied to specify different types of a particular statistic. These qualifiers allow some statistics to be reset while others are not. Using a qualifier is a handy way to keep both periodic and cumulative statistics on a variable or attribute. See **reset** statement.

2.1.1 Histograms

The **accumulate** statement is used to collect data on the total time a variable has held a value within a particular range during the simulation. Histograms can be requested for global variables, system attributes, and attributes of permanent entities, but they cannot be requested for attributes of temporary entities. The system automatically generates a histogram array of one more dimension than that of the accumulated variable.

When requesting histograms, the phrase:

(*quantity*$_1$ to *quantity*$_2$ by *quantity*$_3$)

defines a range of values from ***quantity***$_1$ to ***quantity***$_2$, which is divided into intervals of ***quantity***$_3$ units. SIMSCRIPT II.5 reserves the histogram array with **(*quantity*$_2$ - *quantity*$_1$)/*quantity*$_3$ + 1** elements for each element of the accumulated variable. Each time the value of the accumulated variable changes, the length of time the variable retained the value is added to the pertinent element of the histogram array. If a value is less than ***quantity***$_1$, the time for that value is added to the first element. If a value is greater than or equal to ***quantity***$_2$, it is added to the last element.

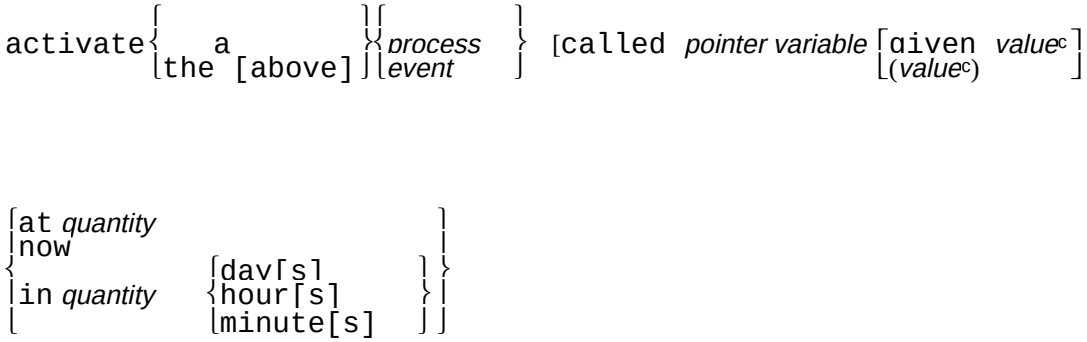
Qualifiers can be used with histograms to identify particular arrays for selective **reset**, as with other accumulated statistics.

2.1.2 Dummy Variables

A dummy variable is a preamble-defined variable or attribute whose value is not accessible to the program. If a program does not require that the value of the tallied variable be accessible, **accumulate** computations can be performed on the variable without having its value stored. Only **number, maximum**, and **minimum** statistics can be collected on dummy variables in an **accumulate** statement. Dummy attributes are declared in **every** or **the system** statements, while dummy global variables are declared in **define ... variable** statements.

2.2 ACTIVATE (*process*) Statement

The **`activate`** statement activates the future occurrence of a process by filing a process notice in the relevant process set.



Keywords	Synonyms
`activate`	`reactivate`
`a`	`an` (For a new process)
`the [above]`	`this` (For an existing process)
`given`	`giving`
`now`	`next`
`in`	`after`
`day[s]`	`unit[s]`

EXAMPLES:

`activate a CUSTOMER now`

Creates a process notice of the class customer and schedules its entry as soon as control is returned to the timing routine. A pointer to this process notice is placed in the variable **`CUSTOMER`**.

`reactivate this SHIP in UNLOADING.TIME(SHIP) hours`

The process **`SHIP`** is to be entered when **`time.v`** has been updated to **`time.a(SHIP)`**, which is **`UNLOADING.TIME(SHIP)/hours.v`** units of simulated time beyond the **`time.v`** at reactivation. The process notice already exists, and a pointer to it is located in variable **`SHIP`**. The **`reactivate`** statement keyword is used by the modeller to document the fact that the process notice already exists.

```
activate a STORM giving LOCATION and DURATION in 2 days
```

Creates a process notice of the class **STORM** which has two user-defined attributes whose values are to be set to the values of **LOCATION** and **DURATION**. The process is scheduled to begin when **time.v** has been updated to **time.a(STORM)**, which is two days beyond the **time.v** at activation.

```
activate a CAR called NEXT.CAR in 5 minutes
```

Creates a process notice of the class **CAR**, but puts the pointer to it in the variable **NEXT.CAR** rather than in **CAR**. The process is scheduled to begin when **time.v** has been updated to **time.a(NEXT.CAR)**, which is 5 minutes beyond the **time.v** at activation.

The **activate** statement schedules the future occurrence of a process and is analogous to the use of the **schedule** statement with events. This statement assigns values to process notice attributes and files the notice in the event set, **ev.s**. It can appear in any routine, but not in the preamble.

2.2.1 CALLED Phrase

If the **the** keyword signifies that the process notice already exists, SIMSCRIPT II.5 uses the variable name in the **called** phrase to locate that process notice. If the the keyword is not used, the variable is assigned a pointer to the process notice created.

2.2.2 GIVEN Phrase

A **given** phrase assigns values to attributes of the process notice, which causes them to be passed as arguments to the process routine. The **activate** statement assigns values of expressions to successive attributes of the process notice, starting with the first programmer-defined attribute declared in the **every** statement. (Use of **in word** phrases to arrange physical storage of attribute values does not influence the order of selection of process arguments.) If there are fewer expressions than attributes, SIMSCRIPT II.5 assigns zeros to the remaining attributes. The keyword **given** can be omitted if expressions are enclosed in parentheses.

2.2.3 AT Phrase

The **at** phrase, which denotes when in the future the named process is to occur, sets attribute **time.a** to the value of an expression. The expression must yield a real value. **time.a** is used to file this process notice in the event set in chronological order (the event set is ranked on low **time.a**).

The value of **time.a** is used to update **time.v**, the current simulation time, whenever a process notice or an event notice is selected from the event set by setting it to the value of of the first process or event in the event set. The system sets **time.v** to zero at the start of

simulation and increases **time.v** as the simulation progresses. An absolute time must always be specified in an **at** phrase.

2.2.4 IN Phrase

An **in** phrase specifies the relative time at which a process is to occur. This phrase enables the programmer to schedule a process at **time.v** plus a designated number of days, hours, or minutes. (The keyword **units** can be used in place of **days**.) The units of **time.a** are days if the **days, hours**, or **minutes** keywords are used in this phrase. If **hours** or **minutes** is specified, SIMSCRIPT II.5 automatically converts the hours or minutes to days using the system variables **hours.v** and **minutes.v**. The system initializes **hours.v** to 24 and **minutes.v** to 60, but the programmer can modify these values. If **time.v** runs in units other than days, the **units** keyword should be used in an **in** phrase.

2.2.5 NOW Phrase

A process scheduled with a **now** phrase occurs as soon as the current process or event returns control to the timing routine. Such a process will occur before any processes or events having the same event time, scheduled previously with **at** or **in** phrases. When **activate** statements include **now** phrases for two or more processes, the processes are ranked according to the **priority** statement if they are of different classes, according to the **break ... ties** statement if that statement appears for the process class, or first-in, first-out if a **break ... ties** statement has not been included for the process class.

2.3 ADD Statement

The **add** statement adds the value of an arithmetic expression to the value of a variable, and the sum becomes the new value of the variable.

`add` *quantity* `to` *variable*

EXAMPLES:

```
add 1 to X
```

Adds 1 to the value of variable **X**.

```
add A * X**2 + B * X + C to Y ''quadratic formula
```

Adds $\mathbf{AX^2 + BX + C}$ `TO` the value of variable **Y**. Commentary text follows the two apostrophe characters.

```
add BAGGAGE.WEIGHT(FLIGHT) to TOTAL.WEIGHT(FLIGHT)
```

Adds the value of attribute **BAGGAGE.WEIGHT** to the value of **TOTAL.WEIGHT** for the entity whose identification number is in the variable.

The **add** statement adds the value of an expression to the value of a variable. An **add** statement can appear in any routine, but not in the preamble. If the expression and the variable differ in mode, SIMSCRIPT II.5 converts the expression to the mode of the variable before assigning the sum to the variable (see the **let** statement for conversion rules).

2.3.1 Complex Subscripted Variables

Before compilation, the **add** statement is translated to:

`let` *variable* = *variable* + *quantity*

If the variable has complex subscript references, it is more efficient to compute the subscripts separately than to have the compiler compute them twice. For example:

```
add 1 to x(y*(ab-2),diff**n)
```

translates to:

```
let x(y*(ab-2),diff**n) = x(y*(ab-2),diff**n) + 1
```

which causes the subscripts **y*(ab-2)** and **diff**n** to be evaluated twice. To conserve storage space and computer time, this **add** statement could be written as:

```
let i = y*(ab-2)
```

```
let j = diff**n
add 1 to x(i,j)
```

2.3.2 Subscripts Containing Functions

If the subscript of the variable is a function, or contains a function, unexpected results can occur when the function has side effects. For example, note what happens because of the following calls to the random number generator.

```
add 1 to TABLE(uniform.f(a,b,1))
```

translates to:

```
let TABLE(uniform.f(a,b,1)) = TABLE(uniform.f(a,b,1)) + 1
```

before compilation. This statement causes two random numbers to be generated, and possibly two different elements of **TABLE** to be accessed. The programmer may have preferred:

```
let I = uniform.f(a,b,1)
add 1 to TABLE(i)
```

2.3.3 Error Messages

An **add** statement having complex subscripted variables or function references can cause duplicate error messages to be produced because of intermediate translations.

2.4 AFTER Statement

See **before/after** statement.

2.5 ALSO Phrase

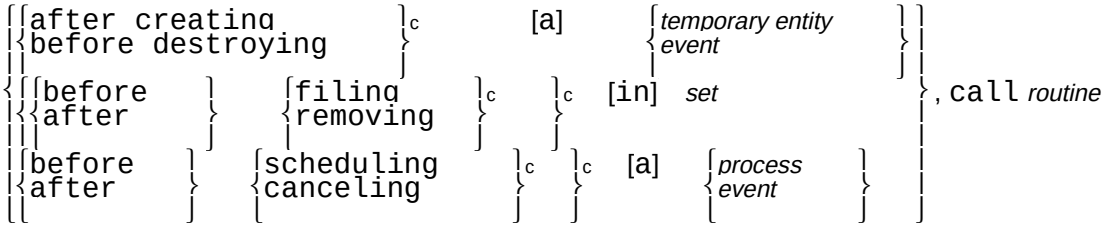
See **do ... loop** construct, **for each (*class*)** phrase, **for ... of (*set*)** phrase, **for ... to (*index*)** phrase, **until** phrase, or **while** phrase.

2.6 ALWAYS Statement

See **if ... else ... always** construct.

2.7 BEFORE/AFTER Statement

The **`before/after`** statement is a debugging statement that can appear only in the preamble. It can trace six types of SIMSCRIPT II.5 statements.



Keywords	Synonyms
a	an
	the
	any
scheduling	activating
	causing
canceling	interrupting
call	perform
	now

EXAMPLES:

`before destroying a FLIGHT, call PLANE.CHECK`

Calls subroutine **`PLANE.CHECK`** each time the **`destroy`** statement is executed for an entity of the entity class **`FLIGHT`**. The identification number of the entity is automatically passed as an argument to the subroutine.

`before filing and removing from RESERVATIONS, call CHECK.RESERVATIONS`

Calls subroutine **`CHECK.RESERVATIONS`** each time the **`file`** and **`remove`** statements are executed for the set class **`RESERVATIONS`**. The identification number of the entity being filed or removed automatically becomes an argument of the subroutine.

`after scheduling a TAKEOFF, call TEST`

Calls subroutine **`TEST`** each time the **`schedule`** statement is executed for an event of the class **`TAKEOFF`**. The identification number of the event being scheduled and the time of occurrence automatically become arguments of the subroutine. The **`routine`** statement could appear as **`routine TEST given NUMBER and TIME`**.

The purpose of the **before/after** statement is to monitor the **create, destroy, file, remove, schedule, activate**, and **cancel** statements. The named subroutine is called whenever one of these statements is executed if a **before/after** statement is included in the preamble for the specified entity class, set class, and event class. SIMSCRIPT II.5 permits one **before/after** statement for each temporary entity class, set class, and event class. To use the **before/after** statement, write a routine with the name that appears in the statement and the number of arguments automatically passed for the statement being monitored. Although the **call** phrase does not show the arguments, the **routine** statement must have the correct number of arguments. Using the arguments, the routine can trace entity identifications, subscripts, or event times.

Table 11 lists the arguments automatically given to the named subroutine. All arguments are integer except the event time, which is real.

Table 11. Arguments Automatically Given to the Subroutine

Statement Form	Arguments
creating an entity	Identification number of the entity. **Before** cannot be used with this form.
destroying an entity	Identification number of the entity. **After** cannot be used with this form.
filing in a set	Identification number or index of the entity to be filed and the set subscripts. In **file ... before** and **file ... after** statements, the identification of the second entity is not passed as an argument.
removing from a set	Identification number or index of the entity to be removed and the set subscripts, if any. The entity identification for the **remove first** and **last** statements is passed as zero.
scheduling an event	Identification number of the event or process notice and the time the event is to occur.
canceling an event	Identification number of the event or process notice.

2.8 BEGIN HEADING Statement

The **begin heading** statement marks the beginning of a heading section within a report section.

```
begin heading
```

Only one form of this statement exists.

The **begin heading** statement can appear only within a report section. A heading section is used, for example, to print titles, column headings, and computational results whenever a page ejects. As shown in figure 1, a heading section starts with a **begin heading** statement and ends with an **end** statement. SIMSCRIPT II.5 executes the program segment included between these two statements the first time the program segment is encountered and whenever a page is ejected by an output statement within the report section. The output statement, such as a **print** statement, must appear after the heading section, but within the enclosing report section. A **begin heading** statement cannot appear in the preamble.

Within a heading section, the logical expressions:

```
page is first
page is not first
```

can be used in an **if ... else ... always** construct to select statements to be executed for the first page of output but not for succeeding pages, or vice versa.

Pages are ejected whenever the current line count (**line.v**) exceeds the maximum number of lines to be printed per page (**lines.v**).

2.8.1 System Variables

See table 12 for a list of global variables frequently used in heading and report sections.

```
begin report
      begin heading
            Program segment executed the first
            time encountered and whenever a
            page is ejected
      end '' heading section
      for i = 1 to 60, print ...
            Page ejects here; heading program
            segment executed
end  '' report section
```

Figure 1. Heading Section Within a Report Section

Table 12. Line and Page System Variables

Name	Value	Description
line.v	Current line number	Refers to the current output device; automatically set to 1 when the device is first used and automatically incremented by 1 whenever a line is printed. The maximum value of **line.v** is **lines.v**. Reset to 1 on page ejection. Separate value maintained for each output device.
lines.v	Maximum lines per page	Automatically set to 55 when a program begins execution, but can be modified within the program. Separate value maintained for each output device.
page.v	Current page number	Refers to the current output device. Can be reset within the program. Numbering then continues sequentially, beginning with the new value. Separate value maintained for each output device.
pagecol.v	Report column number	A single integer variable used to number the printed pages. If **pagecol.v** $\neq$ **0**, **PAGE xxx** (where **xxx** = value of **page.v**) is printed at the top of each new page. The word **PAGE** begins in the column denoted by the value of **pagecol.v**.
heading.v	Name of a routine	A single subprogram variable tested by the system for each new page. If **heading.v** $\neq$ **0**, the system executes the routine whose name is stored in **heading.v**.

2.9 BEGIN REPORT Statement

The **`begin report`** statement marks the beginning of a report section with optional new page and column repetition features.

`begin report [on a new page] [printing` *for phrase*, `in groups of` *integer value* [`per page`]]

EXAMPLES:

`begin report`

Marks the beginning of a report section.

`begin report on a new page`

Marks the beginning of a report section and indicates that a page should be ejected, if necessary.

`begin report on a new page printing for J = 1 to 20 in groups of 8 per page`

Marks the beginning of a report section using column repetition. Groups of eight values of index variable **`J`** are to be used by a **`print`** statement as eight-column indices for each execution of the report section. Report begins a new page, and each group of column indices applies to a new page.

The **`begin report`** statement, which marks the beginning of a report section, can include a phrase that begins each report on a new page, as well as two phrases used in conjunction with a **`print`** statement for column repetition. As shown in figure 1 above, a report section begins with a **`begin report`** statement and ends with an **`end`** statement. Within a report section, **`print`** statements can specify the content and format of the report, and a heading section (denoted by a **`begin heading`** statement) can include titles and column headings to be printed on each page of the report. The **`begin report`** statement cannot appear in the preamble.

2.9.1 ON A NEW PAGE Phrase

Each report can start on a new page by including the optional phrase **`on a new page`**. This phrase ejects a page on the output device unless the current page is blank. SIMSCRIPT II.5 determines whether or not the current page contains information by testing the system variables **`line.v`** (current line number) and **`wcolumn.v`** (column number of the current output pointer); **`line.v = 1`** and **`wcolumn.v = 0`**, if the current page is blank.

2.9.2 PRINTING Phrase

The **printing** phrase in a **begin report** statement is used with the **in groups of** phrase in a **print** statement for column repetition. Together, these phrases can print reports with more columns of data than can be fitted across a single page. After printing, individual pages produced by the **print** statement can be placed side-by-side to obtain a wide page. For example, the statement:

```
begin report printing for j = 1 to 20 in groups of 8 per page
```

used with the pertinent **print** statement will print 20 columns of data, with eight columns on each page. Values generated by the index variable **j** in this **for ... to (*index*)** phrase are considered to be column indices. That is, each value of the index variable corresponds to a column in a printed report. This statement causes the report section to be executed three times: first with **j = 1, ..., 8**; second with **j = 9, ..., 16**; and third with **j = 17, 18, 19, 20**. The column indices are used by the report section in groups. The **print** statement uses one group at a time. As shown in the above **begin report** statement, the number of column indices generated need not be an even multiple of the group size. If the controlling **for** phrase produces no values (for example, none have been selected by a **with** phrase), the entire report section is not executed.

2.9.3 PER PAGE Phrase

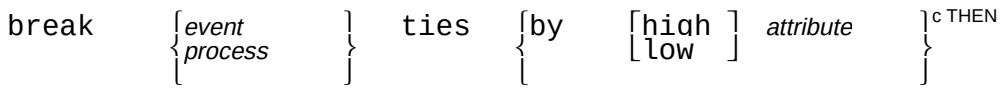
The **per page** phrase denotes that each group of column indices applies to a new page. If this phrase is omitted, printing continues on the current page until that page is completed.

2.9.4 System Variables

Table 12 lists global variables frequently used in heading and report sections.

2.10 BREAK ... TIES Statement

The **`break ... ties`** statement establishes the priority order within a process or event class in case processes or events have the same event time.



Keyword	Synonym
by	on

EXAMPLES:

```
break SEAT.RESERVE ties by high PRIVILEGE.CODE
```

When two or more event notices of event class **`SEAT.RESERVE`** have the same event time, priority is given to the event having the highest value of **`PRIVILEGE.CODE`**.

```
break SEAT.RESERVE ties
          by high PRIVILEGE.CODE, then
          by high FARE, then
          by low CLUB.NUMBER
```

When two or more event notices of event class **`SEAT.RESERVE`** have the same event time, priority is given to the event having the highest value of **`PRIVILEGE.CODE`**; but if several event notices have identical event times, and identical highest values of **`PRIVILEGE.CODE`**, priority is given to the event having the highest value of **`FARE`**. If several event notices have identical event times, identical highest values of **`PRIVILEGE.CODE`**, and identical highest values of **`FARE`**, priority is given to the event having the lowest value of **`CLUB.NUMBER`**.

The **`break ... ties`** statement establishes priorities within an internal event class. When two or more event notices of the same class have the same event time, the **`break ... ties`** statement declares that the event having the highest (lowest) value of the named attribute should occur first. Only internally-generated event notices have ranking attributes, while externally-generated events compete with them on a first-come, first-served basis. Attributes named in the **`break ... ties`** statement must be defined in `every` statements. SIMSCRIPT II.5 permits one **`break ... ties`** statement for each internal event class. The **`break ... ties`** statement can appear only in the preamble. See **`priority`** statement for resolving ties among events of different classes.

2.10.1 THEN BY Phrases

`Then by` phrases can include additional attributes whose values determine priorities when ties also exist in values of the first attribute named in the statement. For example, naming the first attribute declares that when ties occur in event times, the ties should be broken by giving priority to the event having the highest (lowest) value of the named attribute. If there are ties in event times and ties in values of the first attribute, the ties are broken according to values of the second attribute named. Any number of **`then by`** phrases can be included to break successive ties that can exist.

2.10.2 Order of Executing Events at the Same Simulated Time

If several events have been scheduled at the same simulated time, the events are executed in an order corresponding to (1) the **`priority`** statement if they are of different classes, (2) the **`break ... ties`** statement if it has been included, or (3) a first-in, first-out basis if a **`break ... ties`** statement has not been included for the event class.

2.11 CALL Statement

The **call** statement calls a subroutine and can include input and output argument lists.

```
call routine    [given value[c]  ][yielding variable[c]]
                 (value[c])      
```

Keywords	Synonyms
call	perform
	now
given	giving
	the
	this

EXAMPLES:

```
call PROCESS
```

Calls the routine named **PROCESS**.

```
call PRINT.MESSAGE(NUMBER)
```

Calls the routine named **PRINT.MESSAGE**. The variable **NUMBER**, enclosed in parentheses, is an input argument.

```
call PRINT.MESSAGE given MSG.NO(I), 4 * NO.WORDS yielding ERROR.FLAG
```

Calls the routine named **PRINT.MESSAGE**. The variables **MSG.NO(I)** and **4 * NO.WORDS** are input arguments, and **ERROR.FLAG** is an output argument.

The **call** statement calls a subroutine named in a **routine** statement. The **call** statement can include input arguments, output arguments, or both. An input argument is an expression whose value will be transmitted to the corresponding local variable within the routine. An output argument, however, must be a variable, either subscripted or unsubscripted. The **call** statement can appear in any routine, but not in the preamble.

The same variable name can appear as both an input and an output argument. When this is done, the value of the input argument is transmitted to the corresponding local variable within the called routine, the computation is performed, and a new value is assigned to the output argument before control is returned to the calling routine.

2.11.1 Argument Modes

Disagreements in mode between arguments in **call** statements and corresponding arguments in **routine** statements can be difficult to discover because the effects are subtle. For example, an integer number used as a real argument (assumed to be floating point with

an exponent) can effectively be zero. One must be particularly careful to pass constants in the mode expected by the routine.

2.11.2 Argument Definitions

A **`define ... routine`** statement in the preamble can specify the number of arguments for a subroutine. If the number of arguments in the **`call`** and **`define ... routine`** statements disagree, SIMSCRIPT II.5 takes the following corrective action:

1. Disregards additional input and output arguments in the **`call`** statement.
2. Considers omitted input arguments to be zero.
3. Reserves locations for missing output arguments so they can receive output values (although these will be inaccessible to the calling program).
4. Emits a warning message.

If the **`define ... routine`** statement contains only given arguments, the called routine is assumed to yield no values. If the **`define ... routine`** statement contains only yielding arguments, the called routine is assumed to have no given values.

2.12 CANCEL Statement

The **cancel** statement removes a scheduled event notice from the event set.

```
cancel [the [above]] { event } [called pointer variable]
                     { process }
```

Keywords	Synonym
the [above]	this

EXAMPLES:

```
cancel TAKEOFF
```

Cancels an event notice for the event class **TAKEOFF**. The identification number of the event notice cancelled is that assigned to the global variable named **TAKEOFF**.

```
cancel this DELAY called FIXED
```

Cancels an event notice for the event class **DELAY**. The identification number of the event notice is that assigned to the variable named **FIXED**.

The **cancel** statement removes an event notice from the event set. This statement performs the opposite task of the **schedule** statement, which schedules an event by filing the event notice in the event set. The cancelled event notice is not automatically destroyed, but can be destroyed with a **destroy** statement. If an event notice that has not been scheduled is cancelled, SIMSCRIPT II.5 terminates the program with an error message. The **cancel** statement is used only for simulation and cannot appear in the preamble.

When the **called** phrase is omitted, SIMSCRIPT II.5 cancels the event notice pointed to by the global variable having the same name as the event class. If the **called** phrase is included, however, the named variable is assumed to contain the identification number of an event notice of the same class.

2.13 CAUSE Statement

See **schedule** statement.

2.14 CLOSE Statement

The **close** statement closes a **UNIT** previously opened with an **open** statement. The file is closed and internal buffers are flushed to the disk.

```
close [unit] UNIT
```

EXAMPLE

```
close unit .FEX.UNIT
```

2.15 COMPUTE Statement

The **compute** statement calculates requested statistics for an expression and assigns each statistical value to a named variable. This statement must be controlled by a logical control phrase. It computes the indicated statistics of the expression after the **loop** statement if the control is over a **do ... loop** construct.

compute {*name* as the *stat kywd*}^{c} of *variable*

Keyword	Synonym
as	=

EXAMPLES:

```
for I = 1 to 10, compute TOTAL = SUM of X(I)
```

Adds the values in array **X** and stores the sum in variable **TOTAL**.

```
for J = 1 to N, do
    COMPUTE VR = variance, SD as STD.DEV

loop
```

Computes the variance, standard deviation, and number of values in the one-dimensional array **LIST**, and stores these statistics in variables **VR**, **SD**, and **NO**, respectively.

```
for each FLIGHT of DEPARTURES, with DEPARTURE.TIME ls 1200,
compute FMAX as maximum, FMIN as minimum, IMAX as
max(FLIGHT), IMIN as min(FLIGHT) of NO.PASSENGERS(FLIGHT)
```

Selects, from the set named **DEPARTURES**, entities with values of **DEPARTURE.TIME** that are less than 1200, and stores the maximum value of **NO.PASSENGERS** in **FMAX**, the minimum value of **NO.PASSENGERS** in **FMIN**, the value of **FLIGHT** for the entity having the maximum value in **IMAX**, and the value of **FLIGHT** for the entity having the minimum value in **IMIN**.

The **compute** statement calculates requested statistics (described in table 13) on an expression or from values stored in arrays. This statement must be controlled by at least one **for each (class), for ... of (set), or for ... to (*index*)** phrase, or an **until** or **while** termination phrase, any of which can have appended phrases. Termination and selection phrases can terminate the iteration or can select specific values, according to logical expressions in the phrases. The **compute** statement can appear in any routine, but it cannot be included in the preamble.

When a **compute** statement is included with other statements in a **do ... loop** construct, SIMSCRIPT II.5 computes the requested statistics when the first **loop** statement is encountered. Any variables included in the **compute** statement are set to statistical values

after the iterations. The statistics are undefined, however, if control transfers out of the **do ... loop** construct.

Table 13. Statistical Keywords for Compute Statement

Statistical Keyword	Synonym	Computation	Definition
NUMBER	NUM		Number of values selected.
SUM		$\sum$ *expression*	Sum of the selected values of the expression.
MEAN	AVG AVERAGE	SUM/NUMBER	Sum of the selected values of the expression divided by the number of values selected.
SUM.OF.SQUARES	SSQ	$\sum$ *expression*^{2}	Sum of the squares of selected values of the expression.
MEAN.SQUARE	MSQ	SUM.OF.SQUARES/ NUMBER	Sum of the squares of selected values of the expression divided by the number of values selected.
VARIANCE	VAR	MEAN.SQUARE - (MEAN)^2	
STD.DEV	STD	SQRT.F (VARIANCE)	Square root of the variance.
MAXIMUM	MAX		Maximum value of the selected values of the expression.
MIMIMUM	MIN		Minimum value of the selected values of the expression.
MAXIMUM (index)	MAX(*index*)		Value of the index variable that produced the maximum value.
MINIMUM (index)	MIN(*index*)		Value of the index variable that produced the minimum value.

2.16 CREATE Statement

The **`create`** statement allocates a block of storage for one instance of the attributes and set pointers of the specified temporary entity class (or event or process notice), and assigns (to the named variable) a pointer to that block.

`create` [a] { *temporary entity* / *process* / *event* } [`called` *pointer variable*]

Keyword	Synonyms
a	an
	the
	this

EXAMPLES:

`create FLIGHT`

Allocates storage for an entity of the temporary entity class **`FLIGHT`**. The pointer to this block is assigned to the global variable named **`FLIGHT`**. All attributes of this **`FLIGHT`** entity are set to zero.

`create a FLIGHT called JET(I)`

Allocates storage for an entity of the temporary entity class **`FLIGHT`**. The pointer to this block is the I^{th} value of array **`JET`**. All attributes of **`JET(I)`** are set to zero.

`create a STOCK.CERTIFICATE called STOCK`

Allocates storage for an entity of the temporary entity class **`STOCK.CERTIFICATE`**. The pointer to this block is assigned to the variable named **`STOCK`**, and attribute values are set to zero.

`create TAKEOFF`

Allocates storage for an event notice of the event class **`TAKEOFF`**. The pointer to this block is assigned to the global variable named **`TAKEOFF`**, and each word in the block is set to zero.

SIMSCRIPT II.5 allocates storage for each temporary entity as it is created during program execution. The **`create`** statement locates a contiguous block of storage words for use as a temporary entity, or for an event notice, and provides a pointer to these words. This pointer, also called the *entity identification number*, is assigned to a variable. The value of each attribute of this new entity is set to zero by setting each storage word to zero. A temporary entity may have more attributes than the number of computer storage words used if attribute packing or equivalencing is performed.

When the **called** phrase is omitted, the entity identification is assigned to the global variable having the same name as the entity class. If the **called** phrase is included, however, the entity identification will be assigned to the variable named in the phrase.

A word block allocated by a **create** statement can be returned to the system with a **destroy** statement. See the *User's Manual* for the specific storage allocation algorithm. The **create** statement can appear in any routine, but it cannot be included in the preamble.

2.17 CREATE EACH Statement

The **`create each`** statement allocates arrays for the attributes of the permanent entity or resource classes named in the statement.

Keyword	Synonyms
`each`	`a`
`a`	`every`
	`all`

EXAMPLES:

`create each AIRPORT`

Allocates arrays for the attributes of the permanent entity class **`AIRPORT`**. Each array will have **`N.AIRPORT`** elements.

`create every CITY, AIRPORT and RUNWAY`

Allocates arrays for the attributes of the permanent entity classes **`CITY`**, **`AIRPORT`**, and **`RUNWAY`**, with dimensions **`N.CITY`**, **`N.AIRPORT`**, and **`N.RUNWAY`**, respectively.

`create every AIRLINE(5) and STOCKHOLDER(4)`

Allocates arrays for the attributes of the permanent entity classes **`AIRLINE`** and **`STOCKHOLDER`**. Each **`AIRLINE`** array will have 5 elements and each **`STOCKHOLDER`** array will have 4 elements. **`N.AIRLINE`** is set to 5 and **`N.STOCKHOLDER`** is set to 4.

Attributes of permanent entities are stored in arrays. Resources are a special case of permanent entities. The **`create each`** statement allocates storage for the attributes of permanent entities. During program execution, the arrays are reserved together, and the attribute values stored in the arrays are set to zero. Arrays for several entity classes can be created by including several names in this statement. The **`create each`** statement can appear in any routine, but not in the preamble.

The **`create each`** statement must be used before any attributes of a permanent entity or resource are referenced, either explicitly or implicitly. For resources, the **`create each`** statement must be used and the **`U.resource`** attribute set to a nonzero value before any resource units may be requested.

If the value of **`N.`*`entity`*** (number of entities in the entity or resource class) has not been specified previously, an arithmetic expression can be included in the **`create each`** statement to specify attribute arrays. The value of this expression automatically becomes the value of **`N.`*`entity`***. Either **`N.`*`entity`*** must be nonzero or the arithmetic expression must be included when the **`create each`** statement is executed.

Storage space for entity or resource attributes can be returned to the system with a **`destroy each`** statement.

2.18 CYCLE Statement

The **cycle** statement within a **do ... loop** construct of code causes control to pass back to the loop's controlling statements.

```
⎡       ⎤
⎨ cycle ⎬
⎣ next  ⎦
```

This statement consists of exactly one word.

The **cycle** statement is used within a **do ... loop** construct of code to cause premature ending of one iteration of the loop. Control is passed back into the controlling **for** or **until** logic, to select the next index value or to end the loop. In effect, **cycle** functions as if control had been transferred to the **loop** statement, except that no programmer-defined label is needed.

The statements **cycle** and **leave** clarify programs by eliminating labels, and provide the concept of local labels to SIMSCRIPT II.5. These features are especially useful when coupled with the **substitute** statement features of SIMSCRIPT II.5.

2.19 DEFINE ... ROUTINE Statement

The **`define ... routine`** statement declares characteristics of subroutines and functions.

```
                                  ⌈integer     ⌉
                                  |real        |
define routine[c] as [a]          |double      | [fortran] routine[s]
                                  |alpha       |
                                  ⌊text        ⌋

[given integer [argument[s]]] [[,] yielding integer [argument[s]]]
```

Keywords	Synonyms
a	an
routine[s]	function[s]
given	giving
	with
argument[s]	value[s]
fortran	
nonsimscript	non-simscript

EXAMPLES:

```
define DEPOT.SOURCING as a routine
```

Defines **`DEPOT.SOURCING`** as a subroutine. No checking will be performed because argument numbers are omitted.

```
define QUOTA and SALES as routines with 3 values
```

Defines **`QUOTA`** and **`SALES`** as subroutines, each having three input values and no output values.

```
define ANSWER as an alpha function
```

Defines **`ANSWER`** as a function whose value will be alphanumeric.

```
define PRINT.MESSAGE as a routine given 2 arguments yielding 1
```

Defines **`PRINT.MESSAGE`** as a subroutine having two input arguments and one output argument.

```
define square.fn as a double function given 1

define low_level_call3 as a nonsimscript integer function given 4

define speed as fortran routine given 3
```

Notes:

1. When you specify “fortran” routine, parameters are passed in “call by reference” mode as required by FORTRAN compilers.
2. When you specify “nonsimscript” routine, parameters are passed “by value” as required by C-compilers.
3. For more details about interfacing with non-simscript and FORTRAN routines, refer to the *SIMSCRIPT II.5 User’s Manual* for your platform.

The **`define ... routine`** statement is used to declare subroutines and functions. It appears only in the preamble. Each function *must* be declared to distinguish it from a variable in a subroutine. Subroutines need not be declared. This statement can declare the mode (for functions only), whether the subroutine or function is releasable, and the correct number of input and output arguments for subroutines. More than one function or subroutine can be declared in a single **`define ... routine`** statement, but a function or subroutine name can appear only once in a **`define ... routine`** statement. If a function is defined as alphanumeric, the value returned by the function is considered to be a character string, not a numerical value.

A combination of **`normally`** and **`define ... routine`** statements can appear in the preamble. **`Normally`** statements could specify the predominant characteristics, and **`define ... routine`** statements could declare any exceptions, as well as declaring additional characteristics.

2.19.1 GIVEN and YIELDING Phrases

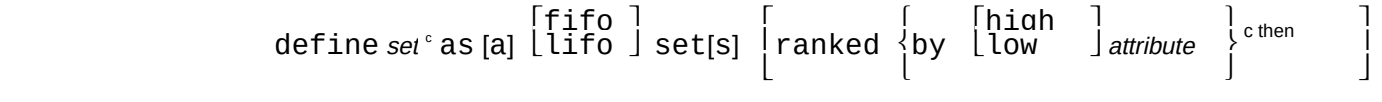
The correct number of arguments for a subroutine can be specified in the **`define ... routine`** statement by including **`given`** and **`yielding`** phrases. Subsequently, if the pertinent **`call`** and **`routine`** statements have a varying number of arguments, SIMSCRIPT II.5 applies the following corrective action:

1. Disregards additional input and output arguments in **`call`** and **`routine`** statements.
2. Considers omitted input arguments to be zero.
3. Reserves locations for missing output arguments so they can receive values (but these values are inaccessible to the calling routine).
4. Emits a warning message.

If the **`define ... routine`** statement contains only given arguments, the called routine is assumed to yield no values. If the **`define ... routine`** statement contains only yielding arguments, the called routine is assumed to have no given value. If no argument numbers are specified, no checking is performed, and variable-length calling sequences can be used.

2.20 DEFINE ... SET Statement

The **`define ... set`** statement names one or more sets and defines ranking, owner and member attributes, generated routines, and optional deletion of owner and member attributes and processing routines.



Keyword	Synonym
a	an

EXAMPLES:

```
define PORTFOLIO and RESERVATIONS as sets
```

Defines **`PORTFOLIO`** and **`RESERVATIONS`** as generalized sets having all the generated set attributes and routines.

```
define WAITING.LINE as a fifo set
```

Defines **`WAITING.LINE`** as a **`fifo`** set having all generated set attributes and routines.

```
define REGULATIONS as a set ranked by low NUMBER
```

Defines **`REGULATIONS`** as a set ranked by low values (ascending order) of attribute **`NUMBER`**.

```
define ARRIVALS as a set ranked by low ARRIVAL.TIME, then by high
NO.PASSENGERS without l and p attributes
```

Defines **`ARRIVALS`** as a set ranked by low values (ascending order) of attribute **`ARRIVAL.TIME`**, and resolves ties of **`ARRIVAL.TIME`** values by high values (descending order) of **`NO.PASSENGERS`**. Attributes **`L.`*`ARRIVALS`*** and **`P.`*`ARRIVALS`*** are not generated.

```
define AIRLINES as a set ranked by low NAME without l and p
attributes, without fl, fb, fa, and r routines
```

Defines **`AIRLINES`** as a set ranked by low values (ascending order) of attribute **`NAME`**. Set attributes **`L.`*`AIRLINES`*** and **`P.`*`AIRLINES`*** and set routines **`fl, fb, fa, rf, rl`**, and **`rs`** are not generated.

The **`define ... set`** statement, which can appear only in the preamble, describes set characteristics, including set discipline, owner and member attributes (see table 14), and generated set handling routines (see table 15). Two or more sets having the same charac-

teristics can be defined in the same statement. Sets can be defined as **`fifo`** or **`lifo`**, or can be ranked by high or low values of member attributes. When the set discipline is omitted, SIMSCRIPT II.5 assigns a generalized set whose organization will be determined by the programmer. A **`define ... set`** statement must follow (not necessarily immediately) the **`every`** statement that declares the owner and member entities of the set.

2.20.1 FIFO Sets

In a **`fifo`** set, entities are filed on a first-in, first-out basis. Each entity is placed last in the set as it is filed, and entities are removed from the top in the order in which they were filed. A **`fifo`** set requires only **`F.set, L.set`**, and **`S.set`** attributes. It generates seven routines: **`ff, fl, fb, fa, rf, rl`**, and **`rs`**.

Table 14. Automatically Generated Set Attributes

Letter	Attribute Name	Definition	Attributes of
F	**`F.set`**	Pointer to first entity in set.	Owner entities
L	**`L.set`**	Pointer to last entity in set.	Owner entities
N	**`N.set`**	Number of member entities currently in the set.	Owner entities
P	**`P.set`**	Pointer to predecessor in set.	Member entities
S	**`S.set`**	Pointer to successor in set.	Member entities
M	**`M.set`**	Membership attribute that is <>0 if an entity is in the set and 0 if the entity is not in the set.	Member entities

Table 15. Automatically Generated Set Routines

Mnemonic	Required Set Attributes	Routine Name	Purpose
FF	F S	**T.set**	Files entity first or ranked
FL	F L S	**U.set**	Files entity last
FB	F P S	**V.set**	Files entity before specified entity.
FA	F S	**W.set**	Files entity after specified entity.
F			Generates no file routines.
RF	F S	**X.set**	Remove first entity.
RL	F L P S	**Y.set**	Remove last entity.
RS	R P S	**Z.set**	Remove specified entity.
R			Generates no file routines.

2.20.2 LIFO Sets

In a **lifo** set, entities are filed in, and removed from, sets on a last-in, first-out basis. Each entity is placed first in the set as it is filed, and entities are removed from the top in the reverse order in which they were filed. A **lifo** set requires only **F.set** and **S.set** attributes. It generates seven routines: **FF, FL, FB, FA, RF, RL**, and **RS**.

2.20.3 Ranked Sets

Ranking is specified by naming the attribute whose values are to control the order of entities in the set. Ranking can be cascaded by including any required number of **then by** phrases. A **then by** phrase specifies how ties are to be resolved when two or more values are identical for a ranking attribute. **... before** and **file ... after** statements cannot be used with a ranked set. Ranked sets generate four routines: **ff, rf, rl**, and **rs**.

2.20.4 WITHOUT ... ATTRIBUTES Phrase

Owns and **belongs** phrases in an **every** statement automatically provide attributes for the owner and member entities, but the **without ... attributes** phrase can be used to delete any unnecessary set attributes. Each specified letter deletes the automatically-generated attribute formed by prefixing that letter and period to the set name. Any or all owner and member attributes can be deleted, but deleting all attributes destroys the concept of a set. See table 16 for attributes required for various set operations.

2.20.5 WITHOUT ... ROUTINES Phrase

Naming a set in an **every** statement with an owns phrase automatically provides the seven set routines, but the **without ... routines** phrase can be included in the **define ... set** statement to delete any unnecessary set routines. Any set routine can be deleted to conserve storage. If the letters **F** or **R** are specified, all file routines or all remove routines are deleted, respectively.

Table 16. Required Set Attributes and Routines

Set Statement	Attributes Required						Routines Required						
	f	l	p	s	m	n	ff	fl	fb	fa	rf	rl	rs
file in a ranked set	x		x	x			x						
file first	x			x			x						
file last	x	x		x				x					
file before	x		x	x					x				
file after	x			x						x			
remove first	x			x							x		
remove last	x	x	x	x								x	
remove specific	x		x	x									x
is empty	x												
is in set					x								
Automatic checking					x								
for each v in set	x			x									
for each v in set in reverse		x	x										
for each v from w in set				x									
for each v from w in set in rev.			x										
for each v after w in set				x									
for each v after w in set in rev.			x										

2.21 DEFINE ... TO MEAN Statement

The **`define ... to mean`** statement allows the programmer to substitute any word for any string in subsequent statements. During compilation, all subsequent occurrences of the word are replaced by the string.

```
define WORD to mean STRING
```

EXAMPLES:

```
define X to mean MATRIX
```

Substitutes the string **`MATRIX`** for the variable **`X`**.

```
define Y to mean A * X**2 + B * X + C
```

Substitutes the expression **`A * X**2 + B * X + C`** for the variable **`Y`**.

```
define SUBSECTION to mean ROUTINE
```

Substitutes the SIMSCRIPT II.5 keyword **`ROUTINE`** for the word **`SUBSECTION`**.

```
define FORMAT.LIST to mean I 4, 3 I 5
```

Substitutes the format list **`I 4, 3 I 5`** for the word **`FORMAT.LIST.`**

The **`define ... to mean`** statement substitutes a string of words for the indicated word in all subsequent statements, until superseded. The word can be a single letter, a name, a number, a special character, or an alphanumeric literal, and the string to be substituted for the word can be any string of characters such as a single character, a name, one or more SIMSCRIPT II.5 statements, an expression, a format string for an input/output statement, or an argument list.

SIMSCRIPT II.5 considers the string to be all the remaining characters of the card on which the **`define ... to mean`** statement appears. During compilation, whenever the compiler detects the specified word, it substitutes the string and compiles the statement with the substitution. Substitution takes place only when the word for which substitutions are to be made appears as a complete token, but not when it appears as part of another word. Substitutions can appear in strings for other **`define ... to mean`** or **`substitute`** statements, thus allowing several levels of substitution.

The **`define ... to mean`** statement can appear in the preamble or in a routine. When this statement appears in the preamble, the substitution affects the entire program. In a routine, the substitution is local and is effective only for that routine until superseded. A **`suppress substitution`** statement can override the effect of a **`define ... to mean`** statement, while a **`resume substitution`** statement can reinstate the effect.

2.21.1 Purposes of DEFINE ... TO MEAN

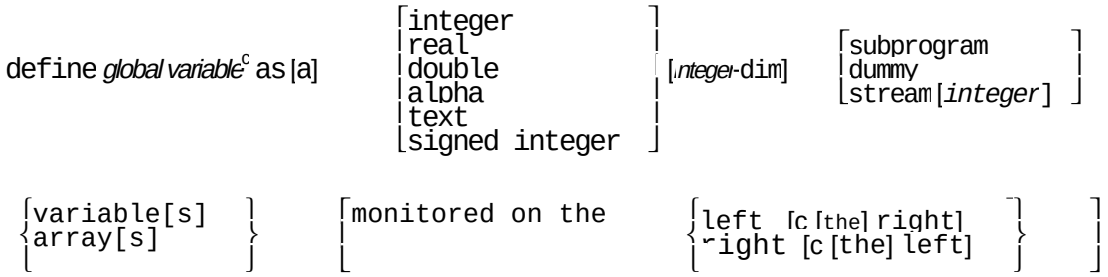
The **`define ... to mean`** statement can be employed for any of the following purposes:

1. To change a word in a routine to the same word used in other routines in a large program.
2. To change statement keywords to another vocabulary.
3. To define a macro instruction, meaning a compound instruction generated from a single keyword.
4. To define format strings in order to call them by name. This can minimize the number of characters that must be written when several statements have identical format lists.
5. To define names as synonyms, substitute one variable name for another, or replace a name with complete statements.

Redefining statement keywords must be handled very carefully to avoid substituting a new string for an optional keyword, or for any other characters that might cause incorrect compilation because the statement syntax was not followed. For example, preceding the statement **`every MAN can own some DOGS`** with the statement **`define CAN to mean BOTTLE`** results in a syntax error when attempting to compile with the substitution: **`every MAN BOTTLE own some DOGS`**.

2.22 DEFINE ... (Global) VARIABLE Statement

The **`define ... (Global) variable`** statement defines the properties of global variables.

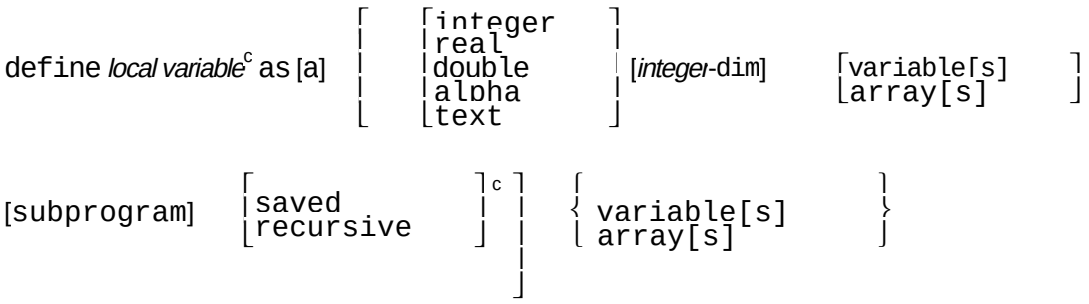


Keywords	Synonyms
a	an
dim	dimensional

For further information, see the **`define ... variable`** statement.

2.23 DEFINE ... (Local) VARIABLE Statement

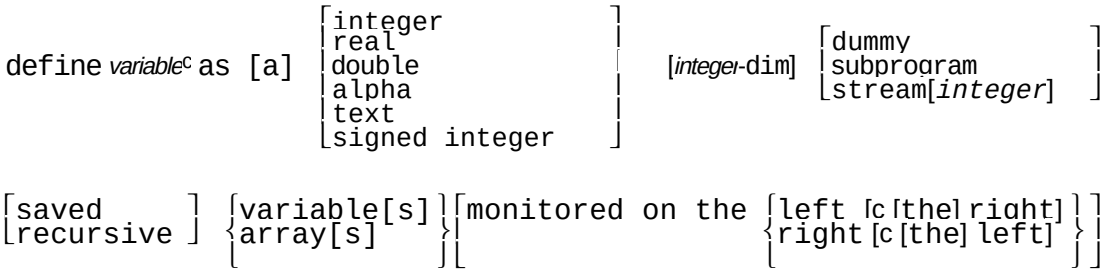
The **`define ... (Local) variable`** statement defines the properties of local variables.



For further information, see the **`define ... variable`** statement.

2.24 DEFINE ... VARIABLE Statement

The **`define ... variable`** statement defines characteristics of global and local variables, of attributes, and of arrays.



Keywords	Synonyms
a	an
dim	dimensional

EXAMPLES:

```
define NUMBER as a variable
```

Defines **`NUMBER`** as a variable having the mode, type, and dimensionality of the current background conditions.

```
define ORIGIN, DESTINATION, and MOVE as alpha variables
```

Defines **`ORIGIN`**, **`DESTINATION`**, and **`MOVE`** as variables having alphanumeric values, type, and dimensionality as specified by background conditions.

```
define BOX as an integer, 3-dimensional array
```

Defines **`BOX`** as a three-dimensional array whose elements have integer values.

```
define a, b, and c as real, recursive variables
```

Defines **`a`**, **`b`**, and **`c`** as variables whose values are real and recursive.

```
define LOCATIONS and ADDRESSES as 1-dimensional, saved, subprogram
arrays
```

Defines **`LOCATIONS`** and **`ADDRESSES`** as one-dimensional arrays containing subprogram variables, whose values (addresses of routines) are saved from one subroutine call to another.

```
define NAME as dummy variable
```

Defines **NAME** as a dummy variable, which has a name but no storage location.

```
define DATA and RESULT as integer, 2-dimensional arrays monitored
on the left and right
```

Defines **DATA** and **RESULT** as two-dimensional arrays monitored on the left and the right; elements of the arrays have integer values.

The **define ... variable** statement defines the mode of one or more variables, declares whether variables are saved or recursive, subprogram or dummy, and specifies the dimensionality of arrays. A given variable has several characteristics, and several variables can be named in a single statement. This statement can be used to define characteristics of global and local variables, and of attributes and arrays that differ from characteristics declared in **normally** statements. Each variable, attribute, or array can appear in only one **define ... variable** statement. The **define ... variable** statement may appear in the preamble and in routines, depending on whether the variable to be defined is global or local.

2.24.1 NORMALLY and DEFINE ... VARIABLE Statements

A combination of **normally** and **define ... variable** statements can appear in the preamble and in routines. In the preamble, **normally** statements usually declare the most prevalent characteristics of the global environment, called the *background conditions*, while **define ... variable** statements declare any exceptions and additions. Characteristics that have been defined in a **define ... variable** statement override those declared in preceding **normally** statements, while the background conditions implicitly fill in unspecified characteristics for each variable.

Characteristics defined in the last **normally** statement in the preamble apply to all routines, unless superseded by **normally** statements within the routines. In a subroutine or function, **normally** and **define ... variable** statements define the local environment, with definitions applying only to that routine. These statements can appear anywhere within the routine, but their relative order is important.

2.24.2 Global Variables

A global variable, which has a common meaning throughout a program, references the same storage location whenever the variable is used. Each global variable must be included in a **define ... variable** statement in the preamble to inform the compiler that it is a global variable. If its characteristics are declared in a preceding **normally** statement, the **define ... variable** statement need not repeat the characteristics. The type specification is meaningless for global variables, as they are always saved.

2.24.3 Attributes

In the preamble, a **define ... variable** statement is used to define characteristics of attributes that differ from those declared in the effective **normally** statement. A **define ... variable** statement must follow **every** and **the system** statements whenever the **define ... variable** statement includes attributes named in either statement. See the **... random ... variable** statement for defining random variable attributes.

2.24.4 Local Variables

Local variables, whose characteristics differ from those of the current local environment or that have the same names as global variables, must be included in **define ... variable** statements in their respective routines. Variables not appearing in a **define ... variable** statement are automatically defined by the compiler as local variables having the current background characteristics. Local variables can be defined as saved or recursive.

2.24.5 Arrays

The dimensionality of an array must be specified in the preamble with either a **normally** or a **define ... variable** statement. Any dimensionality declared in a **normally** statement can be superseded by a subsequent declaration in a **define ... variable** statement, but the dimensionality is permanent after appearing in a **define ... variable** statement. If an array is an argument, the dimensionality must appear in a **define ... variable** statement in the subroutine to which the array is an argument.

Arrays can be declared as saved or recursive. This means that the array base pointer word is either saved, or set to zero each time the routine is called.

2.24.6 Arguments, Recursive Variables, and Saved Variables

Three types of local variables are provided: arguments, recursive variables, and saved variables. Any local variable can be declared as either saved or recursive. Arguments, however, are always stored as recursive variables. They are automatically defined to be recursive when named in a **routine** statement, no matter what is said about them in a **define ... variable** statement. It is always best to define other argument characteristics explicitly to avoid later confusion.

The first time a routine is called, all saved and recursive local variables are set to zero. Thereafter, a recursive local variable has an initial value of zero each time the routine in which it appears is called, but a saved local variable retains the value stored when the routine was last executed.

2.24.7 Subprogram Variables

A subprogram variable is defined as a variable having the address of a subroutine, or of a function, as its value, enabling the routine to be called indirectly. A subprogram variable

can be used as an argument, in **let** statements, in logical expressions, etc., to stand for the specific piece of computation defined by the routine that is the value of this variable.

2.24.8 Dummy Variables

Global variables and attributes can be defined as dummy variables, which have names but no storage locations. These variables are used in **accumulate** and **tally** statements to compute statistics on variables and attributes that otherwise are not used in a program. Dummy global variables must also be defined in a **define ... variable** statement, and dummy attributes must also be declared in **every** or **the system** statements.

2.24.9 Monitored Variables

A monitored variable is a variable whose values are monitored (checked or used) by a subroutine. Each time a monitored variable is accessed, its associated subroutine is executed. Therefore, a monitored variable has both a value and a subroutine associated with it, each with the same name. In fact, because variables can be monitored on the right, on the left, or on both the right and left, both a left- and right-hand subroutine with the same name as the monitored variable might be associated with it. The words **left** and **right** refer to the occurrence of the variable to the left or right of an equal sign in a **let** statement. A **define ... variable** statement can declare a variable, an array, or an attribute as a monitored variable. The **enter** statement allows the value assigned to a left-monitored variable to be used within the monitoring routine. The **move** statement gives access to the value of a right-monitored variable or assigns a value to a left-monitored variable.

Note that monitored variables behave more like variables than routines. For example, one can never **call** a monitored variable directly. When a monitored variable is used, its subscripts are automatically converted to integer (like a subscripted variable), but also passed as arguments to the monitoring routine.

2.25 DESTROY Statement

The **destroy** statement returns a block of storage serving as a temporary entity, process, or an event notice to available system storage. It releases the block of storage for the specified entity pointer variable.

```
destroy [a] { temporary entity } [called pointer value]
            { process          }
            { event            }
```

Keyword	Synonyms
a	an
	the
	this

EXAMPLES:

```
destroy FLIGHT
```

Returns a block of storage for an entity of the temporary entity class **FLIGHT**. The pointer to this block is assigned to the global variable named **FLIGHT**.

```
destroy the FLIGHT called JET(I)
```

Returns a block of storage for an entity of the temporary entity class **FLIGHT**. The pointer to this block is the **I**th value of the array named **JET**.

```
destroy the STOCK.CERTIFICATE called STOCK
```

Returns a block of storage for an entity of the temporary entity class **STOCK.CERTIFICATE**. The pointer to this block is assigned to the variable named **STOCK**.

```
destroy TAKEOFF
```

Returns a block of storage for an event notice of the event class **TAKEOFF**. The pointer to this block is assigned to the global variable named **TAKEOFF**.

The **destroy** statement returns a block of storage, which was created by a **create** statement, to available storage. If the **called** phrase is omitted, SIMSCRIPT II.5 uses the global variable having the same name as the entity class to locate the block to be returned. If the **called** phrase is included, however, SIMSCRIPT II.5 uses the variable named in the **called** phrase to locate the block. In either case, the appropriate variable must contain a pointer to (the address of) the storage block to be returned.

When an entity is destroyed, that piece of storage is made available for use by **`create`** statements (and, in some implementations, **`reserve`** statements as well). References to that entity are invalid after it is destroyed because the storage may have then been allocated to represent a new entity. Depending on the implementation, attribute values may or may not be set to zero on destroying an entity. (Attributes will, of course, be set to zero when this storage block is used to **`create`** an entity.)

An entity that is still in a set cannot be destroyed. Any attempt to do so will result in a fatal error. The **`destroy`** statement can appear in any routine, but it cannot be included in the preamble.

2.26 DESTROY EACH Statement

The **`destroy each`** statement deallocates arrays for the attributes of the permanent entity or resource classes named in the statement.

```
destroy each  { permanent entity }
              { resource        }
```

Keyword	Synonyms
`each`	`every`
	`all`

EXAMPLE:

```
destroy each AIRPORT
```

Releases all arrays associated with the permanent entity **`AIRPORT`**.

The **`destroy each`** statement returns storage allocated for a permanent entity or a resource. When the entity or resource is destroyed, the storage associated with each array forming the entity is made available for later **`create each`** or **`reserve`** statements (and, in some implementations, **`create`** statements as well). Released storage includes the SIMSCRIPT-defined attributes involved in set operations and **`accumulate`** and **`tally`** operations. References to attributes of any entities or resources in that class are invalid and, on some implementations, will be detected as such.

An entity that is still in a set cannot be destroyed. This will be detected as an error. The **`destroy each`** statement can appear in any routine, but it cannot be included in the preamble.

2.27 DO ... LOOP Construct

The **do ... loop** construct designates the beginning of a program segment to be executed repeatedly.

do *statement*^{i} loop

Keyword	Synonym
this	the following

EXAMPLES:

```
for I = 1 to 10, do
```

Executes controlled statements with values of index variable **I** ranging from 1 to 10; that is, the program segment is executed 10 times.

```
for I = 1 to 10, for J = 1 to N , do this
```

Executes controlled statements with values of index variable **I** ranging from 1 to 10, and **J** ranging from 1 to **N**. That is, the inner loop is executed n times for each execution of the outer loop.

```
until X**2 - Y**2 is negative, do '  'calculations
```

Executes controlled statements until $X^2 - Y^2$ yields a negative value. **Negative** is a keyword available for use in logical expressions. Commentary text follows the two apostrophe characters.

```
for each CITY, with POPULATION(CITY) greater than 500000, do
```

Executes controlled statements for each entity of entity class **CITY** if the value of attribute **POPULATION** exceeds 500,000.

```
for each AIRPORT, for each FLIGHT of ARRIVALS, do the following
```

Executes controlled statements for each entity of entity class **FLIGHT** filed in the set named **ARRIVALS** at each airport.

A program segment to be executed repeatedly must begin with a **do** statement and end with a **loop** statement, and **do ... loop** constructs must be preceded by at least one **for each (*Class*)**, **for ... of (*Set*)**, or **for ... to (*Index*)** phrase, or by a **while** or **until** termination phrase. Normally, a **do ... loop** construct is executed over the entire range of controlling **for** phrases, because **for** phrases control a program segment exactly as they control a single statement. Any combination of **for** phrases and **selection** and **termination** phrases can precede the **do ... loop** construct, with the program segment being executed once for each iteration of controlling **for** phrases. The **do ... loop** construct can appear in any routine, but it cannot be included in the preamble.

2.27.1 Nested DO ... LOOP Constructs

A **do ... loop** construct can be nested within other **do ... loop** constructs for control over subscripted variables having two or more subscripts. Subscript indexing occurs within the limits of the **do** statement and **loop** statement. When **do ... loop** constructs terminate at the same place, the **for** phrases can be preceded by the keyword **also** in order to eliminate redundant **loop** statements. In this event, SIMSCRIPT II.5 matches the **do** statement that follows the **also for** phrase with the **loop** statement of the **do ... loop** construct of the preceding **for** phrase. In figure 2, nested **do ... loop** constructs are illustrated on the left, and the same loops are shown using **also for** phrases on the right.

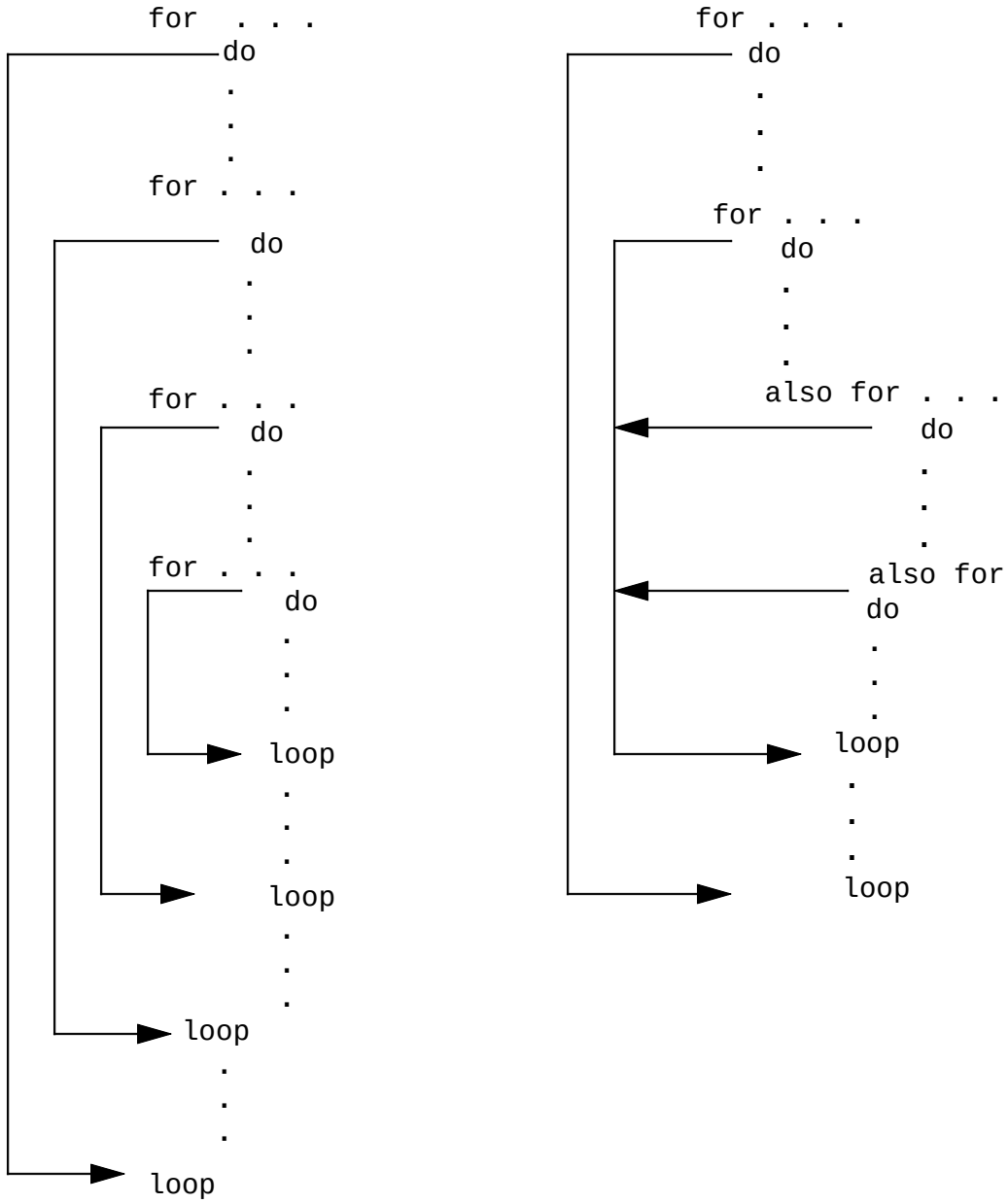


Figure 2. Nested `do ... loop` Constructs and `do ... loop` Constructs Using `also for` Phrases

2.28 ELSE Statement

See **`if ... else ... always`** construct.

2.29 END Statement

The **end** statement designates the physical end of the program preamble, of each event, process, function, and program routine, and of each report and heading section.

```
end
```

Only one form of this statement exists. It consists of exactly one word.

An **end** statement must be placed at the physical end of each SIMSCRIPT II.5 program element to ensure correct compilation. Each of the following must have an **end** statement to mark the physical end of the programming modules:

Preamble	Report section
Main routine	Heading section
Subroutine	External event routine
Function routine	Internal event routine
	Process routine

2.30 ENTER WITH Statement

The **enter with** statement is used to transfer a right-hand expression to a local variable within a left-hand function.

```
enter with variable
```

EXAMPLES:

```
enter with NUMBER
```

Specifies that the value received by a left-hand routine is stored in the variable **NUMBER**.

```
enter with LIST(I)
```

Specifies that a value received by a left-hand routine is to be stored in the **I**th element in the one-dimensional array named **LIST**.

A value can be passed to a left-hand function by writing the function on the left side of the equal sign in a **let** statement, using it as a **yielding** argument, using it in a **read** statement, and so on. The **enter with** statement, which must be the first executable statement in every left-hand function, specifies that the value computed "on the right" is to be assigned to the named variable. The variable can be local or global, subscripted or unsubscripted, or an attribute (for use within the left-hand routine). The mode of the expression assigned to the function must agree with the mode defined for the function in the preamble. After transferring the value with an **enter with** statement, a left-hand function is like any other routine. For example, computations or other processing can be performed using the variable. In order to store a value in a variable monitored on the left, a **move from** statement is used in the left-hand routine.

2.31 ERASE Statement

The **erase** statement is used to release storage designated for text variables.

```
erase text variable
```

EXAMPLES:

```
erase AUTHOR
```

Releases the storage previously designated for the text variable **AUTHOR**, and assigns a null value to the variable.

```
erase NAME, STREET.ADDRESS and CITY.ADDRESS
```

Releases the storage previously designated for the text variables **NAME**, **STREET.ADDRESS**, and **CITY.ADDRESS**, and assigns null values to all three variables.

Text variables may be erased by using the **erase** statement. The statement has the same effect as assigning a null value to the variable. Thus, the first example above is equivalent to:

```
let AUTHOR = ""
```

Note: The name of the text variable itself is still recognized by SIMSCRIPT II.5. The **erase** statement merely erases the stored value.

2.32 EVENT Statement

The **event** statement names an event routine for an internal event, an external event, or both.

```
event [to] event    [given value^c                ] [saving the event notice]
                    [(value^c)                    ]
```

Keywords	Synonyms
event	upon
to	for
given	giving
	the
	this

EXAMPLES:

```
event MANAGEMENT.REPORT
```

Declares **MANAGEMENT.REPORT** as an event routine, and destroys the event notice before entering the event routine.

```
event TAKEOFF given FLIGHT.NO, DESTINATION, NO.PASSENGERS
```

Declares **TAKEOFF** as an event routine having **FLIGHT.NO, DESTINATION**, and **NO.PASSENGERS** as arguments; destroys the event notice.

```
upon TAKEOFF(FLIGHT.NO, DESTINATION, NO.PASSENGERS) saving the
event notice
```

Declares **TAKEOFF** as an event routine having **FLIGHT.NO, DESTINATION**, and **NO.PASSENGERS** as arguments. Saves the event notice.

An event routine, which is declared with an **event** statement, is similar to a subroutine although an event routine does not return output values. It cannot yield values because an event routine is called by the timing routine, and consequently cannot return to another routine. Each event class must have an event routine. When an event is generated (by executing a **schedule** statement or by an external event card), the timing routine is informed by the event notice when the event is to occur in the future. The **schedule** statement includes the simulated time for internal events, and external event data cards specify the time for external events. Events take place instantaneously and do not consume simulated time. The **event** statement cannot appear in the preamble.

2.32.1 Arguments

An internal event is triggered by an event notice, as a result of a **schedule** statement. Besides the five special attributes that all event notices have, an event notice transmits values of any additional attributes from the **schedule** statement to arguments of the event routine. (Additional attributes must be defined in **every** statements.) An event routine that simulates only external events cannot have arguments, but a routine used for both internal and external events can have arguments. The arguments of the event routine are local to the routine. If their specifications differ from the last **normally** specifications in the preamble, they must be defined in the event routine.

2.32.2 SAVING Phrase

Event notices can be reused. An event notice is automatically destroyed just before an event routine is executed unless a **saving the event notice** phrase is included in this statement. The pointer to the event notice will no longer be valid without the **saving** phrase.

2.32.3 Logical Expression for Event Routines

SIMSCRIPT II.5 provides a logical expression to determine, within an event, whether that event was generated internally or externally. The logical expression is of the form:

```
                  ⌈endogenous ⌉
event is [not]    {exogenous  }
                  |internal   |
                  ⌊external   ⌋
```

and yields a true or false value. This logical expression, which tests the event notice, can be included in an **if ... else ... always** construct to make decisions regarding events.

2.33 EVENT NOTICES Statement

An **`event notices`** statement declares that the following **`every`** statements define event notices for internal events.

`event notices` [`include` *event* `c`]

Keywords	Synonyms
`event notices`	`events`
`include`	`are`

EXAMPLES:

`event notices`

Indicates that event notice declarations follow. The declarations will be made with **`every`** statements, and will name user-defined attributes. They denote ownership of, and membership in, sets.

`event notices include ARRIVAL and WEEKLY.REPORT`

Identifies **`ARRIVAL`** and **`WEEKLY.REPORT`** as event notices requiring only the system-defined attributes for event notices. **`Every`** statements may follow this form of the **`event notices`** statement.

An event notice is a temporary entity that has five special attributes. An **`event notices`** statement, which declares that the following **`every`** statements define event notices, must be included in the preamble for internal events. Because SIMSCRIPT II.5 automatically places the special attributes in the event notice record, the programmer must not declare these attributes. One cannot place other attributes in the words of an event notice which contain the special attributes. An event notice may consist of only the special attributes, in which case the event notice triggers the event, but does not transmit values to arguments in the event routine. The attributes of an event notice record are described in table 17.

Event notices are created and destroyed like temporary entities.

An event notice carries information about an internal event. When an event is generated, the event notice transmits this information about the event to the timing routine, and (eventually) from the timing routine to the event routine when the event occurs.

Table 17. Event Notice Attributes

Attribute	Description
time.a	Simulated time at which the event is to occur.
eunit.a	Code to indicate whether internal or external. Number of input unit containing event information for external events; zero for internal events.
p.ev.s	Pointer to predecessor in **ev.s***.
s.ev.s	Pointer to successor in **ev.s***.
m.ev.s	Membership attribute that is <>0 if the event notice is in **ev.s***, and 0 if the event is not in **ev.s***.
Note: ***ev.s** refers to the event set. That is, the set used by the timing routine to keep track of scheduled events.	

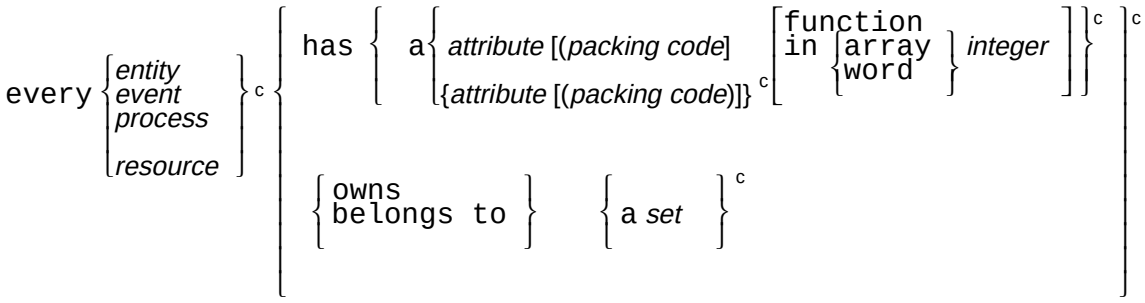
Event notices are used to schedule events for occurrence at some time in the simulated future. Many events of the same event class can be scheduled at the same simulated time. A **break ... ties** statement declares priorities among events of the same event class, while a **priority** statement declares priorities among different classes of events. If no **priority** statement is present, priority is given according to the order in which the event notices have been defined in the **include** phrase or in subsequent **every** statements.

In addition to the five special attributes, event notices can have attributes that are either variables or functions. Event notices can own and belong to sets. **Every** statements are used to declare the attributes and sets.

Event notices with additional attributes must be declared in **every** statements. Event notices with no additional attributes can be named in the **include** phrase of the **event notices** statement, and the first five words of the event notice record will be used for the five special attributes.

2.34 EVERY Statement

The **every** statement declares entity, attribute, and set structure for temporary and permanent entities, resources, and event and process notices. It also specifies optional attribute packing, equivalences, word assignments, and functions.



Keywords	Synonyms
has	have
	can have
	may have
a	an
	the
	some
owns	own
	can own
	may own
belongs to	belong to
	can belong to
	may belong to

EXAMPLES:

```
every CITY has a NAME, an AREA, and a POPULATION
```

Declares that entities of the permanent entity class **CITY** have attributes named **NAME**, **AREA**, and **POPULATION**.

```
every PATRON has a NAME, a DESTINATION, and a FARE, and belongs to
some RESERVATIONS
```

Declares that entities of the temporary entity class **PATRON** have attributes named **NAME, DESTINATION,** and **FARE**. The entities are members of the set class **RESERVATIONS**.

```
every TAKEOFF has a FLIGHT.NO, a DESTINATION, and SOME
(NO.PASSENGERS, NUMBER.OF.PASSENGERS)
```

Declares that event notices of the class **TAKEOFF** have attributes named **FLIGHT.NO, DESTINATION,** and **NO.PASSENGERS**. **NUMBER.OF.PASSENGERS** is a synonym for **NO.PASSENGERS.**

```
every AIRPORT has a NO.OF.RUNWAYS(*/4) and owns some JET.RUNWAYS,
some ARRIVALS, some DEPARTURES, and some TERMINALS
```

Declares that entities of the permanent entity class **AIRPORT** have an attribute named **NO.OF.RUNWAYS** whose values are to be intrapacked in an array, with four consecutive values per word. The entities own sets of the set classes **JET.RUNWAYS, ARRIVALS, DEPARTURES,** and **TERMINALS**.

```
every STOCKHOLDER has a NAME, a FINANCIAL function, owns a
PORTFOLIO, has the F.PORTFOLIO in array 1, the L.PORTFOLIO in array
2, and the N.PORTFOLIO in array 3
```

Declares that entities of the permanent entity class **STOCKHOLDER** have an attribute named **NAME** and a function attribute named **FINANCIAL**. The entities own sets of the set class **PORTFOLIO**, and have attributes (set pointers) **F.PORTFOLIO**, **L.PORTFOLIO**, and **N.PORTFOLIO** that are to be assigned to arrays 1, 2, and 3, respectively.

```
every FLIGHT has some (FIRST.CLASS.SEATS(L/2), TOURIST.SEATS(2/2))
in word 1, and the (NO.PASSENGERS(1-10), NO.IN.CREW(11-16), and
SEATING.CAPACITY(2/2)) in word 2
```

Declares that entities of the temporary entity class **FLIGHT** have five attributes. Values of attribute **FIRST.CLASS.SEATS** are to be stored in the first half of word 1, and values of **TOURIST.SEATS** in the second half of word 1. Values of **NO.PASSENGERS** are to be stored in bits 1-10 of word 2, values of **NO.IN.CREW** in bits 11-16 of word 2, and values of **SEATING.CAPACITY** in the second half of word 2.

The **every** statement, which declares entities, attributes, and sets, is used for permanent and temporary entities, resources, and for event and process notices. **Every** statements name an entity class, or event or process notice class, and define attributes of the entity, sets owned by the entities, and sets of which the entities are members. When compiled, this statement establishes set pointers and set attributes for owner entities that allow set memberships to be constructed. Variations of this statement declare attribute equivalencing and packing factors, array and word assignments, and function and dummy

attributes. In the preamble **temporary entities, permanent entities, resources, process notices**, and **event notices** statements must be followed by their respective groups of **every** statements.

In the statement format, the name of an entity class is followed by attribute phrases, set-owner phrases, and set-member phrases. The phrases can be in any desired order, and more than one of each phrase type can be included in a single statement. Keywords specify the phrase type: **has** denotes an attribute phrase, **owns** denotes a set-owner phrase, and **belongs to** denotes a set-member phrase. In an attribute phrase, packing factors can be included to declare the word portions, or specific bits, to be occupied by values of an attribute. In addition, attributes of permanent entities and resources can be assigned to specific arrays, and attributes of temporary entities can be assigned to specific words. Explicit array and word assignments generate more efficient code than implicit assignments generate.

2.34.1 General Rules

1. The name of an entity, resource, or process class, or of an event notice, can appear in more than one **every** statement.
2. Two entities cannot have an attribute with the same name placed in different words within the entity records, because attribute names specify relative locations in entity records.
3. The current background mode and dimensionality characteristics are assigned to the declared attributes (but may be overridden by a subsequent **define ... variable** statement). Automatically-defined set attributes are of integer mode.
4. Attributes are consecutive words or arrays in entity records in the order in which they are defined unless other specifications appear in the **every** statement. For those implementations requiring hardware alignment of double precision floating point numbers, a used word must be left to properly align a **double** variable.

2.34.2 Compound Entities

Compound entities are entities that jointly have attributes and own sets. They can consist of permanent entities, temporary entities, or a combination of permanent and temporary entities, but the attributes of compound entities that consist only of temporary entities, or combined permanent and temporary entities, must be functions. In the **every** statement, compound entities are specified by naming two or more entity classes, followed by their joint attributes and sets.

2.34.3 Event Notices

An event notice, which is generated for an event, is a temporary entity that has five special attributes. Event notices are declared when an **event notices** statement precedes the pertinent **every** statements in the preamble. The compiler automatically places the system-defined attributes in the first words of each event notice. The number of words re-

quired varies with the implementation. No other attributes can be placed in words used by the system-defined attributes. The general rules for **every** statements also apply to event notices.

2.34.4 Process Notices

A process notice, which is generated for a process, is a temporary entity that has nine special attributes. Process notices are declared when a **processes** statement precedes the pertinent **every** statements in the preamble. The compiler automatically places the system-defined attributes in the first words of each process notice, with the number of words required varying with the implementation. No other attributes can be placed in words used by the system-defined attributes. The general rules for **every** statements also apply to process notices.

2.34.5 Equivalencing

Data values for the same entity can be given different attribute names by placing the names in parentheses, separated by commas, in an attributes phrase. These names are then said to be equivalent.

Equivalent attributes are assigned to the same computer word. Any attribute can be equivalent, except text attributes, which can only be equivalenced to other text attributes.

2.34.6 Common Attributes

Common attributes are attributes that are common to (have the same name in) more than one entity class. Rules applying to common attributes are:

1. Values of common attributes must have the same relative locations in all entity records.
2. Common attributes must have the same packing factors and word assignments.
3. Common attributes *must specify the word* in a temporary entity record in which a value is to be stored.

2.34.7 Packing

Packing is defined as storing two or more values in a single word. Values can be packed into fractions of a word (e.g., a byte), or into specific bits, or several values of one attribute can be packed into a single word. In the **every** statement, packing is specified by appending a packing factor, enclosed in parentheses, to an attribute name.

Field packing designates which fraction of a word — typically a half, quarter, or sixth — is to be occupied by values of the named attribute. For example, the packing factor (l/2) specifies the first half of a word, and (4/4) specifies the fourth quarter of a word.

A bit-packing factor designates bits to be occupied by values. For example, a bit packing factor of (7-10) specifies that values of an attribute are to occupy bits 7 through 10 of a

word. Bits are numbered sequentially from the left (most significant) to the right (least significant) starting with 1.

A factor is denoted by (***/integer**). The asterisk indicates intrapacking, and integer is the number of values to be packed per word. For example, the intrapacking notation (***/2**) packs two consecutive values per word.

The following rules apply to packing:

1. Packing is specified by appending a packing factor, enclosed in parentheses, to an attribute name.
2. If values of more than one attribute are to occupy the same word, the attribute names must be enclosed in parentheses and separated by commas. For example, (**FIRST (1/2), SECOND (4/4)**). Alternatively, they may be assigned the same word number.
3. More than one group of attributes to be packed can be specified in a single **every** statement.
4. Attributes of permanent and temporary entities can be packed, but global variables cannot (see rule 8).
5. Integer and alpha values can be packed. Text and real values cannot be packed. Set pointers may or may not be packed, depending on the implementation.
6. All integer and alpha attributes can have field and bit packing.
7. Overlapping the packing specification is allowed. However, this usage is discouraged. Programs containing overlapped attributes might not be transferrable between different computer types.
8. Only attributes of permanent entities can specify intrapacking.
9. If two attributes placed in the same word have the same packing factors, their names are synonyms.
10. Packing does not apply to function attributes or dummy variables.
11. The default for packed integers is unsigned. Signed integers can be specified in a **define ... variable** statement.

2.34.8 Function Attributes

A function attribute is an attribute whose value is computed by a function routine. Consequently, a routine must be written having the same name as the attribute. Because function attributes designate routines, no storage in entity records is allocated for the values.

2.34.9 Dummy Attributes

A dummy attribute, which does not have a storage location, must be declared in a **`define`** statement. This declaration permits the dummy attribute to be used in **`accumulate/ tally`** statements without having its value stored.

2.34.10 Sets Named in EVERY Statements

The following rules and characteristics apply to sets named in **`every`** statements:

1. Owner entities have attributes **`F.`*`set`*** and **`L.`*`set`*** (pointers to the first and last members of the set, respectively), as well as **`N.`*`set`*** (whose value is the number of member entities currently in the set).
2. Member entities have attributes **`P.`*`set`*** and **`S.`*`set`*** (pointers to predecessor and successor members of the set, respectively), and a membership attribute named **`M.`*`set`*** (which is non-zero if the entity is in the set and 0 if the entity is not in the set).
3. Set pointers have integer values.
4. When entities belong to common sets, the set pointers must be assigned to the same words or arrays.
5. Set members are ranked as first-in, first-out when they are placed in a set, which gives priority to the first entity placed in the set. This may be overridden by a subsequent **`define ... set`** statement.

2.35 EXCEPT WHEN Phrase

See **`unless`** phrase.

2.36 EXTERNAL EVENTS/PROCESSES Statement

The **external events** statement declares events that can be generated externally. The **external processes** statement declares processes that can be generated externally.

```
external { event[s] are event_c
         { process[es] are process_c
```

Keywords	Synonyms
external	exogenous
are	is

EXAMPLES:

```
external event is MANAGEMENT.REPORT
```

Declares **MANAGEMENT.REPORT** as an external event.

```
external events are SNOW and MANAGEMENT.REPORT
```

Declares **SNOW** and **MANAGEMENT.REPORT** as external events.

```
external process is STORM
```

Declares **STORM** as an externally-triggered process.

The **external events** statement, which can appear only in the preamble, declares events that can be generated external to the simulation model. Events can be generated externally, internally, or both. External events must be declared in an **external events** statement, while an **event notices** statement is required for internal events. SIMSCRIPT II.5 permits one **external events** statement for each event class.

If an event is declared as external, and is not declared as an event notice, the system will use an event notice having only the five special attributes. However, if an event is external, and is also declared as an event notice, the event can be triggered both internally and externally and the event notice can have more than the five special attributes. The event notice will have the same name as the declared event, and is prepared each time an external event record containing the event name is read from the external event input unit. (External event input units must be declared in the **external ... units** statement.)

An external event record, which generates an external event or process, contains the name of an event (process) class, the time the event (process) is to occur, and the mandatory **mark.v** character. It may also optionally contain data to be read by the event (process) routine. The system reads these records as free-format data from the external unit input devices. They must be in chronological order. Table 18 describes the possible time formats for external event records. Examples follow:

```
END.SIMULATION 12.35    *
```

Generates an external event of the class **END.SIMULATION** to occur at day 12.35 of the simulation.

```
 WEATHER    1    13    45     SLEET    0.1    *
```

Generates an external event of the class **WEATHER** to occur on the second day of simulation at 1:45 p.m.. **SLEET** and the number 0.1 are data to be read by the event routine named **WEATHER**.

```
MANAGEMENT.REPORT    2/17/91    9    30    WEEK    25.0    89.6    *
```

Generates an external event of the class **MANAGEMENT.REPORT** to occur on February 17, 1991 at 9:30 a.m.. **WEEK**, 25.0 and 89.6 are data to be read by the event routine named **MANAGEMENT.REPORT**.

Table 18. Time Formats for External Event Cards

Format	Definition
Decimal time units	Day of the simulation at which the event is to occur; must be a real number. Examples: 0.0; 12.35; 18.0
Day hour minute	Three integers, separated by blanks, that specify the day, hour of the day, and minute of the hour when the event is to occur. All three numbers must appear. Hours are numbered from 0 to 24, minutes from 0 to 60. Examples: 0 0 0 indicates the start of simulation. 1 13 45 represents the second day at 1:45 p.m.
Calendar time	Date on which the event is to occur, denoted as a calendar day, and the hour and minute of the hour as integers. Years after 1999 and before 1900 must be completely expressed. Example: 2/17/91 9 30 represents February 17, 1991 at 9:30 a.m.

SIMSCRIPT II.5 reads each card as it is required, interprets the contents, and creates an event or process notice of the named class. It stores the time of occurrence in attribute **time.a** of the event (process) notice, and the number of the external unit in **eunit.a**. Then it files the notice in the event set corresponding to the event class. See the **external ... units** statement for action in case of multiple external units.

When an external event (process) becomes the current event (process), SIMSCRIPT II.5 stores the number of the unit containing the event data in the system variable **read.v** and transfers control to the event or process routine, as appropriate. In the event (process) routine, either **read (Free-form)** or **read ((Formatted)** statements can read the optional data. **Rcolumn.v** (current input pointer) is positioned to read the first column after the time. After executing the external event or process, the next event (process) data are read from the **read.v** unit. SIMSCRIPT II.5 skips any unread data included before the **mark.v** character. Reading in the next external event card sets up the event or process notice, as above, and control is then passed to the event and process selection mechanism.

If the events or processes of the named class are generated only externally, an event notice contains only five system-defined attributes, and a process notice contains only nine such attributes. However, if the events or processes can also be generated internally, the event (process) notice may have additional attributes. In the event set, the event (process) notice for an external event (process) is merged with notices for internally generated events (processes) of the same class.

The **external processes** statement declares processes that can be generated external to the simulation model. Processes can be generated externally, internally, or both. External processes must be declared in an **external processes** statement, while a

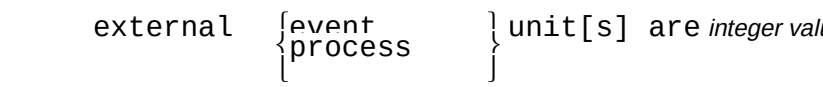
processes statement is required for internal processes. SIMSCRIPT II.5 permits one **external processes** statement for each process class.

If a process is declared as external, and is not declared using the **processes** statement, the system will use a process notice having only the nine system-defined attributes. However, if a process is external and is also declared in the **processes** statement, the process can be triggered both internally and externally and the process notice can have more than the nine system-defined attributes. The process notice will have the same name as the declared processes, and is prepared each time an external process data card containing the process name is read from an external unit.

The **external processes** statement can appear only in the preamble.

2.37 EXTERNAL ... UNITS Statement

The **`external ... units`** statement names input units from which external event and process data are to be read.



Keywords	Synonyms
`external`	`exogenous`
`are`	`is`

EXAMPLES:

```
external event unit is 1
```

Names **`1`** as the unit from which external event and process data are to be read.

```
external process units are 7 and EX.UNIT
```

Names **`7`** and **`EX.UNIT`** as the units from which external event and process data are to be read.

The **`external ... units`** statement declares units from which external event or process data are to be read. Integer constants or unsubscripted variables can name devices, but variables must be initialized to valid device numbers before the start of simulation. If this statement is omitted, SIMSCRIPT II.5 assumes that external event and process data are on the standard input unit. When using several input devices, the standard input unit must be included if that unit is a source of external events or processes.

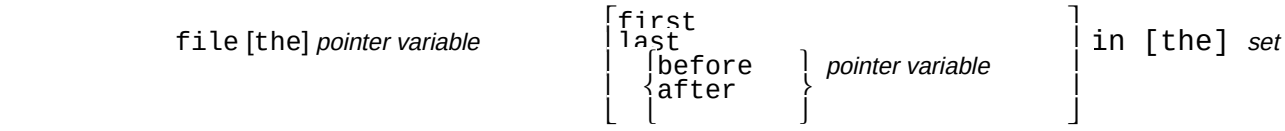
Data for events and processes may be interspersed on the same external unit. All data appearing on the unit must be arranged in order of increasing values of **`time.a`**.

At the start of simulation, the first external event or process record is read from each external input unit. The system creates event or process notices, stores the **`time.a`** and **`eunit.a`** information, and files all the notices in the event set before selecting the first event or process to be executed. Subsequent external event (process) records are read from whatever external unit is required.

The **`external ... units`** statements can appear only in the preamble.

2.38 FILE Statement

The **file** statement files a permanent or a temporary entity in the named set, which can be a **fifo, lifo**, ranked set, or a generalized set whose organization is determined by the programmer.



Keyword	Synonym
the	this

EXAMPLES:

```
file FLIGHT in WAITING.LINE
```

Files the entity, whose identification is the value of global variable **FLIGHT**, in the set named **WAITING.LINE** (owned by the system), according to the discipline declared in a **define ... set** statement (or on a **fifo** basis if no discipline is declared).

```
file this PATRON first in RESERVATIONS
```

Files the entity, whose identification is the value of variable **PATRON**, first in the set named **RESERVATIONS** owned by the system.

```
file the STOCK.CERTIFICATE after UNITED in the PORTFOLIO(CLIENT)
```

Files the entity, whose identification is the value of variable **STOCK.CERTIFICATE**, after the entity whose identification is the value of variable **UNITED** in the set named **PORTFOLIO** owned by the entity **CLIENT**.

When a program begins execution, all sets are empty. The **file** statement is used to alter set pointers to file an entity in the named set during execution. Entities filed in sets can be either permanent or temporary. When filing in sets declared as **fifo, lifo**, or ranked in a **define ... set** statement, the declared set discipline is the default if the **first, last, before**, and **after** phrases are omitted from the **file** statement. Typically, these phrases are used only for generalized sets organized according to a discipline determined by the programmer. The **file** statement can appear in any routine, but it cannot be included in the preamble.

2.38.1 FIRST, LAST, BEFORE, and AFTER Phrases

When **file ... first** is specified, the entity is filed at the beginning of the set. If **file ... last** is specified, however, the entity is filed at the end of the set. **Before** and

after phrases file an entity before or after an entity already in the set. Either of these phrases require the identification of the entity to be filed, as well as the identification of the entity before which or after which that entity is to be filed.

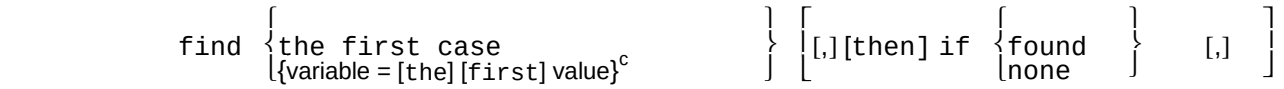
2.38.2 Arithmetic Expressions

An arithmetic expression in this statement must evaluate to an entity identification number. This is either the address of a temporary entity record from a prior **create** statement, or an integer index denoting one of the **N.*entity*** (number of entities in an entity class) permanent entities.

Note that an entity identification number can itself be held in an attribute that is referenced by another identification number. Entities can be nested to any level, but a nested expression must evaluate to an entity identification number.

2.39 FIND Statement

The **find** statement searches for the first value, in a group of values, that satisfies conditions in designated logical expressions. This statement must be controlled by a **for** phrase with a selection phrase, but cannot be within a **do ... loop** construct. The optional **if found** or **if none** phrase directs control after the control phrase has been completed, depending on the outcome of the **find** search.



EXAMPLES:

```
for I = 1 to 10, with X(I) < 5, find the first case
```

Searches for the first value that is less than 5 in the one-dimensional array **X**. Variable **I** will have the index of the first such value.

```
for I back from N to 1, with SALES(I) ls QUOTA, find LAST.VALUE =
first 1, if found, go to REVIEW
```

Starting at the end of the one-dimensional array **SALES**, searches for the first value that is less than the value of **QUOTA**. The index variable value is assigned to **LAST.VALUE**. Control transfers to '**REVIEW**' if any such value is found. Otherwise, the statement following a subsequent **else** statement is executed.

```
for each CITY, when POPULATION(CITY) gr 500000, find LARGE = CITY
```

Searches entities of permanent entity class **CITY** for the first entity having a value of **POPULATION** greater than 500,000, and assigns the index variable value to variable **LARGE**.

```
for each STOCKHOLDER, for each STOCK.CERTIFICATE in PORTFOLIO, when
CORPORATION(STOCK.CERTIFICATE) = "TWA", find HOLDER = STOCKHOLDER
and NUMBER = STOCK.CERTIFICATE
```

Searches each set named **PORTFOLIO**, owned by permanent entities of entity class **STOCKHOLDER**, for the first entity whose value of **CORPORATION** is the alphanumeric literal **TWA**. The value of index variable **STOCKHOLDER** is assigned to variable **HOLDER**, and the value of entity identification variable **STOCK.CERTIFICATE** is assigned to variable **NUMBER**.

A **find** statement must always be controlled by at least one **for each (*class*)**, **for ... of (*set*)**, or **for ... to (*index*)** phrase that has an appended phrase, such as a **with** phrase. The **for** phrase must not control a **do ... loop** construct, however. This statement searches for the first value that satisfies conditions specified in any appended

phrases and sets a variable equal to an expression. The **find** statement cannot appear in the preamble, but can be included in any routine.

The **for** phrase steps the index variable (specified in the **for** phrase) through the group of values, and searches for the first value that satisfies the criteria. When the criteria are satisfied, generation of index values terminates and the expression is computed, based on the selected index assigned to the named variable. If more than one **for** phrase is specified, the **find** statement usually includes more than one variable and arithmetic expression as shown in the fourth example above.

2.39.1 Alternative Forms

The alternative, **find the first case**, can be used when no expression is to be computed. This form bypasses generation of a **let** statement that assigns the value of the "found expression" to the specified variable. The other alternative, **find** ***variable*** **=** ***expression*****,** evaluates the arithmetic expression and assigns that value to the variable in the **find** statement. This evaluation occurs when the first value is found for which the logical expression is true.

Note: If you select a variable named **FIRST**, the optional word **FIRST** must also be specified in order to compile correctly.

2.39.2 IF FOUND and IF NONE Phrases

A **find** statement can include either an **if found** or an **if none** phrase that yields a true or false condition. All rules for **if** statements (e.g., the use of **else** and **always**) apply to these statements.

2.40 FOR EACH (*class*) Phrase

The **for each (*class*)** phrase causes a program segment to be executed for each entity of a permanent entity class.

```
for each { permanent entity } [called pointer variable] [,]
         { resource         }
```

Keyword	Synonyms
each	every
	all

EXAMPLES:

```
for each AIRPORT
```

Executes controlled statements for each entity of entity class **AIRPORT**. Global variable **AIRPORT** is automatically set to the entity indices.

```
for every AIRPORT called STRIP
```

Executes controlled statements for each entity class **AIRPORT**. Variable **STRIP** is automatically set to the entity indices.

```
for each CITY, for every AIRPORT called JET, with NO.RUNWAYS(JET)
> = 5, while COUNTRY(CITY) equals "US"
```

Executes controlled statements for each entity of entity class **CITY**, and for each entity of class **AIRPORT** if the value of attribute **NO.RUNWAYS** is greater than 5. Variables **CITY** and **JET** are automatically set to the respective entity indices. The controlled statements are executed as long as the value of attribute **COUNTRY** is the alpha-numeric literal **US**.

The **for each (*class*)** phrase steps through all entities of a permanent entity class, enabling controlled statements to be executed for each entity of that class. Permanent entities, which have their attributes stored as arrays, are indexed sequentially from 1 through **N.*entity***, and the short form of the **for each (*class*)** phrase is equivalent to:

```
for ENTITY = 1 to N.ENTITY
```

When this short form is used, the sequential index values are assigned to the global variable having the same name as the entity class. If the **called** phrase is included, however, the

index values will be assigned to the variable named in the phrase, and the **`for each (`*`class`*`)`** phrase is then equivalent to:

```
for variable = 1 to N.entity
```

The **`for each (`*`class`*`)`** phrase can appear in any routine, but it cannot be included in the preamble.

2.40.1 Nested FOR EACH (*class*) Phrases

`For each (`*`class`*`)` phrases can be nested within other **`for each (`*`class`*`),for ... of (`*`set`*`)`**, and **`for ... to (`*`index`*`)`** phrases. When two **`for each (`*`class`*`)`** phrases are nested, for example, the first **`for`** phrase controls the outer loop, and the second **`for`** phrase controls the inner loop. When computing values of index variables, the inner index variable is stepped through its entire range of values for each value of the outer index variable. The controlled statements are executed each time.

2.40.2 WITH, UNLESS, WHILE, and UNTIL Phrases

`With, unless, while`, and **`until`** phrases can be appended to a single **`for each (`*`class`*`)`** phrase, as well as to nested **`for`** phrases. When phrases follow **`for`** phrases, each **`unless`** and **`with`** applies to the **`for`** phrase immediately preceding it, but each **`while`** and **`until`** phrase applies to all preceding **`for`** phrases. SIMSCRIPT II.5 permits any combination of phrases to be appended, and allows more than one of each phrase type.

2.41 FOR ... OF (*set*) Phrase

The **`for ... of (set)`** phrase enables a program segment to be executed for all entities stored in the named set.

```
                          ⎡ ⎧           ⎫                ⎤
for  each entity pointer  ⎢ ⎨from       ⎬ entity pointer ⎥ of set [in reverse order] [,]
                          ⎣ ⎩after      ⎭                ⎦
```

Keywords	Synonyms
`each`	`every`
	`all`
To begin processing with the entity identified by the expression:	
`from`	--
To begin processing with the entity that follows the identified entity:	
`after`	--

EXAMPLES:

```
for each FLIGHT of DEPARTURES
```

Executes controlled statements for each entity in the set named **`DEPARTURES`**. **`FLIGHT`** will contain the entity identification number for each iteration.

```
for each RUNWAY after 6 of JET.RUNWAYS
```

Executes controlled statements for each entity filed in the set named **`JET.RUNWAYS`**, starting with the permanent entity after the entity whose index is 6. The variable **`RUNWAYS`** contains the indices of the entities in the set.

```
for each AIRPORT, for each FLIGHT of DEPARTURES in reverse order,
unless DESTINATION(FLIGHT) equals "ASIA"
```

Executes controlled statements for each entity called **`FLIGHT`** filed in the set named **`DEPARTURES`** at each airport. Set members are processed in reverse order, and no statements are executed if the value of attribute **`DESTINATION`** of a **`FLIGHT`** is **`ASIA`**.

All the members of a set can be processed with statements controlled by a **`for ... of (`*`set`*`)`** phrase. Set members can be temporary entities, permanent entities, or both. This phrase selects set members from first to last (i.e., in order of their ranking) and assigns the entity identification numbers as values of the named variable. A set can also be stepped through backward by including the optional phrase **`in reverse order`**. If the set is emp-

ty, the system bypasses the program segment controlled by the **for ... of (*set*)** phrase. This phrase can appear in any routine, but it cannot be included in the preamble.

The **for ... of (*set*)** phrase can specify that set members be processed starting from, or after, a particular entity. In either case, the entity identification must be included in the phrase, and the system terminates the program with an error message if the identified member is not in the set.

2.41.1 Nested FOR ... OF (set) Phrases

For ... of (*set*) phrases can be nested within other **for each (*class*), for ... of (*set*)**, and **for ... to (*index*)** phrases. When two **for ... of (*set*)** phrases are nested, for example, the first **for** phrase controls the outer loop, and the second **for** phrase controls the inner loop. When computing values of index variables, the inner index variable is stepped through its entire range of values for each value of the outer index variable. The controlled statements are executed each time.

2.41.2 WITH, UNLESS, WHILE, and UNTIL Phrases

With, unless, while, and **until** phrases can be appended to a single **for ... of (set)** phrase, as well as to nested **for** phrases. When phrases follow **for** phrases, **each unless** and **with** applies to the **for** phrase immediately preceding it, but each **while** and **until** applies to all preceding **for** phrases. SIMSCRIPT II.5 permits any combination of phrases to be appended, and allows more than one of each phrase type.

2.41.3 Mechanism of FOR ... OF (*set*)

The **for ... of (*set*)** phrase works as follows:

```
   let variable = f.set
   go to test
'again'
   let variable = v
'test'
   if variable = 0, go out
   else
      let v = S.set(variable)
      Controlled statement
      go to again
'out'
```

F.*set* will be subscripted for a subscripted set. A group of controlled statements must be enclosed within a **do ... loop** construct. The value of the variable is retained when control transfers out of the loop.

2.42 FOR ... TO (*index*) Phrase

The **`for ... to (`*index*`)`** phrase, which is a loop control statement, increments the value of a variable for each execution of a program segment.

`for` *variable* { `=` / `back from` } *quantity* `to` *quantity* [`by` *quantity*] [,]

EXAMPLES:

```
for I = 1 to 10
```

Steps index variable **`I`** through 1, 2, ..., 10, using increments of 1.

```
for I back from 10 to 0 by 2
```

Steps index variable **`I`** through 10, 8, ..., 0, using decrements of 2.

```
for N = -0.5 to 0.7 by 0.1
```

Steps index variable **`N`** through -0.5, -0.4, ..., 0.0, 0.1, ..., 0.7, using increments of 0.1.

```
for ROW = 1 to NO.FLIGHTS.FLOWN, for COLUMN = 1 to NO.DAYS.IN.PERIOD
```

Steps index variable **`ROW`** through 1, 2, ..., **`NO.FLIGHTS.FLOWN`**, and index variable **`COLUMN`** through 1, 2, ..., **`NO.DAYS.IN.PERIOD`** for each value of **`ROW`**.

```
for J = A to B by 2 * DELTA, for K = X to Y by EPSILON / 3
```

Steps index variable **`J`** through the values of **`A`** to **`B`**, using increments of **`2 * DELTA`**, and steps index variable **`K`** through the values of **`X`** to **`Y`**, using increments of **`EPSILON/ 3`**, for each value of **`J`**.

The **`for ... to (`*index*`)`** phrase controls the number of times a single statement, or a program segment, is to be executed. It provides an index operation by incrementing the value of a variable for each execution of the controlled statements. *Quantity*$_1$ and *quantity*$_2$ define the range of values assumed by the index variable, while *quantity*$_3$ defines the increment. The **`for ... to (`*index*`)`** phrase is diagrammed in figure 3.

To control a program segment (group of statements), the segment is enclosed in a **`do ... loop`** construct. This phrase can appear in any routine, but it cannot be included in the preamble.

The following rules and characteristics apply to the **`for ... to (`*index*`)`** phrase:

Forward Stepping:

```
   let variable = quantity1
   go to TEST
'AGAIN'
   let variable = variable + quantity3
'TEST'
   if variable > quantity2, go to OUT
   else
      controlled statement
      go to AGAIN
'OUT'
```

Backward Stepping:

```
   let variable = quantity1
   go to TEST
'AGAIN'
   let variable = variable - quantity3
'TEST'
   if variable < quantity2, go to OUT
   else
      controlled statement
      go to AGAIN
'OUT'
```

Figure 3. `For ... to` (*index*) Phrase Execution

1. Mode conversions are automatically performed if all quantities do not have the same mode.
2. If any quantity is real, all computations required to compute values of the index variable will be real.
3. ***Quantity*$_2$** and ***quantity*$_3$** are computed each time the loop is repeated.
4. The index variable, ***quantity***$_2$, and ***quantity*$_3$** can be recomputed within a loop. Recomputing values can affect the subsequent index variable values, and can affect computations performed by the program.
5. A controlled statement is not executed if its terminating condition is satisfied initially. For example, the statement **`for i = 1 to n`** does not execute the controlled segment if **`n`** equals zero.

6. All rules that apply to the incremental option pertain to the **back from** option, the only difference being the direction in which the index variable changes value.

7. Program control can transfer in and out of loops as long as each transfer recognizes the organization of the **for ... to (*index*)** phrase. The value of the index variable is retained when control transfers out of the loop via a **go to** statement or when the loop is exhausted.

2.42.1 Nested FOR ... TO (*index*) Phrases

Any number of **for ... to (*index*)** phrases can be nested for arrays of more than one dimension. When **for ... to (*index*)** phrases are nested, the first **for** phrase controls the outer loop, and the second **for** phrase controls the inner loop. When computing values of **I** and **J**, for example, the inner statement is stepped through its entire range of values for each value of the outer index variable. The controlled statements are executed each time. Index variables of outer statements can appear in any expression of the inner statements because their values are defined within these statements.

2.42.2 WITH, UNLESS, WHILE, and UNTIL Phrases

With, unless, while, and **until** phrases can be appended to a single **for ... to (*index*)** phrase, as well as to nested **for each (*class*)**, **for ... of (*set*)**, and **for ... to (*index*)** phrases. When phrases follow **for** phrases, each **unless** and **with** applies to the **for** phrase immediately preceding it, but each **while** and **until** applies to all preceding **for** phrases. SIMSCRIPT II.5 permits any combination of phrases to be appended, and allows more than one of each phrase type.

2.43 GO TO Statement

The **go to** statement transfers program control to the statement having the specified label.

go [to] *label* [(*integer value*)]

Note: ***label[(value)]*** may be enclosed within apostrophes.

EXAMPLES:

```
go START
```

Directs program control to the statement preceded by the label **START**.

```
go to 'FINISH'
```

Directs program control to the statement preceded by the label **FINISH**. Apostrophe characters are optional.

```
go to 'POINT(INDEX+BASE-1)'
```

Directs program control to the statement preceded by the label **POINT**, subscripted by the value of **(INDEX+BASE-1)**. Apostrophe characters are optional.

The **go to** statement directs program control to the statement preceded by the specified label. This statement can appear in any routine, but it cannot be included in the preamble. Two distinct labels, which are considered to be equivalent, can identify the same statement. Equivalent labels are useful for program segments that have not yet been written, or when segments can be included or omitted without destroying the logic.

2.43.1 Subscripted Labels

Labels can be subscripted. A subscript must be an expression, enclosed in parentheses, that evaluates to an integer value. The complete subscripted label can be optionally enclosed in single apostrophe characters. When the system executes a **go to** statement having a subscripted label, control is transferred to the statement preceded by the same label and subscript equal to the integer value of the expression. Subscripts need not start with the number 1 and need not be in consecutive order within a program.

Subscripted labels are useful when transferring control to various segments in a frequently modified program because they permit labels to be added or deleted without altering the **go to** statements. It is often more economical to use subscripted labels in **go to** statements (rather than **call** statements) to produce the effect of a subroutine without the expense of recursion.

2.43.2 Error Conditions

An error condition will occur if the value of the expression is not within the range 1 to the number of labels, or if a label is undefined. System action on this condition is undefined. An implementation may or may not print an error message, terminate, or continue incorrectly at some unknown point.

2.44 GO TO ... PER Statement

The **go to ... per** statement transfers control to a labeled statement or one of several labeled statements in a list according to the integer value of the transfer expression.

go [to] *LABEL*^{OR} per *QUANTITY*

Note: *LABEL* may be enclosed in apostrophes.

Keyword	Synonym
or	, (comma)

EXAMPLES:

```
go X, Y per A
```

Directs program control to either label **X** or label **Y**, depending on whether the value of **A** is 1 or 2.

```
go to 'ONE' or 'TWO' or 'THREE' per M * N + COUNT / N
```

Directs program control to labels **ONE**, **TWO**, or **THREE**, depending on whether **M * N + COUNT / N** yields a value of 1, 2, or 3. Single apostrophe characters enclosing the label names are optional.

```
go to ADD, SUBTRACT, MULTIPLY or DIVIDE per OPERATOR.CODE
```

Directs program control to labels **ADD**, **SUBTRACT**, **MULTIPLY**, or **DIVIDE**, depending on whether **OPERATOR.CODE** is 1, 2, 3, or 4.

The **go to ... per** statement transfers program control to the first label, to the second label, or to the **n**th label according to the value of the expression. That is, program control transfers to the statement preceded by the first label if the expression yields a value of 1, to the statement preceded by the second label if the expression yields a value of 2, and to the statement preceded by the **n**th label if the expression yields a value of **n**. SIMSCRIPT II.5 rounds a real value.

Two or more distinct labels, which are considered to be equivalent, can identify the same statement. Equivalent labels are useful for program segments that have not yet been written, or when segments can be included or omitted without destroying the logic. Conversely, the same label can appear several times in a **go to ... per** statement. The **go to ... per** statement may appear in any routine, but not in the preamble.

2.44.1 Error Conditions

An error condition will occur if the value of the expression is not within the range 1 to the number of labels, or if a label is undefined. System action on this condition is undefined. An implementation may or may not print an error message, terminate, or continue incorrectly at some unknown point.

2.45 HERE Statement

The **here** statement acts as a label for nearby jump statements.

```
here
```

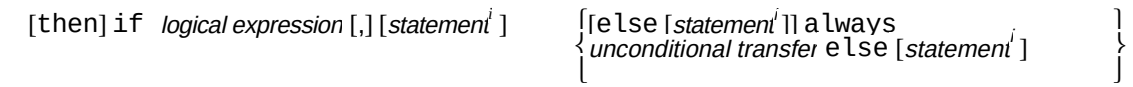
Only one form of this statement exists. It consists of exactly one word.

The **here** statement provides a "proximate label" for preceding **jump ahead** statements and for following **jump back** statements in the same routine. Several **jump** statements may refer to the same **here** statement, and several **here** statements may appear in a single routine.

The **here** statement can appear in any routine, but not in the preamble.

2.46 IF ... ELSE ... ALWAYS Construct

The **`if ... else ... always`** construct determines whether a specified logical expression is true or false. If the condition is true, execution continues with the succeeding statement until the corresponding **`else`** statement. If the condition is false, execution transfers to statements following the corresponding **`else`** statement.



Keywords	Synonyms
`else`	`otherwise`
`always`	`regardless`
	`endif`

EXAMPLES:

```
if X = 0.0
```

Executes succeeding statements when the value of variable **`X`** equals 0 (logical expression is true). Executes the program that follows **`else`** when the expression is false.

```
if b**2 > 4*a*c
```

Executes the succeeding statements when **`b**2 > 4ac`**. Executes the program segment that follows **`else`** when the logical expression is false.

```
if mode is alpha and card is not new
```

Executes the succeeding statements when both logical expressions **`mode is alpha`** and **`card is not new`** are true. Executes the program segment that follows **`else`** when either logical expression is false.

```
if NAME(DOG) eq "FIDO"
 add 1 to FIDO.COUNT
always
```

If attribute **`NAME`** of entity **`DOG`** contains the text literal **`"FIDO"`**, the value of variable **`FIDO.COUNT`** is increased by one. Regardless, the program segment that follows `always` is executed.

```
if X = 0
   for each TEMP in S
   with TYPE (TEMP) = 1
   find the first case
then if found
   let Y = 1
always
```

When **X** equals 0, this statement causes the set **S** to be searched for a member with **TYPE = 1**. If such a member exists, **Y** is set equal to 1.

The structured **if ... else ... always** construct is used to program decisions that transfer control according to the truth value (true or false) of logical expressions. The **if** phrase is immediately followed by a group of statements to be executed if the logical expression is true. It may also be followed by an **else** statement and an alternative group of statements to be executed if the logical expression is false (see figure 4). This construct can appear in any routine, but not in the preamble.

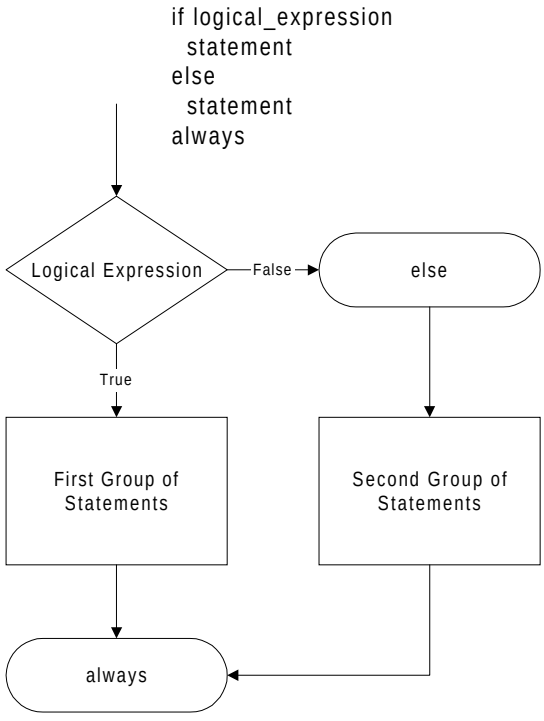


Figure 4. Structured if ... else ... always Construct

Strict adherence to structured programming would require that the first group of statements not end with an unconditional transfer (e.g., a **go to, return**, or **stop** statement). When no unconditional transfer exists, an **always** statement must follow the second group of statements (see figure 4). If the first group of statements ends with an unconditional transfer, no corresponding **always** statement is allowed. In such a case, **else** should be replaced by **otherwise** (see figure 5).

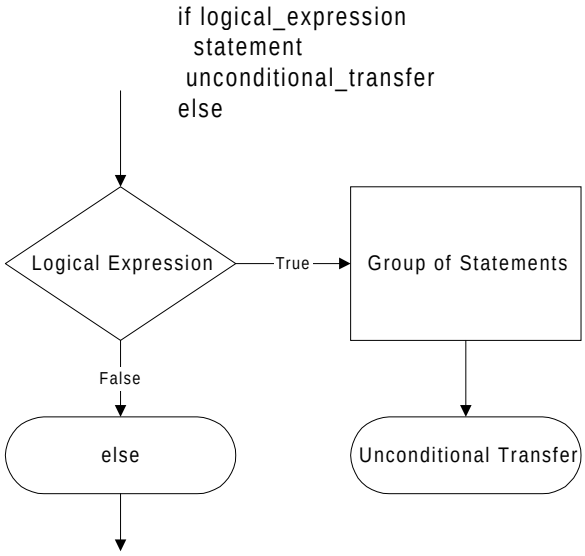


Figure 5. Structured `if ... else ... always` Construct with Unconditional Transfer

If there is no alternative group of statements to be executed when the logical expression is false, the **else** statement can be omitted. (In such a case, some programmers replace **always** with **regardless**, but this is not recommended.)

2.46.1 Nested IF ... ELSE ... ALWAYS Constructs

If ... else ... always constructs can be nested by inserting other **if ... else ... always** constructs. An **always** statement is required for *each* **if ... else ... always** construct. Frequently, when **if ... else ... always** constructs are nested, the optional keyword **then** can precede **if** and redundant **always** statements can be eliminated. However, the **then if** applies only to nested logical tests in which the false condition is the same for each test. When any **then if** has a false condition, control transfers to the `always` statement corresponding to the previous **if ... else ... always** construct. Figure 6 illustrates the logic of **then if ... else ... always** constructs.

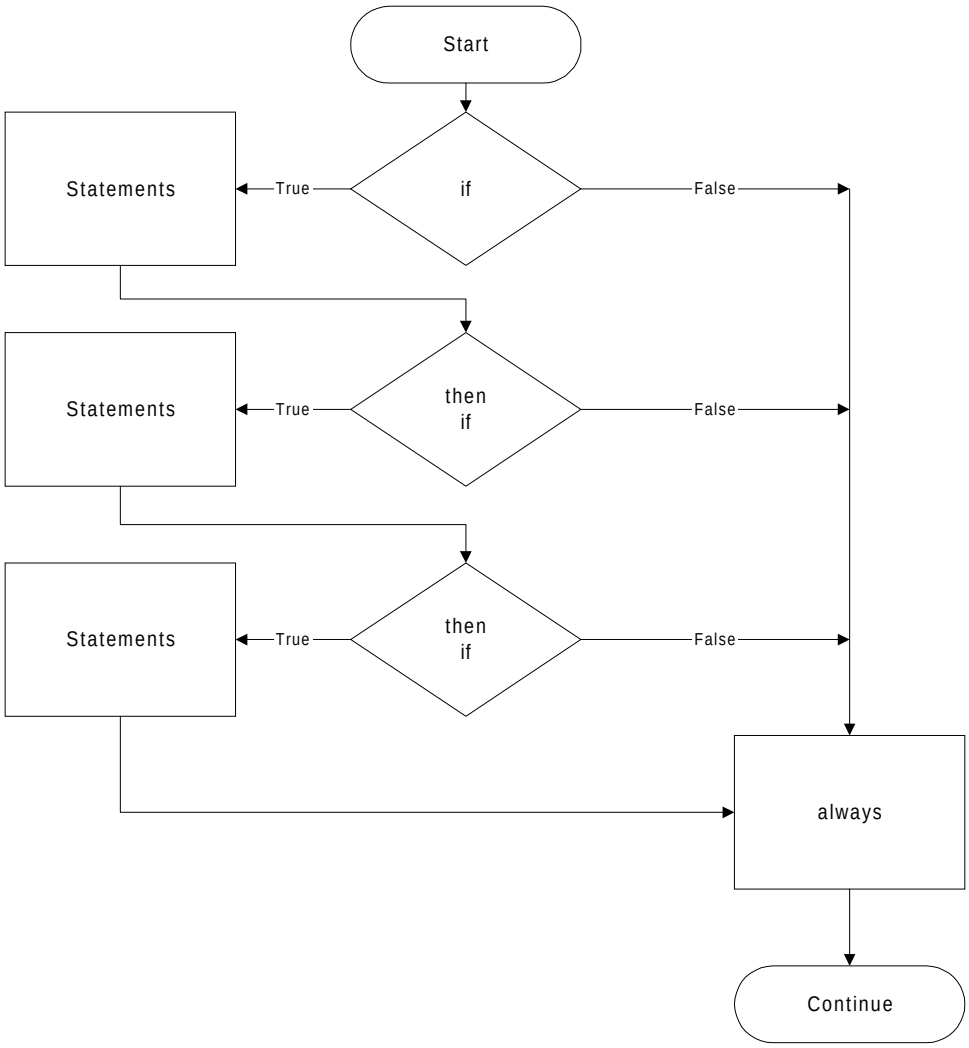


Figure 6. `Then if` Statements

A useful alternative to nested **`if ... else ... always`** constructs is the use of the **`go to`** statement with subscripted labels. While "pure" structured programming would enforce the rule of single-entry/single-exit, this alternative (described under **`go to`**) may be less confusing in some cases.

2.47 INTERRUPT Statement

The **interrupt** statement takes a process in the active state and places it in the interrupted state.

```
interrupt [the [above]] process [called pointer variable]
```

Keyword	Synonym
the [above]	this

EXAMPLES:

```
interrupt the GENERATOR
```

Interrupts a **GENERATOR** that is in the pending list as a result of a **wait** statement. The amount of time remaining until the **GENERATOR** would have been resumed is captured and recorded as a process attribute **time.a**. If the **GENERATOR** is later resumed, this value is used to determine the new waiting period.

```
interrupt SHIP called OTHER.SHIP
```

Interrupts the **SHIP** that has been identified by a pointer variable, **OTHER.SHIP**, by a previous assignment. **OTHER.SHIP** is in the pending list as a result of a **work** statement, and the amount of time remaining until **OTHER.SHIP** would have been resumed is captured and recorded as a process attribute **time.a**. If **OTHER.SHIP** is later resumed, this value is used to determine the new period of work.

Any process in the active state (i.e., in a **work** statement) may be interrupted by any other process, event, or routine. The amount of time remaining to be worked is placed in the **time.a** attribute. If the process is resumed, it works the remaining time before executing subsequent statements.

The **interrupt** statement can appear in any routine, but not in the preamble.

2.48 JUMP Statement

The **jump** statement provides a label-free transfer of control.

```
jump   [ahead  ]
       {back   }
       [       ]
```

EXAMPLES:

`jump ahead`

Causes execution to proceed with the next **here** statement in the program.

`jump back`

Causes execution control to pass to the nearest previous **here** statement in the program.

The **jump** statement can appear in any routine, but not in the preamble.

2.49 LAST COLUMN Statement

The **`last column`** statement designates the last card column used for statements. Characters beyond that column will be disregarded by the SIMSCRIPT II.5 compiler.

```
last column is integer
```

Keyword	Synonym
`is`	`=`

EXAMPLES:

```
last column is 72
```

Designates that column 72 is the last card column containing statements.

```
last column = 60  ''PREAMBLE STATEMENT
```

Designates that column 60 is the last card column containing statements. Commentary text follows the two apostrophe characters.

The **`last column`** statement specifies that program statements do not extend beyond the designated card column. This feature enables programmers to identify cards or to number them sequentially using the right-hand columns. SIMSCRIPT II.5 disregards characters in succeeding card columns during the compilation process, but prints them on the listing. If this statement is omitted, the compiler assumes that all 80 card columns are used for statements.

This statement can appear anywhere in a preamble, but it cannot be included in a routine. Whenever a **`last column`** statement appears in the preamble, the number of card columns pertaining to program statements can change. The final **`last column`** statement, however, applies to all routines that follow the statement.

2.50 LEAVE Statement

The **leave** statement causes exit from within a **do ... loop** construct of code.

```
leave
```

Only one form of this statement exists. It consists of exactly one word.

The **leave** statement is used within a **do ... loop** construct to cause premature exit from the loop. Execution of a **leave** statement causes control to pass to the statement following the next **loop** or **repeat** statement in the program. Controlling index variables in the **for** phrases driving the loop are left unchanged.

The statements **cycle** and **leave** clarify programs by eliminating labels, and provide the concept of local labels to the system. These features are especially useful when coupled with the **substitute** statement features of SIMSCRIPT II.5.

2.51 LET Statement

The **let** statement assigns the value of an expression to a variable. If the variable is of integer mode and the expression is real, the result is rounded before storing.

```
let variable = variable
```

EXAMPLES:

```
let X = X + 1
```

Sets variable **X** equal to **X + 1**.

```
let X1 = (-B + sqrt.f(B**2 - 4 * A * C)) / (2 * A)
```

Sets variable **X1** equal to the positive root of a quadratic equation.

```
let CAPACITY(FLIGHT) = FIRST.CLASS.SEATS(FLIGHT) +
TOURIST.SEATS(FLIGHT) + ECONOMY.SEATS(FLIGHT)
```

Sets attribute **CAPACITY** equal to the sum of the values of **FIRST.CLASS.SEATS**, **TOURIST.SEATS**, and **ECONOMY.SEATS**, for entity class **FLIGHT.**

The **let** statement evaluates the expression specified to the right of the equal sign, performs any required computations, and assigns the computed results to the variable named to the left of the equal sign. In this statement, the equal sign is an assignment operator specifying the replacement of the current value with a new value. A **let** statement can appear in any routine, but it cannot be included in the preamble.

If the expression and the variable differ in mode, SIMSCRIPT II.5 converts the expression to the mode of the variable before setting the variable equal to the value of the expression. Conversion from integer to real is accomplished simply by taking the whole number integer value and converting it to a real number of the same value. Real-to-integer conversion involves rounding the value of the expression (by adding -0.5 if it is negative or +0.5 if positive, and truncating) to a whole number before storing the value in the variable.

2.52 LIST Statement

The **list** statement labels and displays, in a standard format, values of expressions and elements of one- and two-dimensional arrays.

EXAMPLES:

```
list NUMBER
```

Displays the value of variable **NUMBER**.

```
list A, B, C, B * C / LENGTH
```

Displays values of variables **A, B**, and **C**, and of the expression **B * C / LENGTH**.

```
list COLUMN, VERTICAL, and TABLE
```

Displays elements of the one-dimensional arrays **COLUMN** and **VERTICAL**, and of the two-dimensional array **TABLE.**

```
list GRAPH, PI.C * R**2, RADII
```

Displays elements of the two-dimensional array named **GRAPH**, the value of expression **PI.C * R**2**, and elements of the one-dimensional array **RADII**. **PI.C** is a system constant whose value is π.

The **list** statement can appear in any routine, but not in the preamble. The exact format of the output is implementation-dependent.

2.53 LIST ATTRIBUTES Statement

The **`list attributes`** statement displays attribute values for one entity of a permanent or temporary entity class.

```
list attributes of entity [called pointer variable]

[          [ tape  ]                  ]
|using     | unit  |   integer value  |
[          [       ]                  ]
```

EXAMPLES:

```
list attributes of FLIGHT
```

Displays values of all attributes for an entity of the entity class **`FLIGHT`**. The entity identification is contained in the global variable named **`FLIGHT`**.

```
list attributes of CITY called NEW.YORK
```

Displays values of all attributes for an entity of the entity class **`CITY`**. The entity index is contained in the variable named **`NEW.YORK`**.

The **`list attributes`** statement, which displays attribute values for one entity of the designated entity class, can be used for both permanent and temporary entities. This statement displays data in a standard system format rather than in a programmer-defined format. The **`list attributes`** statement is convenient for debugging purposes and whenever tailored reports are not required. Some attributes, the set pointers and random variables, are not printed by the **`list attributes`** statement. However, **`N.`*`set`*** (number in set) and **`M.`*`set`*** (set membership) attributes are printed. The **`list attributes`** statement can appear in any routine, but it cannot be part of the preamble.

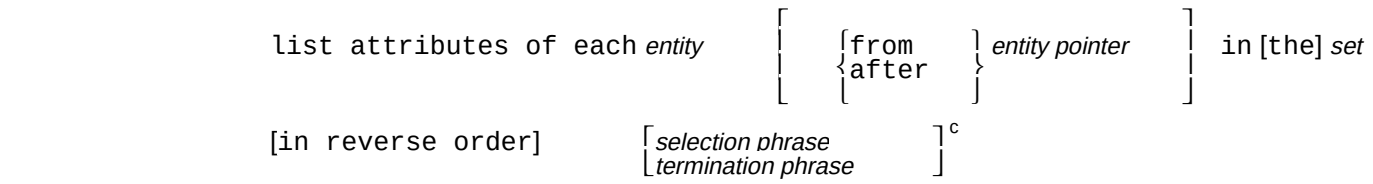
When the **`called`** phrase is omitted, SIMSCRIPT II.5 uses the global variable having the same name as the entity class to locate the specific entity whose values are to be listed. If the **`called`** phrase is included, however, the value of the expression is used to locate the entity.

2.53.1 Function Attributes

When using the **`list attributes`** statement, input/output operations (such as changing output device numbers) should not be performed in attribute functions invoked by this statement. Note also that function attributes invoked within a **`list attributes`** statement can affect control of the listing loop, causing incorrect printing.

2.54 LIST ATTRIBUTES OF EACH Statement

The **`list attributes of each`** statement displays attribute values for all entities in a permanent entity class, or displays values for all permanent or temporary entities filed in a named set.



Keywords	Synonyms
To begin processing with the entity identified by the expression:	
`from`	--
To begin processing with the entity that follows the identified entity:	
`after`	--
`in`	`of`
	`at`
	`on`
`the`	`this`

EXAMPLES:

```
list attributes of each CITY
```

Displays attribute values for all entities of the entity class **`CITY`**.

```
list attributes of each PATRON in RESERVATIONS(FLIGHT)
```

Displays attribute values for all entities of the entity class **`PATRON`** filed in the set named **`RESERVATIONS`**. **`FLIGHT`** is the identification number of the set owner.

```
list attributes of each FLIGHT in DEPARTURES(AIRPORT) in reverse order
```

Displays attribute values for all entities of the entity class **`FLIGHT`** filed in the set named **`DEPARTURES`**. **`AIRPORT`** is the identification number of the set owner. Values are displayed beginning with the last entity.

```
list attributes of each STOCK.CERTIFICATE from DELTA in
PORTFOLIO(STOCKHOLDER), until CORPORATION(STOCK.CERTIFICATE)
equals "TWA"
```

Displays attribute values for entities of the entity class **STOCK.CERTIFICATE** filed in the set named **PORTFOLIO**. **STOCKHOLDER** is the set owner. Values are displayed beginning with the entity whose identification number is assigned to the variable **DELTA**, and entities are processed until attribute **CORPORATION** contains the alphanumeric literal **TWA**.

The **list attributes of each** statement can display attribute values for all entities in a permanent entity class, or it can display attribute values for permanent or temporary entities filed in a set. This statement displays values in a standard system format rather than in a programmer-defined format (as in a **print** statement). The **list attributes of each** statement is convenient for debugging purposes and whenever tailored reports are not required. The attributes used for set pointers and random variables are not printed by the **list attributes of each** statement. However, **N.*set*** (number in set) and **M.*set*** (set membership) attributes are printed. The **list attributes of each** statement can appear in any routine, but it cannot be part of the preamble.

2.54.1 Output for LIST ATTRIBUTES OF EACH Statement

The **list attributes of each** statement prints attribute values in an implementation-defined format. Columns are printed across a page, with the attribute name above its column of values. A specific number of positions are allotted for each column, and as many columns as possible are printed across the page. Columns are continued on a succeeding page if the entity contains more attributes than can be printed across the page. Because a single heading is printed, the labeled output is meaningful only for sets with one entity class filed in them.

2.54.2 Function Attributes

When using the **list attributes of each** statement, input/output operations (such as changing output device numbers) should not be performed in attribute functions invoked by this statement. Note also that function attributes invoked within a **list attributes of each** statement can affect control of the listing loop.

2.55 LOOP Statement

See **do ... loop** construct.

2.56 MAIN Statement

The **main** statement signals the beginning of the main routine in a program.

```
main
```

Only one form of this statement exists. It consists of exactly one word.

The **main** statement, which is optional, informs the SIMSCRIPT II.5 compiler that succeeding cards contain statements of the main routine. Program execution begins with the first executable statement that follows **main**. If the **main** statement is omitted, the compiler assumes that any unlabeled routine is the main routine. In order to distinguish clearly the main routine from subroutines, it is recommended that the main routine be preceded by a **main** statement. A program can have only one **main** statement.

2.57 MOVE Statement

The **move** statement is used within monitoring routines either to access or to set the value of the monitored variable.

```
move     ⌈from ⌉   variable
         {to   }
         ⌊     ⌋
```

EXAMPLES:

```
move from NUMBER
```

Assigns the value of variable **NUMBER** to the monitored variable.

```
move to NUMBER
```

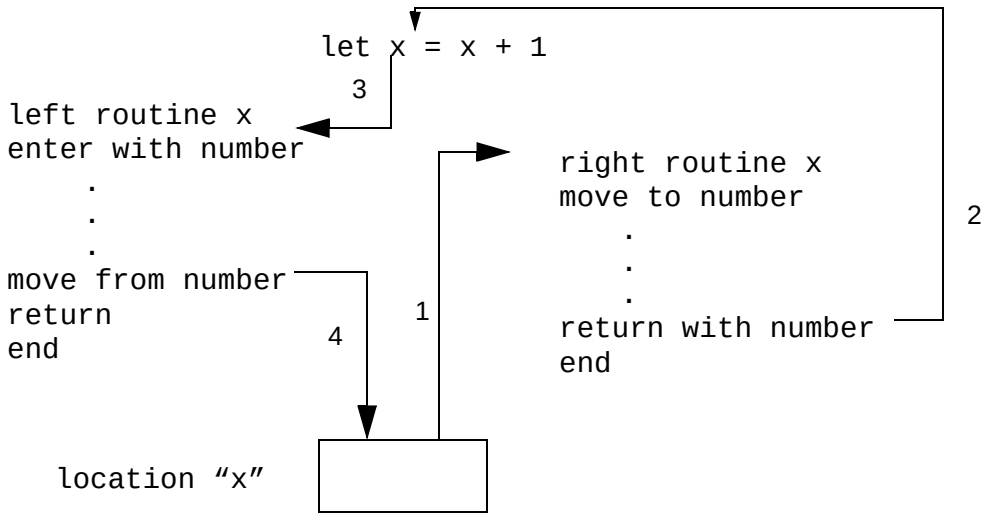
Assigns the value of the monitored variable to variable **NUMBER**.

```
move to LIST(I)   '' SET LIST TO MONITORED VARIABLE
```

Assigns the value of the monitored variable to the **I**th element of the one-dimensional array **LIST**. Commentary text follows the two apostrophe characters.

The **move** statement is used in monitoring routines either to access or to set the value of the monitored variable. The **move from** form of the statement is used in left-hand monitoring routines to assign the value of an expression to the monitored variable, while the **move to** form is used in right-hand monitoring routines to assign the value of the monitored variable to a variable within the routine. Figure 7 illustrates both forms of the **move** statement. Monitored variables must be declared in **define ... variable** statements in the preamble.

Typically, a value is transmitted to the left-hand monitoring routine by the **enter with** statement, computation or other processing is performed, and a value must be assigned to the monitored variable by the **move from** form of the statement. The **move to** form obtains the value of the monitored variable and makes it accessible to the right-hand monitoring routine. In effect, this statement can convert a conventional right-hand routine into a right-hand monitoring routine. After assigning the value of the monitored variable to the named variable, the named variable can be used in the right-hand monitoring routine like any other variable. A value is transmitted from a right-hand monitoring routine to the monitored variable with a **return** statement.



Legend:

1 In a right-hand monitoring routine, the value of the monitored variable is brought from memory with a **`move to`** statement.

2 The possibly-modified value produced by the right-hand routine is returned to the point of reference using the **`return with`** statement.

3 A left-hand monitoring routine obtains its left-function value via an **`enter with`** statement.

4 The value of a monitored variable is placed in memory using the **`move from`** statement.

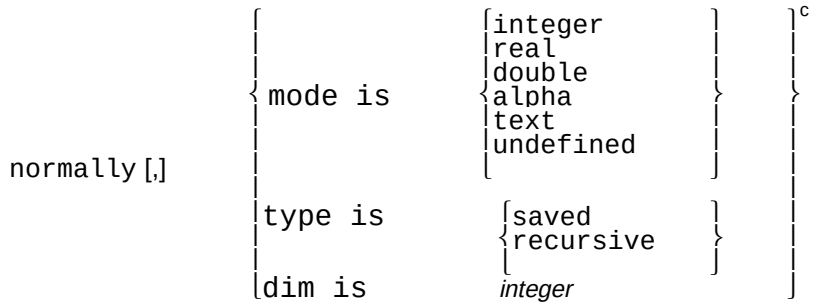
Figure 7. `MOVE` Statements

2.58 NEXT Statement

See **`cycle`** statement.

2.59 NORMALLY Statement

The **normally** statement declares general characteristics for values of variables, attributes, and functions, as well as defining the dimensionality of arrays.



Keyword	Synonym
is	=
dim	dimension

EXAMPLES:

```
normally mode is integer
```

Declares that generally values of variables, attributes, and functions have integer mode.

```
normally, mode = integer, dimension = 1
```

Declares that generally values of variables, attributes, and functions have integer mode and that variables are one-dimensional arrays.

```
normally, dim is 2, mode is real and type is recursive
```

Declares that generally variables are two-dimensional arrays, values of variables, attributes, and functions are real, and local variables are recursive.

The **normally** statement declares general characteristics for values of variables, functions, attributes, and arrays. It declares whether the mode is usually integer, real, double, alpha, or text, specifies whether local variables are saved or recursive, and defines the number of dimensions for arrays. These characteristics remain effective until superseded by subsequent **normally** statements or overridden by **define ... variable** statements. Phrases of the **normally** statement can appear in any desired order, or each declaration can be a separate statement.

2.59.1 NORMALLY and DEFINE ... VARIABLE Statements

Both **normally** and **define ... variable** statements can appear in the preamble to declare characteristics of the global environment, and in routines to declare characteristics

of local environments. In the preamble, **normally** statements can make general "background" declarations, while **define ... variable** statements can declare any exceptions and additions. Characteristics declared in a **define ... variable** statement override those appearing in **normally** statements. Characteristics of variables that are not defined in **define ... variable** statements are assumed to be the background characteristics as specified by the **normally** statements (or by default).

The last **normally** statement in the preamble applies to all routines, unless superseded by declarations within the routines themselves. In a subroutine or function, **normally** and **define ... variable** statements define the local environment, with definitions applying only to that routine. These statements can appear anywhere within a routine, but their relative order is important.

2.59.2 Mode

A phrase in the **normally** statement can declare the mode — integer, real, alpha, or text — for variables; the default is real. Alpha and text variables are treated as character strings, not as numerical quantities. Variables having any mode can be multidimensional, and local variables can be either saved or recursive. If the programmer declares the default mode as **undefined**, the compiler will generate a warning message for all uses of variable names that have not been explicitly defined. This is the recommended mode.

2.59.3 Saved and Recursive Variables

The terms "saved" and "recursive" apply only to local variables and arrays (arguments are automatically stored as recursive variables). Global variables are always saved. All local variables are zero the first time a routine is called. Thereafter, a recursive local variable has an initial value of zero each time the routine in which it appears is called, but a saved local variable retains the value stored when the routine was last executed. A **normally** statement in the preamble can declare whether local variables and arrays are generally saved or recursive, but a **normally** statement in a routine is effective only for the routine in which the **normally** statement appears. If a **type** phrase is not included in a **normally** statement, local variables and arrays are recursive by default.

2.59.4 Dimensionality

The dimensionality of arrays must be specified in the preamble or in routines with either a phrase of the **normally** statement or a **define ... variable** statement. When declared in a **define ... variable** statement, the dimensionality of arrays is permanent and cannot be superseded by other **normally** or **define ... variable** statements. If the dimensionality does not appear in a **normally** statement, variables are zero-dimensional.

Note: Variables whose dimensions do not appear in **define ... variable** statements take on the current background dimensionality. Thus:

```
normally dim = 2
```

```
define X as a real, saved variable
```

defines **X** as a two-dimensional, real, saved array.

2.60 NOW Statement

See **`call`** statement.

2.61 OPEN Statement

The **open** statememt opens a file for input or output and assigns it to a specified unit.

```
open [unit] unit for      ⎡input  ⎤ [options]
                          ⎢output ⎥
                          ⎣       ⎦
```

When the option “noerror” is given, an error (such as a non-existing file) does not result in a SIMSCRIPT II.5 runtime error, but in setting the system variable **ropenerr.v** to a value other than 0. This value can then be checked. Use the **close** statement to close a unit.

Note: On PC WindowsNT and Windows 95 platforms **unit**s can be assigned to files also by means of the **units.cfg** file which must be in the current directory.

EXAMPLE

```
function FILE.EXISTS.F(FNAME)
   define FNAME as text variable
   define SAVEDREAD.V as integer variable

      '' -- remember old READ.V since it gets changed by USE
   SAVEDREAD.V = READ.V
   open unit .FEX.UNIT for input, name = FNAME, noerror
   use .FEX.UNIT for input
   if ROPENERR.V <> 0
      close .FEX.UNIT
      READ.V = SAVEDREAD.V
      return .FALSE
   else
      close .FEX.UNIT
      READ.V = SAVEDREAD.V
      return .TRUE
   endif
end
```

2.62 Routine ORIGIN.R

When the calendar format is used, an origin must be provided with which the calendar time can be compared. This must occur before the **start simulation** statement is executed. The calendar date of the state of simulation is set by calling the system routine **origin.r** as follows:

```
call origin.r(month, day, year)
```

Integer values must be used for month, day, and year.

Because time is stored as a real value in the system variable **Time.v** and in attribute **time.a** of event notices, conversions must be made between the calendar time specifications and the computer representation values. When converting values, the system assumes

that Monday is the origin day, and that simulation starts at the beginning of that day (0000 hours). **`Time.v`** is set to zero at the start of simulation. The functions listed in table 19 convert the expressions representing the year, month, and day into cumulative simulation times, and vice versa. Routine **`origin.r`** must be executed before the functions are used.

Table 19. Time Conversion Functions

Name	Arguments	Value of the Function
`date.f`	Three expressions yielding integer values that are the month, day, and year. Example: `date.f`(6,30,91)	Current simulation day; an integer
`month.f`	Expression yielding a real value that is the cumulative simulation time. Example: `month.f`(539.5)	Current month, 1-12; an integer
`day.f`	Expression yielding a real value that is the cumulative simulation time. Example: `day.f`(539.5)	Day of current month, 1-31; an integer
`year.f`	Expression yielding a real value that is the cumulative simulation time. Example: `year.f`(539.5)	Current year; an integer

2.63 OTHERWISE Statement

See **`if ... else ... always`** construct**.**

2.64 PERFORM Statement

See **`call`** statement**.**

2.65 PERMANENT ENTITIES Statement

The **`permanent entities`** statement indicates that permanent entities are declared in the **`every`** statements that follow.

```
permanent entities [include entity^c]
```

Keyword	Synonym
`include`	`are`

EXAMPLES:

```
permanent entities
```

Indicates that permanent entities are declared in the **`every`** statements that follow.

```
permanent entities include COUNTRY '' country has no attributes
```

Indicates that permanent entities are declared in the **`every`** statements that follow, and that **`COUNTRY`** is an entity class having no attributes. Commentary text follows the two apostrophe characters.

The `permanent entities` statement, which can appear only in the preamble, indicates that permanent entities are declared in the **`every`** statements that immediately follow. A preamble can have several **`permanent entities`** statements, each of which must be followed by its respective group of **`every, define ... variable`**, and **`define ... set`** statements. The **`create each`** statement creates a group of permanent entities, and the attributes of the entities are stored as arrays. Storage for the arrays is allocated by a **`create each`** statement.

2.65.1 INCLUDE Phrase

The **`permanent entities`** statement has an optional **`include`** phrase, which can name one or more permanent entity classes that do not have attributes. (Entity classes that do have attributes must be named in **`every`** statements.) For each entity class named in the **`include`** phrase, SIMSCRIPT II.5 automatically defines a global variable having the same name as the entity class, and another global variable, **`N.`*`entity`*** (the number of entities in the entity class). For example, the statement:

```
permanent entities include COUNTRY and GOVERNMENT
```

causes the global variables **`COUNTRY`** and **`N.COUNTRY`** and **`GOVERNMENT`** and **`N.government`** to be defined. These global variables permit statements such as:

```
for every COUNTRY
for every GOVERNMENT
```

to be used to step through a sequence of values from `1` to `N`. **`Create each`** statements cannot be used to set the values of **`N.`*`entity`*** for these entities.

2.66 PREAMBLE Statement

The **`preamble`** statement marks the beginning of a program preamble.

```
preamble
```

EXAMPLES:

`preamble`

Marks the beginning of the program preamble.

The **`preamble`** statement must be the first statement of a program preamble. All statements in a program preamble are nonexecutable, and provide the compiler with definitions regarding entities, attributes, and sets, events and routines, background mode, type and dimensionality, and global variables and arrays. The preamble can be omitted, in which case the SIMSCRIPT II.5 default conditions are used. In this case, there are no user-defined global variables, and a preamble is implied (consisting of the system-defined functions and variables).

When a preamble is compiled, tables are constructed that define the variables and structures. These tables are used to compile all routines. The preamble also generates routines that support the entity, attribute, set, and event declarations.

Changes to statements in the preamble require that an entire program be recompiled.

2.67 PRINT Statement

The **print** statement, which displays messages, titles, and computational results, must be immediately followed by one or more format lines containing text or format specifications, or both.

```
print integer[double] line [s]  [ with { variable                          }  ]c
                                |      {a group of integer value c fields  }  |
                                [      [                                   ]  ]

[suppressing from column integer ]thus
```

Keyword	Synonyms
thus	like this
	as follows

EXAMPLES:

```
print 1 line thus
      Weekly Flight Report
```

Prints one line containing the text **Weekly Flight Report**, with blank columns on either side.

```
print 3 lines as follows
      Weekly Flight Report

Airline    Flight    Date    Passengers
```

Prints three lines, the first containing the heading **Weekly Flight Report**, with blank columns on either side, the second containing a blank format line, and the third containing the headings referring to **Airline**, **Flight**, **Date**, and **Passengers**.

```
print 1 line with MONTH, DAY, YEAR thus
*** ** 19**
```

Prints one line containing the values of variables **MONTH**, **DAY**, and **YEAR**. The digits 19 in the format line are text.

```
print 3 lines with FARE, DISTANCE, and DISTANCE/HOURS as follows
      FARE IS $****.**
      DISTANCE IS *****
      AVERAGE IS *** MILES PER HOUR
```

Prints three lines. The first contains the text **FARE IS** and a **$**, followed by the decimal value of variable **FARE**. The second contains the text **DISTANCE IS**, followed by the integer value of **DISTANCE**. The third contains the text **AVERAGE IS**, followed by the integer value of the quotient **DISTANCE/HOURS**, followed in turn by the text **MILES PER HOUR**.

```
print 1 double line like this
      ------------------------------------------------------------------
      ------------------------------------------------------------------
```

Prints one line of dashes. The keyword `double` indicates that twice as many format lines follow, as is designated by the line count.

```
for each CITY, print 1 line with NAME, POPULATION and AREA thus
     ************  *****  ******.*
```

Prints, for each entity of the class **`CITY`**, one line containing values of attributes **`NAME`**, **`POPULATION`**, and **`AREA`**. The number of lines printed equals the number of entities of the class **`CITY`**.

```
for I = 1 to 60, print 1 line with I, and a group of X(I,J) fields,
TOTAL(I) suppressing from column 47 as follows

      ** *** *** *** *** *** *** *** *** |****
                                         column 47
```

Assuming the **`print`** statement is preceded by a **`begin report`** statement that specifies **`groups of 8`**, prints 60 lines with nine columns per page (values of index variable **`I`** and eight values of **`X(I,J))`**; prints values of **`I`** according to the first format field, and values of **`X(I,J)`** in the eight succeeding fields. When all column indices have been used, prints the values of **`TOTAL(I)`** according to the last format field.

The **`print`** statement displays titles, column headings, and computational results in a programmer-defined format. This statement can perform both as a means of displaying messages and as a complex report layout statement. It is possible to specify the format of printed results, to control the printing of headings and titles, and to arrange "wide" reports on standard-width paper. Each **`print`** statement must be followed by one or more format lines indicating the text and format specifications for printing values of expressions. **`group`** and **`suppressing`** phrases, which can be optionally included in this statement, are used only in report sections that have column repetition. **`Print`** statements generally appear in heading and report sections, and are often controlled by **`for each (`*`class`*`)`**, **`for ... of (`*`set`*`)`**, and **`for ... to (`*`index`*`)`** phrases, as well as **`until`** or **`while`** phrases.

Values of specific attributes can be displayed by the **`print`** statement. Naming an attribute in this statement causes retrieval and display of a single value, just as a subscripted variable or function reference does. **`Print`** output starts at the current **`wcolumn.v`** pointer location. After output, the system skips to the next output line and positions the pointer at the beginning (sets **`wcolumn.v = 0`**).

2.67.1 Format Lines

One or more format lines must follow the **`print`** statement. A format line can have a maximum of 80 columns of text and format specifications for arithmetic expressions whose values are to be printed. The number of columns from column 1 to the last nonblank column determines the length of a format line. See table 20 for format specifications, rules that ap-

ply to the different specifications, and conventions used by SIMSCRIPT II.5 in displaying values.

The following general rules apply to format lines:

1. A **`print`** statement can appear on a card with prior statements, but each format line must be on a separate card.
2. A blank column in a format line will appear as a blank space on the printout.
3. Blank lines can be inserted between lines of output with a blank format line or with the **`skip`** statement.
4. SIMSCRIPT II.5 prints text exactly as it appears, and any character except an asterisk (*) or a parallel (|) can be included as text.
5. When values are to be printed contiguously, a parallel must terminate a format on the left, or two contiguous formats will merge into a single format. The parallel character always acts as the first asterisk of a new format. For example, two contiguous, five-character integer fields can be indicated as *****|****, and four contiguous one-digit fields can be indicated as ||||.

2.67.2 DOUBLE Keyword

If more than 80 columns must be printed on a line, the keyword `double` can be included in the **`print`** statement. This keyword indicates that twice as many format lines (than are specified by the line count) follow. When the **`double`** keyword appears, SIMSCRIPT II.5 reads the format lines in pairs and interprets each pair as one format line of 160 columns. Assuming that printer paper having 132 columns is used, the first format line would contain text and format specifications for 80 columns, and the second format line would contain text and format specifications for the next 52 columns.

Table 20. PRINT STATEMENT FORMAT SPECIFICATIONS

Value and Examples	Rules and Conventions
Integer * ** *****	1. Prints an integer value. 2. Treats the rightmost character as the low-order position and prints as many digits to the left of the * as possible up to the next consecutive * or text character. 3. Only the rightmost * need appear in a format line. 4. Prints in scientific notation if insufficient space was allocated. 5. If an expression does not yield integer values, prints a rounded integer by adding 0.5 (depending on the sign) to the value of the expression and truncating the result.
Decimal *.* **.* **.***	1. Prints a decimal value. 2. Treats the integer portion according to the rules for integer values. 3. Rounds the decimal portion to the number of asterisks that appear to the right of the decimal point. An expression is rounded in the n^{th} decimal place by adding 0.5 (10^{-n}) and truncating at the n^{th} decimal place. 4. Prints trailing zeros.
Rounded decimal **. ***. *.	Prints a rounded integer.
Fraction .* .** .****	Prints a fraction between 0 and 1.

Table 20. PRINT STATEMENT FORMAT SPECIFICATIONS (Continued)

Value and Examples	Rules and Conventions
Scientific ……..	1. A minimum of eight consecutive periods must appear in a format line. 2. Prints a number of the general form: $xxx.xxxE^{xx}$ 3. The value of the printed expression is $xxx.xxx(10^{xx})$ 4. $0 \leq$ |decimal number| < 10
Alphanumeric ** * ****	1. Prints alphanumeric characters. 2. Each character must be indicated by * or |. 3. Only the leftmost positions are used if more characters are represented than are stored in an alphanumeric word.
Text ********* *** **********	1. Prints alphanumeric characters of a text variable or literal. 2. Each character must be indicated by * or |. 3. If the text string contains fewer characters than are represented, only the leftmost positions are used. If the text string contains more characters than are represented, only its leftmost characters are used.

2.67.3 Expressions

When values of arithmetic expressions are to be printed, the expressions must be listed in a **`print`** statement, and format specifications that correspond to expression values must appear in format lines. Note that integer values can be printed with decimal format and real values can be printed in integer format.

During execution, SIMSCRIPT II.5 evaluates the expressions and then prints the values according to the specifications in left to right order. That is, the system prints the value of the first expression according to the first format specification in the format line, prints the value of the second expression according to the second format specification, and so on.

2.67.4 GROUP Phrase

The a **`group of ... fields`** phrase is used only for column repetition within a report section. Column repetition is defined as repeating a group of format fields for each group of values of an index variable. Groups of values of the index variable are generated by a preceding **`begin report`** statement that contains a **`printing`** phrase. One group of index

variable values is used by the **print** statement at one time in order to print one group of data (see figure 8). When column repetition is used, each column of grouped data is considered to have a column index that is one value of the index variable. The **printing** phrase in a **begin report** statement must specify the number of column indices to be used in a group. This number is the number of columns of grouped data to appear across a page.

In the format line, a format field must appear for each value in a group. All format fields in a group need not be identical. For example, * and ** are permitted, but all values must have the same mode (e.g., ** and *.* are not permitted). Report columns can also contain data that are not part of a group, such as the value of the index variable that controls lines.

2.67.5 SUPPRESSING Phrase

The **suppressing** phrase enables the program to suppress specific values until all grouped data have been printed. This phrase specifies that all format specifications, beginning with the format in the designated column, are to be disregarded until all column indices have been used. Values are then printed using the formats that start in the designated column.

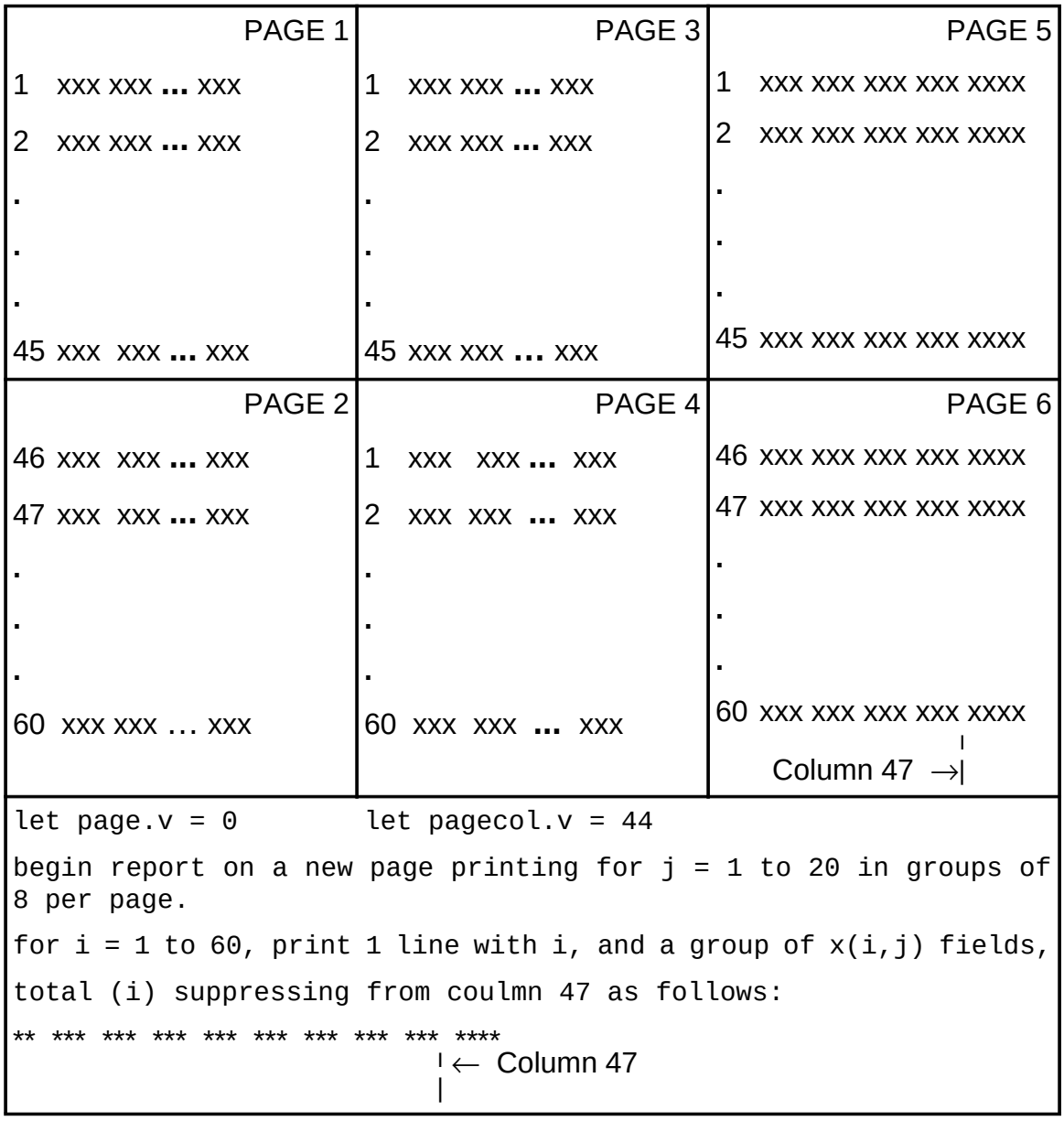


Figure 8. Sample Row and Column Repetition

2.68 PRIORITY Statement

The **priority** statement assigns priorities to different classes of processes or events.

```
priority order is  { event   }^c
                   { process }
```

EXAMPLES:

```
priority order is WEATHER and MANAGEMENT.REPORT
```

If two or more events of different classes are scheduled for the same time, events of the class **WEATHER** will be executed first.

```
priority order is DELAY, LANDING, TAKEOFF, SEAT.RESERVE and
END.SIMULATION
```

If two or more events of different classes are scheduled for the same time, priorities are assigned to event classes in the following order: **DELAY**, **LANDING**, **TAKEOFF**, **SEAT.RESERVE**, and **END.SIMULATION**.

The **priority** statement assigns priorities to classes of processes or events. This statement resolves conflicts that can exist when two or more events of different event classes are scheduled for the same simulated time. Both external and internal processes and events can be named in the same **priority** statement. Priorities cannot be assigned to external event (process) units, however. If a **priority** statement lists event names, it must follow all **event notices, external events**, and **external event unit** statements in the preamble. If it lists process names, it must follow all **process**, **external processes**, and **external event notices** statements.

If a **priority** statement is included, events (processes) of the first-named class have the highest priority, events of the second-named class have the second highest priority, and so on. Subsequently, when event notices have identical simulated occurrence times, SIMSCRIPT II.5 selects events for execution in the order of their priorities.

When the **priority** statement is omitted, priorities are assigned by default in the order in which events and processes are declared in the preamble, using **event notices**, **processes**, and **every** statements. If some event classes are named in **priority** statements and others are not, the named classes automatically have higher priorities than the omitted ones. The omitted event classes are ranked among themselves in the order of their appearance.

When several events or processes are scheduled to occur at the same simulated time, the events are selected to be executed according to the **priority** statement (or default ordering) if they are of different classes, according to the **break ties** statement (if any) for events in the same event class, and first-in, first-out otherwise.

2.69 PROCESS Statement

The **process** statement names a process routine for a process. The process must be declared in the preamble.

```
process [to] process    [ given value [, value]... ]
                        [ (value [, value]...)     ]
```

Keywords	Synonyms
to	for
given	giving
	the
	this

EXAMPLES:

```
process TAKEOFF
```

Declares **TAKEOFF** as a process routine, and destroys the process notice before entering the process routine.

```
process SERVICE.CUSTOMER given DEPARTMENT and CUSTOMER.TYPE
```

Declares **SERVICE.CUSTOMER** as a process routine having **DEPARTMENT** and **CUSTOMER.TYPE** as arguments. Destroys the process notice.

A process routine, which is declared by the **process** statement, is similar to an event routine, in that it can only be called by the timing routine. Each **process** class must have a process routine.

Processes are generated by the **activate** process statement, which specifies the simulated time at which the process is to begin. Control remains within the process routine until one of the following occurs:

1. A **return** statement is encountered.
2. A **work** or **wait** statement is executed.
3. A **request** statement refers to an unavailable resource.

Because a process routine may relinquish control before completion, the process routine should be considered re-entrant. That is, the values of saved or global variables may change upon execution of a **work, wait** or **request** statement.

The **process** statement can only appear at the beginning of a process routine.

2.69.1 Arguments

A process is triggered by a process notice as the result of an **activate** process statement. In addition to the five special attributes an event notice has, every process notice has four attributes that are specific to processes.

Process notices also transmit the value of any **given** arguments from the **activate** statement to the process routine. Each such argument must be declared as an attribute of the process using the **every** statement. The arguments of a process routine are local to the routine and distinct for multiple entries of a process routine.

2.69.2 Logical Expression for Process Routines

SIMSCRIPT II.5 provides a logical expression to determine, within a process, whether that process was generated internally or externally. The logical expression is of the form:

```
                        ⌈endogenous    ⌉
                        |exogenous     |
process is [not]       {internal       }
                        ⌊external      ⌋
```

and yields a true or false value. This logical expression, which tests the process notice, can be included in an **if ... else ... always** construct to make decisions regarding processes.

2.70 PROCESSES Statement

The **processes** statement declares that the following **every** statements define process notices.

```
processes [include process c]
```

Keyword	Synonym
include	are

EXAMPLES:

```
processes
```

Denotes that process declarations follow. The declarations will be made with **every** statements, and will name user-defined attributes.

```
process include TAKEOFF and LANDING
```

Identifies **TAKEOFF** and **LANDING** as processes requiring only the system-defined attributes for processes. **Every** statements may follow this form of the **processes** statement.

The **processes** statement can appear only in the preamble.

Process notices are created and destroyed like temporary entities, and carry information about a process. When a process is generated, the process notice transmits this information to the timing routine, and then to the process routine when the process begins.

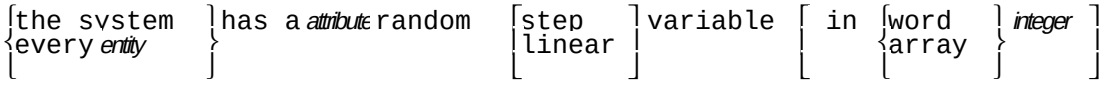
Process notices are used to schedule processes to occur at some time in the simulated future. Many processes of the same process class can be scheduled at the same simulated time. A **break ties** statement declares priorities among processes of the same process class, while a **priority** statement declares priorities among different classes of processes. If no **priority** statement is present, priority is given according to the order in which the process notices have been defined in the **include** phrase or in subsequent **every** statements.

In addition to the nine system-defined attributes, process notices can have attributes that are either variables or functions. Process notices can own and belong to sets. **Every** statements are used to declare the attributes and sets.

Process notices with additional attributes must be declared in **every** statements. Process notices with no additional attributes can be named in the **include** phrase of the **processes** statement. This phrase notifies the system that the following names are names of processes and that standard process notices will be used for them.

2.71 ... RANDOM ... VARIABLE Statement

The **... random ... variable** statement declares a random variable.



Keyword	Synonym
has	have
	can have
	may have
a	an
	the
	some

EXAMPLES:

```
the system has a SAMPLE random step variable
```

Declares that the system attribute **SAMPLE** is a random step variable.

```
every AIRPORT has a TRAFFIC random linear variable
```

Declares that every entity of the class **AIRPORT** has an attribute named **TRAFFIC**, which is a random linear variable.

The **... random ... variable** statement declares a random variable. Random variables are table look-up variables, each of which has a list of possible values and associated probabilities. The system selects a sample value by generating a random number (using the function **random.f**), matching the random number with the possible probabilities, and selecting the corresponding sample value from the look-up table. Sampling takes place by drawing successive pseudo-random numbers from random number stream 1, unless another stream number is requested.

For a random step variable, the system samples from the table values, which can have either integer or real values, in a step-like manner. For a random linear variable, which can have only real values, the system performs sampling by employing linear interpolation between the sample values. The mode of the values can be implied through the background mode at the time the **... random ... variable** statement is encountered, or the mode can be defined in a subsequent **define ... variable** statement. However, linear values may only be real.

2.71.1 Function RANDOM.F

Function **random.f** generates a stream of pseudo-random numbers between 0 and 1. The algorithm used in this function depends on the implementation. The generated numbers are statistically independent of one another. All SIMSCRIPT II.5 programs are initialized with 10 random number streams, and the starting numbers for these streams are contained in the system array **seed.v**. As pseudo-random numbers are generated, new values are assigned to **seed.v** so that it contains the current number as an integer.

Random.f can be viewed in two ways — as generating uniformly distributed pseudo-random numbers between 0 and 1, or as generating probabilities.

2.71.2 Mode and Stream Numbers

Either of the following forms of the **define ... variable** statement is used to declare the mode of a random variable and to designate a stream number.

define *variable* as [a] [real] [, stream *integer*] variable
define *variable* as [an] [integer] [, stream *integer*] variable

where 0< *integer* ≤ 10.

2.71.3 Using Random Variables

Sampling is always automatic. That is, a random variable is similar to a right-hand function. Whenever a random variable appears, a routine that performs sampling is executed. SIMSCRIPT II.5 generates these routines using random number stream 1 unless otherwise specified. If the programmer requires another type of sampling other than step or linear, he must omit the words **step** or **linear** from the **... random ... variable** statement and provide his own sampling function. Random variables can be read and sampled only. Assignments cannot be made to them (i.e., they cannot appear to the left of equal signs or as **yielding** arguments).

2.71.4 Reading Values and Probabilities

Special storage assignments are made for sample values and probabilities. These sample values and probabilities can be read only by a **read (Free-form)** statement and not by the **read (Formatted)** statement. Only one random variable can appear in each read statement. When a random variable appears in a **read (Free-form)** statement, the system:

1. Reads pairs of values until a **mark.v** character (default) appears.
2. Assumes that the first value of each pair is a probability and that the second is a sample value.
3. Creates an entity record for each pair of values. Each entity record has three attribute words: the probability, the sample value, and a successor.
4. Maintains the entities in a set-like list.

5. Ensures that the pointer to the random list occupies the storage declared for the random variable or attribute.

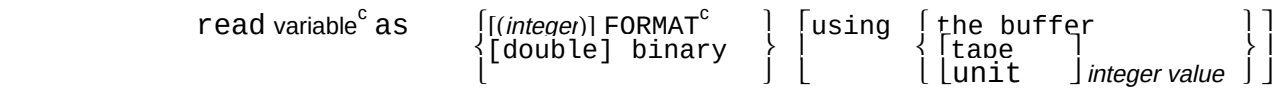
Probabilities read by the system can be cumulative or individual. If the probabilities are cumulative, the last probability must be 1.0; and if the probabilities are individual, they must sum to 1.0. All probabilities are stored cumulatively, but if individual probability values are read, SIMSCRIPT II.5 accumulates the values. Thus, if the last probability = 1, the probabilities are assumed to be cumulative, and if the last probability = 1, the probabilities are summed so that they are stored cumulatively. The last probability is set to 1. Probability values less than 0 or greater than 1 terminate the program with an error message.

2.72 REACTIVATE Statement

See **`activate`** statement.

2.73 READ (Formatted) Statement

The **`read`** **(Formatted)** statement reads formatted or binary data.



Note: **`Double`** is optional on implementations where full precision requires more than one computer word.

EXAMPLES:

```
read A, X, and Y as I 3, 2 I 4
```

Beginning with the column that follows the input pointer, reads a three-character integer value and assigns that value to **`A`**, then reads two successive integer values of four characters each, and assigns the values to **`X`** and **`Y`**.

```
read X, Y, Z and A(1), A(2), A(3), as 3 d(8,2) and 3 a 4
```

Beginning with the column that follows the input pointer, reads three successive decimal values of eight characters each, and assigns the values to **`X`**, **`Y`**, and **`Z`**; then reads three successive alphanumeric values of four characters each, and assigns the values to the first three elements of array **`A`**.

```
read DISTANCE and QUANTITY as B 1, E(10,2), /, E(10,2)
```

After positioning the input pointer at column 1 of the current record, reads a value in scientific notation from columns 1 to 10 and assigns that value to **`DISTANCE`**; then skips to the next record and reads a value from columns 1 to 10 and assigns that value to **`QUANTITY`**.

```
start new card
for I = 1 to N, read X(i), Y(1) as (4) I 5, D(7,2)
```

Beginning with column 1 of a new record, reads four pairs of data (one integer and one decimal value) from each record until **`N`** pairs of values have been read. Assigns the integer values to elements of array **`X`** and the decimal values to elements of array **`Y`**.

```
read A(1), B(2), and X(1), X(2), X(3) as B 5, I 10, S 3, D(7,2),/,
B 25, 3 I 6
```

Beginning in column 5 of the current record, reads an integer value from columns 5 to 14 and assigns that value to the first element of array **`A`**; then skips three columns, reads a decimal value from columns 18 to 24, and assigns that value to the second element of array **`B`**; then skips to the next record, and beginning in column 25 reads three integer values of six characters each, and assigns the values to the first three elements of array **`X`**.

```
for I = 1 to N, for J = 1 to N, read MATRIX(I,J) as binary using
the buffer
```

Reads the two-dimensional array **MATRIX** in binary form from the internal buffer.

```
read WEATHER.TYPE and VISIBILITY as B 35, A 4, D(7,3)
```

Beginning in column 35 of the current record, reads an alphanumeric value from columns 35 to 38 and assigns that value to **WEATHER.TYPE**; then reads a decimal value from columns 39 to 45 and assigns the value to **VISIBILITY.**

The **read (Formatted)** statement reads data from cards according to a format list and assigns these data to variables named in the statement. Formats in a format list must correspond in order to the desired values punched on data cards. Each format includes a descriptor, which is a code defining the type of data (e.g., integer, decimal) to be read from the cards. There are eight descriptors: five are data descriptors that apply to numeric and alphanumeric values, and three are control descriptors used for spacing and for skipping columns and records. In addition to reading formatted data, this statement can also read binary data. Either formatted or binary data can be read from any input device, or data can be read from an internal file.

2.73.1 Format Lists

During execution of a **read (Formatted)** statement, the format list is scanned from left to right, and individual formats are used to read values from data cards. The values, in turn, are assigned to variables. The data descriptors apply to integer, decimal, scientific notation, alphanumeric, and computer representation values, while the control descriptors designate beginning data columns and columns and records to be skipped. Descriptors are further defined in table 21. Values being read must agree in mode with their descriptors, except for integer and alphanumeric modes, which can be interchanged. When interchanged, the mode implied by the descriptor governs.

2.73.2 Skipping to Next Card

A **read (Formatted)** statement does not necessarily start at the beginning of a new data card (record), because records are changed under programmer control and not automatically after each statement. Consequently, data can be split between cards or read from noncontiguous parts of the same card. This statement processes a continuous string of characters and skips to a new data card *only* when directed.

When starting a new input record, the value of **record.v** is incremented by one. Thus, it keeps track of the number of records read in so far.

Table 21. Descriptors for READ (Formatted) Statements

I. INTEGER

Format	Rules
`n i w` where: `n` is the optional number of consecutive fields `i` is the descriptor `w` is the field width	1. Parameter `n` must be an integer, but `w` can be an expression. 2. At least one blank must appear between `n` and `i` and between `i` and `w`. 3. The system treats leading, embedded, and trailing blanks as zeros. 4. If the value to be read exceeds the capacity of one word, only the right-most digits are used. 5. If the value is less than one word, the digits are right-adjusted with leading zeros. 6. Only digits (and an optional sign character) can be used in I data fields.

D: DECIMAL

Format	Rules
`n d(a,b)` where: `n` is the optional number of consecutive fields. `d` is the descriptor `a` is the total field width including the sign, integer digits, decimal point, and fractional digits `b` is the number of fractional digits	1. Parameter `n` must be an integer, but `a` and `b` can be expressions. 2. At least one blank must appear between `n` and `d`. 3. A decimal point is optional in the input data field. 4. If the decimal point is omitted, a decimal point is implied before the first digit of the `b` field. 5. When a decimal point is included, it overrides the location implied by `b`. 6. Very large and small decimal numbers can be input in scientific notation.

Table 21. Descriptors for READ (Formatted) Statements (Continued)

E: SCIENTIFIC NOTATION

Format	Rules
`n e(a,b)` where: `n` is the optional number of consecutive fields `e` is the descriptor `a` is the total field width including the sign, integer digits, the letter `e`, and sign of the exponent `b` is the number of fractional digits	1. Parameter `n` must be an integer, but `a` and `b` can be expressions. 2. At least one blank must appear between `n` and `e`. 3. Data read by the `e` format must have the general form: `xxx.xxxexx` 4. The `e` format is similar to the `d` format, but the `e` format must have an appended scale factor. 5. The scale factor on the input can exclude either the letter `e` or the sign of the exponent, but not both.

A: ALPHANUMERIC

Format	Rules
`n a w` where: `n` is the optional number of consecutive fields `a` is the descriptor `w` is the field width	1. Parameter `n` must be an integer, but `w` can be an expression. 2. At least one blank must appear between `n` and `a` and between `a` and `w`. 3. Characters are left-adjusted in a word with trailing blanks. 4. If more characters are specified than can be stored, the leftmost characters are stored and the remaining characters are excluded.

C: COMPUTER REPRESENTATION

Format	Rules
`n c e` where: `n` is the optional number of consecutive fields `c` is the descriptor `e` is the number of characters in the internal representation of the computer	1. Parameter `n` must be an integer, but `e` can be an expression. 2. At least one blank must appear between `n` and `c` and between `c` and `e`. 3. The `c` descriptor is computer-dependent; for example, on the IBM 360, the format C 4 reads four hexidecimal digits; on the Honeywell 600/6000, the format 2 C 5 reads two fields of five octal digits each.

Table 21. Descriptors for READ (Formatted) Statements (Continued)

B: BEGINNING COLUMN

Format	Rules
`b n` where: `b` is the descriptor `n` is the column number	1. Parameter `n` can be an expression. 2. At least one blank must appear between `b` and `n`. 3. Parameter `n` specifies the column in which the first character of an input value is located, and the system positions the current input pointer to that column. 4. `B` descriptors need not be in ascending order in a format list.

S: SKIP COLUMN

Format	Rules
`s n` where: `s` is the descriptor `n` is the number of columns to skip	1. Parameter `n` can be an expression. 2. At least one blank must appear between `s` and `n`. 3. Parameter `n` specifies the number of columns to be skipped before reading the next field on the card. 4. The system disregards data punched in skipped columns.

/: SKIP TO NEW RECORD

Format	Rules
/	Each slash skips to a new record on the current input unit.

T: `text`

Format	Rules
`n t w` where: `n` is the optional number of consecutive fields `t` is the descriptor `w` is the field width	1. Parameter `n` must be an integer, but `w` can be an expression. 2. At least one blank must appear between `n` and `t` and between `t` and `w`. 3. The length of the text variable is set equal to the field width, and all the characters in the field are stored in the text variable.

Table 21. Descriptors for READ (Formatted) Statements (Continued)

‘1	
Format	Rules
`n t *` where: `n` is the optional number of consecutive fields `t *` is the descriptor	1. Parameter `n` must be an integer. 2. At least one blank must appear between `n` and `t` and between `t` and `*`. `*` may optionally be enclosed in parentheses. 3. The next nonblank input character is treated as a delimiter character; the input is scanned for the next occurrence of that character, and all characters between the two delimiters are stored in the text variable. 4. The length of the string is set to the number of characters between the delimiters. The field may span multiple input records.

2.73.3 Input Buffer

A buffer whose length is one record is provided for each input unit. This buffer contains the data fields. The current input pointer, which is the system variable **`rcolumn.v`**, points to the last column read in the current input buffer. For each new record, **`rcolumn.v`** is zero. (Some implementations initialize **`rcolumn.v`** to -1 before the input unit is read for the first time.) As input is processed, the pointer moves along the buffer according to the format list. For each value read, **`rcolumn.v`** is positioned to the last column read. The value of **`rcolumn.v`** can be advanced by the Beginning Column (B), Skip Column (S), and Skip to New Record (/) descriptors and by the **`start new`** statement. Note that the B format can move the column pointer backward, allowing values to be read more than once. The B, S, and / descriptors can be combined with other formats, or they can appear alone in a **`read`** **(Formatted)** statement.

2.73.4 AS BINARY Phrase

The **`as binary`** phrase reads binary information. The binary data can be read from the current input unit, or a **`using`** phrase can be appended to the **`read`** statement to designate another input unit. Binary and formatted data cannot be read together from the same unit.

For integer, real, or alpha values, the `read` **`as binary`** statement inputs a single computer word of information. For text values, it inputs an integer computer word with the length of the string, followed by successive words of the string until all the characters have been input.

2.73.5 AS DOUBLE BINARY Phrase

On those implementations for which the maximum floating-point precision is more than one computer word, the **`read as double binary`** statement enters two computer words

for a floating-point (real) number. The **read as double binary** statement should only be used for values produced using the **write as double binary** statement.

2.73.6 USING Phrase

A **using** phrase can locally override the current input unit declared in a **use** statement. This phrase designates a device as the current input unit for the duration of the **read (Formatted)** statement execution. After execution of this statement, SIMSCRIPT II.5 automatically reassigns the previous unit as the current input unit. Data can be read from an internal file by including the keywords **the buffer** in a **using** phrase, or from an input unit by specifying the device number of that unit.

2.73.7 The Buffer

A special area of memory can be made available as an internal buffer not connected to any external device for read statements. Its length is the value of **buffer.v** (default 132). Space is allocated (the amount depending on the value) when the first **write ... using the buffer, read ... using the buffer**, or **use the buffer** statement is executed. It is initialized to all blanks. This internal buffer is generally used for data editing employing the formatting capabilities of **read** and **write** statements. All data in **the buffer** must have been put there by **write** statements. **The buffer** is reinitialized to blanks each time its usage changes from input to output.

2.73.8 Controlled Statements

A **read (Formatted)** statement cannot read an entire array by listing only the array name, but a controlled **read (Formatted)** statement that includes a repetition factor can read array elements conveniently (see the fourth example above). A repetition factor, consisting of an expression enclosed in parentheses, must precede a format list that is to be repeated for each card. When a repetition factor is used, the **read (Formatted)** statement must be controlled by a **for ...** statement, and the input must start with a new card. (**a start new** statement, for example, can position the input pointer to a new card.)

With this form of the **read (Formatted)** statement, the system automatically skips to a new card after reading the specified number of fields from an individual card. This statement can terminate with the input pointer positioned in the middle of a record.

2.73.9 End-of-File

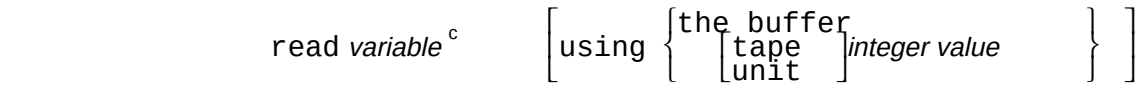
An end-of-file can be encountered when a **read (Formatted)** statement is executed. The value of the global variable **eof.v** determines the action taken by the system when an end-of-file is detected. This variable is automatically defined for each input unit and initialized to zero when a program begins execution. Variable **eof.v** can retain a zero value or can be set equal to 1. Subsequently, when an end-of-file marker is encountered, SIMSCRIPT II.5 interprets the values of **eof.v** as follows:

Value of eof.v	Interpretation
0	Consider an end-of-file marker to be an error.
1	Assign zeros to variables named in the statement, **`set eof.v to 2`**, and returns control to the statement that follows this **`read`** **(Formatted)** statement.
2	An end-of-file has been read; continue as if **`eof.v = 1`**.

By testing **`eof.v`** immediately after a **`read`** **(Formatted)** statement, the programmer can determine whether the statement read actual values or encountered an end-of-file.

2.74 READ (Free-Form) Statement

The **read (Free-form)** statement reads unformatted values and assigns them to the named variables.



EXAMPLES:

```
read A, B, and C
```

Assuming that **A**, **B**, and **C** are variables, reads three values and assigns them to **A**, **B**, and **C**.

```
read CODE, A, B, X(2), Y(4), Z(2,4)
```

Assuming that **CODE**, **A**, and **B** are variables, that **X** and **Y** are one-dimensional arrays, and that **Z** is a two-dimensional array, reads six values and assigns them to **CODE**, **A**, and **B** and to the subscripted variables **X**(2), **Y**(4), AND **Z**(2,4).

```
read A, LIST(3), COLUMN, MATRIX, X, LIST(x)
```

Assuming that **A** and **X** are variables, and that **LIST**, **COLUMN**, and **MATRIX** are arrays, reads and assigns a value to **a**, reads and assigns a value to the third element of **LIST**, reads and assigns as many values as there are elements in **COLUMN**, reads and assigns as many values as there are elements in **MATRIX**, reads and assigns a value to **X**, and reads and assigns a value to the element of **LIST** whose index is the value of **X**.

```
for I = 1 to 3, read X(i) and Y(i)
```

Assuming that **X** and **Y** are one-dimensional arrays, reads six values and assigns them to **X**(1), **Y**(1), **X**(2), **Y**(2), **X**(3), AND **Y**(3).

```
read NUMBER, DATUM for J = 1 to 3, read X(J)  read Y
```

Assuming that **NUMBER** and **DATUM** are variables and that **X** and **Y** are one-dimensional arrays reserved as 3 and 4 elements, respectively, reads nine values and assigns them to **NUMBER**, **DATUM**, **X**(1), **X**(2), **X**(3), **Y**(1), **Y**(2), **Y**(3), and **Y**(4). Three `read` statements are shown here, one of which is controlled by a **for ... to (*index*)** statement.

```
read TABLE using the buffer
```

Assuming that **TABLE** is a two-dimensional array reserved as 3 by 3, reads nine values from the internal buffer and assigns them row by row as follows: **TABLE**(1,1), **TABLE**(1,2), **TABLE**(1,3), **TABLE**(2,1), **TABLE**(2,2), **TABLE**(2,3), **TABLE**(3,1), **TABLE**(3,2), AND **TABLE**(3,3).

```
for each CITY, read NAME(CITY), AREA(CITY), and POPULATION(CITY)
```

Assuming that **CITY** is a permanent entity class, reads attribute values for all entities and assigns them to the attribute arrays named **NAME**, **AREA**, and **POPULATION**.

The **read (Free-form)** statement reads integer, decimal, and alphanumeric values from an input unit and assigns the values to variables named in the statement. During execution, this statement reads as many values from a data card as there are variables — unsubscripted variables, elements of arrays, and attributes — specified or implied in the statement. Combinations of variables, individual elements of arrays, and entire arrays can be read by a single statement. The free-form data cards can be read from a device other than the current input unit, or data can be read from an internal buffer. A **read (Free-form)** statement can be controlled by a **for each (*class*), for ... of (*set*),** or **for ... to (*index*)** phrase, and can appear within a **do ... loop** construct. Implied subscripts cannot be used to read attributes of permanent entities because an attribute name in this statement causes the entire attribute array to be read. The **read (Free-form)** statement may appear in any routine, but not in the preamble.

2.74.1 Data Records

When preparing data records (in an input file), values can occupy any columns, but the permissible magnitude of numbers may vary with each SIMSCRIPT II.5 implementation. Numerical values can be written in either integer or decimal format. For example, the numbers 3, 3.0, and 3.0000 are equivalent. Numbers can also be expressed in scientific notation as long as there are no embedded blanks. [See E format under **read (Formatted)** statement.] It is an error to try to read real values into integer variables.

Successive **read (Free-form)** statements do not necessarily — but could — read new data records, because input data are treated as a continuous stream of values.

The following rules apply to the preparation of data records:

1. Values need not occupy specific columns.
2. Values must be separated from each other by at least one blank column.
3. A value cannot be split between records.
4. Blank characters cannot be read as values of alphanumeric variables because a blank character terminates a field. SIMSCRIPT II.5 reads alphanumeric values by beginning with the first nonblank character and ending with the first blank, or after reading as many characters as the variable can contain.
5. An integer or decimal value to the right of a full word of alphanumeric characters need not be separated from the alphanumeric characters by a blank character, although a blank is permissible.

2.74.2 ARRAYS

The **read (Free-form)** statement can read entire arrays or individual elements. Entire arrays are read by listing only the array name. For example, the statement **read column** reads all elements of the array **column** and assigns the values in ascending subscript order. A multidimensional array is read row by row. That is, SIMSCRIPT II.5 assigns values to successive elements whose subscripts change in ascending order, with the last subscript position varying most rapidly.

Individual elements can be read by listing the array name followed by a subscript enclosed in parentheses. A variable list can include elements whose subscripts are expressions, such as **TABLE(N*M+2/J)** or **TABLE(I)**, if variables in the expressions have assigned values. Variables are processed from left to right, and values are assigned if they appear before being used as subscripts. For example:

```
read N, M, TABLE(N), TABLE(LIST(N) + M)
```

uses the values of **N** and **M** as subscripts after they are read.

2.74.3 USING Phrase

A **using** phrase can locally override the current input unit. This phrase designates a device as the current input unit for the duration of the **read (Free-form)** statement execution. After execution of this statement, SIMSCRIPT II.5 automatically reassigns the previous unit as the current input unit. Data can be read from an internal file by including the keywords **the buffer** in a **using** phrase, or from an input unit by specifying the device number of that unit.

2.74.4 Controlled READ (Free-Form) Statements

FOR phrases can control **read (Free-form)** statements. For example, one may wish to read individual elements of arrays. A `for` phrase controls the entire **read (Free-form)** statement, not just the indices of variables appearing in the statement. Therefore, each variable in a controlled **read (Free-form)** statement must contain the index variable as a subscript, or successive values will be assigned to unsubscripted variables while the index variable iterates over its range of values. This could cause values to be assigned incorrectly to the unsubscripted variables.

2.74.5 System Variables

SIMSCRIPT II.5 provides five system variables that enable the programmer to test characteristics of input data before the data are read. When a system variable is used in a statement, the system automatically determines the characteristics based on the status of the current input data, and the programmer can then use the value in any desired manner. The system variables, as well as definitions and examples, are shown in table 22.

Table 22. System Variables to Test Characteristics of Input Data

Name	Value[1]	Description	Example
sfield.f	0	Starting column of the next data field	**if sfield.f equals 40, go to...**
efield.f	0	Ending column of the next data field	**let N=efield.f-sfield.f+1**
mode	alpha	Mode of the next data field: **integer, real** or **alpha**	**if mode is integer, go to...**
card	new	First data field on card indicator: **card is new** or **card is not new**	**if card is new, go to ...**
data	ended	No data items in data deck indicator: **data is ended** or **data is not ended**	**if data is ended, stop**

[1] When there are no data (e.g., all data have been read and look-ahead is not possible), system variables have the listed values.

2.75 RECORD Statement

Not available for all implementations. See *User's Manual* for your computer type.

2.76 REGARDLESS Statement

See **if ... else ... always** construct.

2.77 RELEASE Statement

The **release** statement releases storage previously reserved for arrays and for attributes of permanent entities.

```
release   { array                          }
          { permanent entity attribute     }
```

EXAMPLES:

```
release COLUMN(*)
```

Returns storage occupied by the array named **COLUMN** to system storage.

```
release CODES and A(*)
```

Returns storage occupied by a two-dimensional array named **CODES** and the one-dimensional array named **A**, to system storage. Note that when releasing an entire array, the star notation for the pointer variable is optional.

```
release STOCKHOLDER, and MATRIX(2,*)
```

Assuming that **STOCKHOLDER** is an attribute of a permanent entity class, returns storage occupied by the **STOCKHOLDER** attribute array and by row 2 of the two-dimensional array named **MATRIX** to system storage.

The **release** statement returns storage (occupied by arrays, parts of arrays, and attributes of permanent entities) to system storage. Names of pointer variables and attributes of permanent entities can be listed in this statement, and a single statement can include several names of the different types. A variable named in this statement must be a pointer to an array, or some portion of an array (e.g., a row of a ragged table). Once storage for an array has been released, that array is undefined until reserved again. The pointer variable is set to zero. Arrays can be reserved and released continually throughout program execution in order to conserve storage by reusing it.

The **release** statement can appear in any routine, but it cannot be included in the preamble. Permanent entities can be deleted by releasing their attributes, but all attributes of a permanent entity should be released at the same time.

Local arrays are not automatically released when a **return** statement is executed in routines in which the arrays appear. A **release** statement should therefore be executed before returning to the calling program unless the array has been declared as saved. In this way fresh recursive arrays will be reserved each time the routine is executed while the same saved arrays will be used each time. If the programmer neglects to release a recursive array, that storage becomes inaccessible because the pointer to it is zeroed out on the next entry to the routine.

2.78 RELINQUISH Statement

The **relinquish** statement makes the specified number of units of the resource available for automatic reallocation of resource units.

relinquish *integer value* [unit [s] of] *resource* [(*integer value*)]

EXAMPLES:

```
relinquish 1 WORKER(2)
```

Relinquishes one unit of the resource named **WORKER**, of the second type (where **N.WORKER** 2).

```
relinquish 2 MACHINE(JOB.TYPE)
```

Relinquishes two units of the resource named **MACHINE**, of the type **JOB.TYPE**, whose value has either been assigned, or defined by a **define ... to mean** statement.

```
relinquish 3 units of MATERIAL
```

Relinquishes three units of the resource named **MATERIAL**, of which only one type exists.

A process that has previously requested some units of a resource may relinquish some or all of them, using the **relinquish** statement. The number of units of the resource being relinquished is added to the total quantity available.

If any processes are enqueued 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 scan is then terminated.

The process relinquishing the resource continues execution at the statement immediately following the **relinquish** statement.

The **relinquish** statement can appear only in process routines.

2.79 REMOVE Statement

The **remove** statement removes an entity from the named set.

```
          ⌈[the] first    ⌉
remove    |[the] last     |pointer variable from [the] set
          ⌊ the [above]   ⌋
```

Keywords	Synonym
`the [above]`	`this`

EXAMPLES:

```
remove first PATRON from RESERVATIONS(FLIGHT)
```

Removes the first entity from a set of the class **RESERVATIONS** and assigns the entity identification to variable **PATRON**. The set owner is an entity whose identification is contained in the **FLIGHT** variable.

```
remove the STOCK.CERTIFICATE from the PORTFOLIO
```

Removes the entity whose identification number is the value of **STOCK.CERTIFICATE** from the set **PORTFOLIO**. The set owner is **the system**.

```
remove the above FLIGHT from this WAITING.LINE(RUNWAY)
```

Removes the entity whose identification is the value of variable **FLIGHT** from a set of the class **WAITING.LINE**. The set owner is an entity of the class **RUNWAY**.

The **remove** statement, which removes an entity from the designated set, can remove the first entity, the last entity, or a specific entity. If this statement specifies removal of the first entity, the system removes the entity pointed to by attribute **F.*set*** (first in set) of the set owner. After removing the first entity, the system stores the pointer to that entity in the named variable and places the second entity first in the set. The attribute values of the first entity (e.g., **S.*set***, the **P.*set*** and pointers) remain the same, except for the membership attribute, **M.*set***, which is set to zero. Conversely, if the statement specifies removal of the last entity, the system removes the entity pointed to by attribute **L.*set*** (last in set) of the set owner. An alternative form of the `remove` statement removes a specific entity by using the designated arithmetic expression, which must evaluate to the identification of the entity to be removed. The successor and predecessor entities have their **P** and **S** attributes changed to reflect the removal. The **remove** statement can appear in any routine, but it cannot be included in the preamble.

The program terminates if the value of the designated arithmetic expression is not the identification of an entity currently in the set (denoted by a <>0 in **M.*set***) or if the named set is empty.

2.79.1 Logical Expressions

The membership attribute **M.*set*** provides both error checking and decisions regarding set membership. The logical expressions:

entity `is in` *set*

entity `is not in` *set*

can be used in **`if ... else ... always`** constructs to determine if a specific entity is filed in the set. In these logical expressions, the set name cannot be subscripted because **M.*set*** denotes class, not specific owner, membership. An entity cannot belong to more than one set of a given class (e.g., the same kind of set owned by different entities) at one time.

The first pointer of each set is used to determine whether or not a specific set has members. The logical expressions:

set `is empty`

set `is not empty`

can be used for these decisions.

2.80 REPEAT Statement

See **`do ... loop`** construct.

2.81 REQUEST Statement

The **request** statement is used by processes to request a specific number of resource units. If not available, the requesting process is enqueued in **priority** order and suspended awaiting availability of the resource.

`request` *integer value* [`unit [s] of`] *resource* [(*integer value*)] [[,] `with priority` *integer value*]

EXAMPLES:

```
request 1 WORKER(2) with priority 2
```

Requests one unit of the resource named **WORKER**, of the second type (where **N.WORKER** 2), and assigns a priority.

```
request 2 MACHINE(JOB.TYPE)
```

Requests two units of the resource named **MACHINE**, of the type **JOB.TYPE**, whose value has either been assigned or defined by a **define ... to mean** statement. The priority is treated as zero.

```
request 3 units of MATERIAL with priority 5
```

Requests three units of the resource named **MATERIAL**, of which only one type exists, and assigns a priority.

A process requests a quantity of any given resource using a **request** statement. If the requested quantity of the resource 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 put in a passive state, and a special temporary entity (**qc.e**) is created and filed in the set of resources associated with this process, (*process*), as well as **X.*resource*** or **Q.*resource*** (depending on whether or not the request has been satisfied).

As the resource name is, in fact, a permanent entity name, it should be subscripted. If it is not, the variable of the same name is used as an implicit subscript. This variable is initialized to 1 at resource creation, but care should be taken if it is subsequently altered. Note that some implementations use an implicit subscript value of 1. It is recommended that explicit subscripting be used in all cases.

An optional **with priority** expression may be added to the `request` statement. The queue is ranked on high priority. If the phrase is not present, the priority is treated as zero.

The **request** statement can appear only in process routines.

2.82 RESCHEDULE Statement

See **schedule** statement.

2.83 RESERVE Statement

The **reserve** statement allocates blocks of storage of the specified size to the variable. If **by *** is specified, only pointer space (for multidimensional arrays) is allocated. Otherwise, the data storage is also allocated.

reserve {*variable* [c] as *quantity* [BY] [by *]} [c]

Note: A real quantity will be rounded to integer.

EXAMPLES:

```
reserve column as 10
```

Allocates storage for a base pointer and ten data elements to the array named **column**.

```
reserve codes(*,*) as 4 by 5
```

Allocates storage for base and row pointers and 20 data elements to the array named **codes**. Note that the optional (*,*) notation is used to emphasize the dimensionality of the pointer variable.

```
reserve X, Y, and Z as 6 by 10
```

Allocates storage for base and row pointers and 60 data elements to each of the arrays named **X**, **Y**, and **Z**.

```
reserve A(*) as N, MATRIX(*,*) as N by 2*N, and B(*) as S*T/2
```

Allocates storage for a base pointer and **N** data elements to the array named **A**, for base and row pointers, and $2\mathbf{N}^2$ data elements to the array named **MATRIX**, and for a base pointer and **ST**/2 data elements to the array named **B**.

```
reserve VALUES(*,*) as 7 by *
```

Allocates storage for seven row pointers to the base pointer **VALUES**(*,*).

```
reserve DATA(1,*) as 10
```

Allocates storage for ten data elements to the row pointer **DATA**(1,*).

```
reserve A(*) as 4, MATRIX(J,*) as N+5
```

Allocates storage for a base pointer and four data elements to the array named **A**, and for **N+5** data elements to the row pointer **MATRIX(J,*)**.

The **reserve** statement, which allocates storage to data elements and to pointer variables, can be used to reserve complete or partial arrays. A typical vector or rectangular array can be reserved simply by naming the array and describing the dimensions; pointers required

for the data elements need not be considered for complete arrays. When the **reserve** statement is executed, the system allocates storage for arrays, sets the value at the base and other pointers, and initializes the elements of each array to zero. A subscripted variable must be reserved before it can be used in any statement. The base pointer must be zero for **reserve** to allocate new storage to the array. The **reserve** statement is an executable statement that can appear in any routine, but not in the preamble.

2.83.1 Dimensionality

The dimensionality of each array can be denoted by asterisks in subscript positions. These asterisks can be omitted, however, and SIMSCRIPT II.5 will automatically supply asterisks, causing the dimensionality to agree with specifications in a prior **normally** or **define ... variable** statement. If the dimensionality has not been previously declared, the system uses the number of expressions as the dimensionality.

2.83.2 AS Phrase

The **as** phrase must include one or more arithmetic expressions, each of which denotes an array dimension. An arithmetic expression can include other subscripted variables if the variables have been previously defined. When an expression yields a real value, that value is rounded to an integer before being used as a dimension. An expression cannot yield a zero or a negative value, as either will cause the program to terminate with an error message.

2.83.3 BY * Phrase

A **by *** phrase is used when allocating storage for pointers. In this event, the pointer variable (array name) must have at least one asterisk in a subscript position. When the **reserve** statement includes a **by *** phrase, the asterisk indicates that pointers are being reserved and that subsequent **reserve** statements will allocate data elements to the pointers. Only one **by *** phrase is permitted to indicate that pointers are being allocated, and this phrase must appear after expressions that define dimensions of preceding subscript positions. If the **reserve** statement does not contain a **by *** phrase, SIMSCRIPT II.5 reserves a data array that is pointed to by the named pointer variable.

2.83.4 Pointers and Array Structures

Figure 9 illustrates sample one- and two-dimensional arrays that include the required pointers. As shown, each array has as many computer words as there are data elements in the array, together with the number of words required for pointers. Data element blocks are stored contiguously in memory. Pointers and data elements are related as follows:

1. For a one-dimensional array, the base pointer points to a vector of data elements (a vector is a one-dimensional array).
2. For a two-dimensional array, the base pointer points to a vector of row pointers, each of which points to a vector of data elements.

3. For a three-dimensional array, the base pointer points to a vector of row pointers, each of which points to a vector of column pointers, and each column pointer points to a vector of data elements.
4. For higher dimensional arrays, similar relationships exist.

An array can be reserved in a piecewise fashion by allocating storage to individual pointers. This is done by reserving a vector of pointers, and then, for each pointer in the vector, reserving a new vector of pointers, and so on. This can be carried on for as many dimensions as are desired. When the desired "depth" is reached for a given pointer, a vector of data can be reserved instead of a vector of more pointers. Note that in this scheme, each pointer points to a "sub-array". By reserving arrays in this manner, the programmer can construct ragged tables, manipulate matrices, and create special data structures. Because pointers are storage addresses, they always have integer values.

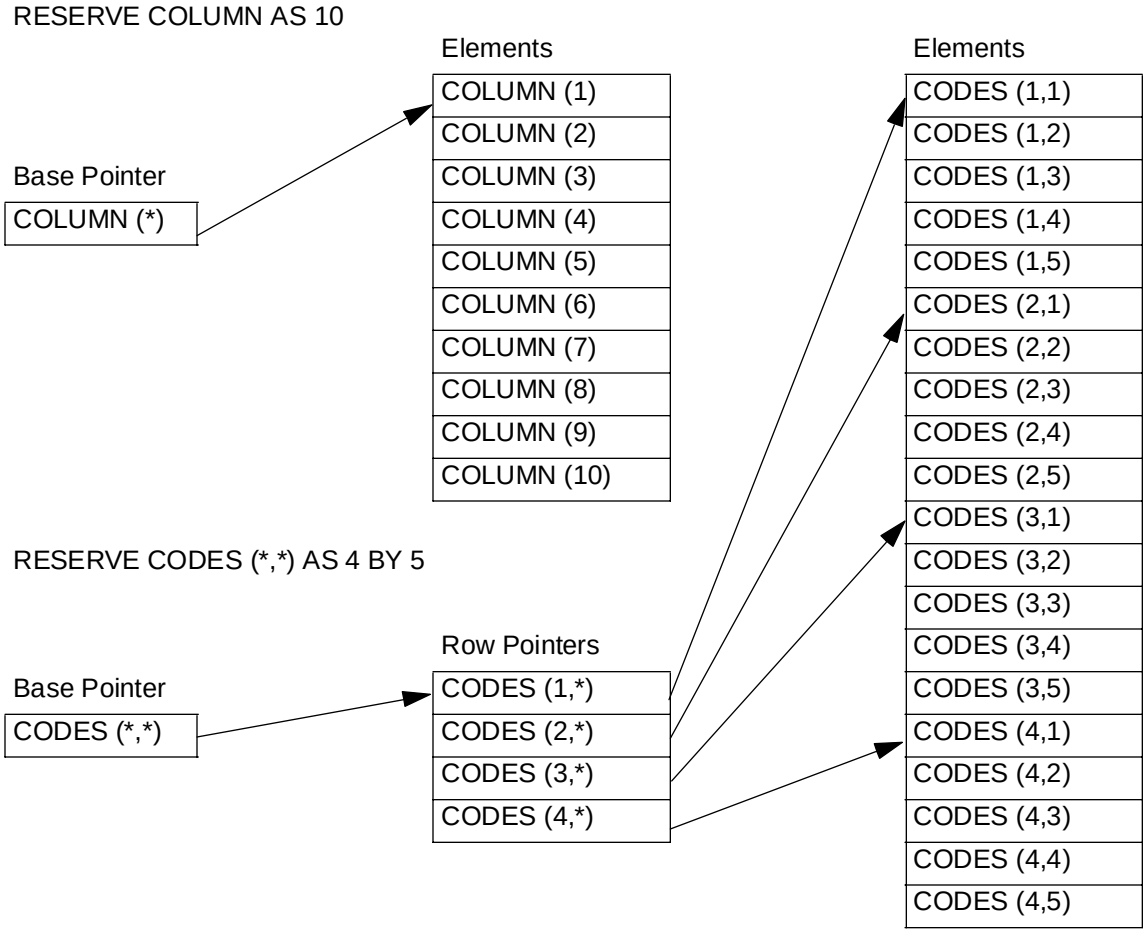


Figure 9. Sample One- and Two-Dimensional Arrays

2.83.5 Function dim.f

Associated with each data or pointer vector is a number indicating the length of that vector. The **dim.f** function accesses this number. This function requires a pointer variable as an argument and returns the length of the vector pointed to. **Dim.f** eliminates the need to save the values of dynamically-computed array sizes in some other variables.

2.83.6 Multiple RESERVE Statements

If a **reserve** statement is executed more than once, SIMSCRIPT II.5 disregards all but the first execution. A nonzero base pointer shows that an array has been reserved. An array reserved by a **reserve** statement can be returned to storage with a **release** statement, and the array (after execution of the **release** statement) is undefined until reserved again. Arrays can be reserved and released continually throughout program execution.

2.84 RESET Statement

The **reset** statement initializes counters used by **accumulate/tally** statements.

```
reset [the][ name ]c total[s] of variable c
```

EXAMPLES:

```
reset totals of LIST
```

Initializes all counters required to **accumulate** or **tally** variable **LIST.**

```
reset weekly totals of POPULATION(CITY)
```

Initializes the **weekly** counters required for attribute **POPULATION** of entities of the class **CITY.**

```
reset weekly and monthly totals of N.QUEUE and POPULATION(CITY)
```

Initializes the **weekly** and **monthly** counters required for attributes **N.QUEUE** and **POPULATION.**

The **reset** statement initializes counters that are associated with variables named in the statement and that are required by **tally** and **accumulate** statements. Each **tally** and **accumulate** variable has a routine that initializes counters used in calculating statistical quantities. Some of these counters are not zero initially. The initializing routines are named **R.*name***, where ***name*** is the variable or attribute being tallied or accumulated.

Qualifying words in a **reset** statement enable reports to be prepared on both a cumulative and periodic basis at the same time. The appearance of one or more qualifiers in a **reset** statement specifies that only the indicated counters, defined in an **accumulate** or **tally** statement for a particular statistic, are to be reset. Any number of qualifiers can be requested. If a **reset** statement does not include qualifiers, all counters associated with the named variables are initialized.

The **reset** statement can appear in any routine, but not in the preamble.

2.85 RESOURCES Statement

The **resources** statement is a preamble statement marking the start of resource entity declarations.

```
resources [include resource^c]
```

Keyword	Synonym
include	are

EXAMPLES:

```
resources
```

Denotes that resource declarations follow. The declarations will be made with **every** statements, and will name user-defined attributes.

```
resources include TELLER and AUTOMATED.TELLER.MACHINE
```

Identifies **TELLER** and **AUTOMATED.TELLER.MACHINE** as resources requiring only the system-defined attributes for resources. **Every** statements may follow this form of the **resources** statement.

The **resources** statement indicates that resource classes are declared in **every** statements that immediately follow, or in the **include** phrase of the **resources** statement. If a resource class has attributes in addition to the system-defined attributes, those attributes must be declared in an **every** statement. Automatically-generated resource attributes appear in table 23.

Table 23. Automatically Generated Resource Attributes

Attribute	Significance	Attribute of
N.*resource*	Number of resources in class.	System
U.*resource*	Number of resource units per resource.	Resource
Q.*resource*	Set of processes waiting for a resource	Resource
N.Q.*resource*	Number of processes waiting for a resource.	Resource
X.*resource*	Set of processes currently using a resource.	Resource
N.X.*resource*	Number of processes currently using a resource.	Resource

2.85.1 Resource Classes

A resource class is similar to a permanent entity class: The preamble declares a group of similar entities, while an individual entity is referred to by an integral subscript in the range 1 to **N.*entity***.

In the **request** or **relinquish** statement, individual resources are referred to by subscripts of the resource class. For example:

```
request LOTSIZE units of CANNER(7)
relinquish 1 PROCESSOR(CPU.NO)
```

As with permanent entities, attributes of resources are specified by subscripting the attribute name with the resource number.

The **create each** statement is used to create the resources in a resource class, which should be done before any resources are requested or any resource attributes are referenced. The number of resources must be specified either in the **create each** statement or by assigning a value to the system attribute **N.*resource***.

2.85.2 Resource Units

Resource units are a measure of the total capacity of a resource or some fraction thereof. Each resource in a resource class has an integral capacity of resource units. The maximum available resource units for a resource is the subscripted attribute **U.*resource***.

In some cases (e.g. bank tellers), the number of resource units may be unity. In others (e.g. a machine capacity), the number of resource units may differ among resources in a resource class.

Each resource of a resource class should be assigned an integral value of the number of resource units available by changing the **U.*resource*** attribute. The following examples illustrate how **U.*resource*** is assigned a value for each resource in a resource class.

```
for each MACHINE, let U.MACHINE = CAPACITY
read U.PRINTER ''reads for I = 1 to N.PRINTER
```

The value of **U.*resource*** is initially zero.

2.86 RESTORE Statement

Not available for all implementations. See *User's Manual* for your computer type.

2.87 RESUME Statement

The **resume** statement is used to restore a previously-interrupted process to the pending list with the remaining "time-to-go" taken from **time.a** (*process*).

```
resume [the [above] ] process [called pointer variable]
```

Keywords	Synonym
the [above]	this

EXAMPLES:

```
resume the GENERATOR
```

Resumes a **GENERATOR** that had previously been interrupted and removed from the pending list. The **GENERATOR** is placed back in the pending list to resume its waiting period.

```
resume SHIP called OTHER.SHIP
```

Resumes the **SHIP** with the pointer variable **OTHER.SHIP** that had previously been interrupted and removed from the pending list. **OTHER.SHIP** is placed back in the pending list to resume its period of work.

A previously-interrupted process may be returned to the active state, that is, replaced in the event set, by a **resume** statement. The statement may be issued by any other routine naming the interrupted process (including another process routine).

The **time.a** attribute at the time of resumption is used to schedule the end of the (active) or (passive) state. It is incremented by the current **time.a** before being used as a ranking attribute for filing in the event set (or any other set).

The **resume** statement can appear in any routine, but not in the preamble.

2.88 RESUME SUBSTITUTION Statement

The **resume substitution** statement reinstates the substitutions previously nullified by a **suppress substitution** statement.

```
resume substitution
```

Only one form of this statement exists.

Substitutions declared by **define ... to mean** and **substitute** statements, which were subsequently nullified by a **suppress substitution** statement, are reinstated with a **resume substitution** statement. This statement should appear on a card by itself because substitutions are nullified for a complete card before the contents are interpreted. If other statements appear on the same card as a **resume substitution** statement, substitutions will not be resumed for these statements before the **resume substitution** statement is recognized. The **resume substitution** statement can appear in the preamble or in any routine.

2.89 RETURN Statement

The **return** statement returns program control to the calling program from a subroutine or from a function routine, and returns to the timing routine from an event routine.

```
return    [with quantity    ]
          [(quantity)       ]
```

EXAMPLES:

```
return
```

Returns control to the calling program if the **return** statement appears in a subroutine, or to the timing routine if the **return** statement appears in an event routine.

```
return(sqrt.f(X))  '  'sqrt.f is a library function
```

Returns control to the calling program from a function routine with the square root of **X**. Commentary text follows the two apostrophe characters.

```
return with NUMBER
```

Returns control to the calling program from a function routine with the value of variable **NUMBER**.

A **return** statement must appear in each subroutine, function routine, and event routine, in order to return control to the calling program or to the timing routine. An event routine always returns to the timing routine, which is the central routine in a simulation model. A routine can have more than one **return** statement to indicate different exits, and each exit can return a different value.

Subroutines and event routines use only the keyword **return**, but a function routine requires either of the following forms:

```
return (quantity)
return with quantity
```

In functions, the value of the quantity is computed before the function returns to the calling program, and the quantity must be in the mode declared in a **define ... routine** statement.

A routine can be both a subroutine and a function if that routine includes both types of **return** statements. In such a case, when control returns from a **return** statement to the calling program, the values assigned to **yielding** arguments are used by the calling program. When the routine returns from a **return with** statement, a single value is returned to the calling program.

2.90 REWIND Statement

The **rewind** statement rewinds an input/output device.

```
rewind   [tape  ] integer value
         [unit  ]
```

EXAMPLES:

```
rewind 5
```

Rewinds the unit identified by the number 5.

```
rewind unit MESSAGES
```

Rewinds the unit whose number is the value of variable **MESSAGES**.

The **rewind** statement "rewinds" the input/output device whose identification is the value of an expression. That is, this statement positions a file on a device to its starting point (first record). **Rewind** can be used to rewind tapes, disks, and drums. After a unit is rewound, that unit must appear in a **use** statement or in the **using** phrase of a **read (Free-form)**, **read (Formatted)**, or **write** statement before reading or writing can occur. If the designated unit is the current input unit when a **rewind** statement is executed, SIMSCRIPT II.5 makes the card reader the input unit. If the designated unit is the current output unit, SIMSCRIPT II.5 changes the output unit to the printer. Of course, the card reader and the printer cannot be rewound. The **rewind** statement can appear in any routine, but it cannot be included in the preamble.

2.91 ROUTINE Statement

The **routine** statement marks the beginning of a subroutine or of a function routine. The prefix **left** or **right** is used for declaring monitoring routines. A routine used as a function has only given arguments.

```
[left  ]routine [to] routine    [given value,c  ][yielding variable,c]
[right ]                        [(value,c)      ]
```

Keywords	Synonyms
routine	function
	subroutine
to	for
given	giving
	the
	this

EXAMPLES:

```
routine SOURCING
```

Names **SOURCING** as a routine having no arguments.

```
left routine RESULT
```

Names **RESULT** as a left-hand routine having no arguments.

```
routine FINANCIAL(SHARES, PRICE)
```

Names **FINANCIAL** as a routine having variables **SHARES** and **PRICE** as input arguments.

```
routine to PRINT.MESSAGE given MESSAGE and LENGTH yielding FLAG
```

Names **PRINT.MESSAGE** as a routine having variables **MESSAGE** and **LENGTH** as input arguments and **FLAG** as an output argument.

The **routine** statement, which must be the first statement of each subroutine and function routine, is used for both left-hand and right-hand routines. If the **routine** statement names a subroutine, that name must appear in a **call** statement in order to execute the subroutine. Subroutines can have both input and output arguments. Arguments are automatically defined as local recursive variables. They need not be declared in **define ... routine** statements in the preamble.

If the **routine** statement names a right-hand function, that function is called by specifying its name, followed by any arithmetic expression enclosed in parentheses. Subsequently, when the function routine is called, values of expressions in the argument list are transferred to local variables in the routine, and the function returns a single value. A function *must* be declared in a **define ... routine** statement in the preamble.

If the **routine** statement names a left-hand function, it is also called by specifying its name and a list of argument expressions. This will appear, however, to the *left* of an equal sign in a **let** statement, as a **yielding** argument to another routine or in a **read** statement. Instead of returning a value to the calling routine with a **with** statement, a left-hand function receives a value from the calling routine via an **enter with** statement. From there, a left routine can perform computations like any other program. Left routines are always used in the function sense rather than as subroutines to be called. All functions *must* be declared in the preamble in a **define ... routine** statement.

2.91.1 Routines Named TO and FOR

If a subroutine or function is named either **TO** or **FOR**, the routine name and the optional words **to** and **for** must be used in a prescribed manner in order to compile correctly. Only the following forms will be compiled:

```
routine to TO          routine for TO
routine to FOR         routine for FOR
```

The practice of naming a routine **TO** or **FOR** should be avoided.

2.91.2 GIVEN Phrase

The **given** phrase can be used for either a subroutine or a function. For both types of routines, arguments in the phrase must be unsubscripted local variables. These will receive values from the calling routine.

2.91.3 YIELDING Phrase

A **yielding** phrase can be appended to the **routine** statement for subroutines. It is not used for functions (although see **return** statement for routines used both as subroutines and functions). Arguments included in a **yielding** phrase must be unsubscripted local variables. These will transmit values from the called routine to the calling routine.

2.91.4 Argument Definitions

A **define ... routine** statement in the preamble can declare the number of arguments for a subroutine or function. If the number of arguments in the **routine** and **define ... routine** statements disagree, SIMSCRIPT II.5 takes the following corrective action:

1. Disregards additional input and output arguments in the **routine** statement.

2. Considers omitted input arguments to be zero.
3. Reserves locations for missing output arguments so they can receive output values.

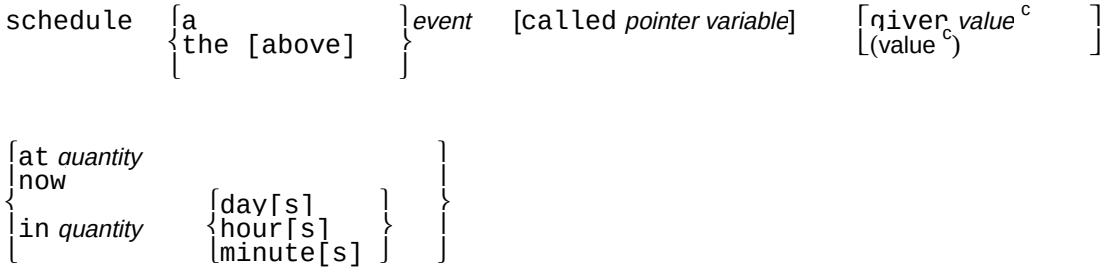
If the **`define ... routine`** statement contains only given arguments, the called routine is assumed to yield no values. If the **`define ... routine`** statement contains only yielding arguments, the called routine is assumed to have no given values.

2.91.5 Argument Modes

Disagreements in mode between arguments in **`call`** statements and corresponding arguments in **`routine`** statements can be difficult to discover because the effects are subtle. For example, an integer number used as a real number (converted to floating point with an exponent) can effectively be zero. SIMSCRIPT II.5 does not automatically check for or convert argument values whose mode differs from that specified in the called routine.

2.92 SCHEDULE (*event*) Statement

The **schedule** statement schedules the future occurrence of an event by filing an event notice in the relevant event set.



Keywords	Synonyms
schedule	reschedule
	cause
For a new event:	
a	an
For an existing event:	
the [above]	this
given	giving
now	next
in	after
day[s]	unit[s]

EXAMPLES:

```
schedule a DELAY at time.v + 0.5
```

Creates an event notice of the class **DELAY**, assigns the identification to global variable **DELAY**, sets attribute **time.a** equal to the value of expression **time.v** + 0.5, and files the notice in the event set.

```
schedule the LANDING called EMERGENCY in 2 * READY minutes
```

Uses an existing event notice of the class **LANDING**, whose identification is assigned to variable **EMERGENCY**, sets attribute **time.a** to the value of the current time plus 2 * **READY** minutes, and files the notice in the event set.

```
schedule a TAKEOFF given FLIGHT.NO, DEST.CODE, and NO.FIRST +
          NO.TOURIST+ NO.ECONOMY in 3 hours
```

Creates an event notice of the class **TAKEOFF**, assigns the identification to global variable **TAKEOFF**, sets the value of attribute **time.a** to the current time plus three hours, stores values of expressions **FLIGHT.NO**, **DEST.CODE**, and **NO.FIRST + NO.TOURIST + NO.ECONOMY** in attributes that immediately follow the special attributes, and files the notice in the event set.

```
schedule a TAKEOFF(FLIGHT.NO, DEST.CODE, and NO.FIRST + NO.TOURIST
+ NO.ECONOMY) now
```

Creates an event notice of the class **TAKEOFF**, assigns the identification to global variable **TAKEOFF**, sets the value of attribute **time.a** to the current time, stores values of expressions **FLIGHT.NO**, **DEST.CODE**, and **NO.FIRST + NO.TOURIST + NO.ECONOMY** in attributes that immediately follow the special attributes, and files the notice in the event set.

The **schedule** statement, which is used only for simulation, schedules the future occurrence of an internal event for the named class. This statement assigns values to event notice attributes and files the notice in the event set, **ev.s**. In each event set, event notices are ranked in ascending order of attribute (the time the event is to occur). Depending on the form selected, this statement can create an event notice or use an existing one, can schedule an event to occur at a future time, at some relative time, or at the current time, and can assign values to attributes of the event notice. It can appear in any routine, but not in the preamble. In this statement, the keywords **an** and **the** are important: **An** designates that an event notice is to be created, while **the** indicates that the event notice already exists.

Figure 10 illustrates a sample event notice which assumes that attributes **FLIGHT.NO**, **DESTINATION**, and **NO.PASSENGERS** were declared in an **every** statement for an event notice of the class **TAKEOFF**. The **schedule** statement assigns the event time to attribute **time.a** and always sets **eunit.a** to zero because the event is being scheduled internally. In addition, this statement assigns values to attributes used only by the timing routine: **p.ev.s** (predecessor in set), **s.ev.s** (successor in set), and **m.ev.s** (membership in set). If the **schedule** statement includes expressions whose values are to be assigned to attributes, the values are assigned in the order of their appearance. When an event notice owns sets, or belongs to sets, owner and member attributes follow attributes declared by the programmer.

2.92.1 CALLED Phrase

A **called** phrase included in a **schedule** statement indicates either 1) that the identification of the event notice is to be assigned to the named variable rather than to the global variable having the event class name; or 2) if a **the** keyword signals that an event notice already exists, SIMSCRIPT II.5 assumes that the variable named in the **called** phrase contains the event notice identification.

2.92.2 GIVEN Phrase

A **given** phrase assigns values to attributes of the event notice. These values in turn are assigned by the timing routine as arguments for the respective event routine when it is executed. The **schedule** statement assigns values of expressions to successive attributes of the event notice, starting with the attribute that follows . Values are assigned to programmer-defined attributes in the same order in which attributes appear in **every** statements in the **preamble**. (Use of **in word** phrases to arrange physical storage of attribute values does not influence the order of selection of event arguments.) If there are fewer expressions than attributes, SIMSCRIPT II.5 assigns zeros to the remaining attributes. If no expressions appear in the statement, and an event notice has defined attributes, the attributes are set to zero. The keyword **given** can be omitted if expressions are enclosed in parentheses.

WORD 1	`time.a` time event is to occur
2	`eunit.a` = 0 for internal; (0 for external
3	`p.ev.s` predecessor in event set
4	`s.ev.s` successor in event set
5	`m.ev.s` membership attribute
6	`flight.no` declared attribute for this event
7	`destination` declared attribute for this event
8	`no. passengers` declared attribute for this event

Note: The first five sample event notices are system maintained.

Figure 10. Sample Event Notices

2.92.3 AT Phrase

The **at** phrase, which denotes when in the future the named event is to occur, sets attribute **time.a** to the value of an expression. The expression must yield a real value. **Time.a** is used to file this event notice in the event set in chronological order (the event set is ranked on low **time.a**) .

The value of **time.a** is used to update **time.a**, the current simulation time, whenever an event notice or a process notice is selected from the event set by setting it to the **time.a** value of the first event or process in the event set. The system sets **time.v** to zero at the start of simulation and increases **time.v** as the simulation progresses. An absolute time must always be specified in an at phrase.

2.92.4 IN Phrase

An **in** phrase specifies the relative time at which an event is to occur. This phrase enables the programmer to schedule an event at **time.v** plus a designated number of days, hours, or minutes. (The keyword **units** can be used in place of **days**.) The units of **time.v** are days if the **days, hours**, or **minutes** keywords are used in this phrase. If **hours** or **minutes** is specified, the hours or minutes are automatically converted to days using the system variables **hours.v** and **minutes.v**. The system initializes **hours.v** to 24 and **minutes.v** to 60, but the programmer can modify these values. If **time.v** runs in units other than days, the **units** keyword should be used in an **in** phrase.

2.92.5 NOW Phrase

An event scheduled with a now phrase occurs as soon as the current event or process returns control to the timing routine. Such an event will occur before any events or processes having the same event time, scheduled previously with **at** or **in** phrases. When **schedule** statements include **now** phrases for two or more events, the events are ranked according to the **priority** statement if they are of different classes, according to the **break ... ties** statement if that statement appears for the event class, or first-in, first-out if a **break ... ties** statement has not been included for the event class.

2.93 SELECT CASE Statement

The **select case** statement is useful for distinguishing several cases in control flow. Instead of writing nested **if** statements, a **select case** statement can be written. The syntax is:

```
select case EXPR
   case CONST_LIST
      STMT_GROUP
   case CONST_LIST
      STMT_GROUP
endselect
```

Meaning of the statement elements:

EXPR Expression of any mode.

CONST_LIST List of constants (of the mode **EXPR**) that will select this case. Multiple values are separated by commas.

The modes of **EXPR** and **CONST** must agree.

If **EXPR** is numeric (**integer, real** or **double**), the values used as **CONST** must be numeric.

If **EXPR** is of mode **alpha** or **text**, then **CONST** must be a literal string delimited by double quotes.

If **EXPR** is a subprogram variable, **CONST** must be a subprogram literal delimited by single quotes.

STMT_GROUP Group of statements to be executed in the selected case (0 or more statements).

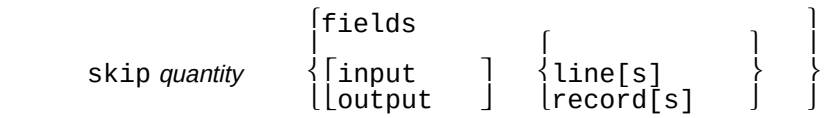
EXAMPLE:

```
'' - - switch on entered CMD

          select case CMD
             case "ON"     STATUS = 1
             case "OFF"    STATUS = 2
             case "E"      '' - - nothing happens here
             default       write as "*** illegal CMD ***",/
          endselect
```

2.94 SKIP Statement

The **skip** statement skips fields, records, and lines, and applies to the current input or current output unit.



Note: A real quantity will be rounded to **integer**. **Record[s]** imply **input**; **line[s]** imply output.

EXAMPLES:

```
skip 4 fields
```

Skips four fields on the current input unit.

```
skip 1 line
```

Skips one line on the current output unit.

```
skip 2 * (n - m + 1) records
```

Skips the remainder of the current record on the input unit, as well as a number of records equal to the value of the expression minus one.

The **skip** statement skips fields, records, and lines on either the current input or the current output unit. The keyword **fields** specifies that some number of input data fields are to be skipped. Data fields are contiguous strings of characters delimited by at least one blank.

When the system reads a field, it waits at (i.e., points to) the end of the field for the next **read (Formatted)** statement. To skip a field means to position the input pointer at the end of the field without reading it. Skipped fields can be on several input cards. Note that when the system reads a field at the end of a data card, that card is retained until the next **read** or **skip** statement is executed.

Using the **skip fields** statement with formatted **read** statements requires keeping very careful track of where the input pointer is, and on which input record. A **read (Free-form)** statement automatically reads data values as they occur on input cards regardless of their exact position.

When skipping records on the current input unit, the **skip** statement can skip the remainder of a current data record when written as **skip 1 record**. The statement:

```
skip QUANTITY records
```

skips the remainder of the current data record and the next ***QUANTITY*-1** records.

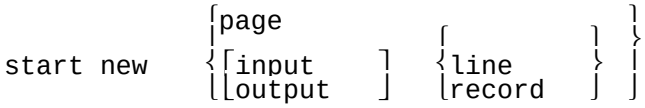
When skipping output lines, the following rules apply:

1. If the value of the expression is negative, the system sets that value to zero.
2. If the value of the expression is greater than the value of **`lines.v`** (number of lines permitted per page), the value of the expression is set to the value of **`lines.v`**. Thus, the maximum number of lines that can be skipped with this statement is one page.

When skipping records, or lines, the input (output) pointer is positioned at the beginning of the next input (output) record.

2.95 START NEW Statement

The **start new** statement starts a new page, a new record, or a new line, and is applied to the current input or output unit.



Note: Record implies input; **line** implies output.

EXAMPLES:

```
start new page
```

Ejects a page on the current output unit.

```
start new record
```

Starts a new **record** on the current input unit.

```
start new output record
```

Starts a new **record** on the current output unit.

The **start new** statement can be used to skip the remainder of a **record** on the current input unit, to eject a page before printing, and to start a new line on the current output unit. This statement can appear in any routine, but it cannot be included in the preamble.

2.96 START SIMULATION Statement

The **start simulation** statement begins a simulation by passing control to the timing routine, which removes the first event notice from the event set and executes that event.

```
start simulation
```

Only one form of this statement exists.

The **start simulation** statement causes the timing routine to begin selecting and executing events and processes from the event set. When a **start simulation** statement is executed, the timing routine first initializes the external event reading mechanism, if there are any external events. It reads information regarding the first event named on each external event unit, schedules the first event from each unit, and initializes the system for subsequent external events. The first event is selected from the event set and executed. While events are being executed, the timing routine, which is called by the **start simulation** statement, controls program execution. When the timing routine finds no more scheduled events in the event set, program control transfers to the statement that follows the **start simulation** statement. Therefore, a simulation can be stopped by simply ceasing to schedule future events. Control will then eventually pass to the statement after the **start simulation** statement when events that are currently scheduled have been executed.

Every simulation model must have a **start simulation** statement. The statement can appear in any routine, but not in the preamble.

2.97 STOP Statement

The **stop** statement halts program execution and is used to signal the logical end of a program and return control to the operating system.

```
stop
```

Only one form of this statement exists.

The **stop** statement, which terminates program execution, can appear in any routine, but cannot be included in the preamble. A program can have any number of **stop** statements, and this statement need not be placed at the physical end of the program deck.

To exit from the program and return status to the operating system, you can use **exit.r** instead of the **stop** statement.

2.98 STORE Statement

The **store** statement sets a variable equal to the value of an expression, without mode conversion.

`store` *value* `in` *variable*

EXAMPLES:

```
store X in Y
```

Stores the value of **X** in variable **Y**.

```
store pi.c * R**2 in CONSTANT   ''pi.c is a system variable
```

Stores πr^2 as the value of **CONSTANT**. Commentary text follows the two apostrophe characters.

```
store "laff" in MOVIE(FLIGHT)
```

Stores the alphanumeric literal **laff** as the value of attribute **MOVIE** for entity class **FLIGHT**.

The **store** statement permits the value of an expression to be assigned to a variable without altering the mode of either the expression or the variable. It is similar to the **let** statement, although when mixed modes appear, the **let** statement converts the expression to the mode of the variable. For example, the **store** statement can be used to store integer or real values in real variables when one does not know *a priori* whether the mode of the value will be real or integer. This statement can appear in any routine, but it cannot be included in the preamble.

An arithmetic expression specified in the `store` statement can be any of the following:

1. An integer expression, which can be a data value, an array pointer, or an entity identification number
2. An integer constant
3. A real constant or expression
4. An alphanumeric variable or literal
5. A subprogram variable.

2.99 SUBSTITUTE Statement

The **substitute** statement permits a string to be substituted for a word in the succeeding statements. The string can appear on one or more cards.

```
substitute   [this (integer) line     ]for string
             {these integer lines     }
             [                        ]
```

EXAMPLES:

```
substitute this line for X
     matrix
```

Substitutes the word **MATRIX** for the character **X**.

```
substitute this line for FORMULA
     A * X**2 + B * X + C
```

Substitutes the expression **A * X**2 + B * X + C** for the word **FORMULA**.

```
substitute these 2 lines for INPUT.FORMAT
     B 5, I 10, S 3, D(7,2),/, B 25, 3 I 6
```

Substitutes the string **B 5, I 10, S 3, D(7,2), /, B 25, 3 I 6** for the word **INPUT.FORMAT**.

```
substitute these 3 lines for ANSWER
     let X = A + B
     call CALCULATE(X)
     go to NEXT
```

Substitutes:

```
let X = A + B
call CALCULATE(X)
go to NEXT
```

for the word **ANSWER**.

The **substitute** statement replaces the designated word, which may be a name, a number, an alphanumeric literal, or a character, with the lines that immediately follow the statement. During compilation, whenever the compiler detects the specified word, it substitutes the string for the word and compiles the statement with the substitution. This statement is similar to the **define ... to mean** statement, but **substitute** permits a string to appear on several individual cards. The **substitute** statement offers extensive capabilities for generating macro instructions; whole sequences of statements can be

inserted directly into a program. Substitution can appear in strings for other **substitute** or **define ... to mean** statements permitting several levels of substitution.

The **substitute** statement can appear anywhere in the preamble and in routines. When it appears in the preamble, the substitution affects the entire program. In a routine, the substitution is local and affects only that routine until superseded. A **suppress substitution** statement can override the effect of a **substitute** statement, while a **resume substitution** statement can reinstate the effect.

2.99.1 Purposes of SUBSTITUTE

The **substitute** statement can be used for any of the following purposes:

1. To change a word in a routine to the same word used in other routines in a large program.
2. To change statement keywords to another vocabulary.
3. To define a macro instruction, that is, a compound instruction generated from a single keyword.
4. To define format strings in order to call them by name, so as to minimize the number of characters that must be written when several statements have identical format lists.
5. To define names as synonyms, substitute one variable name for another, and replace a name with complete statements.

Redefining statement keywords must be handled carefully to avoid substituting a new string for an optional keyword, or for any other characters that might cause incorrect compilation because the statement syntax was not followed.

2.99.2 Rules

The following rules apply to the **substitute** statement:

1. A line to be substituted cannot contain comments, cannot consist entirely of blanks, and cannot be the form line of a **print** statement.
2. Substitution will not take place if the word is embedded in nonblank characters.

2.100 SUBTRACT Statement

The **subtract** statement subtracts the value of an arithmetic expression from the value of a variable, and the difference becomes the new value of the variable.

```
subtract quantity from variable
```

EXAMPLES:

```
subtract 2.5 from X
```

Subtracts 2.5 from the value of variable **X**.

```
subtract COLUMN(I) from LIST(I)   ''one-dimensional arrays
```

Subtracts the **I**th element in array **COLUMN** from the **I**th element in array **LIST**. Commentary text follows the two apostrophe characters.

```
subtract CONSUMED.FUEL(FLIGHT) from TOTAL.FUEL(FLIGHT)
```

Subtracts the value of attribute **CONSUMED.FUEL** from the value of **TOTAL.FUEL** for the entity whose identification number is contained in the variable **FLIGHT**.

The **subtract** statement, which subtracts the value of an arithmetic expression from the value of a variable, is similar to the **let** statement, although the **subtract** statement includes the subtraction operator in the statement itself. The **subtract** statement can appear in any routine, but not in the preamble. If the expression and the variable differ in mode, SIMSCRIPT II.5 converts the expression to the mode of the variable before assigning the difference to the variable (see the **let** statement for conversion rules).

2.100.1 Complex Subscripted Variables

Before compilation, the **subtract** statement is translated to:

```
let variable = variable - quantity
```

If the variable has complex subscript references, it is more efficient to compute the subscripts separately than to have the compiler compute them twice. For example:

```
subtract 1 from X(Y*(AB-2),DIFF**N)
```

translates to:

```
let X(Y*(AB-2),DIFF**N) = X(Y*(AB-2),DIFF**N) - 1
```

which causes the subscripts **`Y*(AB-2)`** and **`DIFF**N`** to be evaluated twice. To conserve storage space and computer time, this `subtract` statement could be written as:

```
let I = Y*(AB-2)
let J = DIFF**N
subtract 1 from X(I,J)
```

2.100.2 Subscripts Containing Functions

If the subscript of the variable is a function, or contains a function, unexpected results can occur. This is especially true when a function is involved that has side effects, such as calling the random number generator. For example:

```
subtract 1 from TABLE(uniform.f(A,B,1))
```

translates to:

```
let TABLE(uniform.f(A,B,1)) = TABLE(uniform.f(A,B,1)) - 1
```

before compilation. This causes two random numbers to be generated, and possibly two different elements of **`table`** to be accessed. The intent may have been:

```
let I = uniform.f(A,B,1)
subtract 1 from TABLE(I)
```

2.100.3 Error Messages

A **`subtract`** statement having complex subscripted variables or function references can cause duplicate error messages to be produced because of intermediate translations.

2.101 SUPPRESS SUBSTITUTION Statement

The **suppress substitution** statement nullifies all current substitutions.

```
suppress substitution
```

Only one form of this statement exists.

The effect of **define ... to mean** and **substitute** statements is nullified by a **suppress substitution** statement. This statement should appear on a card by itself, because a substitution takes place for a complete card before the contents are interpreted. If other statements appear on the same card with a **suppress substitution** statement, substitutions are made for these statements before the **suppress substitution** statement is recognized. The **suppress substitution** statement can appear in the preamble or in any routine. All substitutions are reinstated with a **resume substitution** statement.

2.102 SUSPEND Statement

The **suspend** statement is used to place the current process in the passive state and return control immediately to the timing routine without destroying the current process.

```
suspend [process]
```

EXAMPLES:

`suspend`

Places the current process (corresponding to the routine in which the statement appears) in the passive (created) state, and returns control immediately to the timing routine.

`suspend process  ''ship`

The same as above, with an optional keyword and a comment adding to the clarity of the statement.

When executed in a process routine, the **suspend** statement causes the current process to be placed in a suspended state and control to be relinquished to the timing routine. The process will not resume execution unless the model saves the pointer to the process notice and another routine reactivates the process. Once reactivated, the process resumes execution after the **suspend** statement.

The **suspend** statement may only appear in a process routine.

1.103 SYSTEM Statement

See **`the system`** statement.

1.104 TALLY Statement

See **`accumulate/tally`** statement.

1.105 TEMPORARY ENTITIES Statement

The **temporary entities** statement indicates that **every** statements which follow declare temporary entities.

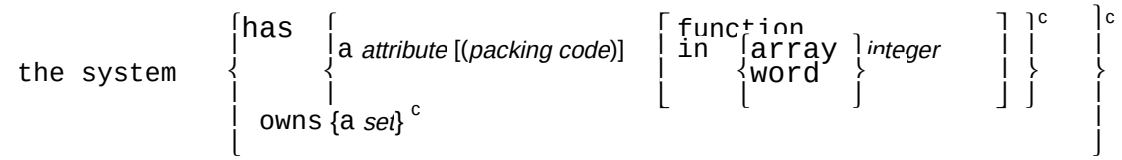
```
temporary entities[include entity^c]
```

Only one form of this statement exists.

The **temporary entities** statement, which can appear only in the preamble, indicates that entity classes named in the following **every** statement are temporary. Storage for each entity of the entity class is allocated individually as the entity is created with a **create** statement. Several **temporary entities** statements may appear in the preamble, and each can be followed by a group of **every**, **define ... variable**, and **define ... set** statements.

1.106 THE SYSTEM Statement

A **`the system`** statement specifies attributes of the system and sets owned by the system. It also specifies optional attribute packing, equivalencing, word assignments, and functions.



Keywords	Synonyms
`has`	`can have`
	`may have`
`a`	`an`
	`the`
	`some`

EXAMPLES:

```
the system has a CONTROL.VALUE and a CODE
```

Declares that **`CONTROL.VALUE`** and **`CODE`** are system attributes.

```
the system has some RULES and owns a QUEUE
```

Declares that **`RULES`** is a system attribute and that **`QUEUE`** is a system-owned set.

```
the system owns a QUEUE and has an F.QUEUE in array 1 and an L.QUEUE
in array 2
```

Declares that **`QUEUE`** is a system-owned set, and that the first-in-set pointers, **`F.QUEUE`**, are to be assigned to array one and the last-in-set pointers, **`L.QUEUE`**, to array two.

```
the system has some FAA.REGULATION.NO(*/4)
```

Declares that values of the system attribute **`FAA.REGULATION.NO`** are to be intra-packed with four consecutive values per word.

```
the system has a (CONTROL.VALUE(L/2), CODE(3/4), NUMBER(17/24))
```

Declares that **`CONTROL.VALUE, CODE`**, and **`VALUE`** are system attributes. A value of **control.value** is to be stored in the first half of a word, a value of **`CODE`** in the third quarter of the same word, and a value of **`NUMBER`** in bits 17 through 24 of the same word.

```
the system has a DECISION function and a NEW.RULE
```

Declares that **DECISION** is a function attribute and **NEW.RULE** is an attribute.

A **the system** statement declares system attributes and sets owned by the system. System attributes are particularly useful as pointers which enable the system as a whole to own sets. Another advantage of system attributes is that they can be packed, equivalenced, or placed in specific array locations, while global variables cannot. In the statement format, the keywords **the system** are followed by attribute phrases and set-owner phrases: **has** denotes an attribute phrase while **owns** denotes a set-owner phrase. The phrases can be in any desired order, and more than one of each phrase type can appear in a single statement. In an attribute phrase, names can be made equivalent (synonymous), and packing factors can be included to declare the word portions, or specific bits, to be occupied by values of an attribute. In addition, values of system attributes can be assigned to specific array locations, enabling the compiler to generate more efficient code than it can when array assignments are omitted.

Other variations of an attribute phrase are used to name function attributes and dummy attributes.

The following general rules apply to a **the system** statement:

1. More than one of **the system** statements can appear in the preamble.
2. The current background mode is assigned to the declared system attributes, except for automatically generated set pointers, which always have integer values. Mode can be overridden with subsequent **define ... variable** statements.
3. System attributes are subscripted if the current background dimensionality is greater than zero. Dimensionality can be overridden in subsequent **define ... variable** statements.
4. Subscripted system attributes used as data, or as set pointers, must be reserved with a **reserve** statement before the attributes can be used.
5. Subscripted system-owned sets can be defined by setting the background dimensionality so that set pointers are arrays.

The same data value can be given several names by placing the names in parentheses, separated by commas, in an attribute phrase. A value will thereby occupy a memory location that is referenced by several names.

Equivalent attributes are assigned to the same computer word. Any attribute can be equivalenced, except text attributes, which can only be equivalenced to other text attributes.

1.106.1 Packing

Packing is defined as storing two or more values in a single word. Values can be packed into fractions of a word (e.g., a byte), or into specific bits, or several values of one attribute can be packed into a single word. In a **the system** statement, packing is specified by appending a packing factor, enclosed in parentheses, to an attribute name.

Field packing designates which fraction of a word — typically a half, quarter, or sixth — is to be occupied by values of the named attribute. For example, the packing factor (1/2) specifies the first half of a word, and (4/4) specifies the fourth quarter of a word.

A bit packing factor designates bits to be occupied by values. For example, a bit packing factor of (7-10) specifies that values of an attribute are to occupy bits 7 through 10 of a word. Bits are numbered sequentially from left (most significant) to right (least significant) starting with 1.

An intrapacking factor is specified by **(*/integer)**. The asterisk denotes intrapacking, and ***INTEGER*** is the number of values to be packed per word. For example, intrapacking notation **(*/2)** packs two consecutive values per word.

The following rules apply to packing:

1. Packing is specified by appending a packing factor, enclosed in parentheses, to an attribute name.
2. If values of more than one attribute are to occupy the same word, the attribute names must be enclosed in parentheses, separated by commas, for example, **(first (1/2), second(4/4))**.
3. More than one group of attributes to be packed can be specified in a single **the system** statement.
4. System attributes can be packed, but global variables cannot.
5. Integer and alpha values can be packed, but text values cannot. Real values and set pointers may or may not be packed, depending on the implementation.
6. All integer and alpha subscripted system attributes can have field, bit, and intrapacking. Zero-dimensional system attributes cannot be packed in any way.
7. Overlapping packing specification is allowed.
8. If two attributes have the same packing factors, their names are synonymous.
9. Packing does not apply to function attributes or dummy variables.
10. The default for packed integers is unsigned. Signed integers can be specified in a **define ... variable** statement.

1.106.2 Function Attributes

A function attribute is defined as an attribute whose value is computed by a function routine. Consequently, a routine must be written having the same name as the attribute, and the system does not allocate storage in entity records for the values.

2.106.3 Dummy Variables

A dummy attribute, which does not have a storage location, must be declared in a **define** statement. The attribute must also appear in either a **the system** statement or an **every** statement. This declaration permits the dummy attribute to be used in **tally** and **accumulate** statements without having its value stored.

2.107 [THEN] IF Statement

See **`if ... else ... always`** construct.

2.108 TRACE Statement

The **trace** statement provides a backtrack of the current function and subroutine calls.

```
trace   [                                  ]
        |using [tape ] integer value       |
        [      [unit ]                     ]
```

EXAMPLES:

`trace`

Beginning with the location of this **trace** statement, provides a backtrack of function or subroutine calls using the current output unit.

`trace using unit CHECK.OUT`

Beginning with the location of this **trace** statement, provides a backtrack of function or subroutine calls using the output unit whose device number is the value of variable **CHECK.OUT**.

The **trace** statement provides a dynamic map of the function and subroutine calls that are in effect when the statement is executed. It can be inserted in programs, for example, where error tests are made. Subsequently, the programmer can reconstruct the flow and locate the source of the error. This statement displays the memory location from which the **trace** statement was executed, as well as the names of all higher-level calling routines. The **trace** statement can appear in any routine, but not in the preamble. SIMSCRIPT II.5 itself uses the **trace** statement whenever it detects an error.

2.108.1 USING Phrase

The **trace** statement normally displays output on the current output unit. A **using** phrase, however, can locally override the current output unit declared in a **use** statement. This phrase designates a device as the output unit for the duration of the **trace** statement execution. After execution of the **trace** statement, SIMSCRIPT II.5 automatically reassigns the previous unit as the current output unit. If a program includes an error test while a tape, disk, or drum is the current unit, a printer should be specified before the **trace** statement is used. SIMSCRIPT II.5 displays the output on the standard output device whenever it uses the **trace** statement.

2.108.2 Output

Output from the **trace** statement includes the following information:

```
at location...
called from...
called from...
```

```
    .
    .
    .
called from...
```

The **`at location`** line displays the memory location of the **`trace`** statement in the object program. The first **`called from`** line displays the name of the routine that called the routine that included the **`trace`** statement, and so on. A **`called from`** line does not appear if the **`trace`** statement is executed in the main routine. (In order to interpret the output, see the *User's Manual* for the particular implementation.)

In addition, for each routine in the backtrack sequence, the system prints the computer representation values (e.g., hexadecimal, octal) of all input and output arguments, as well as the values of any recursive local variables.

Current values are also printed for the following system variables:

`time.v`	Current simulated time.
`event.v`	Zero for an internal event or the number of the external event unit.
`read.v`	Number of the current input unit.
`write.v`	Number of the current output unit.

For each active input/output device, the system displays values of:

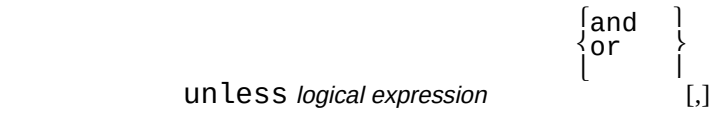
`record.v`	Number of the current input or output record or output record.
`rcolumn.v`	Column number in the current input or output pointer.

or

`wcolumn.v`	
`eof.v`	End-of-file action code.

2.109 UNLESS Phrase

An **unless** phrase is used to control the iterations of a preceding **for** phrase. If the logical expression is true, the controlled statement is not executed. Iteration continues regardless of the value of the logical expression. It may also be used with **while** or **until** phrases.



Keyword	Synonym
unless	except when

EXAMPLES:

```
for I = 1 to N, unless X(I)**2 / Y(I)**2 < MAX
```

Controlled statements will be executed with values generated by the **for ... to (*index*)** phrase when the logical expression **X(I)**2 / Y(I)**2 < MAX** is false.

```
for each AIRPORT, unless NO.OF.RUNWAYS(AIRPORT) less than
MIN.NUMBER and AREA(AIRPORT) less than MIN.AREA
```

The controlled statements will be executed with values generated by the **for ... to (*index*)** phrase when both logical expressions **NO.OF.RUNWAYS(AIRPORT) less than MIN.NUMBER** and **AREA(AIRPORT) less than MIN.AREA** are false.

```
for every PATRON of RESERVATIONS except when FARE(PATRON)
ls LOW or DESTINATION(PATRON) = "sfo" is true
```

The controlled statements will be executed with values generated by the **for ... of (*set*)** phrase only when both logical expressions **FARE(PATRON) is LOW** and **DESTINATION(PATRON) = "sfo"** are false.

An **unless** phrase can be appended to any of the **for** phrases and to **while** and **until** phrases when **while** and **until** are used as independent statements. Logical expressions in an **unless** phrase are tested for each new value of the index variable in the **for** phrases. If the logical expression is true, the controlled program segment is not executed. Conversely, the controlled statements are executed if the logical expression is false.

2.110 UNTIL Phrase

An **until** phrase is used to control iteration in a **for** phrase. As long as the logical expression is false, the controlled statements are executed and iteration continues. An **until** phrase may also be used independently of a **for** phrase.

```
                                    ⎧and  ⎫
                                    ⎨or   ⎬
                                    ⎩     ⎭
[also] until logical expression                [,]
```

EXAMPLES:

```
for I = 1 to N, until X(I) = Y(I)
```

Allows values to be transmitted from the **for ... to (*index*)** phrase to the controlled statements as long as the logical expression **X(I) = Y(I)** is false. The loop terminates when the logical expression is true or the loop is exhausted.

```
for each CITY, until COUNTRY(CITY) ne "us"
```

Values will be transmitted from the **for each (*class*)** phrase to the controlled statements as long as the logical expression **COUNTRY(CITY)** ne **"US"** is false. The loop terminates when the logical expression is true or the loop is exhausted.

```
also until mode is alpha, do
```

Allows values to be transmitted to the controlled statements as long as the logical expression **mode is alpha** is false. Program segment execution terminates when the logical expression is true. The keyword **also** is assumed to eliminate a redundant **loop** statement.

An **until** phrase can be appended to any of the **for** phrases, or it can appear as an independent statement. This phrase allows values to be transmitted from preceding **for each (*class*)**, **for ... of (*set*)**, and **for ... to (*index*)** phrases, to the controlled statements as long as the logical expression is false. Logical expressions in an **until** phrase are tested for each new value of the index variable in **for** phrases. If the logical expression is false, the controlled program segment is executed, and selection terminates when the logical expression is true.

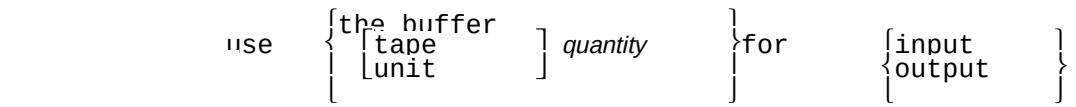
Until phrases can appear as independent statements. In this case, variables specified in the logical expressions are set by computations performed within the range of the phrase, and not from **for** phrase iterations. The range of an independent **until** phrase must be delimited by a **do ... loop** construct. The programmer must be careful to provide for the termination of a loop because SIMSCRIPT II.5 does not automatically terminate an independent **until** phrase. **Until** phrases can be modified by **with** and **unless** phrases, and can be nested with other independent **until** and **while** phrases and with **for** phrases. If nested **until** phrases end on the same **loop** statement, the keyword **also** can precede **until** to eliminate redundant **loop** statements.

2.111 UPON Statement

See **event** statement.

2.112 USE Statement

The **use** statement establishes the indicated input or output device as the current input or output unit. All subsequent input/output statements that do not specify their own devices in **using** phrases use these current units. Specifying **the buffer** causes reading or writing to an internal file.



Note: A real quantity will be rounded to integer.

EXAMPLES:

```
use 5 for input
```

Declares that unit 5 is the current input unit.

```
use tape WRITE for output
```

Declares that the unit whose number is the value of variable **WRITE** is to be the current output unit.

```
use the buffer for output
```

Declares that the buffer is to be used for output; that is, data are written on an internal file.

Use statements designate the current input and output units. Thereafter, all **read (Freeform)**, **read (Formatted)**, and **write** statements that do not include **using** phrases read from and write to these units. Input/output statements can be used with devices other than the standard card reader and printer. Executing a **use** statement causes the designated device to become the current input or output unit.

A device cannot be used simultaneously for both input and output. If a **use** statement names the current input unit as the current output unit, SIMSCRIPT II.5 changes the input unit to the standard card reader. If a use statement names the current output unit as the current input unit, SIMSCRIPT II.5 changes the output unit to the standard line printer. These conventions ensure that error messages are correctly displayed, and aid in detecting errors in device assignments. Of course, the standard output unit (printer) can never be used for input, and the standard input unit can never be used for output.

The **use** statement can appear in any routine, but not in the preamble.

2.113 WAIT/WORK Statement

The **wait/work** statement causes a process to remain in the passive/active state for a specific period of time.

```
⎧wait  ⎫          ⎧day[s]    ⎫
⎨      ⎬ quantity ⎨hour[s]   ⎬
⎩work  ⎭          ⎩minute[s] ⎭
```

Keywords	Synonyms
day[s]	unit[s]

EXAMPLES:

```
wait 7 days
```

Halts execution of the current process for 7 days with **sta.a** set to **0** (passive state).

```
work JOB.TIME minutes
```

Halts execution of the current process for a quantity of minutes equal to the value of the variable **JOB.TIME**; is set to **1** (active state).

```
work exponential.f(MEAN.TIME, 3) hours
```

Halts execution of the current process for a quantity of hours determined by drawing from an exponential distribution with **MEAN.TIME** as its mean, using the third random number stream; **sta.a** is set to **1** (active state).

Within a process routine, a **wait** or **work** statement may be used to halt the process execution for a given lapse of simulated time. The effect of these statements is to file the process notice associated with the process back in the event set, after adjusting the **time.a** attribute to indicate the future time at which execution of the process routine should resume. When simulated time has advanced so that the process notice again becomes eligible for execution, this execution is resumed at the statement following the **wait** or **work** statement. Other events and activities may, of course, be executed during the time lapse. The two statements differ only in the status attributed to the process during the passage of simulated time. This status is recorded in a special attribute of the process notice, **sta.a**, where it may be interrogated by any other executing routine. **Sta.a** is useful for gathering statistics about the states of processes. Values for possible states are shown in table 4 (Part I of this publication).

The **wait/work** statement may only appear in a process routine.

2.114 WHEN Phrase

See **with** phrase.

2.115 WHILE Phrase

A **while** phrase is used to control iterations in a **for** phrase. As long as the logical expression is true, the controlled statements are executed and iteration continues. A **while** phrase may also be used independently of a **for** phrase.

```
                                    ⌈and  ⌉
                                    {or   }
                                    ⌊     ⌋
[also] while logical expression             [,]
```

EXAMPLES:

```
for I = 1 to N, while X(I) is positive
```

Allows values to be transmitted from the **for ... to (*index*)** phrase to the controlled statements as long as the logical expression **X(I) is positive** is true.

```
for each FLIGHT of ARRIVALS while ORIGIN(FLIGHT) ne "lax"
```

Allows values to be transmitted from the **for ... of (*set*)** phrase to the controlled statements as long as the logical expression **ORIGIN(FLIGHT) ne "lax"** is true.

```
while VALUE ls MAX, unless VALUE = X, do
```

Controlled statements will be executed as long as the logical expression **VALUE ls MAX** is true except when the logical expression **VALUE = X** is also true.

A **while** phrase can be appended to any of the **for** phrases or it can appear as an independent statement. This phrase allows values to be transmitted from preceding **for each (*class*)**, **for ... of (*set*)**, and **for ... to (*index*)** phrases to the controlled statements, as long as the logical expression is true. Logical expressions in a **while** phrase are tested for each new value of the index variable in **for** phrases. If the logical expression is true, the controlled program segment is executed. Segment execution terminates when the logical expression is false or when the loop is exhausted.

While phrases can appear as independent statements. In this case, variables specified in the logical expressions are set by computations performed within the range of the phrase, and not from **for** phrase iterations. The range of an independent **while** phrase must be delimited by a **do ... loop** construct. The programmer must be careful to provide for the termination of a loop because SIMSCRIPT II.5 cannot automatically terminate an independent phrase. **While** phrases can be modified by **with** and **unless** phrases and may be nested with other independent **while** and **until** phrases and with **for** phrases. If nested `while` phrases end on the same **loop** statement, the keyword **also** can precede **while** to eliminate redundant **loop** statements.

2.116 WITH Phrase

A **with** phrase is used to control iteration in a `for` phrase. If the logical expression is true, the controlled statement is executed. Iteration continues regardless of the value of the logical expression. A **with** phrase may also be used with **while** or **until** phrases.

```
                        ⎡and ⎤
                        ⎨or  ⎬
                        ⎣    ⎦
with logical expression          [,]
```

Keyword	Synonym
with	when

EXAMPLES:

```
for I = 1 to N, with X(I) * Y(I) > Z(I)
```

Controlled statements are executed with values generated by the **for ... to (*index*)** phrase when the logical expression **X(I) * Y(I) > Z(I)** is true. Controlled statements are not executed when the expression **X(I) * Y(I) > Z(I)** is false.

```
for each CITY with POPULATION(CITY) gr 500000 and AREA(CITY) less
than SQ.MILES/2
```

The statement controlled by the **for each (*class*)** phrase will be executed when both logical expressions **POPULATION(CITY) GR 500000** and **AREA(CITY) less than SQ.MILES/2** are true.

```
for every FLIGHT of DEPARTURES when NO.PASSENGERS(FLIGHT) ls
MINIMUM or DESTINATION(FLIGHT) ne "lax" is true
```

The statements controlled by the **for ... of (*set*)** phrase will be executed when either (or both) of the logical expressions **NO.PASSENGERS (FLIGHT) ls MINIMUM** and **DESTINATION(FLIGHT) ne "LAX"** is true.

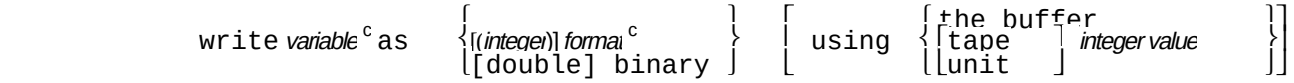
A **with** phrase can be appended to any of the **for** phrases and to **while** and **until** phrases when **while** and **until** are used as independent statements. This phrase selects values to be transmitted from preceding **for each (*class*)**, **for ... of (*set*)**, and **for ... to (*index*)** phrases to the controlled statements when the logical expression is true. Logical expressions in a **with** phrase are tested for each new value of the index variable in **for** phrases. If the logical expression is true, the controlled program segment is executed, but program control in effect transfers around the controlled statements if the logical expression is false.

2.117 WORK Statement

See **wait/work** statement.

2.118 WRITE Statement

The **write** statement writes data to the specified device or the previously established output device according to the specified format.



Note: **Double** optional on implementations where full precision requires more than one computer word.

EXAMPLES:

```
write A, B, and X as 2 i 3 and i 4
```

Beginning with the column that follows the current position of the output pointer, writes the values of **A** and **B** as two integer fields of three characters each, then writes the value of **X** as an integer field of four characters.

```
write INTEGER, DECIMAL, X**2 - Y**2 as i 4, d(8,2), e(9,1)
```

Beginning with the column that follows the current position of the output pointer, writes the value of **INTEGER** as an integer field of four characters, writes the value of **DECIMAL** as a decimal field of eight characters with two fractional digits, and writes the value of $X^2 - Y^2$ as a scientific notation field having a total of nine characters with one fractional digit.

```
write A, B, and X, Y, Z as a 4, s 62, a 2, /,/,/,/, 3 d(10,2)
```

Beginning with the column that follows the current position of the output pointer, writes the values of **A** and **B** as two alphanumeric fields of four characters and two characters, spaced 62 columns apart; then skips three lines and writes the values of **X**, **Y**, and **Z** on the fourth line as three decimal fields of ten characters each with two fractional digits in each field.

```
write CODE as *, /,/,/,/, b 56, "statistical table", a 3
```

Starts a new output page, skips four lines, and, beginning in column 56, writes the character string **statistical table** followed by the value of **CODE** as an alphanumeric field of three characters.

```
for I = 1 to N, write LIST(I) and HEX(I) as (4) I 3 and C 4
```

Beginning with column 1 of a new line, writes four groups of data (four integer elements from array **LIST** and four computer representation elements from array **HEX**) on each line until **N** pairs of values have been printed.

```
for I = 1 to N, for J = 1 to N, write MATRIX(I,J) as binary using
the buffer
```

Writes the two-dimensional array **MATRIX** in binary into the internal buffer.

```
for each CITY, write NAME, AREA, POPULATION as a 3, i 5, i 8
```

For each entity of entity class **CITY**, writes the value of attribute **NAME** as a three-character alphanumeric field, the value of **AREA** as a five-character integer field, and the value of **POPULATION** as an eight-character integer field.

The **write** statement transfers values of expressions to line printers, magnetic tapes, or other output devices according to a format list. In this statement, each expression is evaluated, and the value is printed in the form described by its corresponding format. Formats in a format list must correspond, in order, to the values of expressions to be output. Each format includes a descriptor that is a code defining the type of data (e.g., integer, decimal) to be written on the output device. There are ten descriptors: five are data descriptors that apply to numeric and alphanumeric values, and five are control descriptors used for spacing, skipping columns and records, ejecting pages, and printing character strings. In addition to writing formatted data, this statement can also write binary data. Either formatted or binary data can be written on an output device (perhaps to a unit other than the current output unit) or data can be written to an internal buffer.

During execution of a **write** statement, the format list is scanned from left to right, and individual formats are used to write values. The data descriptors included in the formats apply to integer, decimal, scientific notation, alphanumeric, and computer representation values, while control descriptors designate beginning columns, how many columns and records are to be skipped, where pages are to be ejected, and character strings to be printed exactly as they appear. Descriptors are further defined in table 24. Values being written must agree in mode with their descriptors, except for integer and alphanumeric modes, which can be interchanged. When interchanged, the mode implied by the descriptor governs.

A **write** statement does not necessarily start at the beginning of a new output record as in FORTRAN, because records are changed under programmer control, and not automatically after each statement. This statement processes a continuous string of characters and skips to a new record when directed or when a complete field will not fit on the current record. A buffer, whose length is one record, is provided for each output unit. The current buffer is called **out.f**, and characters in it may be examined or replaced at will. The current output pointer, which is the system variable **wcolumn.v**, points to the column last written in the output buffer. For each new record, **wcolumn.v** starts at zero. (Before an output unit is used for the first time, some implementations set **wcolumn.v** to **-1**.) As output is processed, the pointer moves along the buffer according to the format list. For each value writ-

ten, **wcolumn.v** is positioned to the last column written. The buffer of output data is sent to the output device when a new output record is started. The value of **wcolumn.v** can be advanced by the Beginning Column (b), Skip Column (s), Skip to New Record (/), and Skip to New Page (*) descriptors, and the **start new** statement. The **b, s, /**, and ***** descriptors can be combined with other format descriptors, or they can appear alone in a **write** statement.

Table 24. Descriptors for Write Statement

I: INTEGER

Format	Rules
`n i w` where: `n` is the optional number of consecutive fields `i` is the descriptor `w` is the field width	1. Parameter `n` must be an integer, but `w` can be an expression. 2. At least one blank must appear between n and `i` and between `i` and `w`. 3. On the output, integers are right-adjusted in fields of the specified width. 4. Leading zeros are suppressed, but the rightmost zero in a zero-valued integer is printed. 5. Numbers that exceed the field width are converted to scientific notation. 6. Positive numbers are unsigned; negative numbers are signed.

D: DECIMAL

Format	Rules
`n d(a,b)` where: `n` is the optional number of consecutive fields `d` is the descriptor `a` is the total field width including the sign, integer digits, decimal point, and fractional digits `b` is the number of fractional digits	1. Parameter n must be an integer, but `a` and `b` can be expressions. 2. At least one blank must appear between `n` and `d`. 3. On the output, positive numbers are unsigned, but negative numbers are signed. A minus sign immediately precedes the highest-order digit. 4. Leading zeros are suppressed. Trailing zeros are printed unless the number is exactly zero. 5. Numbers more precise than their allotted output format are rounded. 6. Numbers that exceed the field width are printed in scientific notation.

Table 24. Descriptors for Write Statement (Continued)

E: SCIENTIFIC NOTATION	
Format	Rules
`n e(a,b)` where: `n` is the optional number of consecutive fields `e` is the descriptor `a` is the total field width including the sign, integer digits, decimal points, fractional digits, the letter `e`, sign of the exponent, and the exponent `b` is the number of fractional digits	1. Parameter `n` must be an integer, but `a` and `b` can be expressions. 2. At least one blank must appear between `n` and `e`. 3. On the output, numbers have `b` decimal places, and a scale factor is printed indicating the true value of the number. 4. Positive scale factors are printed without the plus sign. 5. A minus sign is printed after the `e` for negative scale factors. 6. The width of the output, in positions, is equal to `a`; the last four positions contain the scale factor. 7. Table 25 lists the action taken if the field width is insufficient to display the value.

Table 24. Descriptors for Write Statement

A: ALPHANUMERIC	
Format	Rules
`n a w` where: `n` is the optional number of consecutive fields `a` is the descriptor `w` is the field width	1. Parameter `n` must be an integer, but `w` can be an expression. 2. At least one blank must appear between `n` and `a` and between `a` and `w`. 3. The first `w` characters are printed from the leftmost part of the associated expression.

Table 24. Descriptors for Write Statement (Continued)

C: COMPUTER REPRESENTATION

Format	Rules
n c e where: n is the optional number of consecutive fields c is the descriptor e is the number of characters in the internal representation of the computer	1. Parameter n must be an integer, but e can be an expression. 2. At least one blank must appear between n and c and between c and e. 3. The c descriptor is computer-dependent. For example, on the IBM 360, the format C 4 writes four hexadecimal characters; on the Honeywell 600/6000, 2 C 5 writes two fields of five octal characters each.

B: BEGINNING COLUMN

Format	Rules
b n where: b is the descriptor n is the column number	1. Parameter n can be an expression. 2. At least one blank must appear between b and n. 3. Parameter n specifies the position in which the first character of an output value is located, and the system positions the current output pointer to that location. 4. B descriptors need not be in ascending order in a format list.

S: SKIP COLUMN

Format	Rules
s n where: s is the descriptor n is the number of columns to skip	1. Parameter n can be an expression. 2. At least one blank must appear between s and n. 3. Parameter n specifies the number of positions. 4. On the output, skipped positions remain untouched.

Table 24. Descriptors for Write Statement (Continued)

/: SKIP TO NEW RECORD	
Format	Rules
/	Each slash skips to a new record on the current output unit, e.g., the next print line.
*: SKIP TO NEW PAGE	
Format	Rules
*	1. The * descriptor ejects one page on a line printer. 2. This descriptor is disregarded if used in other circumstances.
" ": CHARACTER STRING	
Format	Rules
" "	1. All characters to be printed exactly as they appear are enclosed in quotation marks. 2. The underscore within quotation marks is printed as ". 3. A character string cannot exceed the length of a printed line. 4. A longer character string can be specified as more than one string, each separated by a slash. 5. The spacing of the character string can be indicated by B, S, and / descriptors.

Table 24. Descriptors for Write Statement (Continued)

T: `text`	
Format	Rules
`n t w` where: `n` is the optional number of consecutive fields `t` is the descriptor `w` is the field width	1. Parameter `n` must be an integer, but `w` can be an expression. 2. At least one blank must appear between `n` and `t` and between `t` and `w`. 3. The first `w` characters are printed from the leftmost part of the associated text value. 4. If `w` is greater than the length of the text string, the string is followed by trailing blanks. 3. The length of the string is used as the field width. All the characters of the string are used; when the string is longer than the output record, it is split between successive records.
Format	Rules
`n t *` where: `n` is the optional number of consecutive fields `t` *is the descriptor	1. Parameter `n` must be an integer. 2. At least one blank must appear between `n` and `t` and between `t` and `*`. `*` may optionally be enclosed in parentheses.

Table 25. Order of Character Printing in the "e" Format

Field Width	Characters Printed	Example: -.0234567
1	e	e
2	Sign of number; e	-e
3	Sign of number; e; sign of exponent	-e-
4	Sign of number; e; sign of exponent; d d = digit if 0 ≤ exponent ≤ 9 = * if exponent ≥ 10	-e-2
5	Sign of number; e; exponent	-e-02
6	Sign of number; digit; e; exponent	-2e-02
7	Sign of number; digit; .; e; exponent	-2.e-02
8	Sign of number; digit; .; digit; e; exponent	-2.3 e-02
≥9	Sign of number; digit; . ; additional digits; e; exponent	-2.35e-02 -2.346e-02 -2.3457e-02 -2.34567e-02

2.118.1 AS BINARY Phrase

The **as binary** phrase writes binary information. The binary data can be written on the current output unit, or a **using** phrase can be appended to the **write** statement to designate another output unit. Binary and formatted data cannot be written together on the same unit. Binary information avoids output conversion and hence is a more efficient way to store data temporarily, to be later read in by the same or a different program, as with scratch data.

For integer, real, or alpha values, the **as binary** statement outputs a single computer word of information. For text values, it outputs an integer computer word with the length of the string, followed by successive words of the string until all the characters have been output.

2.118.2 AS DOUBLE BINARY Phrase

On those implementations for which the maximum floating-point precision is more than one computer word, the **write as double binary** statement outputs two computer words for a floating-point (real) number.

2.118.3 USING Phrase

A **using** phrase can locally override the current output unit, which is the printer or the unit declared in the most recently executed **use** statement. This phrase designates a device as the current output unit for the duration of the **write** statement execution. After execution of this statement, SIMSCRIPT II.5 automatically reassigns the previous unit as the current output unit. Data can be written into an internal file by including the keywords **the buffer** in a **using** phrase, or to any output unit by specifying the device number of that unit.

2.118.4 Controlled WRITE Statements

A **write** statement cannot write an entire array by listing only the array name, but a controlled **write** statement that includes a repetition factor can write array elements conveniently (see the fifth example above). A repetition factor, consisting of an expression enclosed in parentheses, must precede a format list that is to be repeated for each record. When a repetition factor is used, the **write** statement must be controlled by a **for** phrase, and the output must start with a new line (record). (A **start new** statement, for example, can position the output pointer to a new line.) With the **write** statement, the system automatically skips to a new line after writing data as prescribed by the format on an individual line. This statement can terminate with the output pointer positioned in the middle of a record.

Index

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z