PART 2 THE SYNTAX AND SEMANTICS OF SIMULA

1 METHOD OF SYNTAX SPECIFICATION

When a SIMULA construction and all its variants can not be economically described in words, its exact range will be shown using a system of notation which is standard throughout Part 2.

The notation is not in itself a part of SIMULA, nor restricted solely to SIMULA. It may be used to describe the syntax (or allowable constructions) of any programming language, and provides a compact, visually clear and precise explanation of the general patterns that the language permits. It is important to realise that it does not describe the meaning of these constructions (their semantics), only their structure. In other words, it indicates:

- the order in which language elements may or must be combined with each other,
- the punctuation that is required, and
- the full range of options.

No such convenient shorthand is yet available for the semantics, so that the interpretation of a legal construction has to be described in words.

We begin by noting that various patterns of basic symbols continually recur in SIMULA. Instead of repeating the listing of the basic symbols each time, they are first grouped together as a named syntactic variable, and from then on we need use only that name. The idea extends itself that further syntactic variables may now be defined in terms of those already defined and possibly basic symbols.

The SIMULA basic symbols are represented by special characters, such as

+ /) ;

by combinations of special characters, such as

:- ** =/=

and by key words in capital letters, such as

PROCEDURE CLASS REAL BEGIN

When we define a syntactic variable, its name will usually be written in lower case letters for distinction:

e.g block statement digit

A certain ambiguity can arise when the name of a syntactic variable consists of two words, such as

prefixed block

To ensure that these are interpreted as one syntactic unit and not for example, an occurrence of a syntactic variable named "prefixed" followed by a syntactic variable named "block", a hyphen is inserted between the words, as:

prefixed-block

In certain cases, when a basic symbol is an integral part of a syntactic variable, it is clearer to use upper-case letters. Again, we avoid possible ambiguities by following the upper-case letters by a hyphen:

e.g. FOR-statement GOTO-statement

We now give two examples to show the uniqueness of interpretations using this technique:

1) IF BOOLEAN-expression THEN
denotes an occurrence of the basic symbol "IF" followed by a "BOOLEAN-expression" followed by an occurrence of the basic symbol "THEN".

2) REF(CLASS-identifier)
denotes an occurrence of the basic symbol "REF" followed by an occurrence of the left parenthesis "(" followed by a "CLASS-identifier" followed by an occurrence of the right parenthesis ")". The separation between "REF" and "(" occurs because if a basic symbol is a key word it may only be composed of capital letters.

Bearing these in mind, the following rules explain the use of the notation as applied to SIMULA.

1. A syntactic variable is a general class of elements in SIMULA. The name of the syntactic variable must consist of:

   a. lower-case letters
   lower-case letters separated by hyphens
   lower-case letters followed by a digit

   e.g. identifier
   compound-statement
   identifier1
   simple-object-expression

   b. a combination of upper-case letters and lower-case letters. There must be one portion of all upper-case letters and at least one portion of all lower-case letters separated, one from another, by hyphens:

   e.g. PROCEDURE-statement

All such units used in this section are defined either formally using this notation or else in words.

2. A basic symbol denotes an occurrence of the characters represented. A basic symbol is either a key word or else one or more special characters

e.g. BEGIN END
+ =/=

N.B. When a basic symbol consists of more than one character no intervening spaces may appear.

Thus, for example,

BEG IN =/ =

are both faulty representations.

3. The term syntactic unit, which is used in subsequent rules, is defined by

   a. a single syntactic variable or basic symbol, or

   b. any collection of syntactic variables, basic symbols, syntax-language symbols (the symbols [,], {, }, |,... whose uses are defined in subsequent rules) surrounded by braces or brackets.

   Examples:

```
digit|letter

⎡digit ⎤
⎣letter⎦

digit

{digit}...
```

4. Braces { } are used as group markers.

Example: the definition of an object-relation is

object-relation

simple-object-expression $\left\{\begin{matrix}\text{IN}\\\text{IS}\end{matrix}\right\}$ CLASS-identifier

The vertical stacking of syntactic units indicates the range of available choices of which exactly one must be taken. The example shows that in an "object-relation", a "simple-object-expression" must be followed by the literal occurrence of either "IN" or "IS" (but not both) and then by a "CLASS-identifier".

5. The vertical stroke | indicates alternatives.

Example: the definition of a digit is

digit

0|1|2|3|4|5|6|7|8|9

This has precisely the same interpretation as

$$\left\{\begin{matrix}0\\1\\2\\3\\4\\5\\6\\7\\8\\9\end{matrix}\right\}$$

but saves considerable space. Both the methods, rule 4 ({ }) and rule 5 (|) are used in this manual to display alternatives. We will usually stick to the use of braces as this notation is clearer, and use | only when the former notation takes up too much space.

6. Square brackets [ ] denote options. Anything enclosed in square brackets may appear once or not at all.

Examples: the definition of a CLASS-declaration is

CLASS-declaration

    [CLASS-identifier] main-part

This denotes a syntactic unit "main-part" optionally preceded by a "CLASS-identifier".

If alternatives are also optional, we use vertical stacking within the square brackets, and omit the braces.

Thus the much simplified version of an activation-statement

```
                ┌                    ┐
    ACTIVATE X  │ BEFORE Y           │
                │ AT time [PRIOR]    │
                └                    ┘
```

would allow the following alternatives:

    ACTIVATE X
    ACTIVATE X BEFORE Y
    ACTIVATE X AT time
    ACTIVATE X AT time PRIOR

7. Three dots ... denote the occurrence of the immediately preceding grouping one or more times in succession.

Examples:

The definition of digits is:

digits

```
     digit ...
```

denoting the occurrence of at least one digit such as

```
     1
     0935
     1970
```

The definition of a compound-statement is:

compound-statement

```
     BEGIN [statement ;]... [statement]  END
```

examples of which are

```
     BEGIN END

     BEGIN  statement  END

     BEGIN  statement;
            statement;
            statement;
     END
```

2 BASIC SYMBOLS AND SYNTACTIC VARIABLES

A program written in SIMULA may contain only

- basic symbols
- identifiers
- constants

Apart from CHARACTER-constants, TEXT-constants and comments where extra latitude is allowed, a program must contain only characters belonging to the language character set. These are either alphanumeric characters, special symbols or key words. Examples of special symbols are

+   -   (   )

which have obvious interpretations. In addition, SIMULA needs many other special symbols and, instead of using peculiar combinations of special characters for their representation, SIMULA uses key words (always written in upper-case letters), such as BEGIN, CLASS. These key words are reserved within the language and may be used only as key words.

2.1 LANGUAGE CHARACTER SET

The SIMULA basic symbol set is built up from a character set of

alphabetic-characters
decimal-digits
special-characters

There are 26 alphabetic-characters - the capital (upper-case) letters A through Z.

There are 10 decimal-digits - the digits 0 through 9.

An alphanumeric-character is either an alphabetic character or a decimal digit.

The 21 special-characters may have an independent meaning within the language (such as + or -) or may be used in combinations (such as := or =/=). Their names and the graphics by which they are represented are:

name	graphic
blank or space	␣
plus	+
minus	-
asterisk or multiply	*
divide	/
equals	=
greater than	>
less than	<
not	¬
comma	,
dot or period	.
exponent	&
colon	:
semicolon	;
dollar	$
left parenthesis	(
right parenthesis	)
character quote	'
text quote	"
hash	#
underscore	

2.2 BASIC SYMBOLS

Any program written in SIMULA may contain only alphanumeric characters and the 19 special characters, except within CHARACTER- or TEXT-constants and comments (see DATA CHARACTER SET section 2.3). Certain combinations of these allowable characters have special significance and are called basic symbols. They fall into two classes:

delimeters
key-words

delimiters

The delimiters used by the language are divided into 6 types:

a) arithmetic-operators
b) logical-operators
c) brackets
d) reference-comparators
e) relational-operators
f) separators

a) arithmetic-operators

The arithmetic-operators are:

+	denoting addition or unary plus
-	denoting subtraction or unary minus
*	denoting multiplication
/	denoting division
**	denoting "raised to the power of"
//	denoting integer division

Note that // may not appear in columns 1-2 (or else that card would be interpreted as a control card).

b) logical-operators

The five logical-operators denoting NOT, OR, AND, EQV and IMP (the last two representing equivalence and implication respectively) are represented by key words. In the case of NOT there is the alternative representation '¬'.

c) brackets

The brackets are:

(
)

which are used in expressions, and for enclosing parameter lists and array bounds

' encloses character constants
" encloses text constants

d) reference-comparators

The reference-comparators are:

== denoting reference equal to
=/= denoting reference not equal to

e) relational-operators

The relational-operators have dual representations as key words and symbol combinations

=	(or EQ)	denoting equal to
¬=	(or NE)	denoting not equal to
>	(or GT)	denoting greater than
>=	(or GE)	denoting greater than or equal to
<	(or LT)	denoting less than
<=	(or LE)	denoting less than or equal to

f) separators

name	graphic	use
comma	,	separating elements in lists
dot	.	denoting decimal point in REAL numbers; remote accessing
colon	;	follows labels; follows VIRTUAL; separates array bounds in array declarations
becomes	:=	in value assignments
denotes	:-	in reference assignments
semicolon	;	separates declarations and statements; separates various parts of procedure and class headings
dollar	$	may be used instead of a semicolon
blank	␣	used as a separator
hash	#	precedes a hexadecimal constant
underscore	_	used in identifiers (e.g. RATE_OF_PAY)

key-words

A key-word is an identifier which is a part of the language and its use is reserved for that purpose. Key-words may be classified as follows:

a) statement-brackets
b) declarators
c) specificators
d) operators
e) key-word-constants

a) statement-brackets

The statement-brackets are:

BEGIN
END

which are used to demark the limits of blocks and compound statements.

b) declarators

The declarators are:

BOOLEAN
CHARACTER
INTEGER
SHORT INTEGER
REAL
LONG REAL
TEXT
REF (CLASS-identifier)
CLASS
PROCEDURE
SWITCH
ARRAY

which are used in declarations and specification lists. The key words SHORT and INTEGER, LONG and REAL must be separated by at least one space, as

SHORT{␣...} INTEGER
LONG {␣...} REAL

c) specificators

The specificators are:

LABEL
NAME
VALUE
VIRTUAL

which are used in specification parts to procedures (LABEL, NAME, VALUE) or to classes (VALUE, VIRTUAL).

d) operators

The operators are divided into 3 classes:

logical-operators
relational-operators
sequential-operators

The logical-operators are:

AND	denoting the logical and
OR	denoting the logical inclusive or
NOT (or ¬)	denoting logical negation
EQV	denoting logical equivalence
IMP	denoting logical implication

The relational-operators are:

EQ (or =)	denoting equal to
NE (or ¬=)	denoting not equal to
GT (or >)	denoting greater than
GE (or >=)	denoting greater than or equal to
LT (or <)	denoting less than
LE (or <=)	denoting less than or equal to

The sequential-operators are:

GOTO
used in GOTO-statements. GOTO may also be written GO{␣...} TO (with any number of blanks between GO and TO, which means that GO and TO are also reserved words),

FOR
STEP
UNTIL
WHILE
DO
used in WHILE- and FOR-statements (DO also appears in connection-statements),

IF
THEN
ELSE
used in conditional-statements and conditional-expressions,

INSPECT
WHEN
DO
OTHERWISE
used in connection-statements. (DO also appears in WHILE- and FOR-statements),

ACTIVATE
REACTIVATE
DELAY
AFTER
BEFORE
AT
PRIOR
used in activation-statements,

INNER
used in CLASS-bodies to alter the order of execution of statements from their textual order,

NEW
is used in generating objects,

THIS
THIS CLASS-identifier
represents a reference to the nearest textually enclosing object of a class equal to or inner to that of the CLASS-identifier,

QUA
defines the scope of a reference expression,

IS
IN
used to test class membership,

COMMENT
used to insert descriptive text among the basic symbols of a program.

The key-word-constants are:

TRUE
FALSE
represent logical values,

NONE
represents the "no object reference",

NOTEXT
represents either the empty text value or no text object.

2.3 DATA CHARACTER SET

Although the language character set is a fixed set defined for the language, the data character set has not been limited. Data may be represented by characters from the language set plus any other EBCDIC characters.

<u>Collating sequence</u>

The 256 members of the data character set have associated with them a unique INTEGER value in the range 0-255. This sequence is known as the collating sequence. It is thus possible to make comparisons of CHARACTERs meaningful by comparing the associated numerical values, such as

```
      'A'   <  'B'
   INCHAR   =  '␣'
```

Parts of the collating sequence are given in Appendix A.

2.4 THE USE OF BLANKS

Identifiers, arithmetic constants, composite operators (e.g. =/=), key words (e.g. BEGIN) may not contain blanks. Blanks are permitted as CHARACTER-constants and in TEXT-constants.

Identifiers, constants and key words may not be immediately adjacent. They must be separated by an arithmetic operator, parenthesis ("(" or ")"), reference comparator, negation ( ), non-key-word relational operator (<,<=,=, =,>,>=,==,=/=), comma, dot, colon, becomes symbol (:=), denotes symbol (:-), semicolon, or blank. Moreover additional intervening blanks are always permitted.

Examples:

```
X + Y          is equivalent to X+Y
A (I )         is equivalent to A(I)
A  :=X :=Y     is equivalent to A:=X:=Y
```

2.5 COMMENT CONVENTIONS

Comments are used for documentation (the insertion of a textual description of part of the program) and do not participate in the execution of a program. The following conventions hold:

Sequence of basic symbols	is equivalent to
delimiter COMMENT {any sequence from the data character set not including a semicolon or dollar} {$ | ;}	delimiter
END {any sequence from the data character set not including END|ELSE|WHEN|OTHERWISE|;|$}	END

Examples:

a)
```
BEGIN COMMENT***THE NEXT BLOCK IS USED FOR PAY-ROLL
                CALCULATIONS***;

     BEGIN .....
           .....
     END OF PAY-ROLL BLOCK;
     ....
END
```

Where the strings "COMMENT .... ;" and "OF PAY-ROLL BLOCK" are treated as comments.

b)
```
IF X > 0 THEN BEGIN .....
                 END OF TRUE PART
            ELSE BEGIN .....
                 END OF ELSE PART;
```

Where the strings "OF TRUE PART" and "OF ELSE PART" are treated as comments.

c)
```
X := X COMMENT**THAT WAS X;**2 COMMENT**SQUARED;;
```
is equivalent, as regards program execution, to
```
X := X**2;
```

2.6 CODING SIMULA PROGRAMS

A SIMULA source program can be written on a standard FORTRAN coding form (IBM Form No. X28-7327). The program may be written in free format from column 1 through 72. Columns 73 through 80 are not significant to the SIMULA compiler and, therefore, may be used for card identification, sequencing or any other purpose. Except with TEXT-constants, column 72 of a card is not considered to immediately precede column 1 of its successor so no basic symbol, identifier nor constant (except a TEXT-constant) may overlap from one card to the next.

Care should be taken not to punch "//" or "/*" in columns 1-2 of a SIMULA source card as these will be taken to be JOB CONTROL cards.

3 IDENTIFIERS

An identifier is a string of alphanumeric or underscore characters, not contained in a comment or constant, preceded and followed by a delimiter - the initial letter must always be alphabetic.

identifier

letter[letter|digit]...[_{letter|digit}...]...

Examples:

valid identifiers
X
SIMULA_67
A15
MORGAN_PLUS_4

invalid identifiers

END	reserved for use as a keyword
SYM␣BOL	contains a blank
3C	does not begin with a letter
APPLE_	underscore cannot appear as the last character

Length of identifiers

Identifiers in a SIMULA program may be composed of any number of alphanumeric or underscore characters, but only the first twelve are significant. Thus if two identifiers contain the same first twelve characters they are considered equivalent.

e.g. BIORTHOGONAL and
BIORTHOGONALISATION

will both be treated as occurrences of the identifier

BIORTHOGONAL

Identifiers and key words

It is not possible to use a key word as an identifier. Every occurrence would be treated as an occurrence of that key word and its use as an identifier would result in errors.

Basic binding rules

Variables, arrays, switches, procedures and classes are said to be quantities. Identifiers serve to identify quantities, or they stand as labels or formal parameters. Identifiers have no inherent meaning and may be chosen freely (except that they may not clash with key words).

Every identifier used in a program must be declared. This is achieved by:

a) a declaration (section 5), if the identifier defines a quantity. It is then said to be a J-variable, J-ARRAY-, PROCEDURE-, J-PROCEDURE-, CLASS-identifier where J stands for the type of the declared quantity.

b) its occurrence as a label (section 5.6) if the identifier stands as a label. It is then said to be a LABEL-identifier.

c) its occurrence in the formal-parameter-list (section 5.4, 5.5) of a PROCEDURE- or CLASS-declaration. It is then said to be a formal-parameter.

The identification of the definition of a given identifier is determined by binding rules. The basic binding rules given below are later extended in the cases of remote accessing (section 6.1), VIRTUAL quantities (section 5.5), and connection (section 7.2).

1. if the identifier is defined within the smallest block (section 7.1) textually enclosing the given occurrence by its occurrence as a quantity or a label, then it denotes that quantity or label.

   The statement following a procedure heading or a class heading is always considered to be a block, which makes the binding to formal parameters a special case.

2. Otherwise if the block is a procedure or a class body and the given identifier is identical with a formal parameter in the associated procedure or class heading, then it stands for that formal parameter.

Otherwise, these rules are applied by considering the smallest block textually enclosing the block which has been previously considered.

If these steps lead to more than one definition or to no definition, then the identification is illegal.

Example:

```
line 1      BEGIN PROCEDURE A;
     2            X := X + 1;
     3            REAL X;
     4            BEGIN REAL X;
     5                  X := 2;
     6            LAB:  A;
     7            END;
     8      END
```

The block spanning lines 4-7 is textually enclosed in the block spanning lines 1-8. The procedure declaration of lines 1-2 is treated as though it were

```
PROCEDURE A;
BEGIN  <dummy-declaration>;
       X := X + 1;
END;
```

Thus the occurrence of X at line 5 is bound to the declarations of line 4, whereas in the invocation of the procedure at line 6 the binding rule for the occurrence of X in the procedure body is to the variable declared at line 3.

The <u>scope</u> of a quantity, label or formal parameter is the set of statements in which occurrences of an identifier may refer to its definition by the above rules.

4 TYPES AND CONSTANTS

Constants and variables possess values and types. Both the value and type of a constant are determined by the way it is written. The value of a variable is the one most recently assigned to it, or its initial value if no assignment has yet been made, and its type is determined by its declaration.

4.1 TYPES

Type is subdivided into two classes by:

<u>type</u>

$$\left\{\begin{array}{l}\text{value-type}\\ \text{reference-type}\end{array}\right\}$$

where value-type and reference-type are defined by:

<u>value-type</u>

$$\left\{\begin{array}{l}\text{[SHORT]INTEGER}\\ \text{[LONG]REAL}\\ \text{BOOLEAN}\\ \text{CHARACTER}\end{array}\right\}$$

<u>reference-type</u>

$$\left\{\begin{array}{l}\text{REF(CLASS-identifier)}\\ \text{TEXT}\end{array}\right\}$$

4.2 CONSTANTS

A constant is a fixed, unvarying quantity that denotes itself, i.e. it can not alter during the course of a program. Each constant has a uniquely defined type. The discussion of constants follows the order:

arithmetic-constants
BOOLEAN-constants
CHARACTER-constants
object-reference-constant
TEXT-constant

<u>arithmetic-constants</u>

arithmetic-constants may be written as decimal-constants (base 10) or hexadecimal-constants (base 16). Note that

a) the use of arithmetic-constants is optimised by the system

b) any '+' or '-' sign preceding an arithmetic-constant is treated separately.

decimal-constants

decimal-constants are interpreted according to conventional notation with '&' representing the exponent sign. If a decimal constant contains either a decimal point, or an exponent sign, or both, it is interpreted as a (LONG) REAL number, if it contains neither a decimal point nor an exponent sign, it is taken to represent a (SHORT) INTEGER number.

decimal-digit

{0|1|2|3|4|5|6|7|8|9}

decimal-digits

{decimal-digit}...

representing a run of at least one decimal digit.

Examples:

```
   000
     1
315730
```

(SHORT)INTEGER-constant

decimal-digits

The range of values is the set of whole numbers from 0 through $2^{31}-1$ (= 2147483467). If the magnitude lies in the range 0 through $2^{15}-1$ (= 32767), the constant is treated as a SHORT INTEGER constant, if the magnitude lies in the range 2^{15} through $2^{31}-1$, it is treated as an INTEGER constant. If the magnitude is equal to or exceeds 2^{32}, the number is interpreted as a REAL constant.

Examples:

0	SHORT INTEGER
91	SHORT INTEGER
814728	INTEGER

(LONG)REAL-constants

$$\left\{\begin{array}{l}\left\{\begin{array}{l}\cdot\text{decimal-digits}\\ \text{decimal-digits}\cdot\text{decimal-digits}\\ \text{decimal-digits}\end{array}\right\}\ \text{exponent}\\ \\ \cdot\text{decimal-digits}\\ \text{decimal-digits}\cdot\text{decimal-digits}\\ \text{exponent}\end{array}\right\}$$

where exponent takes the form

exponent

$$\& \begin{bmatrix} + \\ - \end{bmatrix} \text{ decimal-digits}$$

i.e. the symbol '&', optionally followed by a '+' or '-' sign, followed by a SHORT INTEGER-constant. The exponent represents a scale factor expressed as an integral power of 10.

The range of values of (LONG)REAL-constants is 0 through 10^{75} (approximately). Any such constant has an equivalent representation of the form

A& ±B

where 0.1 <= A < 1, and B is adjusted accordingly. If in this form, A requires from 1 through 7 decimal-digits, then the constant is a REAL-constant. If it requires 8 or more decimal-digits, then the constant is a LONG REAL-constant and has a maximum precision of 16 decimal-digits (any remaining digits are discarded).

Examples

valid	0.0	REAL
	999.999	REAL
	57.6&+21	REAL
	.3&1	REAL
	3&1	REAL
	&-1	REAL
	314.1592653	LONG REAL
	21.2274568&+03	LONG REAL
invalid	3.	no digit after the decimal-point
	3,149.2	embedded comma
	33.4&	no scale factor
	23.4&87	out of range

hexadecimal-constant

#{decimal-digit|A|B|C|D|E|F}...[R]

The hexadecimal digits A through F represent the numbers 10 through 15 respectively. A hexadecimal-constant terminated by the letter 'R' is interpreted as a REAL number if there are 8 or less preceding hexadecimal digits.

e.g. #56789R
#BBFFFFFFR

If there are 9-16 hexadecimal digits, it is treated as a LONG REAL constant

e.g. #00000000FFFFFFFFR

A hexadecimal-constant not containing the letter 'R' is treated as a SHORT INTEGER constant if it contains 4 or less hexadecimal digits

e.g. #0
#FFFF

or as an INTEGER constant if it contains 5 through 8 hexadecimal digits,

e.g. #00FFFFFF

Hexadecimal-constants are taken to be right justified and define a bit pattern.

BOOLEAN-constants

These are the key words FALSE and TRUE whose interpretation is obvious.

CHARACTER-constants are represented by

'{any one member of the data character set}'

Examples:

valid CHARACTER-constants:

```
'X'
'&'
'␣'
'''      the character quote itself
```

invalid CHARACTER-constants:

```
':-'     two data character set members
'A␣'     blanks are significant in character
              constants
X        no embedding character quotes
'4       no terminating character quote.
```

object-reference-constant

There is only one object-reference-constant, namely

NONE

Any object reference variable may take the value NONE.

TEXT-constants have the form

"{any sequence of members of the data character set excluding a text quote (")}"

The length of a TEXT-constant is the number of members of the data character set it contains. The length may be a whole number in the range 0 through $2^{15}-20$. Blanks are significant in TEXT-constants.

Examples:

valid TEXT-constants:

```
"THISTEXTCONSTANTHASNOBLANKS"
"THIS␣ONE␣HAS"
""
```

invalid TEXT-constants:

"ONE␣"␣TOO␣MANY"	contains a text quote
"NEVER␣ENDING	contains no terminating text quote

Text quotes may be introduced into text objects by:

1) inputting a text value containing a text quote.

2) using the procedure "putchar". The following code inserts a text quote into the 14th position of a text object referenced by the TEXT variable T:

```
T.setpos(14);
T.putchar('"');
```