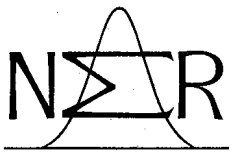
NORSK REGNESENTRAL
NORWEGIAN COMPUTING CENTER



SIMULA®

USERS GUIDE
IBM System/360

REVISED EDITION
1975

® SIMULA is registered trademark of the Norwegian Computing Center

© Norsk Regnesentral 1971, 1975.

Publication no. S-24-1
Norwegian Computing Center

Printed in Norway by
Alstad Offset-trykk

PREFACE

SIMULA is a general purpose programming language. Its defining documents are "The Revised Report on ALGOL 60" [1] and "SIMULA 67 Common Base Language" [2].

This manual, and the companion volume "Programmer's Guide" [3], serve as guides to the implementation on the IBM 360 series, with the "Programmer's Guide" taking care of the implementation dependent parts.

This manual, then, is a guide to the language itself. It has been written in three parts, the first consisting of fully described examples, the second giving the syntax and semantics of SIMULA, and the last consisting of sections (with further examples) on the standard system classes and text handling.

References

[1] "The Revised Report on the Algorithmic Language Algol 60", CACM, Vol. 6., No. 1, 1963.

[2] "SIMULA Common Base Language", NCC Publ. S.22, 1971

[3] "Programmer's Guide - SIMULA for 360", NCC Publ. S.23, 1971.

CONTENTS

PREFACE

PART 1 BASIC SIMULA

COMPILE TIME AND RUN TIME

1 THE HIGHEST MARK

2 NUMBERING A CROSSWORD PUZZLE

3 GEOMETRIC APPLICATIONS

4 SIMULATION OF A QUEUEING SITUATION

PART 2 THE SYNTAX AND SEMANTICS OF SIMULA

1 METHOD OF SYNTAX SPECIFICATION

2 BASIC SYMBOLS AND SYNTACTIC VARIABLES

2.1 LANGUAGE CHARACTER SET
2.2 BASIC SYMBOLS
2.3 DATA CHARACTER SET
2.4 THE USE OF BLANKS
2.5 COMMENT CONVENTIONS
2.6 CODING SIMULA PROGRAMS

3 IDENTIFIERS

4 TYPES AND CONSTANTS

4.1 TYPES
4.2 CONSTANTS

5 DECLARATIONS

5.1 TYPE DECLARATIONS
5.2 ARRAY DECLARATIONS
5.3 SWITCH DECLARATIONS
5.4 PROCEDURE DECLARATIONS
5.5 CLASS DECLARATIONS

6 EXPRESSIONS

6.1 VARIABLES
6.2 FUNCTION DESIGNATORS
6.3 ARITHMETIC EXPRESSIONS
6.4 BOOLEAN EXPRESSIONS
6.5 CHARACTER EXPRESSIONS
6.6 DESIGNATIONAL EXPRESSIONS
6.7 OBJECT EXPRESSIONS
6.8 TEXT EXPRESSIONS AND TEXT VALUES

7 BLOCKS AND STATEMENTS

7.1 BLOCKS AND COMPOUND STATEMENTS
7.2 STATEMENTS

PART 3 SYSTEM CLASSES AND TEXT HANDLING

1 PROGRAM STRUCTURE

2 CLASS SIMSET

3 CLASS SIMULATION

4 TEXT

5 CLASS BASICIO

APPENDIX A : HARDWARE REPRESENTATION

APPENDIX B : SYSTEM PROCEDURES

APPENDIX C : SKELETON OF THE SYSTEM CLASSES AND SYSTEM PROCEDURES

INDEX

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

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

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.

Collating sequence

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 scope 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:

type

```
{ value-type     }
{ reference-type }
```

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

value-type

```
{ [SHORT]INTEGER }
{ [LONG]REAL     }
{ BOOLEAN        }
{ CHARACTER      }
```

reference-type

```
{ REF(CLASS-identifier) }
{ TEXT                  }
```

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

arithmetic-constants

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

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

where exponent takes the form

exponent

$$\&\left[\begin{array}{l}+\\ -\end{array}\right]\ \text{decimal-digits}$$

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	
	57.6&+21	
	.3&1	
	3&1	
	&-1	
	314.1592653	LONG REAL
	21.2274568&+03	
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('"');
```

5. DECLARATIONS

Declarations introduce a measure of security into the language by associating identifiers with variables, arrays, switches, procedures and classes used in the programs.

A declaration determines the type and structure of a quantity.

Declarations must appear in a block head, and upon exit from that block (unless it is the outermost block of a class body), they lose their significance.

declaration

{ type-declaration
| ARRAY-declaration
| SWITCH-declaration
| PROCEDURE-declaration
| CLASS-declaration
| EXTERNAL-declaration }

Common to each of the succeeding sections, which discuss these declarations, is the definition of an identifier-list.

identifier-list

identifier [, identifier]...

in which the identifiers must all be different.

Examples:

valid	AXOLOTL	
	A, B, MARY, B_14	
invalid	A B	no separating comma
	A, END, Q	END is a key word and is not allowed as an identifier.
	B1,B2, B1	the same identifier appears twice
	BIORTHOGONAL, BIORTHOGONALISATION	the identifiers are not distinct (only the first twelve characters are significant).

5.1 TYPE DECLARATIONS

Type declarations associate each identifier in the associated identifier list with a variable of a given type. This restricts the range of values that may be assigned to that variable.

type-declaration

```
{ [LONG]REAL                  }
{ [SHORT]INTEGER              }
{ CHARACTER                   }  identifier-list
{ BOOLEAN                     }
{ REF(CLASS-identifier)       }
{ TEXT                        }
```

Examples:

valid: value-type

```
REAL R1, R2
LONG REAL P1
INTEGER I, J, K
SHORT INTEGER SH1
CHARACTER A, B, C, D
BOOLEAN B1, B2, B3
```

reference-type

```
REF(POINT) P, Q
TEXT MY_STRING, YOUR_STRING
```

invalid:

```
SHORT REAL ONE          no such type
REF LINE_1, LINE_2      no such type - the qualification
                        is omitted.
```

INTEGER and SHORT INTEGER variables may only assume whole numbers.

REAL and LONG REAL variables may only assume numbers.

N.B. An arithmetic expression of any arithmetic type (SHORT INTEGER, INTEGER, REAL, LONG REAL) may be assigned to an arithmetic variable. If the types do not correspond, the expression is first converted to the type of the variable and then transferred, subject to its being in range. If it is not in range, then a run time error will result.

BOOLEAN variables may only assume the values TRUE and FALSE.

CHARACTER variables may only assume values from the data character set.

TEXT variables may only assume strings of characters from the data character set or NOTEXT.

REF variables may only assume the value NONE or references to objects belonging to their qualifying class or its subclasses.

Each variable declared in a type declaration has an initial value (given in the table below). Thereafter the value of a variable is the one last assigned to it, or if no assignment has yet been made, the initial value.

Type	Initial value	Assignable range
INTEGER	0	Whole number in the range -2^{31} through $2^{31}-1$.
SHORT INTEGER	0	Whole number in the range -2^{15} through $2^{15}-1$.
REAL	0.0	Number in the range $\pm(0 \rightarrow 10^{75}$ approx.) to 7 decimal places.
LONG REAL	0.0	Number in the range $\pm(0 \rightarrow 10^{75}$ approx.) to 16 decimal places.
BOOLEAN	FALSE	TRUE, FALSE.
CHARACTER	CHAR(0)	CHAR(0), CHAR(1), ... CHAR(255).
REF(CLASS-identifier)	NONE	NONE or any object of the qualifying class or included in the qualifying class.
TEXT	NOTEXT	NOTEXT or any string of characters from the data character set of length 0 through $(2^{15}-20)$ characters.

5.2 ARRAY DECLARATIONS

An array is a structure of many components (subscripted variables) all of the same type. Each component has the same identifier (the array identifier), and they are distinguished one from another by subscripts. Arrays invite the user to group like data under one identifier.

Arrays are declared with a certain shape. They can have 1 through 8 dimensions (which is the number of subscripts necessary to specify a certain component), and each dimension has a fixed range specified by giving an upper and lower bound.

Pictorial representations of one, two and three dimensional arrays are now given.

One dimensional array

```
INTEGER ARRAY NUMBER (4:9);
```

NUMBER(4)
NUMBER(5)
NUMBER(6)
NUMBER(7)
NUMBER(8)
NUMBER(9)

Dimensions 1:
Lower subscript bound = 4
Upper subscript bound = 9
Declares 6 subscripted variables each of type INTEGER and initialised to 0.

Example:

Find the sum of the components of the array NUMBER.

```
SUM := 0;
FOR I := 4 STEP 1 UNTIL 9 DO
    SUM := SUM + NUMBER(I);
```

Two dimensional array

```
REF(POINT) ARRAY A(0:3, 2:4);
```

A(0, 2)	A(0, 3)	A(0, 4)
A(1, 2)	A(1, 3)	A(1, 4)
A(2, 2)	A(2, 3)	A(2, 4)
A(3, 2)	A(3, 3)	A(3, 4)

Dimensions 2:
Lower subscript bound 1 = 0
Upper subscript bound 1 = 3
Lower subscript bound 2 = 2
Upper subscript bound 2 = 4

Declares 12 subscripted variables each of type REF(POINT) and each initialised to NONE.

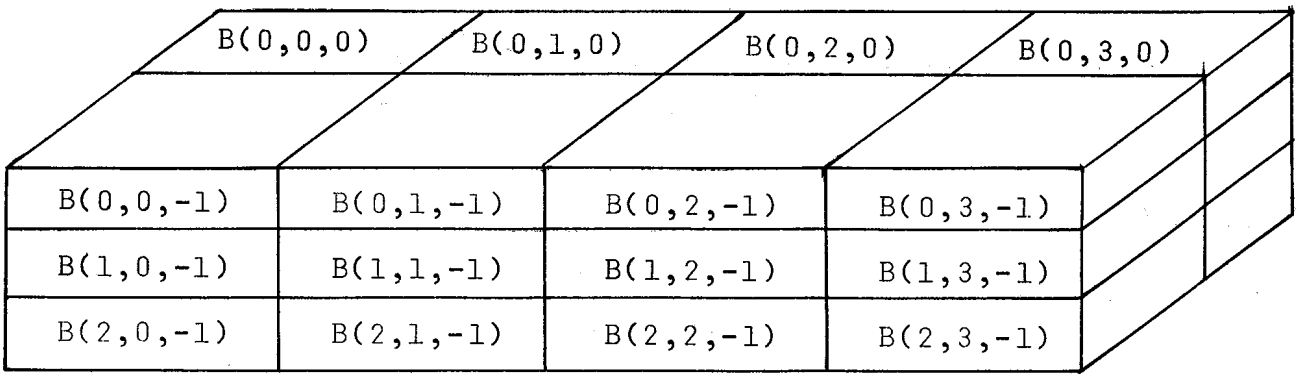
Example:

Scan the array and count how many subscripted variables are currently referencing NONE.

```
COUNT := 0;
FOR I := 0 STEP 1 UNTIL 3 DO
    FOR J := 2 STEP 1 UNTIL 4 DO
        IF A(I, J) == NONE
          THEN COUNT := COUNT + 1;
```

Three dimensional array

```
BOOLEAN ARRAY B(0:2, 0:3, -1:0);
```



(Each component is represented by a cube).

Dimensions 3:
Lower subscript bound 1 = 0
Upper subscript bound 1 = 2
Lower subscript bound 2 = 0
Upper subscript bound 2 = 3
Lower subscript bound 3 = -1
Upper subscript bound 3 = 0

Scan through the array and set the values of each subscripted variable to TRUE.

```
FOR I := 0 STEP 1 UNTIL 2 DO
    FOR J := 0 STEP 1 UNTIL 3 DO
        FOR K := -1 STEP 1 UNTIL 0 DO
            B(I,J,K) := TRUE;
```

ARRAY-declaration

[type] ARRAY ARRAY-segment [,ARRAY-segment]...

ARRAY-segment

identifier-list (lower-bound:upper-bound
[,lower-bound:upper-bound]...)

Examples:

ARRAY-segment
A(1:10)
B,C(1:15, 0:N**2)

ARRAY-declaration
ARRAY ALFA (0:N)
REF(LINE) ARRAY L,M(0:P.X)
TEXT ARRAY R,S(1:5), T(1:4,ININT:IF X=0
THEN 10 ELSE 100)

Each identifier in an array declaration is associated with an array of a given type (if no type is given, the type is taken to be REAL). To enable arrays of the same type, but with different shapes (number of dimensions and subscript ranges) to be declared in the same place, an array declaration contains one or more ARRAY-segments which are identifier-lists followed by their common shape. The number of dimensions is the number of upper-bound lower-bound pairs and the range of each subscript is specified directly by the upper-bound/lower-bound pairs taken in order.

Note that although the lower- and upper-bounds may be arithmetic expressions their value is the rounded integer as evaluated on entry to the block. To be valid, the value of each lower bound must be less than or equal to the value of its associated upper bound. Neither bound may refer to a quantity declared in the same block head. The value of each subscripted variable is initialised according to the type of the array.

5.3 SWITCH DECLARATIONS

A switch is declared with a list of designational-expressions which are accessed by an index. The length of the list, N, gives the number of switch elements. The value of each element is the current value of the designational-expression (a label to a statement, or another switch value).

SWITCH-declaration

```
SWITCH SWITCH-identifier := designational-expression
                                [,designational-expression]...
```

Examples:

```
SWITCH SW := L1, L2, T(N), M1
SWITCH T := M1, IF N<0 THEN L1 ELSE L3, M1
```

A SWITCH declaration contains a list of designational expressions each of which is given a positive index (starting from 1) by counting the items in the list from left to right. The value of the switch-designator (see section 6.6) corresponding to a given index is the current value of the designational expression having the index in the corresponding list. (An expression in a switch list is re-evaluated every time it is referred to).

5.4 PROCEDURE DECLARATION

A procedure declaration defines an action cluster and gives it a formal name. The action cluster is the body of the PROCEDURE. To increase the generality of the concept, there are facilities for transmitting parameters and returning a function value at run time. Thus a procedure is able to interact with its calling environment by bindings established by the calling mechanism.

By means of the procedure concept, special instances of declared action clusters become meaningful units within the SIMULA program.

PROCEDURE-declaration

{proper-procedure-declaration
function-declaration}

proper-procedure-declaration

PROCEDURE PROCEDURE-heading PROCEDURE-body

function-declaration

type PROCEDURE PROCEDURE-heading PROCEDURE-body

PROCEDURE-heading

PROCEDURE-identifier{; formal-parameter-part mode-part spec-part}

formal-parameter-part

(identifier-list);

mode-part

{VALUE identifier-list;
 NAME identifier-list;}...

spec-part

{{type
 [type] ARRAY
 PROCEDURE
 type PROCEDURE
 LABEL
 SWITCH} identifier-list;} ...

PROCEDURE-body

statement

Examples:

proper-procedure-declaration

```
PROCEDURE SWAP(A,B); NAME A, B;
                     REAL A, B;
BEGIN  REAL X;
       X := A;
       A := B; B := X;
END ***SWAP***

PROCEDURE OUTCOLUMN(A,N); REAL ARRAY A; INTEGER N;
BEGIN  INTEGER J;
       FOR J := 1 STEP 1 UNTIL N DO
           BEGIN  OUTFIX (A(J), 5, 12);
                  OUTIMAGE;
           END;
END ***OUTCOLUMN***

PROCEDURE TREETRAVERSE(N); REF(NODE)N;
INSPECT N DO
BEGIN  TREETRAVERSE(LEFTBRANCH);
       DUMP;
       TREETRAVERSE(RIGHTBRANCH);
END ***RECURSIVE TRAVERSE***
```

A proper-procedure is used as a statement in its own right.

```
INIMAGE

OUTTEXT("PROPER-PROCEDURE")

IF B THEN SWAP(TEMP1,TEMP2)
```

function-declaration

```
REF(POINT) PROCEDURE ADD(Q); REF(POINT)Q;
   IF Q =/= NONE THEN ADD := NEW POINT(X+Q.X,Y+Q.Y)

REAL PROCEDURE NORM(A,N); REAL ARRAY A; INTEGER N;
BEGIN  REAL T; INTEGER I;
       FOR I := 1 STEP 1 UNTIL N DO
           T := T + A(I)**2;
       NORM := SQRT(T)
END ***NORM***

INTEGER PROCEDURE FACTORIAL(N); INTEGER N;
IF N < 0 THEN ERROR ELSE
   IF N < 2 THEN FACTORIAL := 1
            ELSE FACTORIAL := N*FACTORIAL(N-1)
```

A function returns a value of the type indicated in its declaration, and may be used wherever a value of that type is legal. (It may also be used as a statement in which case the function value is ignored);

```
P :- R.ADD(S)

X := NORM(MATRIX, 10)

IF NORM (MATRIX, 10) < &-6 THEN
   OUTTEXT ("ELEMENTS␣ALL␣ZERO")
```

A PROCEDURE-declaration defines a procedure associated with a PROCEDURE-identifier. The chief component of a PROCEDURE declaration is the PROCEDURE-body which may be activated through a PROCEDURE-statement or a function-designator.

If the procedure has formal-parameters, then their identifiers are listed in the formal-parameter part. No formal-parameter identifier may appear more than once, and specifications of all formal-parameters must be supplied. Whenever the procedure is activated by a PROCEDURE-statement or function-designator, the formal-parameters are either assigned the values of the actual-parameters (call by value, call by reference) or else replaced by the actual-parameters (call by name).

Identifiers in the PROCEDURE-body which are not formal-parameters are local if declared within that body, otherwise they are non-local. A PROCEDURE-body acts like a block regardless of its format. Consequently, the scope of any label to a statement within the body or to the body itself can never be extended beyond the PROCEDURE-body. In addition, if the identifier of a formal-parameter is redeclared within the PROCEDURE-body (or used as a label), it is given local significance and the corresponding actual-parameter becomes inaccessible.

When a PROCEDURE-declaration is given a type, it designates a function. The type of this function is the type of the PROCEDURE and for any activation its value is the last value assigned to an occurrence of the PROCEDURE-identifier within the PROCEDURE-body. If no such assignment is made, then the result of the function-designator takes a default value which is identical to the initial value of a declared variable of that type.

Within the PROCEDURE-body of a function-procedure, an assignment may be made to an occurrence of the PROCEDURE-identifier. Every other occurrence of a PROCEDURE-identifier denotes another activation of the procedure (recursion).

PARAMETER TRANSMISSION

N.B. There are three possible modes for parameter transmission

call by value
call by reference
call by name

All three are allowable for parameters to procedures, but only call by value and call by reference are valid for parameters to classes. Since call by value and call by reference are common to procedure and class declarations, these sections apply to class declarations as well.

If no mode is specified in the mode part, then the parameter is transmitted by the appropriate default mode which is call by value for value type parameters, and call by reference for other kinds of parameters.

The available transmission modes for legal parameters to procedures are shown in the following table:

PROCEDURE PARAMETERS

Parameter	Transmission		
	by value	by reference	by name
value-type	D	X	0
object-reference	X	D	0
TEXT	0	D	0
value-type ARRAY	0	D	0
reference-type ARRAY	X	D	0
PROCEDURE	X	D	0
type PROCEDURE	X	D	0
LABEL	X	D	0
SWITCH	X	D	0

D : default mode
0 : optional
X : illegal

Call by value

A formal-parameter called by value designates a local copy of the value obtained by evaluating the actual parameter. The evaluation takes place at the time of procedure entry. Throughout the lifespan of the procedure or class body, the formal-parameter acts like a local variable (or value type array) and its contents may be accessed and changed by local assignments. In the case of an object, the contents may be accessed and changed from without by remote accessing.

Value specification is redundant for a parameter of value type. There is no call by value option for object references, reference-type ARRAYS, PROCEDURES, type-PROCEDURES, LABELS and SWITCHES.

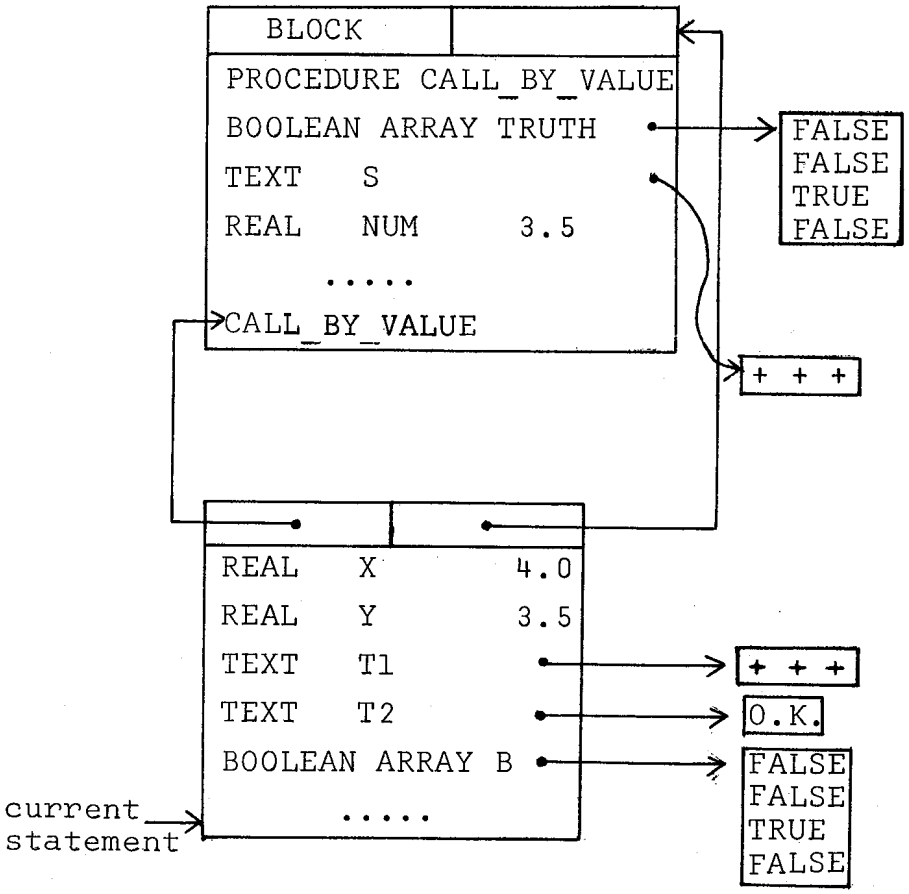
Example:

Given the program:

```
BEGIN  PROCEDURE CALL_BY_VALUE(X,Y,T1,T2,B);
       VALUE T1, T2, B;
       REAL X, Y; TEXT T1, T2; BOOLEAN ARRAY B;
       BEGIN ..................... END;

       BOOLEAN ARRAY TRUTH(0:3);
       TEXT S;
       REAL NUM;
       NUM := 3.5; S :- COPY("+++");
       TRUTH(2) := TRUE;
       CALL_BY_VALUE(4,NUM,S,"O.K",TRUTH);
       ...
END
```

then a snapshot on procedure entry shows the initialised values for the formal-parameters.


BLOCK
PROCEDURE CALL_BY_VALUE
BOOLEAN ARRAY TRUTH
TEXT S
REAL NUM 3.5
.....
CALL_BY_VALUE
FALSE
FALSE
TRUE
FALSE
+ + +
REAL X 4.0
REAL Y 3.5
TEXT T1
TEXT T2
BOOLEAN ARRAY B
.....
+ + +
O.K.
FALSE
FALSE
TRUE
FALSE
current
statement

Call by reference

A formal-parameter called by reference designates a local copy of the reference obtained by evaluating the corresponding actual-parameter. The evaluation takes place at the time of the procedure entry or object generation. Throughout the lifespan of the PROCEDURE- or CLASS-body, the formal-parameter acts like a local quantity.

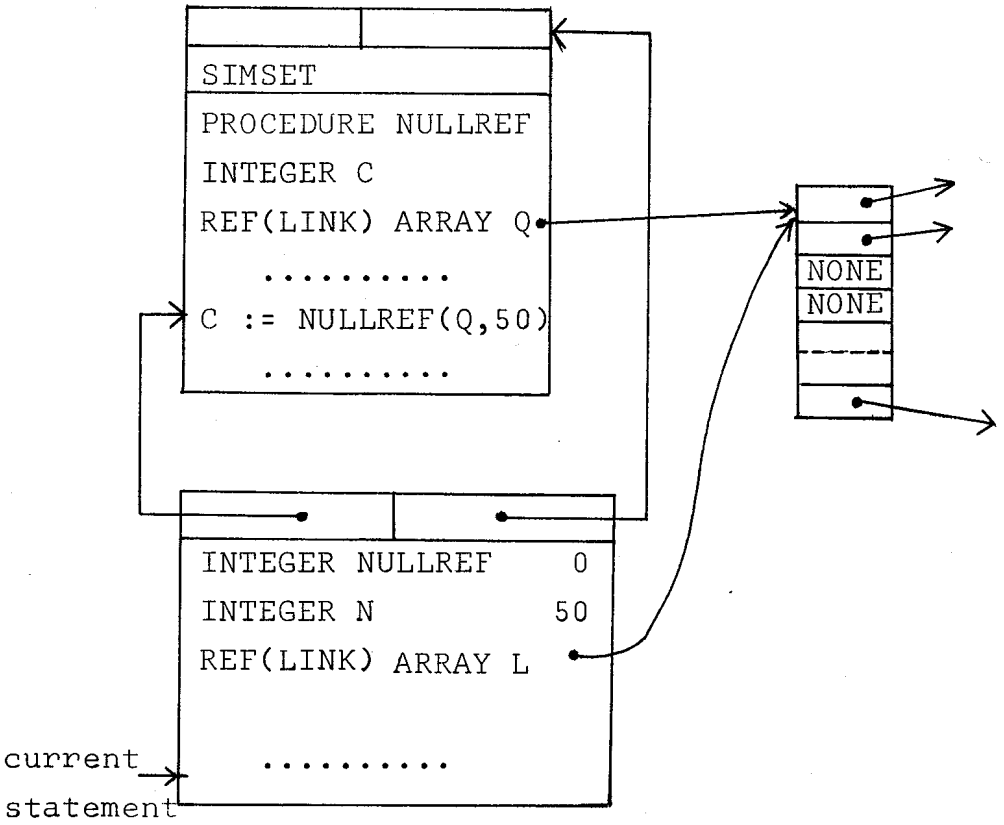
If the formal-parameter is a reference-type variable, its contents may be changed by assignments within the body (or externally by remote accessing in the case of objects).

An ARRAY, PROCEDURE, LABEL or SWITCH parameter called by reference remains fixed and references the same entity throughout its scope. Of course, the contents of an ARRAY called by reference may well be changed through assignments to its components.

For an object-reference type parameter (which may be a variable, PROCEDURE or an ARRAY), the qualification of the matching actual-parameter may coincide with, or be inner to, the qualification of the formal-parameter.

```
SIMSET BEGIN  INTEGER PROCEDURE NULLREF(L,N); REF(LINK) ARRAY L;
                                              INTEGER N;
              BEGIN  INTEGER J, COUNT;
                     FOR J := 1 STEP 1 UNTIL N DO
                         IF L(J) == NONE THEN COUNT := COUNT + 1;
                     NULLREF := COUNT;
              END;
              INTEGER C;
              REF(LINK) ARRAY Q(1:50);
              ......
              C := NULLREF(Q,50);
              ......
```

On the procedure call, a snapshot is



Call by name

Call by name is an optional transmission mode available for parameters to procedures, but not to classes. It represents a textual replacement in that the formal-parameter may be considered replaced throughout the PROCEDURE-body by the corresponding actual-parameter.

Whereas call by value and call by reference operate on variables local to the PROCEDURE-body itself, call by name operates on non-local quantities and can alter global quantities. It is therefore especially useful for the controlled alteration of several variables external to the PROCEDURE-body.

The following rules apply:

1) the type of a name parameter is that prescribed by the corresponding formal specification.

2) if the type of the actual-parameter does not coincide with that of the formal specification, then an evaluation of the expression is followed by an assignment of the value or reference obtained to a fictitious variable of the latter type. This assignment is subject to the rules of section 7.2. The value or reference obtained by the evaluation is the contents of the fictitious variable.

Section 7.2 defines the meaning of an assignment to a variable which is a formal-parameter called by name, or is a subscripted variable whose array identifier is a formal-parameter called by name, if the type of the actual parameter does not coincide with that of the formal specification.

Assignment to a PROCEDURE-identifier which is a formal parameter is illegal, regardless of its transmission mode.

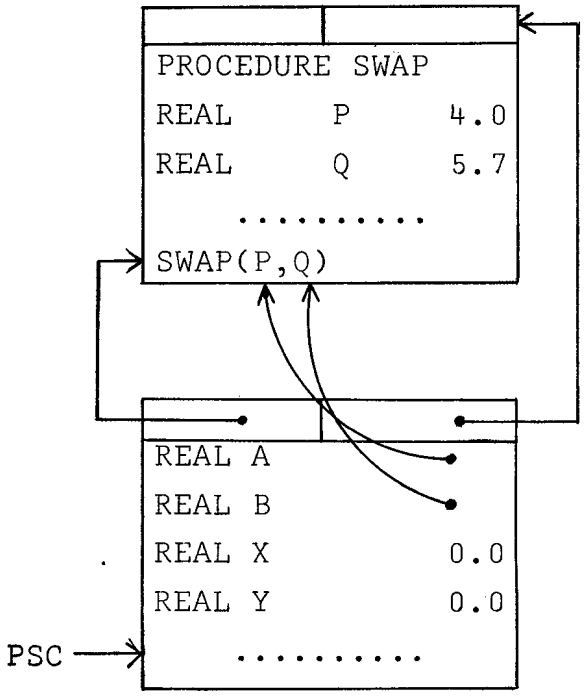
Notice that each dynamic occurrence of a formal-parameter called by name, regardless of its kind, may invoke the execution of a non-trivial expression, e.g. if its actual-parameter is a remote-identifier.

Example:

```
BEGIN  PROCEDURE SWAP(A,B); NAME A, B; REAL A, B;

       BEGIN  REAL X;
              X := A;
              A := B; B := X;
       END ***SWAP***;
       REAL P, Q;
       P := 4.0; Q := 5.7;
       SWAP(P,Q);
       ..........
END
```

A snapshot at the procedure call is:



Program Sequence Control (PSC) references the current statement

No local copies are made. Every occurrence of A or B in the PROCEDURE-body means a re-evaluation of the actual-parameter. Notice that the actual-parameters are evaluated in the context of the procedure call.

5.5 CLASS DECLARATIONS

CLASS declarations define concepts. CLASS instances are called objects and many objects of a given class may exist simultaneously to be examined and manipulated by the program. Objects are generalisations of blocks and procedures, as the environment of either a block instance or a procedure instance can only observe the result of its actions.

CLASS-declaration

[CLASS-identifier]main-part

main-part

CLASS CLASS-identifier { ; / parameter-part } [virtual-part]CLASS-body

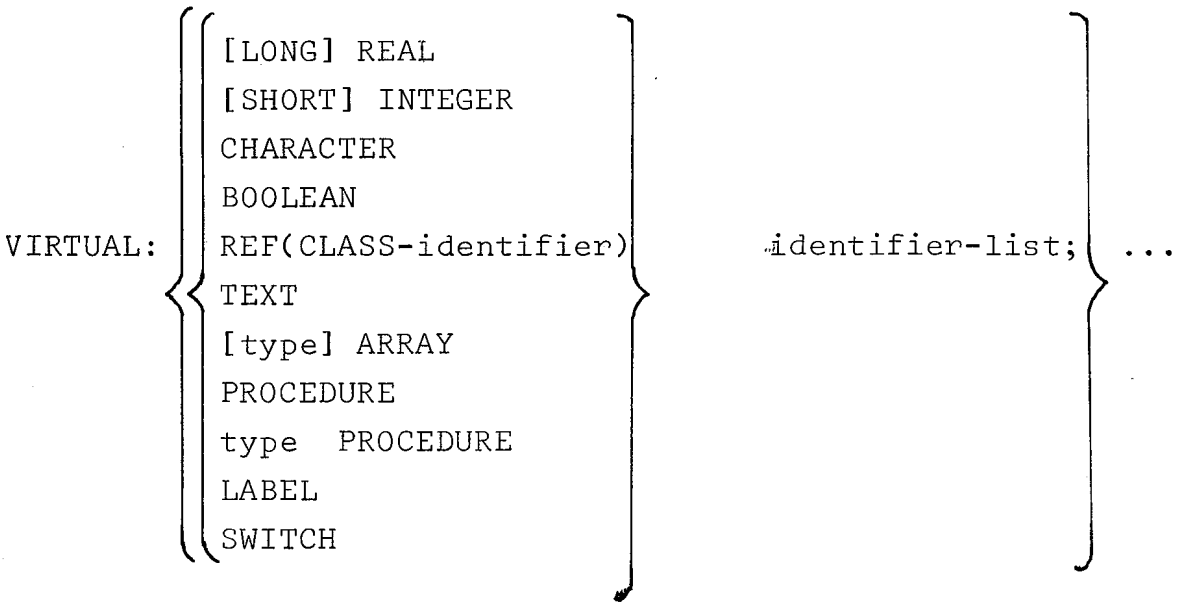
parameter-part

(identifier-list); [VALUE identifier-list;]... CLASS-spec-part

CLASS-spec-part

{ { [LONG]REAL / [SHORT]INTEGER / CHARACTER / BOOLEAN / REF(CLASS-identifier) / TEXT / [type]ARRAY } identifier-list; } ...

virtual-part



CLASS-body

{statement / split-body}

split-body

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

The available transmission modes for legal parameters to classes are shown in the following table:

CLASS PARAMETERS

Parameter	Transmission	
	call by value	call by reference
value-type	D	X
reference-type	X	D
TEXT	O	D
value-type ARRAY	O	D
reference-type ARRAY	X	D

D : default mode
O : optional
X : illegal

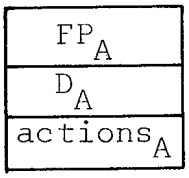
The transmission modes "call by value" and "call by reference" are explained in section 5.4.

The discussion of CLASS-declarations begins by considering two selected examples of increasing scope.

Example 1: Empty prefix and virtual part

The declaration of a class A can have the form:

```
CLASS A(FP_A); SP_A;
BEGIN  D_A;
       actions_A;
END
```



CLASS A declaration A object

A: CLASS-identifier

FP_A: list of the formal parameters of A

SP_A: list of specifications of each of the formal parameters of A

D_A: declarations of the CLASS-body of A

$actions_A$: actions of the CLASS-body of A

The CLASS-body of A is a block (one form of statement).

The quantities passed as parameters or declared in the outermost block of the class body are called the attributes of class A and are attributes of every object of the class. Attributes specified as formal-parameters may only be simple variables or arrays. Declared attributes may be simple variables, arrays, proper-procedures, function-procedures and classes.

The expression

NEW A(...)

creates an object of the class A (the order and number of the actual-parameters must correspond with the order and number of the formal-parameters) and commences execution of its actions. The execution continues until the final END of the class body is encountered, when the execution is terminated. However this may be interrupted in three ways - by a GOTO-statement (which leads out of the object), or by calls on the system procedures "detach", "resume" or "call". A GOTO exit will leave the object in the terminated state. "Detach" suspends the actions of the class body and names it an independent component of the program. Its actions may be continued later by a call on "resume".

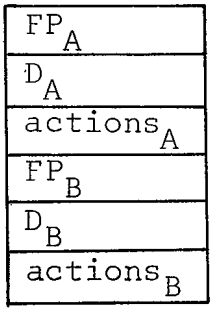
Example 2: Non-empty prefix and empty virtual part

A CLASS-declaration may contain a prefix which is the identifier of another class declared either in the same block, or in the prefix to the same block. The prefixed class is a subclass of the prefix.

e.g.

```
A CLASS B(FP_B); SP_B;
  BEGIN  D_B;
         actions_B;
  END
```

CLASS B declaration



B object

A B-object is a compound object which has all the attributes and actions of the prefix A and the main-part of class B.
A B-object is generated by

```
NEW B(AP_A, AP_B)
```

with a list of actual-parameters corresponding in number and order to those of an A-object and those of the main-part of B. On generation, the actions of the A-part are executed first and then those of the B-part. The actions of the A-part have access to the attributes of A only, the actions of the B-part to those of B and of A.

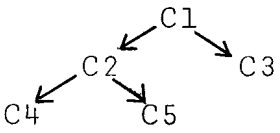
CLASS HIERARCHIES

In general, a hierarchy of classes may be introduced.
e.g.

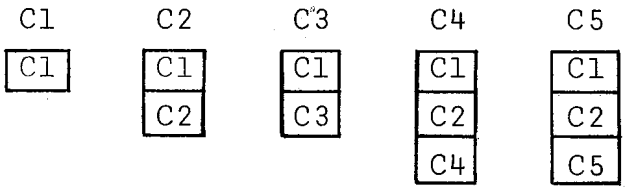
declarations

```
    CLASS C1.....;
C1  CLASS C2.....;
C1  CLASS C3.....;
C2  CLASS C4.....;
C2  CLASS C5.....;
```

hierarchy



objects

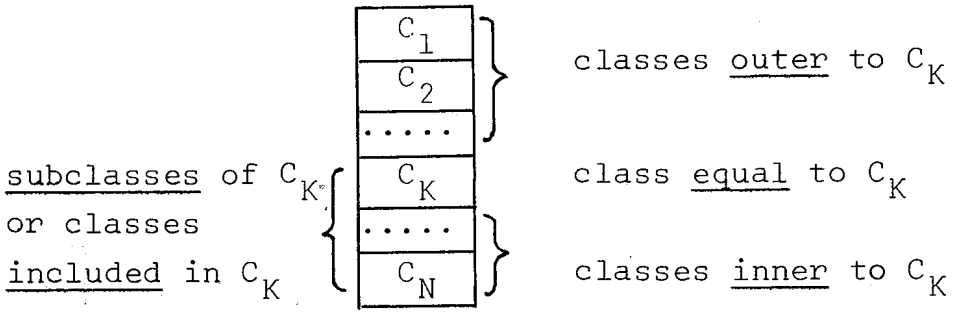


The prefix sequence of a class is the sequence of classes in its prefix chain

e.g. in the above figure, the prefix sequence of

C1 is empty
C2 is C1
C5 is C1 C2

If the prefix sequence of a class C_N is C_1, C_2,....., C_{N-1}, then a C_N object may be depicted by:



An object of a prefixed class is a compound object which is the union of the formal-parameters, declarations and actions of the classes in its prefix sequence together with the structure defined in its own main-part.

The mechanism of concatenation may be extended to the case of prefixed-blocks, e.g.

```
SIMULATION  BEGIN
                ...
            END
```

The execution of a prefixed block begins by executing the actions of the initial operations of the class and then those of the block body. The attributes of the class are accessible inside the block body and this block is thus given a built-in environment in which to operate.

INNER

The order of the execution of actions in an object may be altered from their textual order by use of INNER.

Consider

```
    CLASS A ...;
    BEGIN  ....; S1; INNER; S2  END;

A CLASS B ...;
    BEGIN .....; S3  END
```

In an A object INNER acts as a dummy statement, and the actions executed are

```
    S1; S2
```

In a B object, INNER forces execution of S3 before those of S2, thus

```
    S1; S3; S2
```

VIRTUAL QUANTITIES

The virtual mechanism is a method of extending the rules of accessibility, whilst still retaining security. The virtual mechanism is useful in two circumstances.

1) to redefine attributes
2) to give accessibility to attributes declared at inner levels

1) attribute redefinition

Usually when a class hierarchy is constructed, an attribute has a fixed meaning no matter how many prefix levels are added to the class containing its declaration.

Sometimes, however, we may wish to redefine the attribute throughout its scope.

Example: Consider the description of a CLASS ROW for manipulating vectors.

```
CLASS ROW(A,N); REAL ARRAY A; INTEGER N;
BEGIN  REAL PROCEDURE NORM;
       BEGIN  REAL T; INTEGER I;
              FOR I := 1 STEP 1 UNTIL N DO
                  T := T + A(I)*A(I);
                  NORM := SQRT(T);
       END ***NORM***;
       PROCEDURE NORMALISE;
       BEGIN  REAL T; INTEGER I;
              T := NORM;
              IF T ¬= 0 THEN
                 BEGIN  T := 1/T;
                        FOR I := 1 STEP 1 UNTIL N DO
                            A(I) := A(I)*T;
                 END;
       END ***NORMALISE***;
       IF N < 1 THEN ERROR;
END ***ROW***
```

A new ROW object has four attributes:

1) a REAL ARRAY A running from 1 through N
2) a specified upper bound N
3) a REAL PROCEDURE NORM which computes the square root of the sum of the square of its components. (NORM gives the "magnitude" of the ARRAY A).
4) a PROCEDURE NORMALISE which divides through each array element by the current value of NORM.

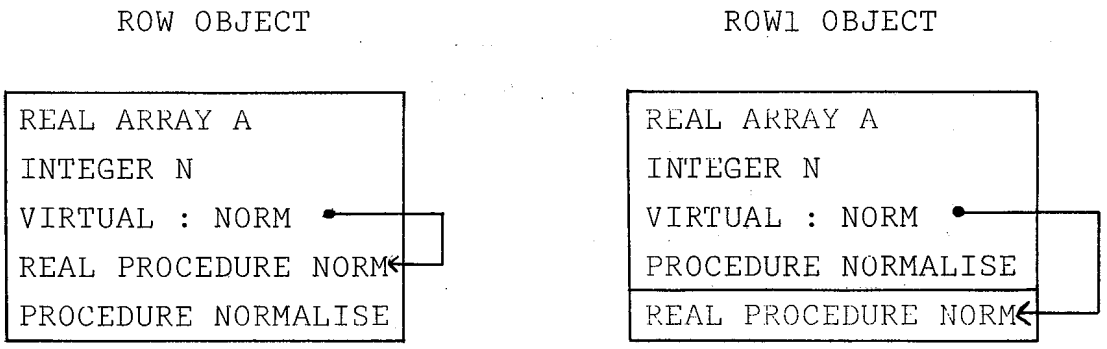
If the user wishes to use a different NORM then the class has to be rewritten.

However by altering the class outline to

```
CLASS ROW(A,N); REAL ARRAY A; INTEGER N;
                VIRTUAL : REAL PROCEDURE NORM;
BEGIN  REAL PROCEDURE NORM...;
       PROCEDURE NORMALISE...;
       IF N < 1 THEN ERROR;
END ***ROW***
```

it becomes possible to alter the definition of NORM in a subclass of ROW, and yet have the new meaning available at the ROW level where it is needed in any call on NORMALISE e.g.

```
ROW CLASS ROW1;
    BEGIN  REAL PROCEDURE NORM;
           BEGIN  REAL S,T; INTEGER I;
                  FOR J := 1 STEP 1 UNTIL N DO
                      BEGIN  T := ABS(A(I));
                             IF T > S THEN S := T;
                      END;
                      NORM := S;
           END ***NORM*** ;
    END ***ROW1***
```



In a ROW1 object, the NORM attribute declared at the ROW level is deleted and is never available. Through the VIRTUAL mechanism, the only valid NORM attribute is that declared at the ROW1 level.

2) Inner accessibility

This is illustrated by an example.

In processing a linked structure which contains objects of various classes, (say A, B) the links are provided by their common part CLASS NODE.

```
CLASS NODE; BEGIN  REF(NODE)NEXT;  END;
NODE CLASS A ....;
NODE CLASS B ....;
```

If the structure is linked by following REF(NODE) NEXT references, then, in the normal way, the information in the structure would be dumped by

```
X :- FIRST;
WHILE X =/= NONE DO
BEGIN  IF X IS A THEN X QUA A.DUMP
                 ELSE X QUA B.DUMP;
       X :- X.NEXT;
END;
```

which is tedious to write. By writing the classes by

```
CLASS NODE; VIRTUAL : PROCEDURE DUMP;
BEGIN  REF(NODE) NEXT;
       ...
END ***NODE*** ;
NODE CLASS A;
     BEGIN  PROCEDURE DUMP ...;
     END;
NODE CLASS B;
     BEGIN  PROCEDURE DUMP ...;
     END;
```

we can write instead

```
X :- FIRST;
WHILE X =/= NONE DO
BEGIN  X.DUMP;
       X :- X.NEXT;
END;
```

A virtual quantity is either unmatched or else matched with an attribute (with the same identifier) declared at the same or an inner prefix level. The matching attribute must be of the same type and the same kind (variable, array, procedure, label or switch) as that specified in the virtual part, except in the case of object-reference-function-procedures. The type REF(C) may be matched with the type REF(C) or type REF(D), where D is a subclass of C. A VIRTUAL proper-PROCEDURE may be matched with any type of procedure. In both these cases, the type of the match is the type of the matching declaration.

A virtual quantity of a given object can have at most one matching attribute. If matching declarations are given at more than one prefix level, then the one at the innermost prefix level is taken. The match is valid at all prefix levels of the object equal or inner to that of the specification.

5.6 EXTERNAL DECLARATIONS

External declarations will be a part of future system development.

6. EXPRESSIONS

Expressions are rules for computing values. Expressions may occur on the right hand sides of assignment statements and as actual parameters.

expression

{ arithmetic-expression
condition
CHARACTER-expression
designational-expression
object-expression
TEXT-expression
TEXT-value }

IF-clause

IF condition THEN

The constituents of expressions are operators and operands. The operands are constants, variables or function designators. No two operands may occur side by side - they must be separated by at least one operator. A discussion of variables and function designators precedes the discussion of expressions.

6.1 VARIABLES

The value of a variable may be used in expressions for forming values and may be changed by assignments to that variable. The value of a variable is the value of the last assignment to it, or its initial value if no assignment has yet been made.

variable

{ simple-variable
{ subscripted-variable
{ remote-variable

simple-variable

identifier

subscripted-variable

{ identifier
{ remote-variable } (subscript-list)

subscript-list

arithmetic-expression [, arithmetic-expression]...

remote-variable

{ simple-text-expression . }
{ simple-object-expression . } identifier

Examples:

simple-variable:	EPS
	A15
	A16_C
subscripted-variable:	SON(2)
	ELEMENT(I,J)
	THIS SQUARE.LONG_SIDE(2)
remote-variable:	ANNA.FATHER
	APOSTLE(6).LEADER
	C.P.X
	LINE(2).SIDE(3)

A variable is a designation for a single value.

simple-variable

The type of the value of a particular simple variable is given in the declaration for that variable.

remote-variable

An attribute of an object is identified by the following information:

1) a reference to the object
2) a class which includes that of the object
3) an attribute identifier defined in that class or any class belonging to its prefix sequence.

Let X be a simple object expression qualified by class C, and A an attribute identifier.

The X.A is valid if X =/= NONE and class C has at least one attribute identified by A. If there is only one attribute with identifier A contained in class C or in its prefix chain, then it is designated by X.A. If class C contains more than one attribute with identifier A, then X.A designates the attribute with identifier A at the innermost prefix level.

N.B. The main part of any class declaration can contain at most one attribute with that identifier.

Example:

```
    REF(C1)X1; REF(C2)X2; REF(C3)X3; REF(C4)X4

    CLASS C1(A); REAL A;
    BEGIN ..................... END;
C4  CLASS C2;
    BEGIN  TEXT A; ... END;
C2  CLASS C3;
    BEGIN  ..................... END;
C3  CLASS C4;
    BEGIN  BOOLEAN A; ........ END;
X1 :- X2 :- X3 :- X4 :- NEW C4(6.0);
```

remote-variable	type
X1.A, X2 QUA C1.A, X3 QUA C1.A, X4 QUA C1.A	REAL
X1 QUA C2.A, X2.A, X3 QUA C2.A, X4 QUA C2.A	TEXT
X1 QUA C3.A, X2 QUA C3.A, X3.A, X4 QUA C3.A	TEXT
X1 QUA C4.A, X2 QUA C4.A, X3 QUA C4.A, X3 QUA C4.A, X4.A	BOOLEAN

It is not possible to access either label or switch identifiers remotely.

If class C or a class in its prefix chain contains a class attribute, then accessing of any attribute in class C is only possible through a connection statement.

Example:

```
     CLASS C1; BEGIN  REAL D;  END;
  C1 CLASS C2;
     BEGIN  CLASS F.....;
            BOOLEAN Q;
     END;
     REF(C2)X;
```

Then X.F, X.Q, X.D
are all illegal attempts at remote accessing.

But it is legal to write

```
INSPECT X DO
BEGIN  ...F...
       ...Q...
       ...D...
END
```

or, for the REAL attribute of C1, X QUA C1.D.

A simple-text-expression is itself a compound structure in the sense that it has attributes accessible through the dot notation. TEXT-value assignments may be made through the use of the system-defined procedure SUB, STRIP and MAIN to sub-fields (T.SUB, T.STRIP) and super-fields (T.MAIN) of the text object referenced by T.

e.g. T.SUB(1,24) := "CHANGE␣IN␣FIRST␣24␣CHARS"
T.MAIN.STRIP := NOTEXT

Subscripted variables

A subscripted variable is an array component. The type of a subscripted variable is the type of the array. Each arithmetic expression in the subscript list is called a subscript and the complete list of subscripts is enclosed in parentheses ( ). Each subscript acts as a variable of type INTEGER and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable.

No attempt is made to ensure that each subscript lies within the corresponding subscript bounds in an array of two or more dimensions. Such arrays are transformed into one dimensional table and the subscripts computed to give access to the appropriate entry. Provided this rule gives an entry within the bounds of the table, then the subscripts will be accepted. Otherwise, a runtime error will result.

6.2 FUNCTION DESIGNATORS

A function designator denotes the value obtained by evaluating the associated procedure body when supplied with the associated actual parameter list (if any).

function-designator

{identifier / remote-identifier} actual-parameter-list

actual-parameter-list

(actual-parameter[, actual-parameter]...)

actual-parameter

{expression / ARRAY-identifier / SWITCH-identifier / PROCEDURE-identifier / LABEL-identifier}

Examples:

```
ININT
INTEXT(20)
SIN(A+B)
H.FIRST
CURRENT.NEXTEV.SUC
POINT(5).MODULUS
```

Function-designators denote single arithmetic, BOOLEAN, TEXT or CHARACTER values or object- or TEXT-references. These values are obtained by applying a set of rules defined by the procedure declaration to a set of actual parameters. The actual parameter list must correspond in number and order to the formal parameter list in the corresponding procedure declaration.

Certain identifiers, expressed as procedures, are pre-defined in the SIMULA system. A list is given as Appendix B. Calls to standard procedures conform to the syntax of calls to declared procedures and are equivalent to normal procedure calls. The identifier of a system defined procedure is not reserved, and may be declared to have another meaning at any level. (This is not recommended practice as it will obviously make the program more difficult to read and understand). The identifier then assumes the new meaning throughout the scope of the block in which it was declared.

6.3 ARITHMETIC EXPRESSIONS

An arithmetic expression is a rule for computing a number.

arithmetic-expression

```
{simple-arithmetic-expression                          }
{IF-clause simple-arithmetic-expression                }
{          ELSE arithmetic-expression                  }
```

simple-arithmetic-expression

```
[±] arithmetic-primary [arithmetic-operator arithmetic-
                                              primary] ...
```

arithmetic-primary

```
{arithmetic-constant                 }
{arithmetic-variable                 }
{arithmetic-function-designator      }
{(arithmetic-expression)             }
```

arithmetic-operator

```
{+ | - | * | ** | / | //}
```

Examples:

arithmetic-primary

```
15.7
#4
X
SIN(23*PI/180)
(IF X>0 THEN X ELSE -X)
```

arithmetic-expression

```
X**2 + Y**2
X + A(J)**2
EXP(2 + (IF B THEN 3 ELSE 4))
IF X > 0 THEN 1 ELSE IF X < 0 THEN -1 ELSE 0
-3*(-5)
```

Examples of incorrect arithmetic expressions:

(A + B) C)	left parenthesis missing
-3*-5	two operators together
(A + B) & - 0.5	exponent is not a primary
X + (IF Y < 0 THEN 2)	ELSE alternative is missing
X + IF Y < 0 THEN 2 ELSE 3	the IF expression must be enclosed in parentheses

The value of a simple-arithmetic-expression is the value obtained by executing the arithmetic operations on the actual numeric values of the primaries of the expression. The actual numeric value of a primary is obvious in the case of numbers.

For variables, it is the current value (assigned last in the dynamic sense); and for function-designators it is the value arising from the computing rules defining the function-procedure when applied to the current values of the procedure parameters in the expression. For arithmetic-expressions enclosed in parentheses, the value must, through a recursive analysis, be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic-expressions, which include IF-clauses, a simple-arithmetic-expression is selected on the basis of the actual values of the BOOLEAN conditions.

If the form of the arithmetic expression is

IF condition THEN simple-arithmetic-expression
ELSE arithmetic-expression

the selection of the appropriate value is made according to the rules:

1. The condition in the IF-clause is evaluated.

2. If the result of 1 is TRUE, then the value of the expression is the value of the simple arithmetic expression following the IF-clause. Otherwise it is the value of the arithmetic expression following the ELSE.

   e.g. IF X < 0 THEN 4 ELSE 17

   if X < 0 then the value is 4
   if X = 0 or X > 0 then the value is 17.

Arithmetic operators and types

Apart from the CONDITIONS of IF-clauses, the constituents of arithmetic expressions are of types SHORT INTEGER, INTEGER, REAL or LONG REAL.

The following hierarchy may be associated with these types.

type number	type	shortened notation
1	SHORT INTEGER	SI
2	INTEGER	I
3	REAL	R
4	LONG REAL	LR

The types are ranked according to their associated type number. Thus LONG REAL is higher than INTEGER, INTEGER is lower than REAL.

The meanings of the basic operators are:

+	addition
-	subtraction
*	multiplication
**	exponentiation
/	division
//	integer division

The types of arithmetic expressions are given by the following rules:

1) The operators + and - may be used as unary operators

```
+ operand
- operand
```

The type of the resulting expression is the type of the operand.

2) The type of the arithmetic expression

```
operand1 {+|-} operand2
```

is the higher of the types of operand1 and operand2. This is displayed in the table:

type of operand1	type of operand2			
	SI	I	R	LR
SI	SI	I	R	LR
I	I	I	R	LR
R	R	R	R	LR
LR	LR	LR	LR	LR

3) / denotes division, and

```
operand1 / operand2
```

always results in an arithmetic expression of type REAL or LONG REAL. If either of operand1 or operand2 are of type LONG REAL then so is the resulting expression, otherwise it is of type REAL.

This is displayed in the table:

operand1 / operand2

type of operand1	type of operand2			
	SI	I	R	LR
SI	R	R	R	LR
I	R	R	R	LR
R	R	R	R	LR
LR	LR	LR	LR	LR

A run time error will always result if the value of operand2 is zero.

4) // denotes integer division. The operator is defined only between two operands of type INTEGER and gives the INTEGER result

```
A // B = SIGN (A*B) * ENTIER (ABS(A/B))
```

If, in

```
operand1 // operand2
```

the types of operand1 and operand2 are not INTEGER, then they are rounded to type INTEGER and then the operation, as defined above, is carried out.

Examples:

```
 10 // 5      =  2
  9 // 5      =  1
 -9 // 5      = -1
  9 //-5      = -1
```

5) The operation

operand1**operand2

denotes exponentiation, where operand1 is the base and operand2 the exponent.

e.g.	2**3	means	8 (=2*2*2)
	2**(3**4)	means	$2^{(3)^4}$
	2**3**4	means	$(2^3)^4$

The type is real unless one of operand1 or operand2 is of type LONG REAL when it is LONG REAL.

6) When the type of an arithmetic expression can not be determined at compile-time, it is considered to be the highest possible type in the hierarchy of the alternatives.

If in

IF X > 0 THEN P.R ELSE S

the result of an expression is of one arithmetic type and the expression is assigned to a variable of another arithmetic type, the result is transformed to the appropriate type.

Arithmetic operator precedence

The evaluation of an arithmetic-expression can be considered as taking place in a number of scans through the expression, each scan going from left to right:

1) The values of the constants, variables and function designators are evaluated.
2) Any subexpression between a left parenthesis and its matching right parenthesis is evaluated (according to rules 2, 3, 4, 5) and is used as a primary in subsequent calculations.
3) All exponentiations are evaluated.
4) All multiplications, divisions and integer divisions are evaluated.
5) All additions and subtractions are evaluated.

The operator precedences may be summarised by

1. ( )
2. **
3. * / //
4. + -

Arithmetics of REAL and LONG REAL quantities

Quantities of type REAL and LONG REAL are defined to a finite accuracy. Thus any operation performed on such quantities is subject to a possible deviation. The control of the possible consequences should be carried out by a numerical analyst.

6.4 CONDITIONS

Conditions (or BOOLEAN-expressions) return the values TRUE or FALSE. Their prime use is in conditional expressions and conditional statements.

condition

```
{BOOLEAN-primary [BOOLEAN-operator  BOOLEAN-primary]...}
{IF-clause BOOLEAN-primary ELSE condition               }
```

BOOLEAN-primary

```
                  {BOOLEAN-variable          }
                  {BOOLEAN-function-designator}
                  {TRUE                      }
[¬]               {FALSE                     }
                  {relation                  }
                  {(condition)               }
```

BOOLEAN-operator

```
{AND|OR|EQV|IMP}
```

relation

```
{simple-value-expression     {=  }     simple-value-expression     }
{                            {>  }                                 }
{                            {>= }                                 }
{                            {<  }                                 }
{                            {<= }                                 }
{                            {=  }                                 }
{                                                                  }
{simple-reference-expression {== }     simple-reference-expression }
{                            {=/= }                                }
{                                                                  }
{simple-reference-expression {IS }     CLASS-identifier            }
{                            {IN }                                 }
```

Note the equivalent modes of writing:

¬	or	NOT
=		EQ
¬=		NE
<		LT
<=		LE
>		GT
>=		GE

Examples:

```
TRUE
SON AND HOLY_GHOST
X = 0 OR ¬ (Y > C)
¬ (X IN HEAD)
T == NOTEXT
FATHER =/= NONE
C = 'A'
```

BOOLEAN OPERATORS

The meanings of the logical operators

NOT	not
AND	and
OR	or
IMP	implication
EQV	equivalent

are given in the table:

P	T	T	F	F	
Q	T	F	T	F	
¬P	F	F	T	T	IF P THEN FALSE ELSE TRUE
P AND Q	T	F	F	F	IF P THEN Q ELSE FALSE
P OR Q	T	T	T	F	IF P THEN TRUE ELSE Q
P IMP Q	T	F	T	T	IF P THEN Q ELSE TRUE
P EQV Q	T	F	F	T	IF P THEN Q ELSE Q

Precedence of operators

The sequence of operations within one condition is from left to right with the following additional rules:

first:	arithmetic expressions (+, -, **, *, /, //) according to section 6.3
second:	relations (<, <=, =, ¬=, >, >=, ==, =/=, IS, IN)
third:	¬
fourth:	AND
fifth:	OR
sixth:	IMP
seventh:	EQV

The use of parentheses is interpreted as in section 6.3.

Relations

1) IS, IN

The operators IS and IN test the class membership of a referenced object:

X IS C — is TRUE only if X is a reference to an object belonging to the class C, otherwise the value is FALSE

X IN C — is TRUE only if X is a reference to an object belonging to either the class C or to a subclass of C, otherwise the value is FALSE.

Both X IN C and X IS C are FALSE if X == NONE.

2) Reference comparators (==, =/=)

These operators compare either two simple text references or two simple object expressions. Two object (text) references X and Y are identical if they refer to the same object (text value instance) or if both are NONE (NOTEXT). In these cases, the relation X == Y is TRUE. Otherwise it is FALSE.

The relation X =/= Y has the value of ¬(X==Y).

If X and Y are two text references, then X =/= Y and X = Y may both be TRUE if X and Y refer to physically distinct character sequences which are equal.

3) Value comparators (=, ¬=, >, >=, <, <=)

The relations take the value TRUE if the corresponding relation is satisfied for the actual values of the expressions involved, otherwise FALSE.

Examples:

5	=	3	is	FALSE
'␣'	=	'+'	is	FALSE
X	<	0	is	FALSE if the current value of X is positive or zero, otherwise it is TRUE

If the values of two arithmetic expressions are compared and they are of different types, then the value lower down in the hierarchy is converted to the type of the higher before comparison is made.

Character values may be compared with respect to the collating sequence. A relation

```
X REL Y
```

where X and Y are character values, and REL is a relational operator has the same truth value as

```
RANK(X) REL RANK(Y)
```

Example:

```
'+' < '='      is TRUE        RANK('+') = 78
                              RANK('=') = 126
```

Two text values are equal if they are both empty or if they are both instances of the same character sequence. Otherwise they are unequal. A text value T ranks lower than a text value U if they are unequal in length and one of the following conditions is fulfilled (otherwise it ranks higher):

1) T is empty
2) U is equal to T followed by one or more characters
3) If the first i-1 characters of T and U are the same, and the ith character of T ranks lower than the ith character of U.

Examples:

NOTEXT = ""	TRUE
"0" < "9"	TRUE
"ABCDE"="ABCDEF"	FALSE
"+12" =/= "=12"	TRUE
"ABC" = "ABC␣"	FALSE

6.5 CHARACTER EXPRESSIONS

CHARACTER-expression

```
{simple-character-expression                    }
{IF-clause simple-character-expression          }
{   ELSE character-expression                   }
```

simple-character-expression

```
{CHARACTER-constant                             }
{CHARACTER-variable                             }
{CHARACTER-function-designator                  }
{(CHARACTER-expression)                         }
```

CHARACTER-constant

```
'{any character from the data character set}'
```

Example:

```
'''
'A'
IF X < 0 THEN '␣' ELSE 'Y'
```

The set of internal characters is ordered according to the collating sequence (Appendix A). The collating sequence defines a one-one mapping between internal characters and integers expressed by the function procedures:

```
INTEGER PROCEDURE RANK (C); CHARACTER C;
```

whose value is in the range 0 through 255 and

```
CHARACTER PROCEDURE CHAR (N); INTEGER N;
```

The parameter value must be in the range 0 through 255 otherwise a run time error results.

Two character subsets are defined by the standard procedures

```
BOOLEAN PROCEDURE DIGIT (C); CHARACTER C;
```

which is TRUE only if C is a digit, and

```
BOOLEAN PROCEDURE LETTER (C); CHARACTER C;
```

which is TRUE only if C is an upper case letter.

Example:

The following program scans an input file on SYSIN until "end of file" is met and records the number of occurrences of the digits 0 - 9.

```
BEGIN  INTEGER ARRAY INCIDENCES (RANK('0') : RANK('9'));
       CHARACTER C; INTEGER R;
       INSPECT SYSIN DO
       WHILE ¬ ENDFILE DO
       BEGIN  C := INCHAR;
              IF DIGIT (C) THEN
                 BEGIN  R := RANK(C);
                        INCIDENCES(R) := INCIDENCES(R) + 1;
                 END;
       END;
       ...
END
```

6.6 DESIGNATIONAL EXPRESSIONS

A designational expression is a rule for obtaining the label of a statement.

designational-expression

```
{simple-designational-expression                    }
{IF-clause simple-designational-expression          }
{   ELSE designational-expression                   }
```

simple-designational-expression

```
{LABEL-identifier                  }
{SWITCH-identifier (subscript)     }
{(designational-expression)        }
```

Examples:

valid
```
LAB
CHOOSE (I)
SELECT (IF X < 0 THEN 2 ELSE 3)
IF B < 0 THEN LAB_1 ELSE SLAB(1)
```

invalid
```
X.LAB                remote SWITCH- or LABEL-
                     identifiers are not allowed
```

The principle of evaluation is analogous to that of arithmetic-expressions. In the general case, the conditions in the IF-clauses will select a simple-designational-expression. If this is a label the desired result is found. A switch designator, on the other hand, refers to the corresponding switch declaration and by use of the actual numerical value of its subscript (an arithmetic-expression) selects one of the designational-expressions listed in the switch declaration by counting from left to right. Since this value itself may be a switch designator, the definition is recursive.

The evaluation of the subscript is analogous to that of subscripted variables.

6.7 OBJECT EXPRESSIONS

An object-expression is a rule for computing a reference to an object or NONE. The type of the expression is REF (qualification).

object-expression

```
{ simple-object-expression                    }
{ IF-clause simple-object-expression          }
{     ELSE object-expression                  }
```

simple-object-expression

```
{ NONE                                                    }
{ variable                                                }
{ function-designator                                     }
{ NEW CLASS-identifier[actual-parameter-part]             }
{ THIS CLASS-identifier                                   }
{ simple-object-expression QUA CLASS-identifier           }
{ (object-expression)                                     }
```

Examples:

```
X
X.Y
SIDE (3)
THIS HEAD.SUC
NEW INFILE("CARDS")
THIS LINK QUA SUB_CLASS.ATTRIB
IF SUC IN LINK THEN SUC ELSE NONE
IF X =/= NONE
   THEN (IF X.F =/= NONE THEN X.F.F ELSE NONE)
   ELSE NONE
```

Qualification

The qualification of an object-expression is defined by the following rules:

1) The expression NONE is qualified by a fictitious class which is inner to all declared classes.

2) A variable, array or function designator is qualified as stated in the declaration (or specification, see below) of the variable or array or procedure in question.

3) An object-generator, local-object or qualified-object is qualified by the class of the identifier following the symbol NEW, THIS or QUA respectively.

4) A conditional object-expression is qualified by the class at the innermost prefix level which includes the qualification of both alternatives. If there is no such class, the expression is illegal.

   In

   ```
   IF B THEN NONE ELSE S
   ```

   the qualification is that of S (by rule 1).

5) A formal parameter of object-reference type is qualified according to its specification regardless of the qualification of the corresponding actual parameter.

6) The qualification of a function-designator whose procedure identifier is that of a virtual quantity, depends on the access level. The qualification is that of the matching declaration, if any, occurring at the highest prefix level equal to or less than the access level, or if none, that of the virtual specification.

Object generators (NEW A....)

An object-generator invokes the generation and execution of an object belonging to the identified class. The object is a new instance of the corresponding (concatenated) class body. The evaluation of an object generator consists of the following actions.

1) The object is generated and the actual parameters of the object generator are evaluated. The parameter values and/or references are transmitted. (For parameter transmission modes, see section 5.4).

2) Control enters the body of the object through its initial BEGIN whereby it becomes operating in the "attached" state.

The evaluation of the object generator is completed:

Case a: Whenever the basic procedure "detach" is executed "on behalf of" the generated object (see PART 3, section 1), or

Case b: upon exit through the final end of the object.

The value of an object-generator is the object generated as the result of its evaluation. The state of the object after the evaluation is either "detached" (case a) or "terminated" (case b).

Local objects

A local-object "THIS C" is a meaningful expression within

1) the class body of C or that of any subclass of C, or
2) a connection block whose qualification is C or a subclass of C.

The value of a local-object in a given context is the object which is, or is connected by, the smallest textually enclosing block instance, in which the local object is a meaningful expression. If there is no such block, the local object is illegal. For an instance of a PROCEDURE- or CLASS-body, "textually enclosing" means containing its declaration.

Instantaneous qualification

Let X represent any simple reference expression, and let C and D be class identifiers such that D is the qualification of X. The qualified object "X QUA C" is then a legal object expression, provided that C includes D or is a subclass of D. Otherwise, i.e. if C and D belong to disjoint prefix sequences, the qualified object is illegal.

The evaluation of X QUA C at run time gives an error if the value of X is NONE, or an object belonging to a class not included in C. Otherwise the value of X QUA C is that of X. (Note the qualification of X is D, whilst that of X QUA C is C).

6.8 TEXT EXPRESSIONS AND TEXT VALUES

A TEXT-expression is a rule for computing a reference to a TEXT object. A TEXT-value is a rule for computing the value of a text (i.e. string of characters).

TEXT-expression

```
{ simple-text-expression                                              }
{ IF-clause simple-text-expression ELSE TEXT-expression               }
```

simple-text-expression

```
{ NOTEXT                  }
{ TEXT-variable           }
{ TEXT-function-designator}
{ (TEXT-expression)       }
```

TEXT-value

```
{ simple-text-expression }
{ TEXT-constant          }
```

Examples:

valid

TEXT-expression

```
NOTEXT
IF T.LENGTH < 5 THEN NOTEXT ELSE BLANKS(80)
(RUNE,T)
T.SUB(1,12)
```

TEXT-value

```
"ANOTHER␣CONSTANT"
NOTEXT
```

invalid

```
IF T.LENGTH < 5 THEN "MAX␣FOUR"
                ELSE "FIVE"
```

TEXT-constants may not be constituents of conditional expressions.

7. BLOCKS AND STATEMENTS

A program is structured into blocks. A block consists of a block head which defines properties of the block, and a compound tail which defines the actions of the block. A block may be prefixed which means that the block is built into a predefined environment.

There is no way for a block environment to interact with an inner block (i.e. examine its current state). The environment can only observe the result of its actions. A block may only be entered through its initial BEGIN and attempts to do so by GOTO statements are thus illegal.

7.1 BLOCKS AND COMPOUND STATEMENTS

program

{block / compound-statement}

block

{prefixed-block / main-block}

prefixed-block

CLASS-identifier [actual-parameter-list]{main-block / compound-statement}

main-block

BEGIN {declaration;}...[statement;]... [statement]END

compound-statement

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

A block automatically introduces a new level of nomenclature. Any identifier may be declared or appear as a label in the block and is then said to be local to it. An entity represented inside a block has no existence outside it and any entity represented by this identifier outside the block cannot be directly accessed by use of it (see the explanation of the binding rule in section 3).

When a block is prefixed, the identifiers declared in the corresponding CLASS are made available. Nevertheless, an identifier in the CLASS may be redefined in the main-block or compound-statement.

The execution of a block is as follows:

step 1: if the block is prefixed then the actual parameters if any are evaluated.

2: if the declarations of the block contain array bounds then these are evaluated. (They may make reference to parameters of the prefix).

3: Execution of the statements body begins with the first statement of the prefix, if any, otherwise with the first statement of the main block. After execution of the block body (unless it is a GOTO statement) a block exit occurs and the statement textually following the entire block is executed.

A CLASS identifier possibly followed by an actual parameter list can prefix a main-block or compound-statement. This results in concatenating the object of the stated class with that main-block or compound-statement, which means that the capabilities of the stated class and its including classes are available within that main-block or compound statement.

When an instance of a prefixed block is generated, the formal parameters of the class are initialised as indicated by the actual parameters of the block prefix. A virtual quantity is identified by the quantity defined by a matching declaration in the block head of the main-block or compound-statement, or by the matching definition at the innermost prefix level of the prefix sequence. The operation rule of the concatenated object is defined by principles similar to those given in section 5.5.

A prefixed-block plays a particular role in the quasi-parallel sequencing of SIMULA: it defines a quasi-parallel system, a system whose components have interactions independent from the program outside the prefixed block, see PART 3.

The following restrictions must be observed when prefixing blocks:

An object in which reference is made to the object itself through use of THIS is an illegal block prefix.

The CLASS-identifier of a prefix must refer to a class local to the smallest block enclosing the prefixed block. If that CLASS-identifier is that of a system class, it refers to a fictitious declaration of that system class occurring in the block head of the smallest enclosing block.

A compound-statement is a means of grouping several statements together to act as one unit, as in

```
FOR I := 1 STEP 1 UNTIL 10 DO
    BEGIN  SUM := SUM + A(I);
           SUMSQ := SUMSQ + A(I)*A(I);
    END
```

where the controlled-statement is a compound-statement,

or of legalising a statement structure

```
IF X > 0 THEN BEGIN  IF Y > 0 THEN QUADRANT := 1;
              END
```

The structure is illegal without the BEGIN - END pair.

7.2 STATEMENTS

A statement is a unit of action in a program. Sequences of statements may be grouped together to form compound statements or blocks.

Special notice must be taken of conditional statements which effectively prevent a certain ambiguity from arising. For consider

```
IF condition1 THEN IF condition2 THEN S2 ELSE S3
```

which has two interpretations

1) IF condition1 THEN IF condition2 THEN S2 ELSE S3

2) IF condition1 THEN IF condition2 THEN S2 ELSE S3

This ambiguity is resolved by not allowing a conditional statement to follow the THEN symbol. Now the two meanings are resolved by writing

```
1) IF condition1 THEN
      BEGIN  IF condition2 THEN S2 ELSE S3  END

2) IF condition1 THEN BEGIN  IF condition2 THEN S2  END
                     ELSE S3
```

The same sort of ambiguity can arise with a WHILE-statement, a FOR-statement, or a connection statement following a THEN.

```
IF condition1 THEN
    WHILE J < 10 DO
       IF condition2 THEN S1 ELSE S2

IF condition3 THEN
   INSPECT X DO
   IF condition4 THEN S3 ELSE S4
```

Here the syntax forces the ELSE-branch to belong to the nearest THEN, but means that whereas we can write

IF condition THEN { WHILE-statement / FOR-statement / connection-statement } ,

we can not write

IF condition THEN { WHILE-statement / FOR-statement / connection statement } ELSE S

In this case, BEGIN - END are necessary round the WHILE-, FOR-, or connection-statement.

statement

{open / closed / IF-statement}

open

[label:]... {FOR-statement / WHILE-statement / connection-statement}

closed

[label:]... {block / compound-statement / activation-statement / PROCEDURE-statement / dummy-statement / GOTO-statement / assignment-statement / object-generator}

label

LABEL-identifier

ASSIGNMENTS

Assignment-statements assign the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may occur only within the body of the function procedure defining the value of that function. The process of assignment takes place in three steps:

1) any subscript expressions in the left variables are evaluated in sequence from left to right
2) the expression is evaluated
3) the value of the expression is assigned to the left part variables from right to left (see example below under Types).

assignment-statement

{value-assignment / reference-assignment}

value-assignment

{variable := / PROCEDURE-identifier := / TEXT-function-designator :=} ... value-expression

Examples:

```
X := 3.45
C := D(I) := '+'
P.X := R.X + S.X
T := "MESSAGE"
T.SUB(1,5) := S := "12345"
Y := IF X > 0 THEN X ELSE -X
```

TEXT-value-assignment

Consider the TEXT-value-assignment

```
R := T
```

let the length of R be Lr and the length of T be Lt. The assignment is legal if Lr >= Lt, otherwise a run time error results.

Lr = Lt the text value of T is copied into R

Lr > Lt The text value of T is copied into R left justified and the remaining Lr-Lt positions are filled with blanks

Assignments to overlapping texts are unpredictable.

e.g. with

```
        T.SUB(10,10) := T.SUB(15,10)
or      T.SUB(15,10) := T.SUB(10,10)
```

The value assignment

```
T := NOTEXT
```

sets the value of T to blanks.

Types

The type of the expression and all the left part variables or PROCEDURE-identifiers must be identical except in the case of arithmetic assignments. In this case, the assignment

```
v1 := v2 := ..... vn := expression
```

is equivalent to

```
vn := expression
vn-1 := vn
  .....
v1 := v2
```

So that if X is REAL and I INTEGER,

```
X := I := 3.57
```

is equivalent to

```
I := 4;
X := 4.0
```

and

```
I := X := 3.57
```

is equivalent to

```
X := 3.57;
I := 4
```

GOTO-STATEMENTS

A GOTO statement interrupts the normal sequence of operations. The value of a designational expression indicates the value of the label of the next statement to be executed.

GOTO-statement

```
GOTO designational-expression
```

Examples:

```
GOTO L
GOTO S(4)
GOTO S(IF N=0 THEN 2 ELSE 6)
GOTO IF N=0 THEN L1 ELSE L2
```

A GOTO statement may not lead into

- a connection statement
- a block which is not already active
- a FOR-statement
- a WHILE-statement

A GOTO statement leaving an attached object leaves the object in the terminated state.

When the value of a designational expression is a switch with an undefined value (the index is out of range), the GOTO statement is equivalent to the dummy statement
e.g.

```
;GOTO S(-1);          is equivalent to
; ;
```

Example:

The program below prints out the first three verses of "The Twelve Days of Christmas". The program logic is built around a SWITCH.

```
BEGIN  SWITCH CASE := LINE1, LINE2, LINE3;
       INTEGER VERSE;
       TEXT ARRAY T(1:3);
       T(1) :- COPY("FIRST");
       T(2) :- COPY("SECOND");
       T(3) :- COPY("THIRD");

IF_WE_USE_FOR_HERE_THE_LABELS_BECOME_INVISIBLE:
       VERSE := VERSE + 1;
       OUTTEXT("ON␣THE␣"); OUTTEXT(T(VERSE));
       OUTTEXT("␣DAY␣OF␣CHRISTMAS"); OUTIMAGE;
       OUTTEXT("MY␣TRUE␣LOVE␣SENT␣TO␣ME"); OUTIMAGE;
       GOTO CASE(VERSE);
LINE3: OUTTEXT("THREE␣FRENCH␣HENS");
       OUTTEXT("TWO␣TURTLE␣DOVES,␣AND");
       OUTTEXT("A␣PARTRIDGE␣IN␣A␣PEAR␣TREE");
       EJECT(LINE + 2);
       IF VERSE < 3 THEN
          GOTO IF_WE_USE_FOR_HERE_THE_LABELS_BECOME_INVISIBLE;
END
```

OUTPUT:

```
ON THE FIRST DAY OF CHRISTMAS
MY TRUE LOVE SENT TO ME
A PARTRIDGE IN A PEAR TREE

ON THE SECOND DAY OF CHRISTMAS
MY TRUE LOVE SENT TO ME
TWO TURTLE DOVES, AND
A PARTRIDGE IN A PEAR TREE

ON THE THIRD DAY OF CHRISTMAS
MY TRUE LOVE SENT TO ME
THREE FRENCH HENS
TWO TURTLE DOVES, AND
A PARTRIDGE IN A PEAR TREE
```

DUMMY STATEMENTS

A dummy-statement executes no actions. Its main use is to place a label before an END

dummy statement

Examples:

```
IF X > 0 THEN ELSE X := -X;

BEGIN ..... LAB : END;
```

WHILE STATEMENTS

The WHILE-statement is used when a statement is to be executed while a condition is TRUE.

WHILE-statement

WHILE condition DO statement

Examples:

```
WHILE ¬ LASTITEM DO
BEGIN  X := ININT;
       COUNT := COUNT + 1;
       SUM := SUM + X;
END;

WHILE X =/= NONE DO
BEGIN  IF X IN POINT THEN NOP := NOP + 1 ELSE
          IF X IN LINE THEN NOL := NOL + 1 ELSE
             ERROR("FALSE␣ENTRY");
       X :- X.SUC;
END;
```

ACTIVATION STATEMENTS

Activation statements are only valid inside a SIMULATION block. They are fully described in part 3, section 3.

FOR STATEMENTS

FOR-statements are control statements which cause controlled-statements to be executed zero or more times.

There are three basic types of FOR statement element.

1) the controlled-statement is executed for a list of values (usually irregular).

```
FOR I := 2, 13, 17 DO controlled-statement
```

2) the controlled-statement is executed a known number of times

```
FOR I := 1 STEP 1 UNTIL 10 DO
    X(I) := I*I
```

3) the controlled-statement is executed until a condition is met

```
FOR X :- X.SUC WHILE X =/= NONE DO
    X QUA LINK.OUT
```

To increase the generality of the concept, these elements themselves are allowed to form a list

```
FOR I := 1, 2, 4 STEP 1 UNTIL 10, I*I WHILE I < 200 DO
  controlled statement
```

FOR-statement

[label:]... FOR controlled-variable for-right-part DO controlled statement

for-right-part

{:= value-element [,value-element]
:- object-element [,object-element]}

value-element

{value-expression
arithmetic-expression STEP arithmetic-expression UNTIL arithmetic-expression
value-expression WHILE condition}

object-element

{object-expression
object-expression WHILE condition}

Each execution of the controlled statement is preceded by an assignment to the controlled-variable. Assignments may change the value of this controlled variable during execution of the controlled-statement.

for list elements

The for list elements are considered in the order in which they are written. When one for list element is exhausted, control proceeds to the next, until the last for list element in the list has been exhausted. Execution then continues after the controlled statement.

The effect of each type of for list element is defined below using the following notation:

C: controlled variable
V: value expression
O: object expression
A: arithmetic expression
B: Boolean expression
S: controlled statement

The effect of the occurrence of expressions in for list elements may be established by textual replacement in the definitions.

α, β, σ are different identifiers which are not used elsewhere in the program. σ identifies a non-local simple variable of the same type as A_2.

1. V

```
    C := V;
    S;
    next for list element
```

2. A_1 step A_2 until A_3

```
    C := A1;
    σ := A2;
α : if  σ*(C-A3) > 0 then goto β;
    S;
   σ := A2;
    C := C + σ;
    goto α;
β : next for list element
```

3. V while B

```
α : C := V;
    if ¬ B then goto β;
    S;
    goto α;
β : next for list element
```

4. 0

```
C :- 0;
S;
next for list element
```

5. 0 while B

```
α : C :- 0;
    if ¬ B then goto β;
    S;
    goto α;
β : next for list element
```

The controlled variable

The controlled-variable is a simple-variable which is not a formal-parameter called by name, a PROCEDURE-identifier, a remote-identifier nor a TEXT-identifier.

To be valid, all for list elements in a for statement must be semantically and syntactically valid.

In particular each implied assignment is subject to the rules of section 7.2.1.3.

The value of the controlled variable upon exit

Upon exit from the for statement, the controlled variable will have the value given to it by the last (explicit or implicit) assignment operation.

Labels local to the controlled statement

The controlled statement always acts as if it were a block. Hence, labels on or defined within the controlled statement may not be accessed from without the controlled statement.

PROCEDURE STATEMENTS

A procedure-statement calls for the execution of the PROCEDURE-body. Before execution, the formal-parameters of the procedure are replaced by the actual-parameters.

PROCEDURE-statement

[simple-object-expression.] PROCEDURE-identifier [(expression
[, expression]...)]

Examples:

```
INTO(H)
OUTTEXT("***0***")
SYSIN.INIMAGE
```

The procedure statement must have the same number of actual parameters in the same order as the formal-parameters of the procedure heading.

Restrictions

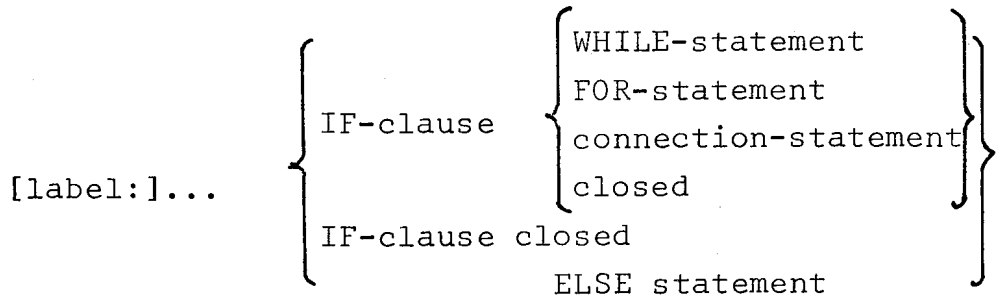
1) An actual-parameter corresponding to a formal-parameter called by NAME which is assigned to within the PROCEDURE-body must be a variable.

2) If the formal-parameter is an ARRAY (PROCEDURE), then the number of dimensions (actual-parameters) used within the PROCEDURE-body must correspond to the number of dimensions (actual-parameters) of the actual ARRAY (PROCEDURE).

CONDITIONAL STATEMENTS

Conditional statements cause certain statements to be executed or skipped depending on the current values of certain conditions. They provide an important structural framework in the language.

IF-statement



Examples:

```
    L:  IF X > 0 THEN X := -X;

        IF X = 0 AND Y = 0 THEN
    L:      BEGIN  OUTTEXT("ORIGIN");
                   OUTIMAGE;
            END;

            X :- T;
            WHILE X =/= NONE DO
            BEGIN  IF X QUA PERSON.MALE THEN MAN := MAN + 1
                                        ELSE WOMAN := WOMAN + 1;
                   X :- X.SUC;
            END;
```

The statement sequence

```
IF condition THEN S1; S2
```

is equivalent to

```
(evaluate condition) S1; S2     if the condition is TRUE
(evaluate condition) S2         if the condition is FALSE.
```

The statement sequence

```
IF condition THEN S1 ELSE S2; T
```

is equivalent to

```
(evaluate condition) S1; T      if the condition is TRUE, and
(evaluate condition) S2; T      if the condition is FALSE.
```

A GOTO statement may lead directly into a conditional statement e.g

```
IF B THEN  BEGIN  OUTTEXT("TRUE");
           L:     OUTIMAGE;
           END
     ELSE  BEGIN  OUTTEXT("FALSE");
                  GOTO L;
           END
```

CONNECTION STATEMENTS

Connection-statements are a form of remote accessing which are mainly used as a user convenience if there are to be many accesses to a particular object.

e.g.

we may replace

```
SYSBIN.INIMAGE;
X := SYSBIN.INREAL;
Y := SYSBIN.INREAL;
C := SYSBIN.INCHAR;
```

by

```
INSPECT SYSBIN DO
BEGIN  INIMAGE;
       X := INREAL;
       Y := INREAL;
       C := INCHAR;
END
```

Connection statement

```
INSPECT object-reference DO { statement
                            { [WHEN CLASS-identifier DO statement]...
                            {                [OTHERWISE statement]
```

Examples:

```
INSPECT SYSOUT DO
BEGIN  OUTTEXT("TITLE");
       OUTIMAGE;
       EJECT(LINE+10);
END;

INSPECT X WHEN A DO OUTTEXT("X IN A")
          WHEN B DO OUTTEXT("X IN B")
          OTHERWISE OUTTEXT("ERROR");

INSPECT X DO
  INSPECT Y DO P := Q
            OTHERWISE OUTTEXT("Y==NONE");
```

To avoid ambiguity, an OTHERWISE refers back to the nearest INSPECT.

The remote accessing of objects of classes may be accomplished by the dot notation or by connection. In most cases the methods are interchangeable if the object contains a class attribute at a certain level, then attributes at that level and levels inner to it can only be accessed by connection.

To access an attribute of an object we have to know:

1) a reference to the object
2) the qualification of the reference
3) the attribute identifier.

The two variations are now explained:

1) a) INSPECT X DO S1 OTHERWISE S2
   b) INSPECT X DO S1

If X =/= NONE then the statement S1 is executed. During execution of this statement the reference variable of X is evaluated once and stored. Access is now gained to the denoted object at the qualification level of X. All attributes of the qualifying class are now available by the occurrence of their identifiers.

If X == NONE, then the statement S1 is skipped and the statement S2 is executed in case a), and the whole connection statement is skipped in case b).

In an otherwise branch, no connection holds.

e.g.

```
        CLASS A;
        BEGIN  REAL X; ..... END;
      A CLASS B
        BEGIN  BOOLEAN Y; .. END;
        REF(A)Q; REF(B)Z;
```

```
INSPECT Q DO
BEGIN  X := 2;
       THIS B.Y := FALSE;
       Z :- NONE;
END
```

```
INSPECT Q QUA B DO
BEGIN  X := 2;
       Y := FALSE;
       Z :- NONE;
END
```

2) INSPECT X WHEN A DO S1
WHEN B DO S2
.....
OTHERWISE T

This discriminates on class membership at run time. The when clauses are considered in turn. If X is an object belonging to a class equal to or inner to the one identified in the clause then the associated statement is executed and the rest of the connection-statement is skipped. The OTHERWISE-clause (if present) is only executed of X == NONE or all preceding WHEN clauses are skipped.

OBJECT GENERATORS

Object-generators were discussed in section 6.5. They may also stand as statements in their own right, in which case their reference value is not assigned on exit. This does not mean that they cannot be referenced as the following example shows:

```
LINK CLASS B(H); REF(HEAD)H;
BEGIN  PTR :- THIS B;
       INTO(H);
END
```

After execution of the statement

```
NEW B(HD);
```

then the generated object is referenced by PTR and HD.LAST (if H =/= NONE).

PART 3 PROGRAM STRUCTURE AND SYSTEM FEATURES

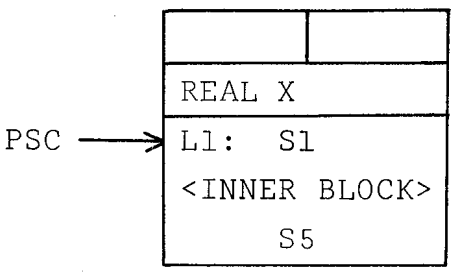
1 PROGRAM STRUCTURE

Without the CLASS concept, the run time structure of a SIMULA program is essentially that of a stack of blocks with the last created block "active". Given the program skeleton

```
BEGIN   REAL X;
L1:     S1;
        BEGIN   BOOLEAN B;
                S2;
                BEGIN   TEXT X;
        L3:             S3;
                END;
L2:             S4;
        END;
        S5;
END
```

(where the S's denote statement sequences) then the structures at run time are

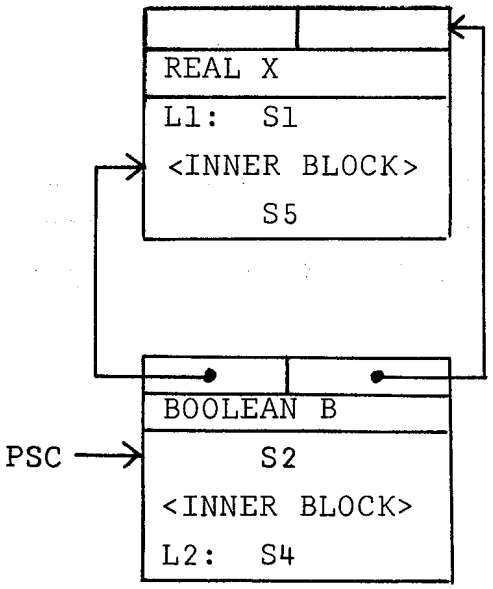
1. during execution of S1 or S5



PSC denotes the current statement

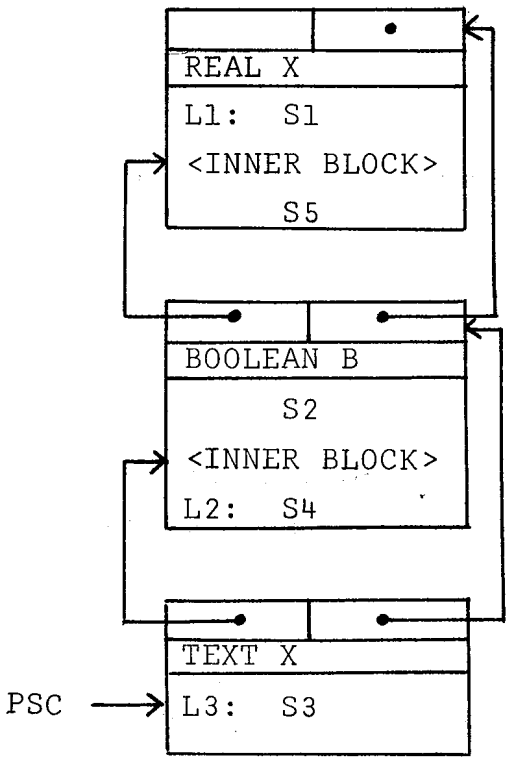
Only one variable REAL X exists.

2. during execution of S2 (or S4)



Two variables REAL X and BOOLEAN B exist.

3. during execution of S3



STACK OF
"ATTACHED"
BLOCKS

Three variables exist REAL X, BOOLEAN B, TEXT X but only two are accessible by actions in the active block, for the REAL X of the outermost block has become inaccessible due to an identifier clash with TEXT X of the innermost block.

Note that besides exiting from a block via its final END, it is also possible to exit via a GOTO-statement. If the statement sequence S3 includes the statement

```
GOTO L2
```

then on execution of that statement, the innermost block is deleted (the variable TEXT X will no longer exist).

If the statement sequence S3 includes the statement

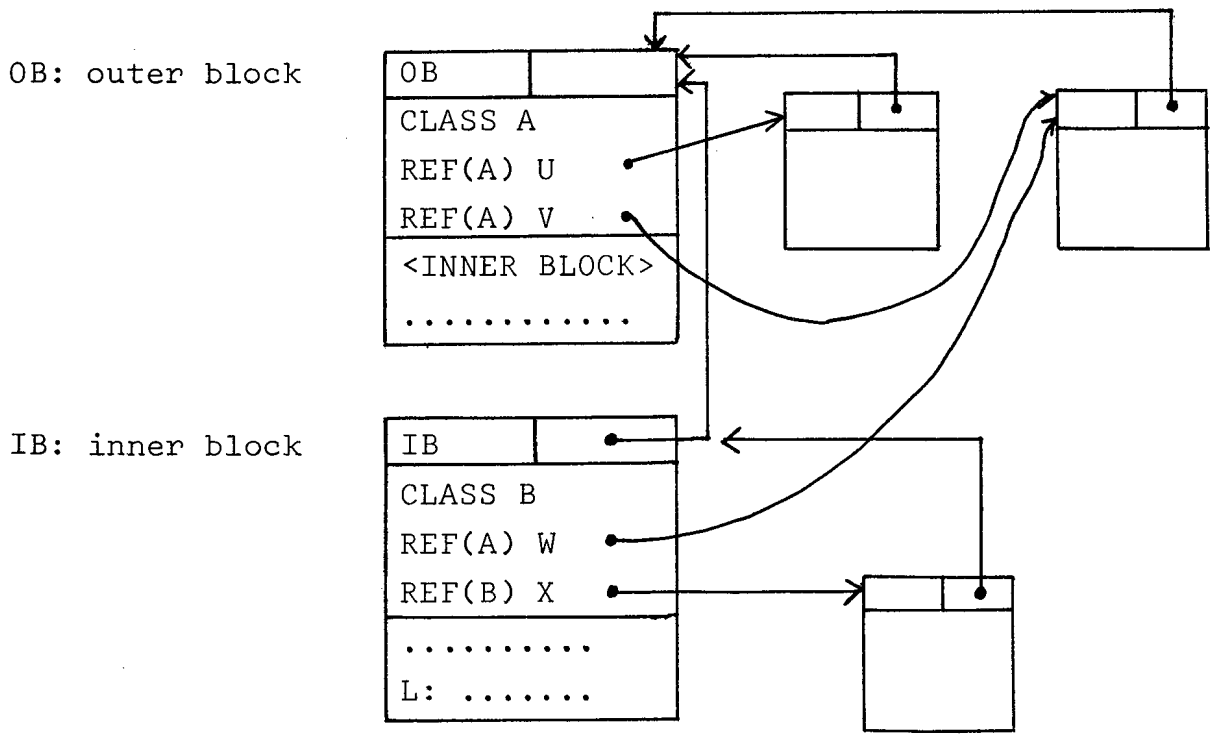
```
GOTO L1
```

then both the inner blocks will be deleted when that statement is executed and both BOOLEAN B and TEXT X will no longer exist.

The CLASS concept initially gives the opportunity for data structures to exist in parallel. The program

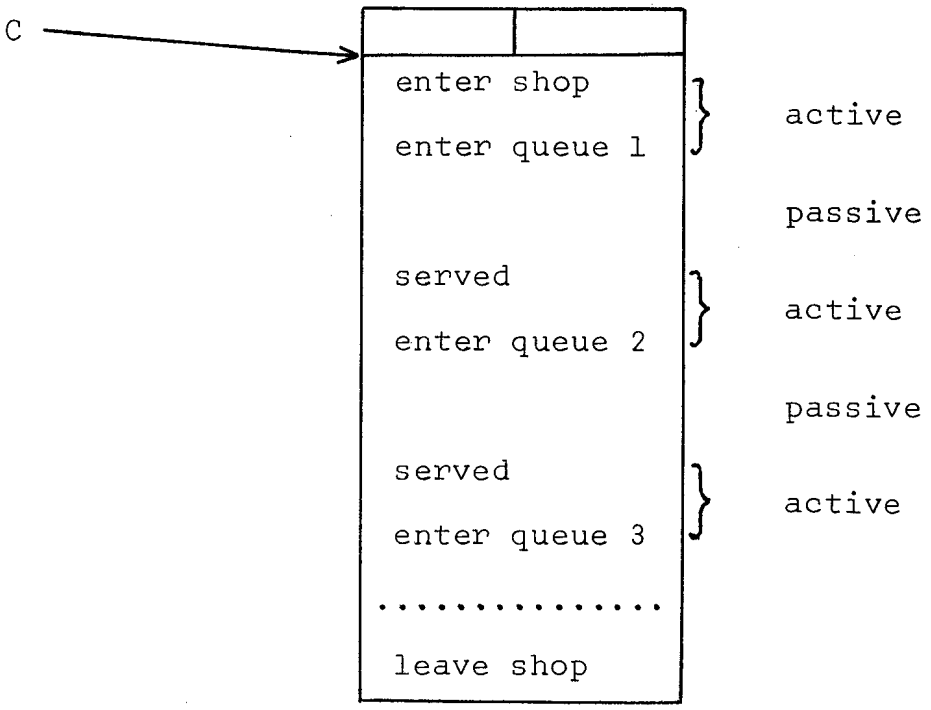
```
BEGIN  CLASS A;.........;
       REF(A)U,V;
       U :- NEW A;
       BEGIN  CLASS B;.........;
              REF(B)X;
              REF(A)W;
              V :- W :- NEW A;
              X :- NEW B;
       L:     .........
       END ***INNER BLOCK*** ;
       .........
END ***PROGRAM*** ;
```

has the representation below at the label L.



When the inner block is deleted, the second A object is still available as it is referenced by V. On the other hand, the B object must be deleted as its declaration and therefore all its references do not exist.

At this level, the actions of each object are executed until exhaustion and then the object is left as an attribute structure in the terminated state. But there is a broad class of problems which cannot be modelled by this mechanism, e.g. when the actions are executed in phases corresponding to the actual concept represented being active or passive. For example, a customer in a shop goes through the stages of queuing (passive) and buying (active), players in a game of cards play, and then are passive until their turn comes again. We could picture a customer object, C, by:



When C is passive its actions may be made active again by a call RESUME(C). The actions of C are resumed from where they were left off last. To mark off this program point objects can be made into program components supplied with a LOCAL SEQUENCE CONTROL (LSC) which marks the current stage of execution of their actions.

A very simple example is that of two players playing a game. Their actions may be loosely described by

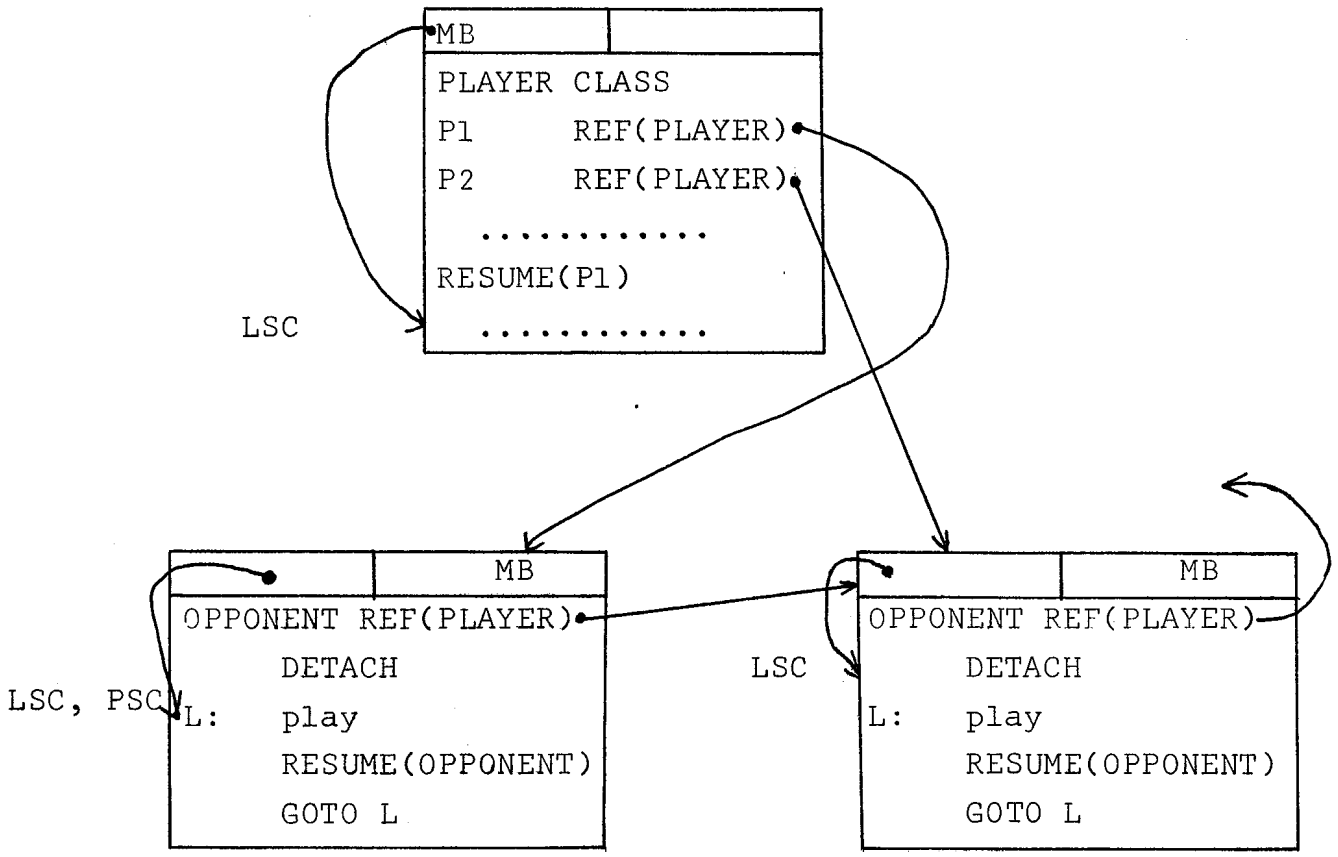
```
        .........
    L:  play;
        RESUME(opponent);
        GOTO L
```

When the first player is generated, we do not wish to execute these actions as the opponent is not yet generated. So we have to return control to the main block and return a reference to this player, and then create the second player. A second system PROCEDURE DETACH serves this purpose. On meeting DETACH, the object becomes a system component with an LSC referencing the next statement, control is returned to its object generator and with it a reference to the object.

The outline of a simple program for a two man game is:

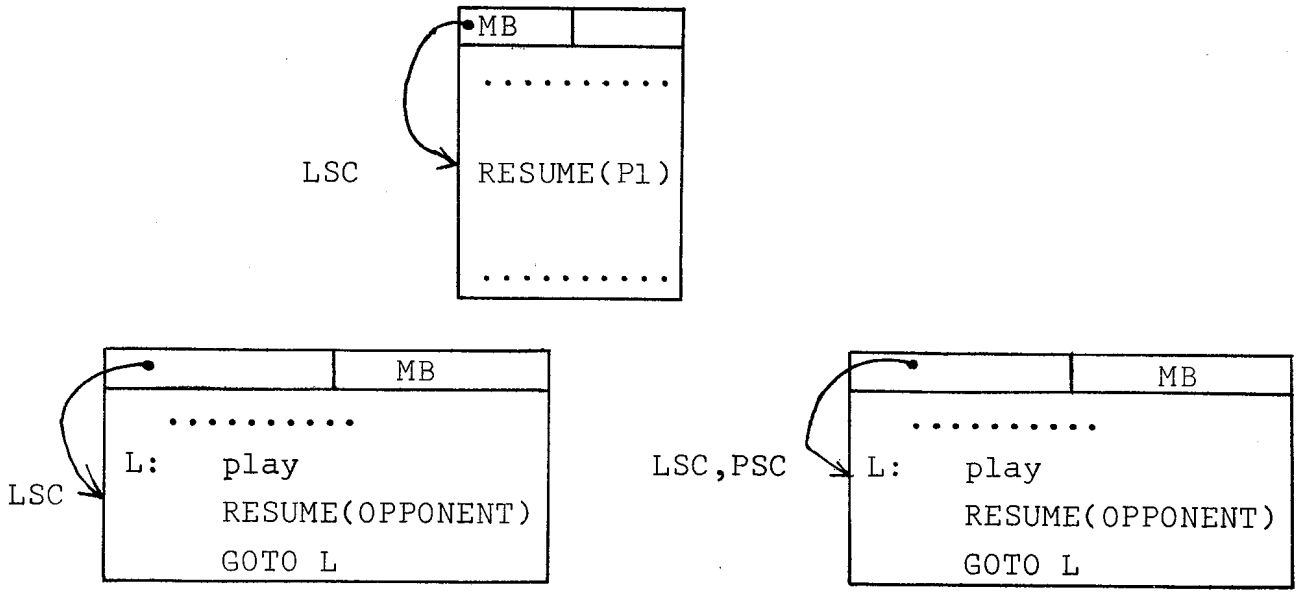
```
BEGIN  CLASS PLAYER;
       BEGIN  REF(PLAYER) OPPONENT;
              DETACH;
       L:     play;
              RESUME(OPPONENT);
              GOTO L
       END;
       REF(PLAYER)P1,P2;
       P1 :- NEW PLAYER;
       P2 :- NEW PLAYER;
       P1.OPPONENT :- P2;  P2.OPPONENT :- P1;
       RESUME(P1);
       ..........
END
```

A snapshot at the point where RESUME(P1) has just been executed is:



The PSC coincides with the LSC of the currently active component.

After P1 has played, P2 is resumed and the new snapshot is:



Note that there are three components here with LSC's: the two players and the program block. The current state of the two objects is "detached" - when they are program components with LSC's. When their actions are exhausted, they lose their LSC's and become "terminated". Control returns to the main program block and continues from its LSC. Only a "detached" object may be resumed.

One question remains: how to transfer control back to the (unreferenced) main program block without terminating an object. This is achieved by a further call on DETACH.

In addition to sequencing PROCEDURES, "DETACH" and "RESUME", there is the PROCEDURE CALL which has one reference parameter which must be a reference to a detached object.

The execution of CALL(Y) from within a block X, will "attach" the detached object Y to X and continue execution of the actions of Y.

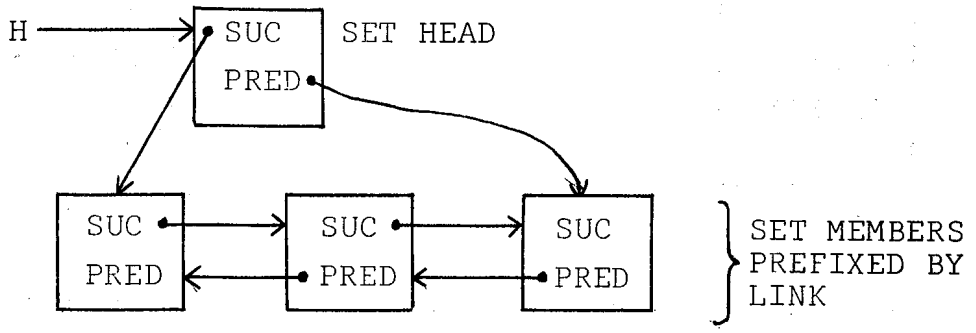
The detailed description of program sequencing given in the "67 Common Base Language" is not repeated here. Further enquiries are directed to that document §9.

2 THE SYSTEM CLASS SIMSET

List processing concepts and list manipulating procedures are declared in the system classes within the CLASS SIMSET. SIMSET can be used as a block prefix or as a prefix to another CLASS at one and only one block level in a program.

In SIMSET are provided the concepts for manipulating two-way lists called "sets". Besides the set members which carry information and are prefixed by LINK, a set also has a HEAD which has attributes giving global information about the set (e.g. how many members it has).

A set is organised on the basis of references SUC and PRED which are common to LINK and HEAD.



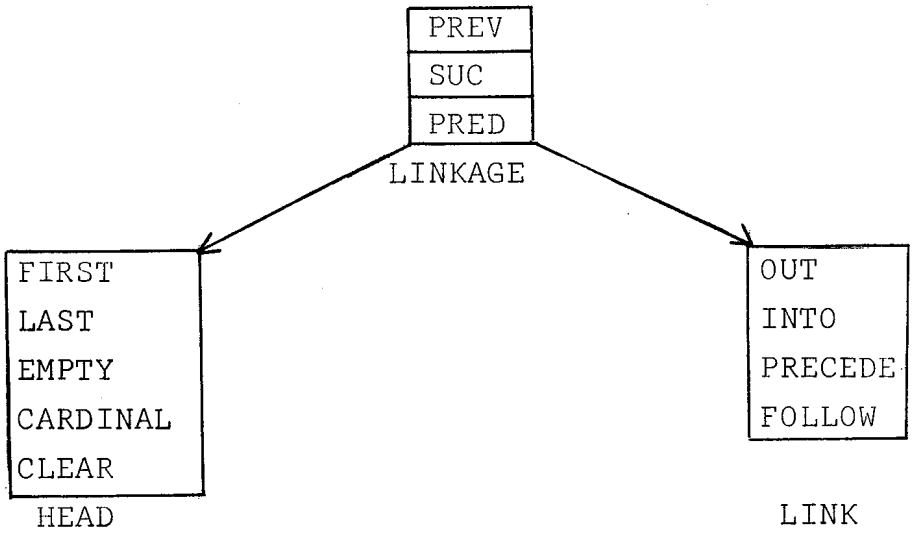
To protect the user from certain kinds of error, SUC and PRED are made REF(LINK) PROCEDURES and so may not be assigned to.

The part common to both HEAD and LINK is declared separately in CLASS LINKAGE which is then used as a prefix to HEAD and LINK.

A skeleton of the CLASS SIMSET is thus

```
CLASS SIMSET;
BEGIN  CLASS LINKAGE.....;
       LINKAGE CLASS LINK.....;
       LINKAGE CLASS HEAD.....;
END ***SIMSET***
```

This hierarchy may be pictorially represented by:



in which the procedures local to each of the classes are denoted by their identifiers.

An outline of the individual classes is now given containing the procedure-headings and a prose description of their actions. Throughout the prose descriptions which are illustrated by representative calls, we assume that the declarations

```
REF(HEAD)HD;
REF(LINK)LK;
REF(LINKAGE)LG;
```

are valid.

CLASS LINKAGE

```
CLASS LINKAGE;
BEGIN  REF(LINK) PROCEDURE SUC;.....;
       REF(LINK) PROCEDURE PRED;....;
       REF(LINKAGE) PROCEDURE PREV;.;
END ***OF LINKAGE*** ;
```

REF(LINK) PROCEDURE SUC;.....;

LK.SUC — returns a reference to the succeeding set member if LK is in a set, and LK is not the last member of the set, otherwise it returns NONE.

HD.SUC — returns a reference to the first set member if the set is not empty, otherwise NONE.

REF(LINK) PROCEDURE PRED;....;

LK.PRED — returns a reference to the preceding set member if LK is in a set and LK is not the first member of the set, otherwise NONE.

HD.PRED — returns a reference to the last set member if the set is not empty, otherwise NONE.

REF(LINKAGE) PROCEDURE PREV;....;

LK.PREV — returns NONE if LK is not in a set, a reference to the set head if LK is first member of a set, otherwise it returns a reference to LK's predecessor.

HD. PREV returns a reference to HD if HD is empty, otherwise a reference to the last member of the set with head HD.

Note: by following PREV it is possible to give a reference to the head of a set in which a LINK object is a member, as shown in the following procedure (not local to LINKAGE, LINK or HEAD).

```
REF(HEAD) PROCEDURE THESETHEADOF(LK); REF(LINK)LK;
BEGIN  REF(LINKAGE)X;
       IF LK =/= NONE THEN
       BEGIN  X :- LK.PREV;
              WHILE X IN LINK DO
                    X :- X.PREV;
              THESETHEADOF :- X;
       END;
END ***THESETHEADOF***
```

CLASS HEAD

```
LINKAGE CLASS HEAD;
BEGIN  PROCEDURE CLEAR;.............;
       REF(LINK) PROCEDURE FIRST;.....;
       REF(LINK) PROCEDURE LAST;......;
       BOOLEAN PROCEDURE EMPTY;.......;
       INTEGER PROCEDURE CARDINAL;....;
END ***OF HEAD***
```

REF(LINK) PROCEDURE FIRST;.....;
HD.FIRST — is equivalent to HD.SUC

REF(LINK) PROCEDURE LAST;.....;
HD.LAST — is equivalent to HD.PRED

BOOLEAN PROCEDURE EMPTY;.....;
HD.EMPTY — returns TRUE if HD references a set with no members, FALSE if HD references a set with one or more members.

INTEGER PROCEDURE CARDINAL;.....;
HD.CARDINAL — returns how many members the set HD contains (0 if HD is empty).

PROCEDURE CLEAR;.....;
HD.CLEAR — removes all members from the set, making it empty.

CLASS LINK

```
LINKAGE CLASS LINK;
BEGIN  PROCEDURE OUT;.........;
       PROCEDURE INTO(H); REF(HEAD)H;.........;
       PROCEDURE PRECEDE(X); REF(LINKAGE)X;....;
       PROCEDURE FOLLOW(X); REF(LINKAGE)X;.....;
END ***OF LINK***
```

PROCEDURE OUT;.....;

LK.OUT — removes LK from a set and re-establishes the SUC, PRED connections between its previously neighbouring members. If LK was not a set member, no action is taken.

PROCEDURE INTO(H); REF(HEAD)H;.....;

LK.INTO(HD) — LK.OUT is called first. If HD == NONE no action is taken. If HD =/= NONE, LK goes into the set HD as the new last member.

PROCEDURE PRECEDE(X); REF(LINKAGE)X;.....;

LK.PRECEDE(LG) — LK.OUT is called first. If LG == NONE or is not in a set then no action is taken, otherwise LK goes into the same set as LG as the new LG.PRED (LG may reference either a HEAD or a LINK object).

PROCEDURE FOLLOW(X); REF(LINKAGE)X;.....;

LK.FOLLOW(LG) — as PRECEDE except that LK becomes the new LG.SUC

The use of reference variables and the fact that an object of any CLASS inner to CLASS LINK may be inserted in a set give the following desirable features:

1) ordered sets can be manipulated by efficient standard procedures

2) both the successor and predecessor of a LINK object are immediately accessible

3) the set members can be objects of different classes.

Note that a LINK object can only be in one set at a time.

Example on the use of SIMSET

The example is concerned with the dealing of a hand of cards, 13 cards each in rotation, to a table of four players. It shows how the concepts HEAD and LINK of SIMSET may be used as a platform upon which to build more general concepts.

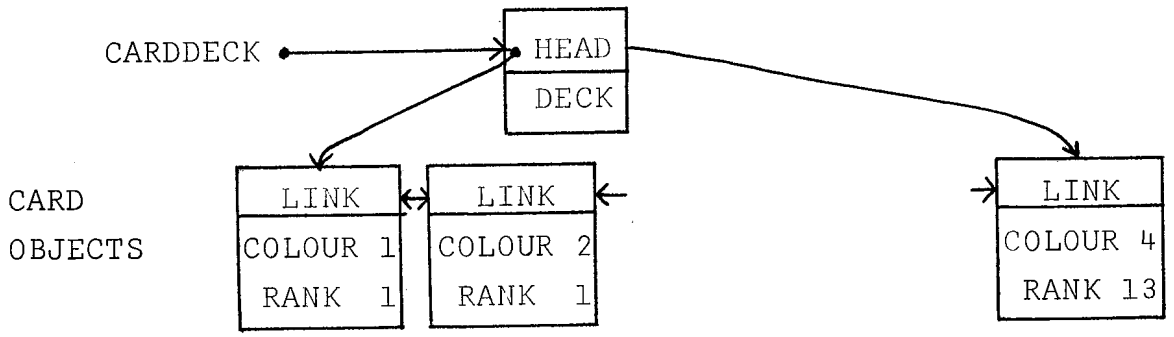
The patterns to be described are

```
CLASS CARD
CLASS DECK
CLASS HAND
```

Initially the CARD objects are created and inserted into a DECK in the order of their generation (which is non-random). Skeletons of these two CLASSES are

```
LINK CLASS CARD(COLOUR,RANK); INTEGER COLOUR, RANK;
BEGIN ..................................... END;
HEAD CLASS DECK;
BEGIN ..... END;
```

Immediately prior to the dealing, the situation may be represented by

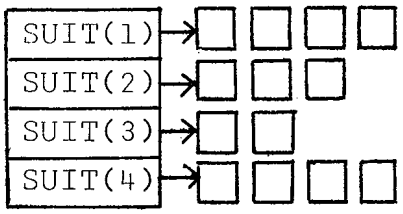


A deal consists of removing the 52 cards in random order from CARDDECK and inserting them into the HANDS in rotation.

It is convenient to describe each HAND by

```
CLASS HAND;
BEGIN  REF(HEAD) SUIT(1:4);
       generate four heads representing the four
            possible suits;
END ***HAND***
```

After generation and the dealing of the CARDS, a HAND object can be visualised by:



4 CLUBS
3 DIAMONDS
2 HEARTS
4 SPADES

Now the CARDS may be ranked in order of their attribute RANK and in the appropriate SUIT. The actions of the program are:

```
generate the deck;
deal the hands;
```

generate the deck;

These actions may be made local to CLASS DECK, which now has the outline

```
HEAD CLASS DECK;
BEGIN  INTEGER I,J;
       FOR J := 1 through 13 DO
           FOR I := 1 through 4 DO
               NEW CARD(I,J).INTO(THIS HEAD)
END ***DECK***
```

On generation of a DECK object, the representation of the CARDS is automatically generated.

deal the hands;

This part consists of randomly selecting the Nth card and placing it in the appropriate SUIT of the current PLAYER. Let J denote the index of the current PLAYER (J = 1,2,3 or 4), then the actions are:

```
J := 0;
FOR I := 52 STEP -1 UNTIL 1 DO
BEGIN  C :- the randomly selected card;
       IF J = 4 THEN J := 1 ELSE J := J + 1;
       COMMENT ***SELECT INDEX OF CURRENT PLAYER*** ;
       place C in PLAYER(J)
END;
```

There remains the tasks of writing the selection procedure, and the procedure to place the CARD C in the current HAND. These are done by PROCEDURE SELECT local to CLASS DECK and PROCEDURE PLACE local to CLASS HAND.

The final program is:

```
SIMSET
BEGIN  HEAD CLASS DECK;
       BEGIN  REF(CARD) PROCEDURE SELECT(N); INTEGER N;
              BEGIN  REF(CARD)X; INTEGER I;
                     X :- FIRST QUA CARD;
                     FOR I := 2 STEP 1 UNTIL N DO
                         X :- X.SUC;
                     SELECT :- X;
              END ***SELECT*** ;
              INTEGER I,J;
              FOR I := 1 STEP 1 UNTIL 13 DO
                  FOR J := 1 STEP 1 UNTIL 4 DO
                      NEW CARD(J,I).INTO(THIS HEAD);
       END ***DECK*** ;

       LINK CLASS CARD(COLOUR,RANK); INTEGER COLOUR, RANK;;
       COMMENT ***COLOUR = 1  REPRESENTS CLUBS
                           2  REPRESENTS DIAMONDS
                           3  REPRESENTS HEARTS
                           4  REPRESENTS SPADES
                    RANK = 1  REPRESENTS ACE
                         2-10  OBVIOUS
                           11  JACK
                           12  QUEEN
                           13  KING*** ;
```

```
CLASS HAND;
BEGIN  PROCEDURE PLACE(C); REF(CARD)C;
       BEGIN  REF(HEAD)S; REF(CARD)X;
              S :- SUIT(C.COLOUR);
              IF ¬ S.EMPTY THEN
              BEGIN  X :- S.FIRST QUA CARD;
                     WHILE X =/= NONE DO
                     BEGIN  IF X.RANK > C.RANK THEN
                                BEGIN  C.PRECEDE(X);
                                       GOTO L;
                                END;
                     END;
              END;
              COMMENT ***WE ENTER HERE IF S IS EMPTY OR
                         IF C.RANK IS THE HIGHEST MET
                         SO FAR*** ;
              C.INTO(S);
   L:  END ***PLACE*** ;

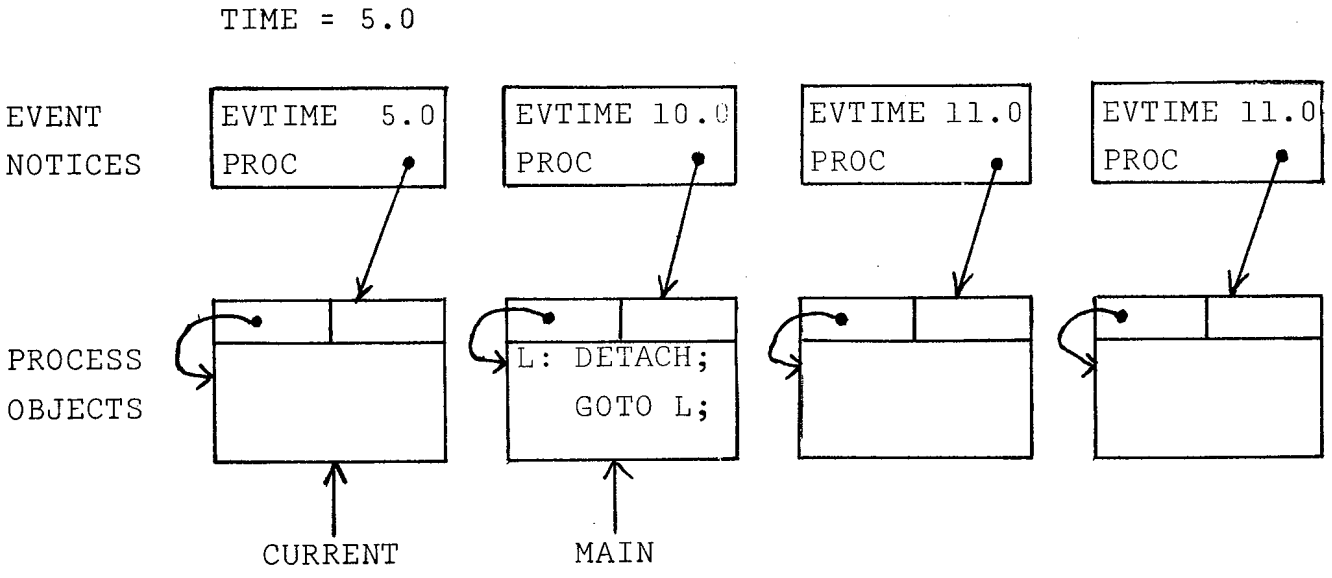
       REF(HEAD) ARRAY SUIT(1:4);
       SUIT(1) :- NEW HEAD;
       SUIT(2) :- NEW HEAD;
       SUIT(3) :- NEW HEAD;
       SUIT(4) :- NEW HEAD;
END ***HAND*** ;
```

```
        REF(CARD)C;
        REF(DECK)CARDDECK;
        INTEGER I,J,U;
        REF(HAND) ARRAY PLAYER(1:4);
        U := ININT; COMMENT ***INPUT THE RANDOM STREAM BASE*** ;
        CARDDECK :- NEW DECK;
        COMMENT ***GENERATES THE WHOLE PACK OF 52 CARDS IN
                        NON-RANDOM ORDER*** ;
        FOR I := 1 STEP 1 UNTIL 4 DO
            PLAYER(I) :- NEW HAND;
        FOR I := 52 STEP -1 UNTIL 1 DO
        BEGIN  C :- CARDDECK.SELECT(RANDINT(1,I,U));
               IF J = 4 THEN J := 1 ELSE J := J + 1;
               PLAYER(J).PLACE(C);
        END;
END ***PROGRAM***
```

3 THE SYSTEM CLASS SIMULATION

The CLASS SIMULATION is prefixed by SIMSET and provides, in addition to SIMSET's set concepts, the notions of a time axis and processes (entities which interact over a period of time).

The time axis consists of a set of event notices which have two attributes, a reference to the PROCESS they represent and the time of its next scheduled event. The event notices are ranked according to the values of the time variable (EVTIME). We can picture the time axis with four scheduled events by:



The first PROCESS represented in the time axis is always referenced by CURRENT and the system time is the value of its scheduled next event (here 5.0). An object of any class prefixed by PROCESS may take an active and passive part in a simulation. The organisation is so framed that the PSC lies within CURRENT and its

actions are executed. When the active phase is over, that PROCESS may be rescheduled for a later active phase (for example, by REACTIVATE or HOLD) or removed from the timing tree (by PASSIVATE or WAIT). It is apparent that RESUME is too primitive for this purpose as it involves rescheduling or removing EVENT NOTICES as well as switching the PSC from one PROCESS object to another.

However RESUME and DETACH do form the basis for the scheduling procedures. To prevent the user from destroying system security, event notices may not be explicitly referenced by the user - he must use the system procedures for scheduling or rescheduling. In addition, it is strongly recommended that explicit use of "DETACH", "RESUME" and "ATTACH" be avoided within a SIMULATION block.

There is one special PROCESS object which plays a key role in any SIMULATION - one referenced by MAIN. Whenever MAIN becomes CURRENT, it causes the actions of the SIMULATION block itself to be continued. The corresponding event notice can then be rescheduled (typically by a call on HOLD) and then the action switches from the SIMULATION block to the new CURRENT. Thus the SIMULATION block is itself treated as a program component during the SIMULATION.

The class outline is

```
SIMSET CLASS SIMULATION;
BEGIN  LINK CLASS PROCESS.....;
       REF(PROCESS) PROCEDURE CURRENT;.....;
       REAL PROCEDURE TIME;.....;
       COMMENT ***SCHEDULING PROCEDURES*** ;
       PROCEDURE HOLD;.......;
       PROCEDURE PASSIVATE;..;
       PROCEDURE WAIT;.......;
       PROCEDURE CANCEL;.....;
       PROCEDURE ACTIVATE;...;
       PROCEDURE ACCUM;......;
       REF("the main program")MAIN;
       COMMENT ***HERE FOLLOW ACTIONS WHICH SET UP THE
                  TIME AXIS AT TIME ZERO*** ;
END ***SIMULATION***
```

We now give a prose discussion of the attributes of CLASS SIMULATION:

CLASS PROCESS

```
LINK CLASS PROCESS;
BEGIN  BOOLEAN PROCEDURE IDLE;.........;
       BOOLEAN PROCEDURE TERMINATED;....;
       REAL PROCEDURE EVTIME;..........;
       REF(PROCESS) PROCEDURE NEXTEV;...;
       DETACH;
       INNER;
       PASSIVATE;
END ***PROCESS***
```

An object of a class inner to CLASS PROCESS is a PROCESS object. A PROCESS object has the properties of CLASS LINK and can be manipulated by sequencing statements. Sequencing statements are used to insert or delete a PROCESS object from the time axis. The state of a PROCESS object after generation is "detached" and its LSC is positioned to the first statement of the user defined operations rule. When the actions of the user defined subclass are exhausted, the unfinished actions of the PROCESS level are continued (following the INNER). These remove the object from the time axis (PASSIVATE) and its state becomes "terminated".

If a PROCESS object is not represented in the time axis, then it is terminated or passive (its actions are not yet exhausted) and the BOOLEAN PROCEDURE IDLE returns TRUE. All PROCESS objects represented in the time axis are said to be suspended except for the first (CURRENT) which is said to be active.

A call on EVTIME for a suspended or active PROCESS object returns the scheduled time of its next event. If the object is passive or terminated, then a call on EVTIME results in a run time error.

PROCESS object	result of procedure call		
state	IDLE	TERMINATED	EVTIME
active	FALSE	FALSE	time of current event
suspended	FALSE	FALSE	time of next event
passive	TRUE	FALSE	run time error
terminated	TRUE	TRUE	run time error

REF(PROCESS) PROCEDURE CURRENT;.....;

returns a reference to the currently active PROCESS object. There is one special PROCESS object in the system referenced by MAIN. Every time MAIN becomes CURRENT it causes the actions of the SIMULATION block to be resumed.

REAL PROCEDURE TIME;.....;

always returns the current value of the system time.

PROCEDURE HOLD(T); REAL T;.....;

HOLD(N) reschedules CURRENT so that its next active phase will occur at TIME + N. If the value of the actual parameter N is negative, the call is equivalent to HOLD(0). After the rescheduling, the actions of CURRENT are resumed.

Notice that HOLD(T) can be called from the user defined SIMULATION block in which case MAIN will be rescheduled - i.e. the actions of the program block are suspended for N time units.

PROCEDURE PASSIVATE;.....;

removes CURRENT from the time axis and resumes the actions of the new CURRENT. A run time error will occur if the time axis is now empty.

PROCEDURE WAIT(S); REF(HEAD)S;.....;

```
PROCEDURE WAIT(S); REF(HEAD)S;
BEGIN  CURRENT.INTO(S);
       PASSIVATE
END ***WAIT***
```

WAIT includes the currently active PROCESS object (this could be MAIN) into a referenced set, and then calls PASSIVATE.

PROCEDURE CANCEL(X); REF(PROCESS)X;.....;

CANCEL(P) where P is a reference to a PROCESS object will delete the corresponding event notice if any. If P is currently active or suspended, it thus becomes passive. If P is a reference to a passive or terminated PROCESS object or NONE, CANCEL(P) has no effect. Thus CANCEL(CURRENT) is equivalent to PASSIVATE.

PROCEDURE ACTIVATE

For user convenience, calls on the procedure ACTIVATE are written in terms of the corresponding activation-statements.

activation-statement

{ ACTIVATE / REACTIVATE } PROCESS-expression1 [{AT|DELAY time}PRIOR / {BEFORE|AFTER}PROCESS-expression2]

Let X be the value of PROCESS-expression1. If the activator ACTIVATE is used, then the activation-statement will have no effect (other than evaluating X) unless X is passive. If the activator REACTIVATE is used, then X may be active, suspended, or passive (in which latter case, the activation-statement acts as an ACTIVATE statement).

The type of scheduling is determined by the scheduling clause.

Direct activation

$$\left\{\begin{array}{l}\text{ACTIVATE X}\\ \text{REACTIVATE X}\end{array}\right\}$$

X becomes the new CURRENT and the system time is unchanged. The formerly active PROCESS object from where the call was made becomes suspended.

Timing clause

$$\left\{\begin{array}{l}\text{ACTIVATE}\\ \\ \text{REACTIVATE}\end{array}\right\} \text{X} \left\{\begin{array}{l}\text{AT}\\ \\ \text{DELAY}\end{array}\right\} \text{T} \quad \left[\text{PRIOR}\right]$$

The timing clause AT specifies the system time of the scheduled active phase. The clause

DELAY T

is equivalent to

AT current-system-time + T

The corresponding EVENT NOTICE is inserted according to the specified time, normally after any EVENT NOTICE with the same system time; the symbol PRIOR may be used to specify insertion in front of any EVENT NOTICE with the same system time.

Default actions

"AT T", when T < the current-system-time, is equivalent to "AT current-system-time".

DELAY T when T < 0 is equivalent to DELAY 0.

Relative activation

$$\left\{\begin{matrix}\text{ACTIVATE}\\ \text{REACTIVATE}\end{matrix}\right\} \text{X} \quad \left\{\begin{matrix}\text{BEFORE}\\ \text{AFTER}\end{matrix}\right\} \text{Y}$$

If Y is a reference to an active or suspended PROCESS object, then the clause BEFORE Y or AFTER Y is used to insert an event notice for X before or after that of Y and at the same system time.

Default actions

If Y is neither active nor suspended, then the activation-statement is equivalent to

```
CANCEL(X).
```

If X == Y, then the activation-statement is equivalent to

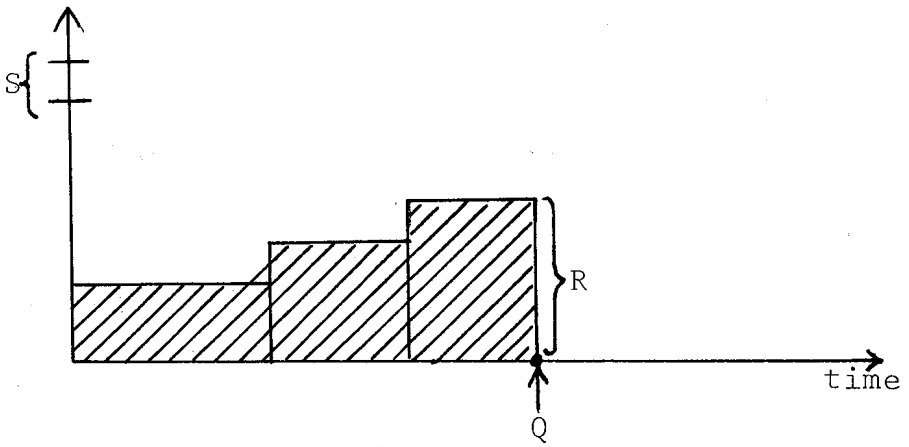
```
CANCEL(X).
```

PROCEDURE ACCUM

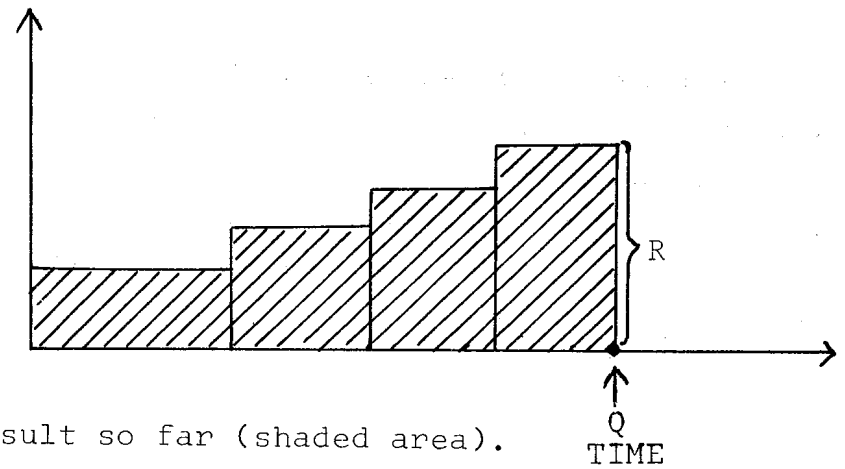
```
PROCEDURE ACCUM(A,B,C,D); NAME A,B,C; REAL A,B,C,D;
BEGIN  A := A + C*(TIME-B);
       B := TIME;
       C := C + D
END ***ACCUM***
```

ACCUM(P,Q,R,S) is used to accumulate the "system time integral" of the variable R. The accumulation takes place in P. The integral is interpreted as a step function of the system time with Q holding the system time when P was last updated. The value of S is the current increment of the step function.

before



after a call on ACCUM(P,Q,R,S) we have

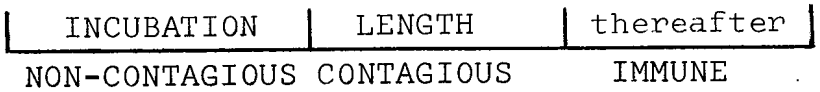


P contains result so far (shaded area).

Example on the use of SIMULATION

The program is a description of a simple epidemic model. A contagious, non-lethal disease is spreading through a POPULATION of a fixed size. Certain countermeasures are taken by a public health organisation. Each individual infection has a given INCUBATION period, during which the subject is noncontagious and has no SYMPTOMS, followed by a contagious period of a given LENGTH.

COURSE OF INFECTION IN DAYS



Each DAY of the contagious period the subject may seek treatment from the public health organisation and get cured. The probability of his seeking treatment is stored in REAL ARRAY PROBTREAT (1:LENGTH). Each person has an expected number of CONTACTS per day. At one such contact the probability of infecting a previously uninfected person is PRINF.

Once cured a person becomes immune. If untreated, the infection ceases after the given period and the person becomes immune.

SICKP (sick persons) appear as PROCESSes in the system. When CURED, or when the disease has run its course, they leave the system. The very first infection is generated by the main program (user block). A person infected by another person is included as a member in a set belonging to the infector (his environment (ENV)). A person can be a member of at most one set.

As people are cured, they are removed from these sets which gradually split up into smaller sets. The latter grow independently, disintegrate further, and so on. As the number of UNINFECTED people decreases, the growth of the contagion slows down until it finally dies out.

The public countermeasures are represented by TREATMENTs which are also PROCESSes. A patient is removed from the environment set to which he belongs, if any. If he has visible symptoms, he is cured. In addition his environment is searched and each member is subjected to a full treatment which may cause other environments to be searched etc. A patient displaying no symptoms is given a mass treatment which has a probability PROBMASS of success. His environment is not searched. In the present model, treatments act instantaneously. The simulation ends after SIMPERIOD units of time.

An outline of the problem description is

```
SIMULATION
BEGIN  PROCESS CLASS SICKP;
       BEGIN  REF(HEAD)ENV;
              PROCEDURE INFECT;
              ......
       END ***SICK PERSON*** ;
       PROCESS CLASS TREATMENT(PATIENT);
                     REF(SICKP)PATIENT;
       BEGIN ..................... END;
       ..........
END ***SIMULATION BLOCK*** ;
```

We may now outline the class actions

```
PROCESS CLASS SICKP;
BEGIN  PROCEDURE INFECT;......;
       INTEGER DAY; BOOLEAN SYMPTOMS;
       REF(HEAD)ENV;
       COMMENT ***wait incubation days for symptoms to
                  appear*** ;
       HOLD(incubation period);
       COMMENT ***now the symptoms are apparent.  If
                  "treatment today" is TRUE then a cure
                  is sought which also implies that the
                  environment of this SICK PERSON is
                  examined.  Also a number of contacts
                  are infected*** ;
       SYMPTOMS := TRUE;
       FOR DAY := 1 through LENGTH DO
       BEGIN
            IF treatment today THEN ACTIVATE NEW TREATMENT;
            INFECT(todays contacts);
            HOLD(1);
       END;
END ***SICK PERSON*** ;
```

```
PROCEDURE INFECT(N); INTEGER N;
 BEGIN  INTEGER I;
        COMMENT ***N gives the number of contacts who can
                  be infected.  N random drawings are made
                  to see if the N contacts are to be in-
                  fected.  If so a NEW SICKP is generated
                  and included in this SICKPs ENV and
                  activated*** ;
        FOR I := 1 through N DO
            IF Ith contact is infected THEN
            BEGIN  NEW SICKP.INTO(ENV);
                   ACTIVATE ENV.LAST;
            END;
 END ***INFECT***
```

```
PROCESS CLASS TREATMENT(PATIENT); REF(SICKP)PATIENT;
BEGIN  REF(SICKP)X;
       INSPECT PATIENT WHEN SICKP DO
       BEGIN  OUT;
              IF SYMPTOMS THEN
              BEGIN  CANCEL(PATIENT);
                     FOR X :- ENV.FIRST WHILE X =/= NONE DO
                         ACTIVATE NEW TREATMENT(X);
              END ELSE IF probmass successful
                          THEN CANCEL(PATIENT)
       END;
END ***TREATMENT***
```

Explanation:

A treatment tests the SYMPTOMS attribute of its parameter PATIENT. If TRUE then the instantaneous successful treatment is given. The patient is removed from the set he is in (OUT) and becomes passive for the rest of the simulation. In addition his environment is searched and a new treatment is activated for each member. If there are no symptoms, the patient is given a cheap pill which has a probability of being successful. If successful the patient is instantaneously cured and takes no further part in the simulation, but his environment is not searched.

A complete description of the program now follows:

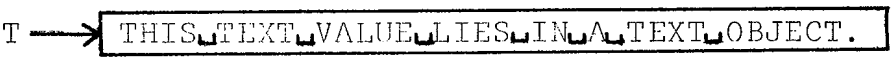
```
BEGIN  INTEGER POPULATION, LENGTH, CONTACTS, INCUBATION,
               U1, U2, U3, U4;
       REAL PRINF, PROBMASS, SIMPERIOD;
       COMMENT ***THE RANDOM STREAM NUMBERS ARE READ IN*** ;
       U1 := ININT; U2 := ININT;
       U3 := ININT; U4 := ININT;
       POPULATION := ININT;
       INCUBATION := ININT; LENGTH := ININT;
       CONTACTS := ININT;
       SIMPERIOD := INREAL;
       PRINF := INREAL; PROBMASS := INREAL;
SIMULATION BEGIN  REAL ARRAY PROBTREAT(1:LENGTH);
                  PROCESS CLASS SICKP;
                  BEGIN  INTEGER DAY;
                         BOOLEAN SYMPTOMS;
                         REF(HEAD)ENV;
                         PROCEDURE INFECT(N); INTEGER N;
                         BEGIN  INTEGER I;
                                FOR J := 1 STEP 1 UNTIL N DO
                                    IF DRAW(PRINF*UNINFECTED/
                                            POPULATION,U3) THEN
                                    BEGIN  NEW SICKP.INTO(ENV);
                                           ACTIVATE ENV.LAST;
                                    END;
                         END ***INFECT*** ;
```

```
                IF UNINFECTED > 0 THEN
                   UNINFECTED := UNINFECTED-1;
                ENV :- NEW HEAD;
                COMMENT ***NO SYMPTOMS APPEAR UNTIL
                          AFTER INCUBATION DAYS*** ;
                HOLD(INCUBATION);
                COMMENT ***NOW SYMPTOMS APPEAR AND THIS
                           SICK PERSON MAY SEEK A CURE
                           AND INFECT OTHERS EACH DAY*** ;
                FOR DAY := 1 STEP 1 UNTIL LENGTH DO
                BEGIN  IF DRAW(PROBTREAT(DAY),U1) THEN
                          ACTIVATE NEW TREATMENT(CURRENT);
                       INFECT(POISSON(CONTACTS,U2));
                       HOLD(1)
                END;
            END ***SICK PERSON*** ;
            PROCESS CLASS TREATMENT(PATIENT);
                     REF(SICKP)PATIENT;
            BEGIN  REF(SICKP)X;
                   INSPECT PATIENT WHEN SICKP DO
                   BEGIN  OUT;
                          IF SYMPTOMS THEN
                          BEGIN  CANCEL(PATIENT);
                                 FOR X :- ENV.FIRST
                                     WHILE X =/= NONE DO
                                     ACTIVATE NEW TREATMENT(X);
                          END ELSE IF DRAW(PROBMASS,U4)
                                      THEN CANCEL(PATIENT);
                   END;
            END ***TREATMENT*** ;

            ACTIVATE NEW SICKP;
            HOLD(SIMPERIOD);
        END ***SIMULATION BLOCK*** ;
END ***PROGRAM***
```

4 TEXT HANDLING FACILITIES

The concept of TEXT is the key to SIMULA's input/output facilities. For example when a card is read in, the internal representation is held as a string of 80 CHARACTERS with a one to one correspondence between the nth column of the card and the nth CHARACTER in the string. Such a string is called a TEXT-value and it is housed in a referenced TEXT-object.



With TEXTs we have thus a combination of both reference (to the housing object) and value (the string of CHARACTERS) properties. TEXTs resemble objects (of classes) in that they possess attributes which are accessed by the normal remote accessing (dot notation) technique, but TEXTs and their attributes are wholly system defined.

Whereas a CHARACTER ARRAY is oriented towards accessing single characters at a time by direct means (subscripts), the TEXT concept is oriented towards groupings of characters and sequentially accessing these groups.

TEXT variables are declared in the usual fashion.

e.g. TEXT R,S,T

and the initial value of each of these variables is NOTEXT.

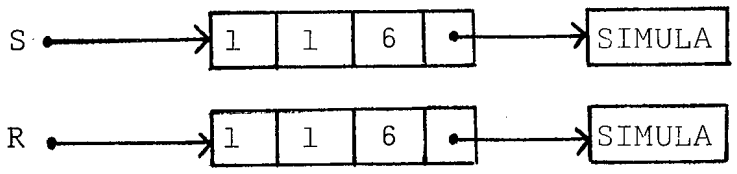
TEXT variables are capable of referencing TEXT objects which may be created by two system defined PROCEDURES - BLANKS and COPY:

```
T :- BLANKS(N)
```

creates a TEXT object of length N characters, each initialised to the blank character. After creation of the object, its reference value is assigned to T.

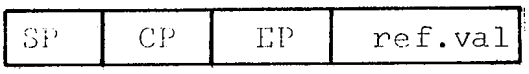
```
S :- COPY("SIMULA");
R :- COPY(S)
```

COPY will accept either a TEXT value or a TEXT reference expression as parameter, creates a TEXT object with value identical to that of the actual parameter and of the same length, and returns a reference to it. The result of the last two statements may be pictured by



magic box

Each TEXT variable has its own "magic box" which gives information about the start position (SP), end position (EP) and current position (CP) of the object it currently references. The box also contains the reference value of the object itself. The formal pattern of these magic boxes is:

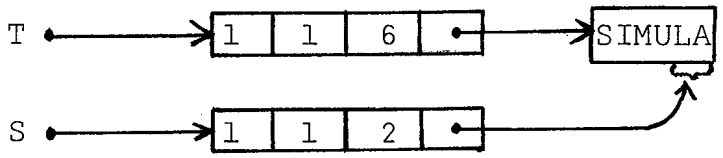


A TEXT object may be referenced in subfields by use of the procedure SUB

After T :- COPY("SIMULA")

then S :- T.SUB(5,2)

results in

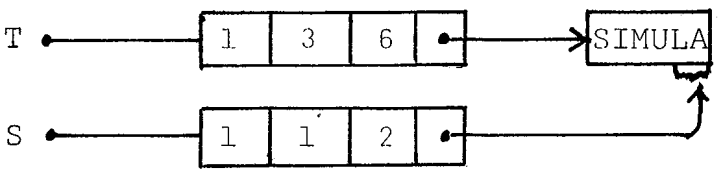


S references the subfield of T beginning from character 5 and of length 2.

The characters may be accessed one at a time by calls on GETCHAR which returns the value of the current character and increments the CP by one.

After C := T.GETCHAR;
D := T.GETCHAR

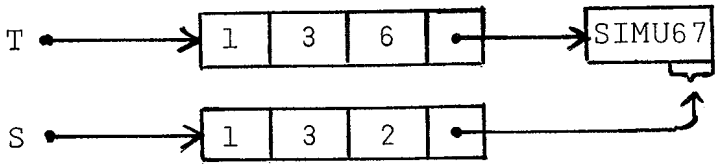
the snapshot is



and the values of C and D are 'S' and 'I' respectively.
As access was made through T, only its C has been incremented (twice). The reverse process of inserting a character value into the current position is achieved through use of PUTCHAR, which also increments the CP.

After
```
S.PUTCHAR('6');
S.PUTCHAR('7')
```

the snapshot is



Note that the value of T has been changed. The CP of S is now out of range. A further call

```
S.PUTCHAR
```
or
```
S.GETCHAR
```

will result in a run time error. To provide a check, a BOOLEAN PROCEDURE MORE is provided which returns FALSE if the CP is out of range and TRUE otherwise. Currently,

```
T.MORE = TRUE          S.MORE = FALSE
```

Other useful system defined procedures are:

LENGTH which returns the length of the currently referenced value
```
(T.LENGTH = 5
 S.LENGTH = 2)
```

POS which returns the value of the CP
```
(T.POS = 3
 S.POS = 3)
```

SETPOS which resets the CP. (To reset the CP's of T and S back to their initial character, we write
T.SETPOS(1)
S.SETPOS(1))

Text values may be transferred from one object to another by

```
S := T
```

or a value into an object by

```
S := "TEXT␣VALUE"
```

Both are left justified.

The only restriction being that the TEXT object receiving the value must be long enough to accept the value or else a run time error occurs. Any positions not directly copied into are filled with blanks.

Several editing and de-editing procedures are defined within SIMULA. These convert numbers to external form and vice versa. They are designed to operate repetitively across a field and are thus oriented towards formatted output and input.

The further detailed description of the TEXT handling facilities is given under the sub-sections

LENGTH and MAIN
subtexts
character access
text generation
TEXT assignment
TEXT editing

Throughout these subsections X, Y, Z denote TEXT references.

LENGTH and MAIN

INTEGER PROCEDURE LENGTH;

The value of X.LENGTH is the number of CHARACTERS in the TEXT object referenced by X.

```
e.g.      after X :- BLANKS(10), then X.LENGTH = 10
            if  Y == NOTEXT    , then Y.LENGTH =  0
```

TEXT PROCEDURE MAIN;

X.MAIN is a reference to the TEXT object which is or contains the text value referenced by X.

```
e.g.      after X :- BLANKS(20);
                  Y :- X.SUB(1,10);
                  Z :- NOTEXT;

           then X.MAIN == X
                Y.MAIN == X
                Z.MAIN == NOTEXT
```

The following relations hold for any TEXT reference X

```
X.MAIN.LENGTH >= X.LENGTH
X.MAIN.MAIN == X.MAIN
```

SUBTEXTS

TEXT PROCEDURE SUB(I,N); INTEGER I,N;

The call

X.SUB(J,M)

designates that part of the TEXT object referenced by X starting in CHARACTER position J and of length M characters.

e.g. after X :- COPY("MAIN␣NOT␣SUB-TEXT");
T :- X.SUB(10,8);

then T = "SUB-TEXT"

For X.SUB(J,M) to be a legal call, the subtext must be included in X. Thus

J > 0
J + M-1 <= X.LENGTH

If these conditions do not hold a run time error results.

TEXT PROCEDURE STRIP;

STRIP is used to return a reference to a subfield of a TEXT object which differs from the original in that all blanks on the right are ignored. X.STRIP is thus equivalent to X.SUB(1,N) where the remaining CHARACTERS of X (from position N+1 and of length X.LENGTH-N), if any, are all blanks.

CHARACTER access

The CHARACTERS, values housed in a TEXT object, are accessible one at a time. Any TEXT reference contains a "position indicator" which identifies the currently accessible CHARACTER of the referenced TEXT object.

The position indicator of NOTEXT is 1. A TEXT reference obtained by calling any system defined TEXT procedure has its position indicator set to 1. The position indicator of a given TEXT reference may be altered by the PROCEDURES SETPOS, GETCHAR, PUTCHAR, TEXT-reference-assignment and any editing or de-editing PROCEDURE. Position indicators are left unaltered by TEXT reference relations, TEXT value relations and TEXT value assignments.

INTEGER PROCEDURE POS;

X.POS is the current value of the position indicator of X.
The following relation is always TRUE.

```
1 <= X.POS <= X.LENGTH + 1
```

PROCEDURE SETPOS(I); INTEGER I;

The effect of X.SETPOS(M) is to assign the value of M to the position indicator of X, if 1 <= M <= X.LENGTH + 1. If M is out of this range, then the value X.LENGTH + 1 is assigned.

BOOLEAN PROCEDURE MORE;

X.MORE is TRUE if the position indicator of X is in the range 1 through X.LENGTH, otherwise the value is FALSE.

CHARACTER PROCEDURE GETCHAR;

The value of X.GETCHAR is a copy of the currently accessible CHARACTER of X provided X.MORE is TRUE. In addition, the position indicator of X is then increased by one. A run time error results if X.MORE is FALSE.

PROCEDURE PUTCHAR(C); CHARACTER C;

The effect of X.PUTCHAR(C) is to replace the currently accessible CHARACTER of X by the value of C provided that X.MORE is TRUE. In addition the position indicator of X is then increased by one. If X.MORE is FALSE, a run time error results.

Example:

The PROCEDURE COMPRESS rearranges the CHARACTERS of the TEXT object referenced by the actual parameter by collecting non-blank CHARACTERS in the leftmost part of the TEXT object and filling in the remainder, if any, with blanks. Since the parameter is called by reference (and not by name), its position indicator is unaltered.

```
PROCEDURE COMPRESS(T); TEXT T;
BEGIN  TEXT U; CHARACTER C;
       T.SETPOS(1); U :- T;
MOVELEFT:  WHILE U.MORE DO
           BEGIN  C := U.GETCHAR;
                  IF C ¬= '␣' THEN T.PUTCHAR(C);
           END;,
           COMMENT ***WE NOW FILL IN THE RIGHT WITH
                      BLANKS*** ;
           T.SUB(T.POS,T.LENGTH-T.POS+1) := NOTEXT;
END ***COMPRESS***
```

Note the use of a value assignment to T.SUB, and the use of NOTEXT on a right hand side as a neat way of filling a TEXT value to blanks.

```
After     X :- COPY("GET␣RID␣OF␣ALL␣BLANKS");
          COMPRESS(X);

then      X = "GETRIDOFALLBLANKS␣␣␣␣"
          X.STRIP = "GETRIDOFALLBLANKS"
```

TEXT generation

N.B. The PROCEDURES are non-local.

TEXT PROCEDURE BLANKS(N); INTEGER N;

The reference value is a new TEXT object of length N, filled with blank CHARACTERS.

The value of the actual parameter, M, is restricted to

$$0 <= M <= 2^{15}-20 = 32748$$

otherwise a run time error results.

TEXT PROCEDURE COPY(T); VALUE T; TEXT T;

The referenced value is a new TEXT object which is a copy of the TEXT value which is (or is referenced by) the actual parameter.

Example:

```
T :- COPY("360SIMULA");
```

is equivalent to,

```
T :- BLANKS(9);
T := "360SIMULA";
```

TEXT assignment

a) TEXT-reference-assignment

A TEXT-reference-assignment causes a copy of the TEXT-reference obtained by evaluating the right part to be assigned to the left part variable - this includes a copy of its position indicator.

```
e.g.     after     X :- COPY("ABCD");
                   X.SETPOS(3);
                   Y :- X;

          then     X.POS = 3
                   Y.POS = 3
```

In general, after

```
                   X :- P;      where P is a TEXT reference,

             then  X == P
                   X = P
                   X.POS = P.POS
```

are all TRUE.

b) TEXT-value assignment

Consider the value assignment

```
T := P;
```

let the length of T be L1, and the length of the right part be a TEXT value of length Lr. There are three cases to consider:

L1 = Lr:	the character contents of the right part TEXT are copied to the left part TEXT
L1 > Lr:	the character contents of the left part are copied into the leftmost Lr characters of the left part TEXT, whose remaining L1-Lr CHARACTERS are filled with blanks.
L1 < Lr:	a run time error results.

After
```
T :- COPY("EIGHT␣CHARS");
T := "WRONG:11";
```

then
```
T = "WRONG:11␣␣␣"
```

Note that

```
T := NOTEXT;
```

would set all the character positions of T to blanks.

In a multiple TEXT value assignment

```
T1 := T2 := ..... TN := P;
```

then

$$T_J.\text{LENGTH} >= T_{J+1}.\text{LENGTH}$$

for $J = 1,2,...,N-1$

Text editing and de-editing

TEXT editing and de-editing procedures are provided to transform binary values into field data and vice versa. The syntax for numeric-text-values (external data) follows:

numeric-text-values

{grouped-item / real-item / integer-item}

grouped-item

sign-part[[groups].]groups

groups

[digits blank]... digits

real-item

sign-part {[[digits].]digits[E sign-part digits] / E sign-part digits}

integer-item

sign-part digits

sign-part

[blank]... [+ -] [blank]...

where 'E' represents an exponent sign. This CHARACTER may be altered by the user by use of the PROCEDURE LOWTEN (see Appendix B).

A numeric-text-value is a character sequence under the above rules.

De-editing procedures

A de-editing procedure operating on a given TEXT reference X operates in the following way:

1) the longest numeric item of the given form is located, contained within X and containing the first character of X. If such error can be found, a run time error results.

2) the numeric item is interpreted as a number. If it is outside the accepted range (see PART 2, section a run time error results.

3) the position indicator of X is made one greater than the last character of the numeric item.

N.B. Unless otherwise stated, the de-editing procedures are illustrated in the context:

```
T :- COPY("1234.5+7.3&4AB");
S :- T.SUB(7,6);
R :- T.SUB(5,2);
```

INTEGER PROCEDURE GETINT;

Locates an integer-item.

```
T.GETINT = 1234
S.GETINT = 7
R.GETINT   causes a run time error
```

REAL PROCEDURE GETREAL;

locates a real-item

```
T.GETREAL = 1234.5
S.GETREAL = 73000.0
R.GETREAL = 0.5
```

INTEGER PROCEDURE GETFRAC;

Locates a grouped item. In its interpretation, any number of blanks, commas, and one decimal point are ignored and the resulting value is an INTEGER.

```
After     T := COPY(1␣␣013.42");

 then     T.GETFRAC = 101342
```

Editing procedures

Editing procedures in a given text reference X convert arithmetic values to numeric items. After an editing operation, the numeric item obtained is right adjusted in the TEXT X preceded by padding blanks. The final value of the position indicator is X.LENGTH+1.

A positive number is edited with no sign. If X == NOTEXT then a run time error results, otherwise if X is too short to contain the numeric item, an edit overflow is caused (X is filled with asterisks) and a warning message is given at the end of program execution.

```
Let     T :- BLANKS(10);
```

PROCEDURE PUTINT(I); INTEGER I;

T.PUTINT(VAL) converts the value of the parameter to an integer-item of the designated value.

```
T.PUTINT(-37)          ␣␣␣␣␣␣␣-37
T.PUTINT(118.8)        ␣␣␣␣␣␣␣119
```

PROCEDURE PUTFIX(R,N); REAL R; INTEGER N;

T.PUTFIX(VAL,M) results in an integer-item of M=0, or a real-item (with no exponent) if M>1 with M digits after the decimal point. It designates a number equal in value to VAL rounded to M decimal places. A run time error results if M<0.

```
T.PUTFIX(18,0)          ␣␣␣␣␣␣␣␣18
T.PUTFIX(-1375,4,3)     ␣-1375.400
```

PROCEDURE PUTREAL(R,N); REAL R; INTEGER N;

T.PUTREAL(VAL,M) results in a real-item to M significant places with an exponent

X.XXXXXXXXE±XX

M figures

If M<0, a run time error results
If M=0, the exponent is preceded by a sign-part
If M=1, the exponent is preceded by an integer-item of one digit.

T.PUTREAL(16,0)	E+01
T.PUTREAL(-25.32,1)	-3E+01
T.PUTREAL(-0.001472,3)	-1.47E-03

PROCEDURE PUTFRAC(I,N); INTEGER I,N;

T.PUTFRAC(VAL,M) results in a grouped-item

XXX XXX.XXX XXX

If M=0, there is no decimal point. If M>0, there are M digits after the decimal point. Each digit group consists of 3 digits except possibly the first and the last. The numeric item is an exact representation of $I*10^{-M}$.

T.PUTFRAC(10012416,3)	10␣012.416

The editing and de-editing procedures are oriented towards "fixed field" text manipulation.

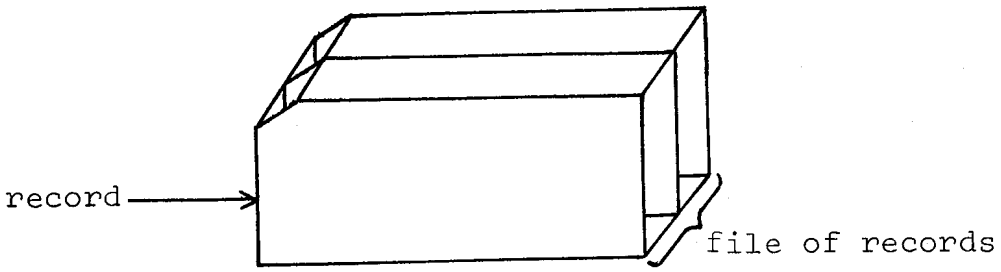
Example:

```
TEXT TR,TYPE,AMOUNT,PRICE,PAYMENT;
INTEGER PAY,TOTAL;
TR :- BLANKS(80);
      TYPE :- TR.SUB(1,5);
      AMOUNT :- TR.SUB(20,5);
      PRICE :- TR.SUB(30,6);
      PAYMENT :- TR.SUB(60,10);
.....
IF TYPE = "ORDER" THEN
BEGIN  PAY := AMOUNT.GETINT*PRICE.GETFRAC;
       TOTAL := TOTAL + PAY;
       PAYMENT.PUTFRAC(PAY,2);
END;
.....
```

5 THE SYSTEM CLASS BASICIO

Files or data sets are collections of data external to a program. They may be organised in a sequential manner (a batch of cards) or direct access manner (collection of items on a disc where each item is specified directly).

A file is composed of several records each of which is an ordered sequence of CHARACTERS.



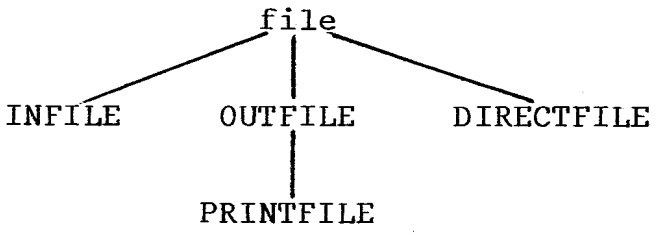
The internal representation of a record is naturally held in a TEXT object, but TEXT handling facilities alone are not enough for treating input and output to secondary storage. We need in addition

a) means for tying the external medium to the internal representation,

b) for transferring information (record-by-record) either from the external file or to the external file, and

c) either interpreting the information in the internal TEXT object in a sequential manner, or else filling the TEXT object in a sequential manner.

A SIMULA system provides system classes for these purposes. The system classes have the hierarchy



The identifier "file" is not accessible by the user - it defines the parts common to the subclasses.

The four types of defined file are:

INFILE — a sequential input file which transfers data from an external file to the program

OUTFILE — a sequential output file which transfers data from the program to an external file

PRINTFILE — (a subclass of OUTFILE) a sequential file with special extra facilities for transmitting information to a line printer

DIRECTFILE — a direct file with facilities for input and output

Each file object has a TEXT parameter called "name" - again this is NOT accessible by the user. When the file object is created, the external file associated with this file object is the file name appearing in a data set control card. The actual parameter must be a valid DDNAME of up to eight CHARACTERS.

The CLASS file has the declaration:

```
CLASS file(name); VALUE name; TEXT name;
                  VIRTUAL : PROCEDURE OPEN, CLOSE;

BEGIN  TEXT IMAGE;

       PROCEDURE SETPOS(I); INTEGER I;
            IMAGE.SETPOS(I);

       INTEGER PROCEDURE POS;
            POS := IMAGE.POS;

       BOOLEAN PROCEDURE MORE;
            MORE := IMAGE.MORE;

       INTEGER PROCEDURE LENGTH;
            LENGTH := IMAGE.LENGTH;

END ***file***
```

The variable IMAGE references a TEXT object value which acts as a buffer containing the information currently being processed.

The PROCEDURES SETPOS, POS, MORE and LENGTH defined local to file operate on the buffer IMAGE. Given a reference to X to an object belonging to a subclass of file, then it is now possible to write the more convenient

```
X.MORE          X.LENGTH          .........
```

Instead of (the still valid)

```
X.IMAGE.MORE    X.IMAGE.LENGTH    .........
```

The PROCEDURES OPEN and CLOSE, which are specified as VIRTUAL but have no matching declaration at the "file" level, complete the definition of CLASS file. The matching PROCEDURES declared in the subclasses of "file" conform to the patterns below with possible minor variations depending upon the subclass. The variations are listed in the appropriate following sub-sections.

The PROCEDURE outlines are:

```
PROCEDURE OPEN(BUF); TEXT BUF;
BEGIN  IF OPEN THEN ERROR;
       IMAGE :- BUF;
END
PROCEDURE CLOSE;
BEGIN .........
       IMAGE :- NOTEXT;
END
```

No information can be processed through a "file" object until it has not only been generated but also opened. This can only be achieved by a call on the PROCEDURE OPEN whose actual parameter is assigned to IMAGE and acts as the buffer. A call on OPEN when a "file" is already open gives a run time error.

The PROCEDURE CLOSE closes a file and releases the buffer (by the assignment IMAGE :- NOTEXT). No information may be transmitted through a closed "file" object, but it may be opened again by a further call on OPEN.

CLASS INFILE

```
file CLASS INFILE; VIRTUAL : PROCEDURE INIMAGE;
                     BOOLEAN PROCEDURE ENDFILE;

BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;.....;
       PROCEDURE CLOSE;......;
       BOOLEAN PROCEDURE ENDFILE;.........;
       CHARACTER PROCEDURE INCHAR;........;
       BOOLEAN PROCEDURE LASTITEM;........;
       INTEGER PROCEDURE ININT;...........;
       REAL PROCEDURE INREAL;.............;
       INTEGER PROCEDURE INFRAC;..........;
       TEXT PROCEDURE INTEXT(W); INTEGER W;......;
END ***INFILE***
```

PROCEDURE OPEN

conforms to the pattern listed with CLASS file but in addition positions the current position indicator to "LENGTH+1".

PROCEDURE CLOSE

conforms to the pattern listed with CLASS file.

PROCEDURE ENDFILE

returns TRUE before the INFILE is opened (by OPEN), if the end of external file marker has been met, and if the INFILE has been closed (by a call on CLOSE).

PROCEDURE INIMAGE

transfers an external file record into the TEXT IMAGE. A run time error will occur if the TEXT object referenced by IMAGE is too short to contain the record. If the record is shorter

than IMAGE, it is left adjusted and the remainder of IMAGE is filled with blanks. Finally the position indicator of IMAGE is set to 1. When the last record has been read in, and INIMAGE is called again, a call on ENDFILE will return TRUE. Any further call on INIMAGE, INCHAR, INTEXT, ININT, INREAL or INFRAC will result in a run time error.

BOOLEAN PROCEDURE LASTITEM;

returns FALSE only if the external file contains more information (non-blank CHARACTERS). It scans past all blank CHARACTERS (calling INIMAGE if need be). If LASTITEM returns FALSE then the currently accessible CHARACTER of IMAGE is the first non-blank CHARACTER. If ENDFILE returns TRUE, a call on LASTITEM also returns TRUE.

CHARACTER PROCEDURE INCHAR;

gives access to the next available CHARACTER and scans past it. If IMAGE.MORE is FALSE, the INIMAGE is called once and the value of the call is the first CHARACTER of the new image. INCHAR gives a run time error if an attempt is made to read past the last record in the file.

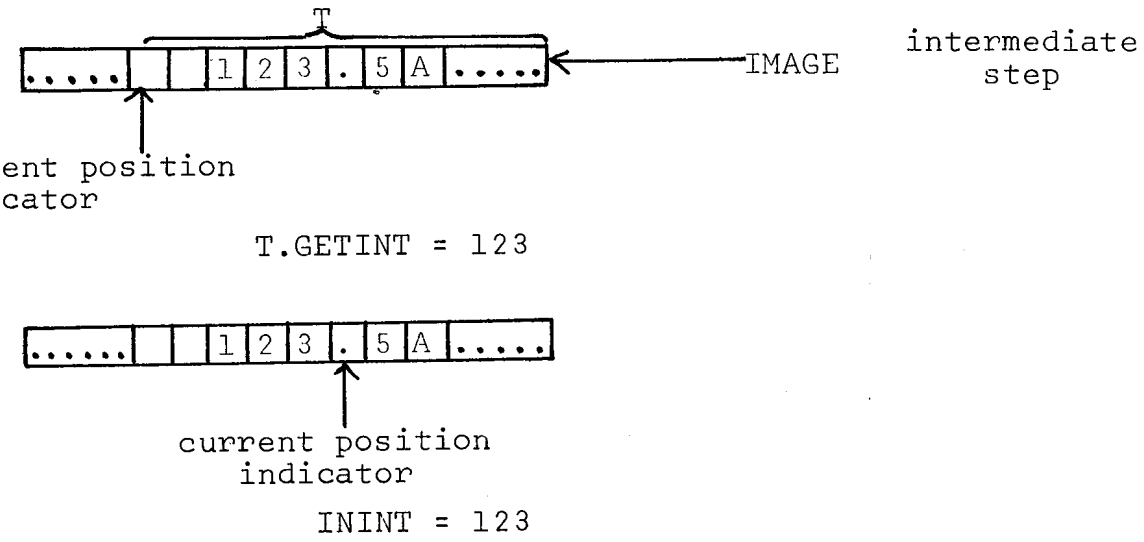
TEXT PROCEDURE INTEXT(W); INTEGER W;

INTEXT(M) creates a copy of the next M CHARACTERS (which may be spread over several records) and returns a reference to this copy. If M < 0 or M > 2^{15}-20 then a run time error results. A run time error will also result if the file does not contain M more CHARACTERS, i.e. an attempt is made to read past the last record.

The remaining PROCEDURES treat the file as a continuous stream of records. They scan past any number of blanks (calling INIMAGE if need be) and then de-edit a numeric item lying in one image. This is done by calling LASTITEM (which scans past the blanks)

and then referencing the remainder of the current IMAGE by a temporary TEXT variable, say T. The value of the "IN***"-PROCEDURE call is the value of the corresponding call on T."GET***". On exit, the current position indicator is updated to reference past the de-edited field, i.e. to reference the first CHARACTER which is not a part of the de-edited numeric item.

e.g.



Run time errors will result if the remaining CHARACTERS in the file are blanks (LASTITEM = TRUE) or if the item is not numeric.

An outline of ININT is:

```
INTEGER PROCEDURE ININT;
BEGIN  IF LASTITEM THEN ERROR;
       T :- IMAGE.SUB(POS,LENGTH-POS+1);
       ININT := T.GETINT;
       SETPOS(POS+T.POS-1);
END *** ININT***
```

INREAL and INFRAC follow the same pattern.

CLASS OUTFILE

```
file CLASS OUTFILE; VIRTUAL : PROCEDURE OUTIMAGE;
BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;....;
       PROCEDURE CLOSE;.................;
       PROCEDURE OUTIMAGE;.............;
       PROCEDURE OUTINT(I,W); INTEGER I,W;.............;
       PROCEDURE OUTFIX(R,N,W); REAL R; INTEGER N,W;....;
       PROCEDURE OUTREAL(R,N,W); REAL R; INTEGER N,W;...;
       PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;.........;
       PROCEDURE OUTTEXT(T); VALUE T; TEXT T;...........;
       PROCEDURE OUTCHAR(C); CHARACTER C;..............;
END ***OUTFILE***
```

PROCEDURE OPEN(BUF); TEXT BUF;

Follows the pattern set by the PROCEDURE OPEN listed with CLASS file.

PROCEDURE CLOSE;

Conforms to the pattern set by the PROCEDURE CLOSE listed with CLASS file but in addition checks the value of POS. If POS ¬= 1 then presumably extra information has been copied into IMAGE since the last call on OUTIMAGE.

Accordingly, if POS ¬= 1, OUTIMAGE will be called once before the OUTFILE is closed.

PROCEDURE OUTIMAGE;

OUTIMAGE transfers the contents of IMAGE to the external file creating a copy as a new record. IMAGE is then cleared to blanks and its current indicator set to one.

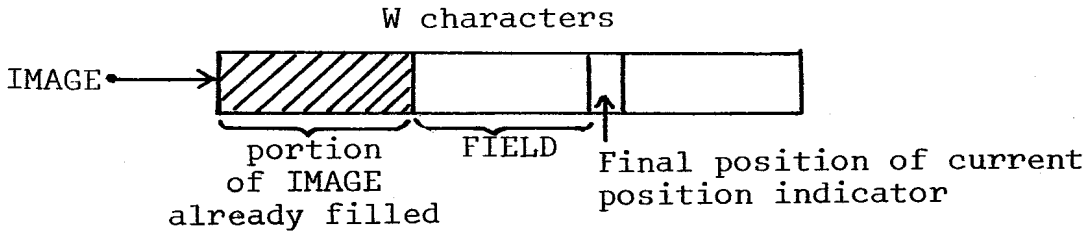
PROCEDURE OUTTEXT(T); VALUE T; TEXT T;

A copy of the CHARACTER sequence represented by the actual parameter is edited into IMAGE from the current position. If the remaining length of IMAGE is insufficient, INIMAGE is called and the editing process proceeds. Thus the TEXT value may be split over several external records.

PROCEDURE OUTCHAR(C); CHARACTER C;

Outputs the value of C into the current position of IMAGE (if MORE = FALSE, then OUTIMAGE is called first). In either case, the current position indicator is then incremented.

The remaining PROCEDURES are all based upon the PUT-PROCEDURES local to TEXTs. The corresponding PUT-PROCEDURES are augmented by an extra parameter W which specifies the field width.



The editing PROCEDURE commences by establishing a temporary TEXT reference (FIELD) to the next sequence of W CHARACTERS lying in one IMAGE. If the current IMAGE has not enough space left, INIMAGE is called. Then the value is edited by calling FIELD."PUT***" where "PUT***" is the PUT-PROCEDURE corresponding to the OUT-PROCEDURE. Finally the current position indicator is increased by W to reference past FIELD - past the just-edited field.

```
PROCEDURE OUTINT(I,W); INTEGER I,W;
     FIELD(W).OUTINT(I);

PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;
     FIELD(W).PUTFRAC(I,N);

PROCEDURE OUTREAL(P,N,W); REAL P; INTEGER N,W;
     FIELD(W).PUTREAL(P,N);

PROCEDURE OUTFIX(P,M,W); REAL P; INTEGER M,W;
     FIELD(W).PUTFIX(P,M);
```

CLASS PRINTFILE

```
OUTFILE CLASS PRINTFILE;
BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;.............;
       PROCEDURE CLOSE;.....;
       PROCEDURE LINESPERPAGE(N); INTEGER N;.......;
       INTEGER PROCEDURE LINE(N); INTEGER N;.......;
       PROCEDURE SPACING(N); INTEGER N;...........;
       PROCEDURE EJECT(N); INTEGER N;.............;
       PROCEDURE OUTIMAGE;........................;
END ***PRINTFILE***
```

CLASS PRINTFILE further orients the prefixing CLASS OUTFILE towards line printer output. The PROCEDURES OPEN and CLOSE take the pattern of those local to OUTFILE but in addition OPEN positions to the top of the next page.

PROCEDURE LINESPERPAGE(N); INTEGER N;

Initially the number of printable lines per page is fixed at some value (V) dependent upon the installation. A call LINESPERPAGE(M) will alter this figure to allow only M printable lines per page. A run time error results if M < 0 or M > V.

PROCEDURE SPACING(N); INTEGER N;

Initially the spacing is 1 and successive images are printed on successive lines. A call SPACING(M) will alter this to separate successive lines by M-1 blank lines. This becomes effective after the next call on OUTIMAGE. If M > "current value of lines per page", or M < 0, then a run time error results. If M = 0, overprinting will occur - successive images being printed on the same physical line.

PROCEDURE EJECT(N); INTEGER N;

This PROCEDURE skips to a certain line on the page - (it avoids calling OUTIMAGE several times). EJECT(L) will position to line L on this page if this is further down the current page (if L > LINE), or else skip to LINE L of the next page if L <= LINE.

A run time error occurs if L < 0. If L > LINESPERPAGE, EJECT(L) is equivalent to EJECT(1).

INTEGER PROCEDURE LINE;

This PROCEDURE returns the INTEGER value of the line number which indicates the next line to be printed. Thus EJECT(LINE+3) will skip three lines and not alter spacing. After each call on OUTIMAGE, the line number is incremented by the current spacing.

PROCEDURE OUTIMAGE;

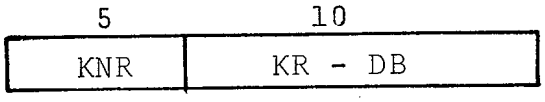
This PROCEDURE acts like the OUTIMAGE of OUTFILE but in addition increments the line number by spacing, and will position to the top of the next page if the current page is filled.

Example:

This example shows the use of three types of file and how to open and close them. The example was chosen to demonstrate these features and how formatting is available by use of the sub-text concept. The logic of the example is particularly simple. A file of transactions has been punched on cards in the format

col 1-5	customer number	KNR
col 7-16	sum of debit transactions	DB
col 19-28	sum of credit transactions	KR

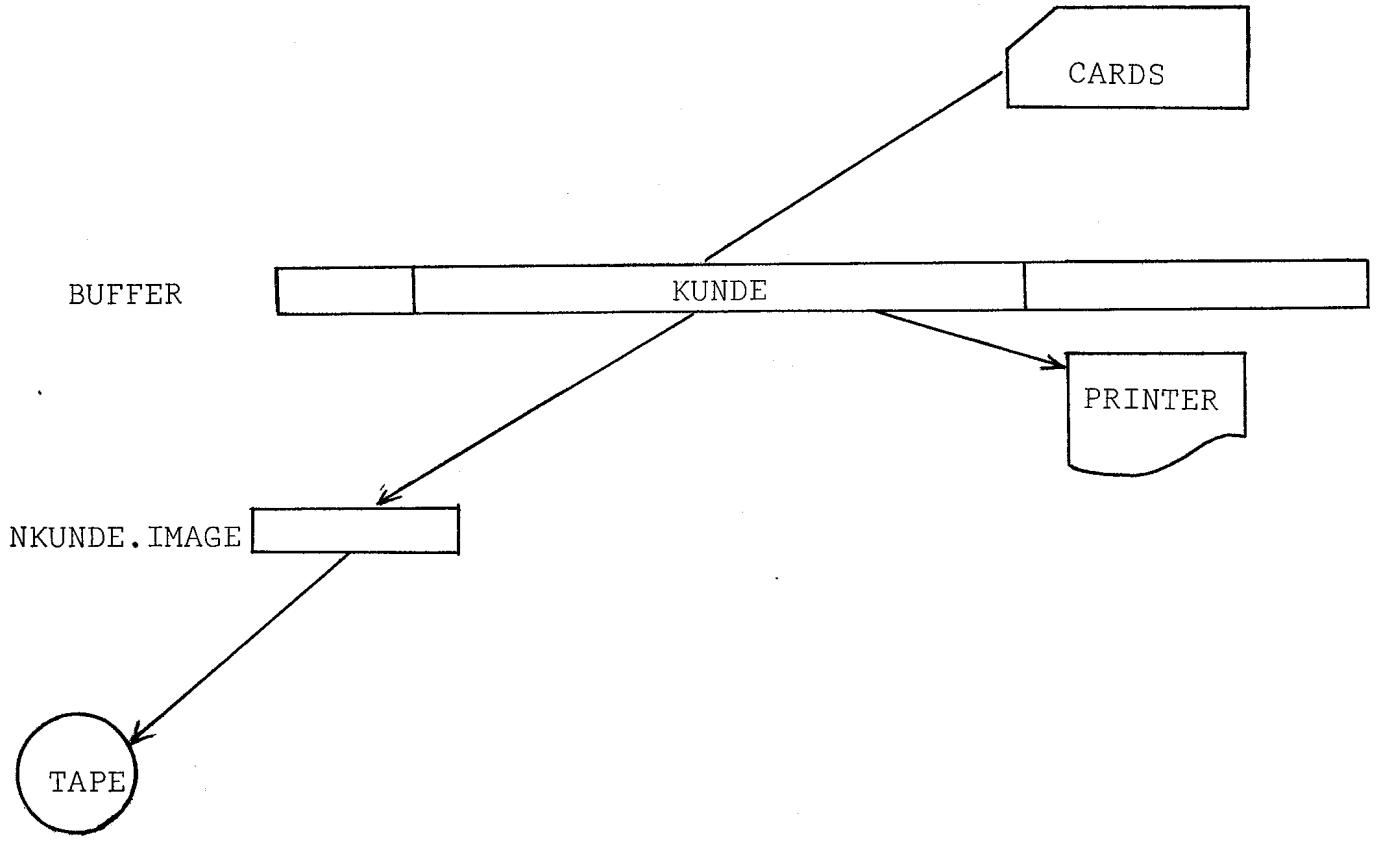
Each transaction is on a fresh card. The information is to be compressed and recorded on a new sequential file on tape. The length of each tape record is 15 CHARACTERS, and its format is:



customer number — total sum of transactions

Checks are made that the customer number is valid (1-3-7 digit check) and that the card is validly punched. If not, a copy of the card is printed on a line printer, and the scan continues.

Note that the INFILE and PRINTFILE share the same buffer.




```
//TAPE   DD   DSN=A,DISP=(NEW,CATLG),LABEL=(1,SL),UNIT=TAPE,VOL=SER=TAPE1

//PRINTER  DD  SYSOUT=A
//CARDS    DD     *

    <cards>

/*
```

```
BEGIN  TEXT KNR, DB, KR, NR, T1, T2, T3, T4, KT, TSUM,
            TS, BUFFER;
       INTEGER SUM, SALDO;
       REF(INFILE) KUNDEKORT;
       REF(PRINTFILE) PRINT;
       REF(OUTFILE) NKUNDE;

       BOOLEAN PROCEDURE NONNUMERIC(T); TEXT T;.........;
       COMMENT ***THIS PROCEDURE RETURNS TRUE IF THE TEXT
                  PARAMETER CAN NOT BE NUMERICALLY INTER-
                  PRETED FROM THE LEFT*** ;
       BUFFER :- BLANKS(132);
       BUFFER := "ERROR";
       COMMENT ***CONSTRUCT THE CARDFILE*** ;
       KUNDEKORT :- NEW INFILE("CARDS");
       KUNDEKORT.OPEN(BUFFER.SUB(10,80));
       KNR :- BUFFER.SUB(1,5);
           NR :- KNR.SUB(1,4);
              T1 :- NR.SUB(1,1);
              T2 :- NR.SUB(2,1);
              T3 :- NR.SUB(3,1);
              T4 :- NR.SUB(4,1);
           KT :- KNR.SUB(5,1);
       DB :- BUFFER.SUB(7,10);
       KR :- BUFFER.SUB(19,10);
```

```
COMMENT ***CONSTRUCT PRINTFILE*** ;
PRINT :- NEW PRINTFILE("PRINTER");
PRINT.OPEN(BUFFER);

COMMENT ***CONSTRUCT TAPEFILE*** ;
NKUNDE :- NEW OUTFILE("TAPE");
NKUNDE.OPEN(BLANKS(15));

COMMENT ***CONSTRUCT WORKING TEXT TSUM*** ;
TSUM :- BLANKS(3);
TS :- TSUM.SUB(3,1);
```

```
        INSPECT NKUNDE DO
        BEGIN
              KUNDEKORT.INIMAGE;
              WHILE ¬KUNDEKORT.ENDFILE DO
              BEGIN  IF    NONNUMERIC(KNR)  OR
                           NONNUMERIC(DB)   OR
                           NONNUMERIC(KR)
                     THEN  ERROR:  PRINT.OUTIMAGE
              ELSE   BEGIN  COMMENT ***137 DIGIT CHECK*** ;
                           SUM := 7*(T1.GETINT + T4.GETINT) +
                           3*T3.GETINT + T2.GETINT;
                           TSUM.PUTINT(SUM);
                           IF TS¬= KT THEN GOTO ERROR;

                           COMMENT ***OUTPUT TO TAPE*** ;
                           SALDO := KR.GETINT - DB.GETINT;
                           OUTTEXT(KNR);
                           OUTINT(SALDO,10);
                           OUTIMAGE;
                     END;
                    KUNDEKORT.INIMAGE;
              END;
        END ***INSPECT NKUNDE*** ;

SLUTT:  KUNDEKORT.CLOSE;
        NKUNDE.CLOSE;
        PRINT.CLOSE
END
```

CLASS DIRECTFILE

```
file CLASS DIRECTFILE;
     VIRTUAL: PROCEDURE LOCATE, ENDFILE, INIMAGE, OUTIMAGE;
BEGIN  PROCEDURE OPEN(BUF); TEXT BUF;.....;
       PROCEDURE CLOSE;.....;
       INTEGER PROCEDURE LOCATION;.....;
       PROCEDURE LOCATE(I); INTEGER I;.....;
       BOOLEAN PROCEDURE ENDFILE;.....;
       PROCEDURE INIMAGE;.....;
       CHARACTER PROCEDURE INCHAR;.....;
       BOOLEAN PROCEDURE LASTITEM;.....;
       INTEGER PROCEDURE ININT;.....;
       REAL PROCEDURE INREAL;.....;
       INTEGER PROCEDURE INFRAC;.....;
       TEXT PROCEDURE INTEXT(W); INTEGER W;.....;
       PROCEDURE OUTIMAGE;.....;
       PROCEDURE OUTCHAR(C); CHARACTER C;.....;
       PROCEDURE OUTINT(I,W); INTEGER I,W;.....;
       PROCEDURE OUTFIX(R,N,W); REAL R; INTEGER N,W;.....;
       PROCEDURE OUTREAL(R,N,W); REAL R; INTEGER M,W;.....;
       PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;.....;
       PROCEDURE OUTTEXT(T); VALUE T; TEXT T;.....;
END ***DIRECTFILE***
```

A direct file represents an external file in which individual records are addressed by indices (ordinal numbers). The index of the current record is returned by a call on LOCATION. The current record may be copied into the program by a call on INIMAGE, or overwritten by a call on OUTIMAGE. In either case, the sequentially next record is then taken as the current record. This sequential accessing may be altered at any time a call LOCATE(M) which will locate the Mth external record and make it the new current record.

PROCEDURE OPEN

conforms to the pattern of OPEN in CLASS file but in addition locates the first record.

PROCEDURE CLOSE

conforms to the pattern of CLOSE in CLASS file.

PROCEDURE ENDFILE

is FALSE if the current index locates a record in the file.

Calls on the PROCEDURES INIMAGE and OUTIMAGE will cause run time errors if ENDFILE is TRUE otherwise they conform to these of the same identifiers in INFILE and OUTFILE but in addition increment the index of the current record by one.

The remaining PROCEDURES are analogous to the corresponding PROCEDURES of INFILE and OUTFILE.

CLASS BASICIO

The system defined file facilities are grouped together in the CLASS BASICIO whose skeleton reads:

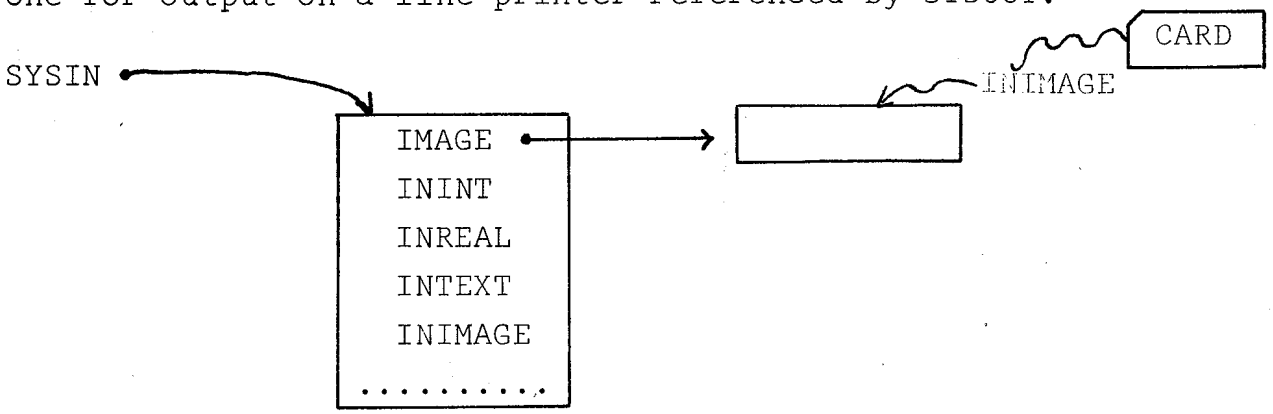
```
CLASS BASICIO(LINELENGTH); INTEGER LINELENGTH;
BEGIN  CLASS file.....................;
       file CLASS INFILE.............;
       file CLASS OUTFILE............;
       file CLASS DIRECTFILE.........;
       file CLASS PRINTFILE..........;
       REF(INFILE)sysin;
       REF(PRINTFILE)sysout;
       REF(INFILE) PROCEDURE SYSIN; SYSIN :- sysin;
       REF(PRINTFILE) PROCEDURE SYSOUT; SYSOUT :- sysout;
       sysin :- NEW INFILE("SYSIN");
       sysin.OPEN(BLANKS(80));
       sysout :- NEW OUTFILE("SYSOUT");
       sysout.OPEN(BLANKS(LINELENGTH));
       INNER;
       sysin.CLOSE; sysout.CLOSE;
END ***BASICIO***
```

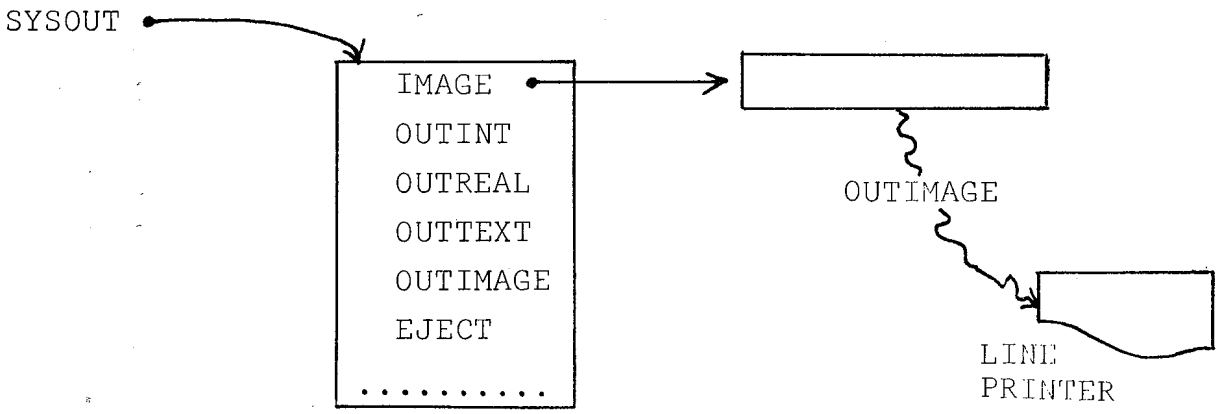
BASICIO contains actions to generate an INFILE (SYSIN for cards), and a PRINTFILE (SYSOUT for line printer). These objects are accessible only through PROCEDURES which copy the values of certain identifiers (sysin, sysout) which are otherwise not accessible by the user.

A user's program behaves as though it is enclosed as follows:

```
BASICIO(132)  BEGIN  INSPECT SYSIN DO
                        INSPECT SYSOUT DO
                          <program>
              END;
```

When a user program begins the system automatically generates two files - one INFILE for card input referenced by SYSIN, and one for output on a line printer referenced by SYSOUT.





When the actions of the user defined program are exhausted, control returns to the prefix level of the BASICIO object and continues after the INNER. The following three statements close the three system generated files.

The inspect statements enclosing the program allow the user to write ININT, INIMAGE,..... instead of SYSIN.ININT, SYSIN.IMAGE and OUTREAL, OUTIMAGE,.... instead of SYSOUT.OUTREAL, SYSOUT.OUTIMAGE. There are attribute name clashes

OPEN, CLOSE	which should never be used for SYSIN or SYSOUT
IMAGE	
SETPOS	
POS	
MORE	
LENGTH	

When these occur they are naturally bound to SYSOUT and the corresponding attributes of SYSIN may be obtained by writing SYSIN.SETPOS, SYSIN.IMAGE etc. Alternatively, an input section may be written as

```
INSPECT SYSIN DO
BEGIN
        input - in this block occurrences IMAGE, SETPOS,
        POS, MORE and LENGTH are bound to SYSIN
END;
```

APPENDIX A: HARDWARE REPRESENTATION AND COLLATING SEQUENCE FOR THE LANGUAGE CHARACTER SET

SIMULA symbol	VALUE		Punched card code
	decimal	hexadecimal	
␣	64	40	no punch
.	75	4B	12 - 3 - 8
<	76	4C	12 - 4 - 8
(	77	4D	12 - 5 - 8
+	78	4E	12 - 6 - 8
&	80	50	12
$	91	5B	11 - 3 - 8
*	92	5C	11 - 4 - 8
)	93	5D	11 - 5 - 8
;	94	5E	11 - 6 - 8
¬	95	5F	11 - 7 - 8
-	96	60	11
/	97	61	0 - 1
,	107	6B	0 - 3 - 8
_	109	6D	0 - 5 - 8
>	110	6E	0 - 6 - 8
:	122	7A	2 - 8
#	123	7B	3 - 8
'	125	7D	5 - 8
=	126	7E	6 - 8
"	127	7F	7 - 8
A	193	C1	12 - 1
B	194	C2	12 - 2
C	195	C3	12 - 3
D	196	C4	12 - 4

SIMULA symbol	VALUE		Punched card code
	decimal	hexadecimal	
E	197	C5	12 - 5
F	198	C6	12 - 6
G	199	C7	12 - 7
H	200	C8	12 - 8
I	201	C9	12 - 9
J	209	D1	11 - 1
K	210	D2	11 - 2
L	211	D3	11 - 3
M	212	D4	11 - 4
N	213	D5	11 - 5
O	214	D6	11 - 6
P	215	D7	11 - 7
Q	216	D8	11 - 8
R	217	D9	11 - 9
S	226	E1	0 - 2
T	227	E2	0 - 3
U	228	E3	0 - 4
V	229	E4	0 - 5
W	230	E5	0 - 6
X	231	E6	0 - 7
Y	232	E7	0 - 8
Z	233	E8	0 - 9
0	240	F0	0
1	241	F1	1
2	242	F2	2
3	243	F3	3
4	244	F4	4
5	245	F5	5
6	246	F6	6
7	247	F7	7
8	248	F8	8
9	249	F9	9

N.B. Characters are compared by their value (column 2 or column 3).
The initial value of CHARACTER variables is 0 (punched card code 12-0-1-8-9).
The data character '&' has punched card code 12, the data character '-' (minus) has punch card code 11.

APPENDIX B: THE SYSTEM DEFINED PROCEDURES

Calls to system defined procedures conform to the syntax of calls to declared procedures. The identifier of a standard procedure may be redefined to have another meaning at any block level. The identifier then assumes the new meaning throughout the scope of the block. Standard procedures are available to any SIMULA program.

The system defined procedures detailed below are grouped into the following sections

Arithmetic functions
CHARACTER handling
Random drawing procedures
Utility procedures

For details of other procedures:

for	TEXT handling	see Part 3, section 4
for	Sequencing procedures	see Part 3, section 1
for	Procedures local to SIMSET	see Part 3, section 2
for	Procedures local to SIMULATION	see Part 3, section 3
for	Procedures local to subclasses of CLASS file	see Part 3, section 5

A skeleton outline of all system defined procedures and classes is given as APPENDIX C.

Arithmetic functions

Certain identifiers, expressed as procedures are defined by the Simula system for standard arithmetic functions.

ABS(E)	modulus (absolute value) of E
ARCCOS(E) ARCSIN(E) ARCTAN(E)	return the principal values of the arc-cosine, arc-sine, arc-tangent of E (E is measured in radians)
COS(E) SIN(E) TAN(E)	return the cosine, sine tangent of E (E is measured in radians)
COSH(E) SINH(E) TANH(E)	return the hyperbolic cosine, hyperbolic sine, hyperbolic tangent of E (E is measured in radians)
EXP(E)	exponential function of E (e^E)
LN(E)	natural logarithm of E ($\log_2 E$, or ln E). If E <= 0, a run time error results.
SQRT(E)	returns the square root of E if E >= 0. If E < 0, a run time error results.

The above 13 functions operate on arithmetic arguments. If the type of E is [SHORT]INTEGER or REAL, then the function value is of type REAL. If the type of E is LONG REAL, then the function value is of type LONG REAL.

SIGN(E) sign of the value of E

$$= \begin{cases} 1 & \text{if } E > 0 \\ 0 & \text{if } E = 0 \\ -1 & \text{if } E < 0 \end{cases}$$

ENTIER(E) largest whole number less than or equal to E (equal to or to the left of E on the real axis).

e.g. ENTIER(5.3) = 5
ENTIER(-4.7) = -5
ENTIER(1) = 1

The above 2 functions operate upon [LONG] REAL or [SHORT] INTEGER values of E and yield values of type [SHORT] INTEGER depending upon their magnitude.

MOD(M,N) M modulo N, that is

M-ENTIER(M/N)*N

e.g. MOD (7,3) is 1
MOD (-48,5) is 2

The function operates on [SHORT]INTEGER arguments, [LONG]REAL arguments being rounded. The result is [SHORT]INTEGER.

CHARACTER handling

Two CHARACTER subsets are defined by the standard procedures:

BOOLEAN PROCEDURE DIGIT(C); CHARACTER C;

which is TRUE if C is a digit, FALSE otherwise.

BOOLEAN PROCEDURE LETTER(C); CHARACTER C;

which is TRUE if C is a capital letter, FALSE otherwise.

The collating sequence defines a one-one mapping between the INTEGERS and the internal CHARACTER representation (see APPENDIX A).

INTEGER PROCEDURE RANK(C); CHARACTER C;

returns a value in the range 0 through 255.

CHARACTER PROCEDURE CHAR(N); INTEGER N;

if the parameter value is not in the range 0 through 255, a run time error results. Otherwise the procedure returns the CHARACTER with value N.

Examples:

```
        RANK('+')  =  78
        RANK('A')  = 193
        CHAR(249)  = '9'
        CHAR(#40)  = '␣'
RANK(CHAR(127)) = 127
```

```
DIGIT(C)  = RANK(C) >= 240 AND RANK(C) <= 249
DIGIT(C)  = C >= '0' AND C <= '9'
LETTER(C)  =  (C >= 'A' AND C <= 'I')
              OR (C >= 'J' AND C <= 'R')
              OR (C >= 'S' AND C <= 'Z')
```

Random drawing procedures

The random drawing procedures produce, in successive calls, a stream of random numbers taken from a specified distribution. As a side effect, the procedures update the stream variable U (which must always be an INTEGER variable) thus advancing the specified stream by one step.

BOOLEAN PROCEDURE DRAW(A,U); NAME U; REAL A; INTEGER U;

If 0 < A < 1, the value is TRUE with probability A, FALSE with probability 1-A.

If A <= 0, the value is always FALSE.

If A >= 1, the value is always TRUE.

INTEGER PROCEDURE RANDINT(A,B,U); NAME U; INTEGER A,B,U;

If A <= B, the value is one of the INTEGERS

A, A+1, A+2, ..., B-1, B

with equal probability.

If A > B, a run time error results.

REAL PROCEDURE UNIFORM(A,B,U); NAME U; REAL A,B; INTEGER U;

If A < B, the value is uniformly distributed between A and B.

If A >= B, a run time error results.

REAL PROCEDURE NORMAL(A,B,U); NAME U; REAL A,B; INTEGER U;

The value is normally distributed with mean "A" and standard deviation "B". An approximation function is used for the normal distribution.

REAL PROCEDURE NEGEXP(A,U); NAME U; REAL A; INTEGER U;

If A > 0, the value is drawn from a negative exponential distribution with mean "1/A". This is the same as a random waiting time in a Poisson distributed arrival pattern with an expected number of arrivals per time unit equal to "A".

If A <= 0, a run time error results.

INTEGER PROCEDURE POISSON(A,U); NAME U; REAL A; INTEGER U;

The value is a drawing from the Poisson distribution with parameter "A".

If A > 20.0, the value is approximated by

ENTIER(0.5 + NORMAL(A,SQRT(A),U)).

If A < 0, the value is zero.

REAL PROCEDURE ERLANG(A,B,U); NAME U; REAL A,B; INTEGER U;

If A <= 0 or B <= 0, a run time error results.

If A > 0 and B > 0, then the value is a drawing from the Erlang distribution with mean "1/A" and standard deviation "1/(A*SQRT(B)).

INTEGER PROCEDURE DISCRETE(A,U); NAME U; REAL ARRAY A; INTEGER U;

The one dimensional REAL ARRAY A, augmented by a unit element one to the right, is interpreted as a step function of the subscript, defining a discrete (cumulative) distribution function. The function value is an INTEGER in the range

"A.lower bound" through "A.upper bound+1"

It is defined as the smallest I such that

A(I) > U

where U is a basic drawing and A(upper bound+1) = 1.0.

REAL PROCEDURE LINEAR(A,B,U); NAME U; REAL ARRAY A,B; INTEGER U;

The value is a drawing from a cumulative distribution function F, which is obtained by linear interpolation in a non-equidistant table defined by A and B, such that

A(I) = F(B(I))

It is assumed that A and B are one dimensional REAL ARRAYS with the same bounds, that the first and last elements of A are equal to 0.0 and 1.0 respectively, and that

$$\left.\begin{array}{l} A(I) >= A(J) \\ B(I) >= B(J) \end{array}\right\} \text{ for } I > J$$

INTEGER PROCEDURE HISTD(A,U); NAME U; REAL ARRAY A; INTEGER U;

The value is an INTEGER in the range "A.lower bound" through "A.upper bound", where A is a one dimensional REAL ARRAY interpreted as a histogram defining the relative frequencies of values.

Utility procedures

PROCEDURE HISTO(A,B,C,D);{INTEGER / REAL}ARRAY A,B;
{INTEGER / REAL}C,D;

A call on HISTO updates a histogram defined by the one dimensional ARRAYS (INTEGER or REAL) A,B according to observation C with weight D. A(I) is incremented by D, where J is the smallest INTEGER such that C <= B(I). It is assumed that the length of A is one greater than the length of B. The last element of A corresponds to those observations which are greater than all the elements of B.

PROCEDURE LOWTEN(C); CHARACTER C;

Without use of LOWTEN, the CHARACTER 'E' represents the exponent sign in any numeric item to be edited or de-edited.
A call on "LOWTEN" with actual parameter "EXPSIGN" will replace 'E' by the value of EXPSIGN in future editing and de-editing.

APPENDIX C: SKELETON OF THE SYSTEM CLASSES AND SYSTEM PROCEDURES

Contents:

SYSTEM DEFINED PROCEDURES
- ARITHMETIC FUNCTIONS
- CHARACTER HANDLING
- TEXT HANDLING
- SEQUENCING PROCEDURES
- RANDOM DRAWING PROCEDURES
- UTILITY PROCEDURES

SYSTEM DEFINED CLASSES
- CLASS SIMSET
- CLASS SIMULATION
- SUBCLASSES of file

PROGRAM ENVIRONMENT

SYSTEM DEFINED PROCEDURES

ARITHMETIC FUNCTIONS

```
REAL PROCEDURE ABS(X); REAL X;
REAL PROCEDURE ARCCOS(X); REAL X;
REAL PROCEDURE ARCSIN(X); REAL X;
REAL PROCEDURE ARCTAN(X); REAL X;
REAL PROCEDURE COS(X); REAL X;
REAL PROCEDURE COSH(X); REAL X;
REAL PROCEDURE EXP(X); REAL X;
REAL PROCEDURE LN(X); REAL X;
REAL PROCEDURE SIN(X); REAL X;
REAL PROCEDURE SINH(X); REAL X;
REAL PROCEDURE SQRT(X); REAL X;
REAL PROCEDURE TAN(X); REAL X;
REAL PROCEDURE TANH(X); REAL X;
```

N.B. If the actual parameter on a call to any of the above procedures is LONG REAL, then a LONG REAL result will be returned.

```
INTEGER PROCEDURE ENTIER(X); REAL X;
INTEGER PROCEDURE MOD(X,Y); INTEGER X,Y;
INTEGER PROCEDURE SIGN(X); REAL X;
```

CHARACTER HANDLING

```
  BOOLEAN PROCEDURE DIGIT(C); CHARACTER C;
  BOOLEAN PROCEDURE LETTER(C); CHARACTER C;
CHARACTER PROCEDURE CHAR(N); INTEGER N;
  INTEGER PROCEDURE RANK(C); CHARACTER C;
```

TEXT HANDLING

TEXT OBJECT GENERATION

```
TEXT PROCEDURE COPY(T); VALUE T; TEXT T;
TEXT PROCEDURE BLANKS(N); INTEGER N;
```

TEXT ATTRIBUTES

```
CHARACTER PROCEDURE GETCHAR;
  INTEGER PROCEDURE GETFRAC;
  INTEGER PROCEDURE GETINT;
     REAL PROCEDURE GETREAL;
  INTEGER PROCEDURE LENGTH;
     TEXT PROCEDURE MAIN;
  BOOLEAN PROCEDURE MORE;
  INTEGER PROCEDURE POS;
          PROCEDURE PUTCHAR(C); CHARACTER C;
          PROCEDURE PUTFIX(X,N); REAL X; INTEGER N;
          PROCEDURE PUTFRAC(I,N); INTEGER I,N;
          PROCEDURE PUTINT(I); INTEGER I;
          PROCEDURE PUTREAL(X,N); REAL X; INTEGER N;
          PROCEDURE SETPOS(N); INTEGER N;
     TEXT PROCEDURE STRIP;
     TEXT PROCEDURE SUB(I,N); INTEGER I,N;
```

SEQUENCING PROCEDURES

```
PROCEDURE CALL(X); REF(anyclass)X;
PROCEDURE DETACH;
PROCEDURE RESUME(Y); REF(anyclass)X;
```

RANDOM DRAWING PROCEDURES

```
INTEGER PROCEDURE DISCRETE(A,U); NAME U; ARRAY A; INTEGER U;
BOOLEAN PROCEDURE DRAW(A,U); NAME U; REAL A; INTEGER U;
   REAL PROCEDURE ERLANG(A,B,U); NAME U; REAL A,B; INTEGER U;
INTEGER PROCEDURE HISTD(A,U); NAME U; ARRAY A; INTEGER U;
   REAL PROCEDURE LINEAR(A,B,U); NAME U; ARRAY A,B; INTEGER U;
   REAL PROCEDURE NEGEXP(A,U); NAME U; REAL A; INTEGER U;
   REAL PROCEDURE NORMAL(A,B,U); NAME U; REAL A,B; INTEGER U;
INTEGER PROCEDURE POISSON(A,U); NAME U; REAL A; INTEGER U;
INTEGER PROCEDURE RANDINT(A,B,U); NAME U; INTEGER A,B,U;
   REAL PROCEDURE UNIFORM(A,B,U); NAME U; REAL A,B; INTEGER U;
```

UTILITY PROCEDURES

```
PROCEDURE HISTO(A,B,C,D); ARRAY A,B; REAL C,D;
PROCEDURE LOWTEN(C); CHARACTER C;
```

SYSTEM DEFINED CLASSES

CLASS SIMSET:

```
 CLASS SIMSET;
 BEGIN  CLASS LINKAGE;
        BEGIN  REF(LINK) PROCEDURE SUC;
               REF(LINK) PROCEDURE PRED;
               REF(LINKAGE) PROCEDURE PREV;
        END ***LINKAGE*** ;

LINKAGE CLASS LINK;
        BEGIN  PROCEDURE OUT;
               PROCEDURE INTO(H); REF(HEAD)H;
               PROCEDURE PRECEDE(X); REF(LINKAGE)X;
               PROCEDURE FOLLOW(X); REF(LINKAGE)X;
        END ***LINK*** ;

LINKAGE CLASS HEAD;
        BEGIN  REF(LINK) PROCEDURE FIRST;
               REF(LINK) PROCEDURE LAST;
               PROCEDURE CLEAR;
               BOOLEAN PROCEDURE EMPTY;
               INTEGER PROCEDURE CARDINAL;.
        END ***HEAD*** ;

 END ***SIMSET*** ;
```

CLASS SIMULATION

```
SIMSET CLASS SIMULATION;
       BEGIN  LINK CLASS PROCESS;
              BEGIN  BOOLEAN PROCEDURE IDLE;
                     BOOLEAN PROCEDURE TERMINATED;
                     REAL PROCEDURE EVTIME;
                     REF(PROCESS) PROCEDURE NEXTEV;
              END ***PROCESS*** ;

              REF(PROCESS) PROCEDURE CURRENT;
              LONG REAL PROCEDURE TIME;
              PROCEDURE HOLD(T); REAL T;
              PROCEDURE PASSIVATE;
              PROCEDURE WAIT(Q); REF(HEAD)Q;
              PROCEDURE CANCEL(X); REF(PROCESS)X;
              PROCEDURE ACCUM(A,B,C,D); NAME A,B,C;
                                        REAL A,B,C,D;
              REF(main program)MAIN;
              <ACTIVATION-statements>
       END;
```

SUBCLASSES of file

```
CLASS file(name); VALUE name; TEXT name;
                  VIRTUAL : PROCEDURE OPEN, CLOSE;
BEGIN  TEXT IMAGE;
       INTEGER PROCEDURE LENGTH;
       BOOLEAN PROCEDURE MORE;
       INTEGER PROCEDURE POS;
       PROCEDURE SETPOS(I); INTEGER I;
END ***file*** ;

file CLASS INFILE; VIRTUAL : BOOLEAN PROCEDURE ENDFILE;
                                    PROCEDURE INIMAGE;

BEGIN             PROCEDURE CLOSE;
         BOOLEAN PROCEDURE ENDFILE;
       CHARACTER PROCEDURE INCHAR;
                 PROCEDURE INIMAGE;
         INTEGER PROCEDURE ININT;
         INTEGER PROCEDURE INFRAC;
            REAL PROCEDURE INREAL;
            TEXT PROCEDURE INTEXT(N); INTEGER N;.
         BOOLEAN PROCEDURE LASTITEM;
                 PROCEDURE OPEN(T); TEXT T;
END ***INFILE*** ;
```

```
file CLASS OUTFILE; VIRTUAL : PROCEDURE OUTIMAGE;
BEGIN  PROCEDURE CLOSE;
       PROCEDURE OPEN(T); TEXT T;
       PROCEDURE OUTCHAR(C); CHARACTER C;
       PROCEDURE OUTFIX(R,N,W); REAL R; INTEGER N,W;
       PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;
       PROCEDURE OUTIMAGE;
       PROCEDURE OUTINT(I,W); INTEGER I,W;
       PROCEDURE OUTREAL(X,N,W); REAL R; INTEGER N,W;
       PROCEDURE OUTTEXT(T); VALUE T; TEXT T;
END ***OUTFILE*** ;

OUTFILE CLASS PRINTFILE;
BEGIN          PROCEDURE CLOSE;
               PROCEDURE EJECT(N); INTEGER N;
       INTEGER PROCEDURE LINE;
               PROCEDURE LINESPERPAGE(N); INTEGER N;
               PROCEDURE OPEN(T); TEXT T;
               PROCEDURE OUTIMAGE;
               PROCEDURE SPACING(N); INTEGER N;
END ***PRINTFILE*** ;
```

```
file CLASS DIRECTFILE; VIRTUAL : PROCEDURE LOCATE OUTIMAGE, INIMAGE;
                                 BOOLEAN PROCEDURE ENDFILE;
BEGIN            PROCEDURE CLOSE;
         BOOLEAN PROCEDURE ENDFILE;
       CHARACTER PROCEDURE INCHAR;
         INTEGER PROCEDURE INFRAC;
                 PROCEDURE INIMAGE;
         INTEGER PROCEDURE ININT;
            REAL PROCEDURE INREAL;
            TEXT PROCEDURE INTEXT(N); INTEGER N;
         BOOLEAN PROCEDURE LASTITEM;
                 PROCEDURE LOCATE(I); INTEGER I;
         INTEGER PROCEDURE LOCATION;
                 PROCEDURE OPEN(T); TEXT T;
                 PROCEDURE OUTCHAR(C); CHARACTER C;
                 PROCEDURE OUTFIX(X,N,W); REAL X; INTEGER N,W;
                 PROCEDURE OUTFRAC(I,N,W); INTEGER I,N,W;
                 PROCEDURE OUTIMAGE;
                 PROCEDURE OUTINT(I,W); INTEGER I,W;
                 PROCEDURE OUTREAL(X,N,W); REAL X; INTEGER N,W;
                 PROCEDURE OUTTEXT(T); VALUE T; TEXT T;
END ***DIRECTFILE*** ;
```

PROGRAM ENVIRONMENT

A user's program behaves as though enclosed as below:

```
basicio(132) BEGIN  INSPECT SYSIN DO
                      INSPECT SYSOUT DO
                        <program>
             END
```

where "basicio" is defined by

```
CLASS basicio(linelength); INTEGER linelength;
BEGIN  REF(INFILE) PROCEDURE SYSIN;
       REF(OUTFILE) PROCEDURE SYSOUT;
  file CLASS INFILE;
  file CLASS OUTFILE;
OUTFILE CLASS PRINTFILE;
  file CLASS DIRECTFILE;
       sysin :- NEW INFILE("SYSIN");
       SYSIN.OPEN(BLANKS(80));
       sysout :- NEW OUTFILE("SYSOUT");
       SYSOUT.OPEN(BLANKS(132));
       INNER;
       SYSIN.CLOSE;
       SYSOUT.CLOSE;
END ***basicio*** ;
```

INDEX

```
*****************************************
* N.B. THE SECTIONS THAT THE INDEXED    *
* WORD APPEAR IN ARE GIVEN ON THE RIGHT*
* AS A LIST IN THE FORMAT:              *
*      <PART NUMBER>:<SECTION NUMBER>   *
* E.G. 2:7.2 MEANS PART 2,SECTION 7.2   *
*****************************************
```


ACCESSIBLE 3:1.0
ACCESSING 1:3.0
ACCUM 3:3.0
ACTIVATE 2:2.2, 3:3.0
ACTIVATION STATEMENTS 2:7.2
ACTIVATION-STATEMENT 2:7.2
ACTIVE 1:4.0, 3:1.0
ACTUAL-FORMAL PARAMETER CORRESPONDENCE 2:6.2
ACTUAL-PARAMETER 1:3.0, 2:5.4, 2:6.2
ACTUAL-PARAMETER-LIST 2:6.2
ACTUAL-PARAMETER-PART 2:6.7, 2:7.1
ADDITION SIGN (+) 2:2.2
AFTER 2:2.2, 3:3.0
ALPHABETIC CHARACTER 2:2.1
ALPHANUMERIC CHARACTER 2:2.1
ALTERNATIVES 2:1.5
AND 1:2.0, 2:2.2, 2:2.2
2:6.4
ARITHMETIC EXPRESSIONS 2:6.3
ARITHMETIC OPERATOR PRECEDENCE 2:6.3
ARITHMETIC OPERATORS AND TYPES 2:6.3
ARITHMETIC TYPE CONVERSION 2:5.1
ARITHMETICS OF (LONG)REAL QUANTITIES 2:6.3
ARITHMETIC-CONSTANTS 2:4.2
ARITHMETIC-CONSTANTS,USE OF 2:4.2
ARITHMETIC-CONSTANT 2:6.3
ARITHMETIC-EXPRESSIONS 2:6.0
ARITHMETIC-EXPRESSION 2:6.1, 2:6.3, 2:7.2
ARITHMETIC-FUNCTION-DESIGNATOR 2:6.3
ARITHMETIC-OPERATORS 2:2.2
ARITHMETIC-OPERATOR 2:6.3
ARITHMETIC-PRIMARY 2:6.3
ARITHMETIC-VARIABLE 2:6.3
ARRAY 1:2.0, 2:2.2, 2:5.2
2:5.4, 2:5.5
ARRAY COMPONENT 2:5.2, 2:6.1

ARRAY DECLARATIONS 2:5.2
ARRAY DIMENSION 2:5.2
ARRAY SEGMENT 2:5.2
ARRAY-DECLARATION 2:5.0, 2:5.2
ARRAY-IDENTIFIER 2:5.2, 2:6.2
ARRAY,INITIALISATION OF 2:5.2
ASSIGNABLE RANGE,BOOLEAN 2:5.1
ASSIGNABLE RANGE,CHARACTER 2:5.1
ASSIGNABLE RANGE,INTEGER 2:5.1
ASSIGNABLE RANGE,LONG REAL 2:5.1
ASSIGNABLE RANGE,REAL 2:5.1
ASSIGNABLE RANGE,REF 2:5.1
ASSIGNABLE RANGE,SHORT INTEGER 2:5.1
ASSIGNABLE RANGE,TEXT 2:5.1
ASSIGNMENT 2:6.1
ASSIGNMENTS 2:7.2
ASSIGNMENT-STATEMENT 1:1.0, 2:7.2
ASTERISK (*) 2:2.1
AT 2:2.2, 3:3.0
ATTACHED STATE 3:1.0
ATTRIBUTE 1:3.0, 2:5.5, 2:6.1
ATTRIBUTE REDEFINITION (VIRTUAL) 2:5.5
BASIC BINDING RULES 2:3.0
BASIC SYMBOL 2:1.0, 2:1.2, 2:2.0
BASIC SYMBOL SET 2:2.1
BASIC SYMBOLS AND SYNTACTIC VARIABLES 2:2.0
BASIC SYMBOLS 2:2.2
BASICIO 3:5.0
BECOMES SIGN (:=) 2:2.2
BECOMES (:=) 2:7.2
BEFORE 2:2.2, 3:3.0
BEGIN 2:2.2, 2:5.5, 2:7.1
BINDING RULE 1:1.0
BINDING RULES,BASIC 2:3.0
BINDING RULES,CONNECTION 2:7.2
BINDING RULES,REMOTE ACCESSING 2:6.1
BINDING RULES,VIRTUAL QUANTITIES 2:5.5
BLANK 3:4.0
BLANK (␣) 2:2.1
BLANKS 3:4.0
BLANK,IN CHARACTER-CONSTANT 2:4.2
BLANK,IN TEXT-CONSTANT 2:4.2
BLOCK 1:1.0, 2:7.0, 2:7.1
2:7.2
BLOCK HEAD 2:5.0, 2:7.0

BLOCK INSTANCE 1:1.0
BLOCK PREFIX 3:2.0
BLOCKS AND COMPOUND STATEMENTS 2:7.1
BLOCKS AND STATEMENTS 2:7.0
BOOLEAN 2:2.2, 2:4.1, 2:5.1 2:5.5
BOOLEAN OPERATORS 2:6.4
BOOLEAN-CONSTANTS 2:4.2, 2:4.2
BOOLEAN-EXPRESSIONS 2:6.4
BOOLEAN-FUNCTION-DESIGNATOR 2:6.4
BOOLEAN-OPERATOR 2:6.4
BOOLEAN-PRIMARY 2:6.4
BOOLEAN-VARIABLE 2:6.4
BRACES 2:1.4
BRACKETS 2:2.2
BUFFER 3:5.0
CALL 2:5.5, 3:1.0
CALL BY NAME 2:5.4, 2:5.5, 2:7.2
CALL BY REFERENCE 2:5.4, 2:5.5
CALL BY VALUE 2:5.4, 2:5.5
CANCEL 3:3.0
CAR WASH SIMULATION 1:4.0
CARD READER 3:5.0
CHARACTER 2:2.2, 2:4.1, 2:5.1 2:5.5
CHARACTER ACCESS (TO TEXTS) 3:4.0
CHARACTER ARRAY 3:4.0
CHARACTER EXPRESSIONS 2:6.5
CHARACTER QUOTE (>) 2:2.1, 2:4.2
CHARACTER STRING 3:4.0
CHARACTER VALUES,COMPARISON 2:6.4
CHARACTER-CONSTANTS 2:4.2
CHARACTER-CONSTANT 2:4.2, 2:6.5
CHARACTER-EXPRESSIONS 2:6.0
CHARACTER-EXPRESSION 2:6.5
CHARACTER-FUNCTION-DESIGNATOR 2:6.5
CHARACTER-VARIABLE 2:6.5
CLASS 1:3.0, 2:2.2, 2:5.5
CLASS BASICIO 3:5.0
CLASS CAR 1:4.0
CLASS CARD 3:2.0
CLASS CIRCLE 1:3.0
CLASS DECK 3:2.0
CLASS DECLARATION 1:3.0
CLASS DECLARATIONS 2:5.5

CLASS DIRECTFILE 3:5.0
CLASS FILE 3:5.0
CLASS HAND 3:2.0
CLASS HEAD 1:4.0, 3:2.0
CLASS HIERARCHIES 2:5.5
CLASS INFILE 3:5.0
CLASS INSTANCE 2:5.5
CLASS LINE 1:3.0
CLASS LINK 1:4.0, 3:2.0
CLASS LINKAGE 3:2.0
CLASS OUTFILE 3:5.0
CLASS PARAMETERS (TABLE) 2:5.5
CLASS PLAYER 3:1.0
CLASS POINT 1:3.0
CLASS PRINTFILE 3:5.0
CLASS PROCESS 1:4.0
CLASS ROW 2:5.5
CLASS ROW1 2:5.5
CLASS SICKP 3:3.0
CLASS SIMSET 3:2.0, 3:3.0
CLASS SIMULATION 1:4.0, 2:7.2, 3:3.0
CLASS STACKABLE 1:3.0
CLASS TREATMENT 3:3.0
CLASS WASH 1:4.0
CLASS-BODY 1:3.0, 2:3.0, 2:5.0
2:5.5, 2:5.5, 2:6.7
CLASS-DECLARATION 2:5.0, 2:5.5
CLASS-IDENTIFIER 2:5.5, 2:6.4, 2:6.7
2:7.1
CLASS-SPEC-PART 2:5.5
CLOSE 3:5.0
CLOSED 2:7.2
CODING FORM 2:2.6
CODING SIMULA PROGRAMS 2:2.6
COLLATING SEQUENCE 2:2.3
COLON 1:1.0
COLON (:) 2:2.1, 2:5.5, 2:7.2
COMMA (,) 2:2.1, 2:6.1, 2:6.2
2:7.2
COMMENT 1:2.0, 2:2.2
COMMENT CONVENTIONS 2:2.5
COMPILE TIME AND RUN TIME 1: .0
COMPILE TIME 1: .0
COMPOUND TAIL 2:7.0
COMPOUND-STATEMENT 1:2.0, 2:7.1, 2:7.1

	2:7.2
CONDITION	2:6.0, 2:6.0, 2:6.3 2:6.4, 2:7.2
CONDITIONAL STATEMENTS	2:7.2
CONDITIONAL-STATEMENT	1:1.0, 2:7.2, 2:7.2
CONDITIONS	2:6.4
CONNECTION BLOCK	2:6.7
CONNECTION STATEMENTS	2:7.2
CONNECTION-STATEMENT	2:7.2, 2:7.2
CONSTANT	2:2.0, 2:4.0, 2:6.0
CONSTANTS	2:4.2
CONTROL CARD	2:2.2
CONTROLLED-STATEMENT	2:7.2
CONTROLLED-VARIABLE	2:7.2
COPY	3:4.0
CURRENT	3:3.0
CURRENT POSITION INDICATOR (TEXT)	3:5.0
DATA CHARACTER SET	2:2.3
DATA SET	3:5.0
DDNAME	3:5.0
DECIMAL DIGIT	2:2.1
DECIMAL-CONSTANTS	2:4.2
DECIMAL-DIGIT	2:4.2
DECIMAL-DIGITS	2:4.2, 2:4.2
DECLARATIONS	2:5.0
DECLARATION	2:5.0, 2:5.5, 2:7.1
DECLARATORS	2:2.2
DEFAULT ACTIONS (AFTER/BEFORE)	3:3.0
DEFAULT ACTIONS (AT/DELAY)	3:3.0
DEFAULT MODE OF PARAMETER TRANSMISSION	2:5.4
DEFAULT VALUE OF A FUNCTION-DESIGNATOR	2:5.4
DELAY	2:2.2, 3:3.0
DELIMETERS	2:2.2
DENOTES SIGN (:-)	2:2.2
DENOTES (:-)	2:7.2
DESIGNATIONAL EXPRESSIONS	2:6.6
DESIGNATIONAL-EXPRESSION	2:5.3, 2:6.6, 2:7.2
DESIGNATIONAL-EXPRESSIONS	2:6.0
DETACH	2:5.5, 2:6.7, 3:1.0 3:3.0
DETACHED STATE	2:6.7, 3:1.0
DE-EDITING PROCEDURES	3:4.0
DIGIT	2:3.0
DIGITS	3:4.0
DIRECT ACCESS FILES	3:5.0

DIRECT ACTIVATION 3:3.0
DIRECTFILE 3:5.0
DIVIDE (/) 2:2.1
DIVISION SIGN (/) 2:2.2
DO 2:2.2, 2:7.2, 2:7.2
DOLLAR ($) 2:2.1
DOT NOTATION 1:3.0
DOT (.) 2:2.1, 2:4.2, 2:6.1
2:7.2
DUMMY STATEMENTS 2:7.2
DUMMY-STATEMENT 2:5.5, 2:7.2, 2:7.2
EDITING PROCEDURES 3:4.0
EJECT 3:5.0
ELSE 2:2.2, 2:6.3, 2:6.4
2:6.5, 2:6.7, 2:6.8
EMPTY PART AND VIRTUAL PART 2:5.5
EMPTY TEXT VALUE (NOTEXT) 2:2.2
END 2:2.2, 2:5.5, 2:7.1
END OF FILE 3:5.0
ENDFILE 3:5.0
EPIDEMIC 3:3.0
EQ (=) 2:2.2, 2:6.4
EQUAL TO SIGN (=) 2:2.2
EQUAL (W.R.T. QUALIFICATION) 2:5.5
EQUALS (=) 2:2.1
EQUIVALENCE (EQV) 2:2.2
EQV 2:2.2, 2:2.2, 2:6.4
EVENT NOTICE 3:3.0
EVTIME 3:3.0
EXAMPLE ON THE USE OF FILES 3:5.0
EXAMPLE ON THE USE OF SIMULATION 3:3.0
EXIT FROM A BLOCK 2:5.0
EXPONENT 2:4.2
EXPONENT SIGN IN DECIMAL-CONSTANT 2:4.2
EXPONENT SIGN (&) 2:4.2
EXPONENT (&) 2:2.1
EXPRESSION 2:6.2, 2:7.2
EXPRESSIONS 2:6.0
EXTERNAL DATA 3:5.0
EXTERNAL DECLARATIONS 2:5.6
EXTERNAL FILE 3:5.0
EXTERNAL RECORD 3:5.0
EXTERNAL-DECLARATION 2:5.0, 2:5.6
E,EXPONENT SIGN 3:4.0
FALSE 2:2.2, 2:4.2, 2:6.4

FIELD 3:5.0
FILE 3:5.0, 3:5.0
FIXED FIELD TEXT EDITING 3:4.0
FOLLOW 3:2.0
FOR 2:2.2, 2:7.2
FOR LIST ELEMENTS 2:7.2
FOR STATEMENTS 2:7.2
FORMAL-ACTUAL PARAMETER CORRESPONDENCE 2:6.2
FORMAL-PARAMETER 2:3.0
FORMAL-PARAMETER-PART 2:5.4
FORTRAN CODING FORM 2:2.6
FOR-RIGHT-PART 2:7.2
FOR-STATEMENT 1:1.0, 2:7.2, 2:7.2
FOR-STATEMENT,LOCAL LABELS 2:7.2
FUNCTION DESIGNATORS 2:6.2
FUNCTION-DECLARATION 2:5.4
FUNCTION-DESIGNATOR 2:6.0, 2:6.2, 2:6.7
GE (>=) 2:2.2, 2:6.4
GENERATED OBJECT 2:6.7
GEOMETRICAL APPLICATIONS 1:3.0
GETCHAR 3:4.0
GETFRAC 3:4.0
GETINT 3:4.0
GETREAL 3:4.0
GO 2:2.2
GOTO 2:2.2, 2:7.2
GOTO EXIT 2:5.5, 3:1.0
GOTO STATEMENTS 2:7.2
GOTO-STATEMENT 1:1.0, 2:7.2, 3:1.0
GREATER THAN SIGN (>) 2:2.2
GREATER THAN (>) 2:2.1
GREATER THEN OR EQUAL TO SIGN (>=) 2:2.2
GROUP MARKERS 2:1.4
GROUPED-ITEM 3:4.0
GROUPS 3:4.0
GT (>) 2:2.2, 2:6.4
HASH SIGN (#) 2:4.2
HASH (#) 2:2.1
HEAD 3:2.0
HEXADECIMAL-CONSTANTS 2:4.2
HEXADECIMAL-CONSTANT 2:4.2
HISTORY OF A MODEL 1:4.0
HOLD 3:3.0
IDENTIFIERS AND KEY WORDS 2:3.0
IDENTIFIER 2:2.0, 2:3.0, 2:5.0

	2:6.1, 2:6.2
IDENTIFIER CLASH	3:1.0
IDENTIFIERS	2:3.0
IDENTIFIER-LIST	2:5.0, 2:5.1, 2:5.2 2:5.4, 2:5.5
IDLE	3:3.0
IF	2:2.2, 2:6.0, 2:7.2
IF-CLAUSE	2:6.0, 2:6.3, 2:6.4 2:6.5, 2:6.7, 2:6.8 2:7.2
IF-STATEMENT	2:7.2, 2:7.2
IMAGE	3:5.0, 3:5.0
IMP	2:2.2, 2:2.2, 2:6.4
IMPLICATION (IMP)	2:2.2
IN	2:2.2, 2:6.4
INCHAR	3:5.0
INCLUDE	2:5.5
INFILE	3:5.0
INFRAC	3:5.0
ININAGE	3:5.0
ININT	3:5.0
INITIAL VALUE	1:1.0, 2:4.0
INITIAL VALUE,BOOLEAN	2:5.1
INITIAL VALUE,CHARACTER	2:5.1
INITIAL VALUE,INTEGER	2:5.1
INITIAL VALUE,LONG REAL	2:5.1
INITIAL VALUE,REAL	2:5.1
INITIAL VALUE,REF	2:5.1
INITIAL VALUE,SHORT INTEGER	2:5.1
INITIAL VALUE,TEXT	2:5.1
INNER	2:2.2, 2:5.4, 2:5.5 2:5.5, 2:7.2, 3:5.0
INNER ACCESSIBILITY (VIRTUAL)	2:5.5
INNER BLOCK	1:2.0
INREAL	3:5.0, 3:5.0
INSPECT	1:3.0, 2:2.2, 2:7.2
INSTANTANEOUS QUALIFICATION	2:6.7
INTEGER	2:2.2, 2:4.1, 2:5.1 2:5.5, 2:6.3
INTEGER DIVISION SIGN (//)	2:2.2
INTEGER-CONSTANT	2:4.2
INTEGER-ITEM	3:4.0
INTEXT	3:5.0
INTO	3:2.0
IS	2:2.2, 2:6.4

JOB CONTROL CARDS 2:2.6
KEY WORD 2:1.0, 2:3.0
KEY WORDS 2:2.2
KEY-WORD-CONSTANTS 2:2.2
LABEL 2:2.2, 2:3.0, 2:5.4 2:5.5, 2:7.2, 2:7.2
LABELS LOCAL TO A FOR-STATEMENT 2:7.2
LABEL-IDENTIFIER 2:6.2, 2:6.6, 2:7.2
LANGUAGE CHARACTER SET 2:2.0, 2:2.1
LASTITEM 3:5.0
LE (<=) 2:2.2, 2:6.4
LEFT PARENTHESIS (() 2:2.1, 2:5.2, 2:5.4 2:5.5, 2:6.1, 2:6.3 2:6.4, 2:6.5, 2:6.6 2:6.7, 2:6.8
LENGTH 3:4.0, 3:5.0
LENGTH AND MAIN 3:4.0
LENGTH OF A TEXT VALUE 3:4.0
LENGTH OF IDENTIFIERS 2:3.0
LESS THAN OR EQUAL TO SIGN (<=) 2:2.2
LESS THAN SIGN (<) 2:2.2
LESS THAN (<) 2:2.1
LETTER 2:3.0
LINE 3:5.0
LINE PRINTER 3:5.0
LINELENGTH 3:5.0
LINESPERPAGE 3:5.0
LINK 3:2.0
LINKAGE 3:2.0
LIST PROCESSING 3:2.0
LOCAL OBJECTS 2:6.7
LOCAL QUANTITY 2:7.1
LOCAL SEQUENCE CONTROL (LSC) 3:1.0
LOCAL-OBJECT 2:6.7
LOCATE 3:5.0
LOCATION 3:5.0
LOGICAL AND (AND) 2:2.2
LOGICAL EQUIVALENCE (EQV) 2:2.2
LOGICAL IMPLICATION (IMP) 2:2.2
LOGICAL INCLUSIVE OR (OR) 2:2.2
LOGICAL NEGATION (NOT,¬) 2:2.2
LOGICAL-OPERATORS 2:2.2, 2:2.2
LONG 2:5.1
LONG REAL 2:2.2, 2:4.1, 2:5.5 2:6.3

LONG REAL-CONSTANT 2:4.2
LOWER SUBSCRIPT BOUND 2:5.2
LOWER-BOUND 2:5.2
LOWTEN 3:4.0
LSC 3:1.0
LT (<) 2:2.2, 2:6.4
MAGIC BOX (TEXT) 3:4.0
MAIN 3:3.0, 3:4.0
MAIN-BLOCK 2:7.1
MAIN-PART 2:5.5
MATCHING ATTRIBUTE (VIRTUAL) 2:5.5
MATCHING QUALIFICATION 2:5.4
METHOD OF SYNTAX SPECIFICATION 2:1.0
MINUS (-) 2:2.1
MODE 2:5.4
MODE-PART 2:5.4
MORE 3:4.0, 3:5.0
MULTIPLE ASSIGNMENT 1:2.0
MULTIPLICATION SIGN (*) 2:2.2
MULTIPLY (*) 2:2.1
NAME 1:3.0, 2:2.2, 2:5.4
NE (¬=) 2:2.2, 2:6.4
NEW 2:2.2, 2:6.7
NEXTEV 3:3.0
NONE 1:3.0, 2:2.2, 2:4.2
2:6.1, 2:6.7
NON-EMPTY PREFIX AND EMPTY VIRTUAL PART 2:5.5
NOT 2:2.2
NOT EQUAL TO SIGN (¬=) 2:2.2
NOT (¬) 2:2.1, 2:2.2, 2:6.4
2:6.4
NOTEXT 2:2.2, 2:6.8, 3:4.0
NUMBERING A CROSSWORD PUZZLE 1:2.0
NUMERIC ITEM 3:4.0
NUMERIC-TEXT-VALUES 3:4.0
OBJECT 2:5.5, 2:6.7
OBJECT EXPRESSIONS 2:6.7
OBJECT GENERATORS 2:6.7, 2:7.2
OBJECT PROGRAM 1: .0
OBJECT-ELEMENT 2:7.2
OBJECT-EXPRESSIONS 2:6.0
OBJECT-EXPRESSION 2:6.7, 2:6.7, 2:7.2
OBJECT-GENERATOR 2:7.2
OBJECT-REFERENCE-CONSTANT 2:4.2, 2:4.2
OBJECT-REFERENCE 2:7.2

ONE DIMENSIONAL ARRAY 2:5.2
OPEN 2:7.2, 3:5.0
OPERAND 2:6.0
OPERATOR 2:6.0
OPERATORS 2:2.2
OPTIONS 2:1.6
OR 1:2.0, 2:2.2, 2:2.2 2:6.4
OTHERWISE 2:2.2, 2:7.2
OUT 3:2.0
OUTCHAR 3:5.0
OUTER 2:5.5
OUTER BLOCK 1:2.0
OUTFILE 3:5.0
OUTFIX 3:5.0
OUTFRAC 3:5.0
OUTIMAGE 3:5.0, 3:5.0
OUTINT 3:5.0
OUTREAL 3:5.0
OUTTEXT 3:5.0
PARAMETER 2:5.4
PARAMETER TRANSMISSION 2:5.4
PARAMETERS TO CLASSES 2:5.4
PARAMETERS TO PROCEDURES 2:5.4
PARAMETER-PART 2:5.5
PASSIVATE 3:3.0
PASSIVE 1:4.0, 3:1.0
PERIOD (.) 2:2.1
PLUS (+) 2:2.1
POS 3:4.0, 3:5.0
POWER SIGN (**) 2:2.2
PRECEDE 3:2.0
PRECEDENCE OF OPERATORS 2:6.4
PRED 3:2.0
PREFIX CHAIN 2:5.5
PREFIX SEQUENCE 2:5.5
PREFIXED BLOCK 1:3.0, 2:7.1
PREFIXED-BLOCK 2:5.5, 2:7.1
PREV 3:2.0
PRINTFILE 3:5.0
PRIOR 2:2.2, 3:3.0
PROCEDURE 1:3.0, 2:2.2, 2:5.4 2:5.5
PROCEDURE ADD (REF(POINT)) 2:5.4
PROCEDURE CALL_BY_VALUE 2:5.4

PROCEDURE CHAR 2:6.5
PROCEDURE COMPRESS 3:4.0
PROCEDURE DECLARATIONS 2:5.4
PROCEDURE DIGIT 2:6.5
PROCEDURE FACTORIAL (INTEGER) 2:5.4
PROCEDURE INFECT 3:3.0
PROCEDURE LETTER 2:6.5
PROCEDURE NORM (REAL) 2:5.4
PROCEDURE NULLREF (INTEGER) 2:5.4
PROCEDURE OUTCOLUMN 2:5.4
PROCEDURE PARAMETERS (TABLE) 2:5.4
PROCEDURE PLACE 3:2.0
PROCEDURE RANK 2:6.5
PROCEDURE SELECT 3:2.0
PROCEDURE STATEMENTS 2:7.2, 2:7.2
PROCEDURE SWAP 2:5.4
PROCEDURE THESETHEADOF (REF(HEAD)) 3:2.0
PROCEDURE TREETRAVERSE 2:5.4
PROCEDURE-BODY 1:3.0, 2:3.0, 2:5.4
PROCEDURE-DECLARATION 1:3.0, 2:5.0, 2:5.4
PROCEDURE-HEADING 2:5.4
PROCEDURE-IDENTIFIER 2:5.4, 2:6.2, 2:7.2
2:7.2
PROCEDURE-STATEMENT 2:7.2, 2:7.2, 2:7.2
PROCESS 3:3.0
PROCESS OBJECT (TABLE OF STATES) 3:3.0
PROCESS-EXPRESSION 3:3.0
PROGRAM 2:7.0, 2:7.1
PROGRAM CONTENT 2:2.0
PROGRAM SEQUENCE CONTROL 3:1.0
PROGRAM STRUCTURE 3:1.0
PROPER-PROCEDURE-DECLARATION 2:5.4
PSC 3:1.0
PUTCHAR 3:4.0
PUTFIX 3:4.0, 3:5.0
PUTINT 3:4.0, 3:5.0
PUTREAL 3:4.0, 3:5.0
QUA 2:2.2, 2:6.7
QUALIFICATION 1:3.0, 2:6.7
QUALIFICATION OF ACTUAL PARAMETER 2:5.4
QUALIFYING CLASS 2:5.1
QUANTITY 2:3.0
QUASI-PARALLEL SYSTEM 2:7.1
R (REAL IN HEXADECIMAL-CONSTANTS) 2:4.2
RAISED-TO-THE-POWER-OF SIGN (**) 2:2.2

RANGE OF VALUES,BOOLEAN VARIABLES 2:5.1
RANGE OF VALUES,CHARACTER VARIABLES 2:5.1
RANGE OF VALUES,INTEGER VARIABLES 2:5.1
RANGE OF VALUES,INTEGER-CONSTANT 2:4.2
RANGE OF VALUES,LONG REAL VARIABLES 2:5.1
RANGE OF VALUES,LONG REAL-CONSTANT 2:4.2
RANGE OF VALUES,REAL VARIABLES 2:5.1
RANGE OF VALUES,REAL-CONSTANT 2:4.2
RANGE OF VALUES,REF VARIABLES 2:5.1
RANGE OF VALUES,SHORT INTEGER VARIABLES 2:5.1
RANGE OF VALUES,SHORT INTEGER-CONSTANT 2:4.2
RANGE OF VALUES,TEXT VARIABLES 2:5.1
RANK 2:6.4
REACTIVATE 2:2.2, 3:3.0
REAL 2:2.2, 2:4.1, 2:5.1
2:5.5, 2:6.3
REAL-CONSTANT 2:4.2
REAL-ITEM 3:4.0
RECORD 3:5.0
RECURSION 2:5.4
REF 2:2.2, 2:4.1, 2:5.1
2:5.5
REFERENCE COMPARATORS 2:6.4
REFERENCE EQUAL (==) 2:2.2
REFERENCE NOT EQUAL (=/=) 2:2.2, 2:6.4
REFERENCE VARIABLE 1:3.0
REFERENCE-ASSIGNMENT 2:7.2
REFERENCE-COMPARATORS 2:2.2
REFERENCE-CONSTANT 2:4.2
REFERENCE-EQUAL (==) 2:6.4
REFERENCE-TYPE 2:4.1, 2:5.1
RELATION 2:6.4
RELATIONAL-OPERATORS 2:2.2, 2:2.2
RELATIONS 2:6.4
REMOTE ACCESSING 1:3.0, 2:6.1, 2:7.2
REMOTE-IDENTIFIER 2:6.2
REMOTE-VARIABLE 2:6.1
REPETITION 2:1.7
RESUME 2:5.5, 3:1.0, 3:3.0
RIGHT PARENTHESIS ()) 2:2.1, 2:5.2, 2:5.4
2:5.5, 2:6.1, 2:6.3
2:6.4, 2:6.5, 2:6.6
2:6.7, 2:6.8
RTS BLOCK 1:2.0
RTS INFO 1:1.0, 1:3.0

RUN TIME 1: .0
RUN TIME ERROR 1:3.0, 3:5.0
RUN TIME STRUCTURE 3:1.0
RUN TIME SYSTEM 1: .0
RUN TIME SYSTEM INFORMATION 1:1.0
SCOPE 2:3.0
SECURITY 1:3.0, 2:5.0
SEMANTICS 2:1.0
SEMICOLON 1:1.0
SEMICOLON (;) 2:2.1, 2:5.4, 2:5.5
2:5.5
SEPARATORS 2:2.2
SEQUENTIAL FILE ORGANISATION 3:5.0
SEQUENTIAL-OPERATORS 2:2.2
SET 3:2.0
SETPOS 3:4.0, 3:5.0
SHORT INTEGER 2:2.2, 2:4.1, 2:5.1
2:5.5, 2:6.3
SHORT INTEGER-CONSTANT 2:4.2
SIGN-PART 3:4.0
SIMPLE VARIABLE 2:7.2
SIMPLE-ARITHMETIC EXPRESSION 2:6.3
SIMPLE-CHARACTER-EXPRESSION 2:6.5
SIMPLE-DESIGNATIONAL-EXPRESSION 2:6.6
SIMPLE-OBJECT-EXPRESSION 2:6.1, 2:6.4, 2:6.7
2:6.7, 2:7.2
SIMPLE-REFERENCE-EXPRESSION 2:6.4, 2:6.7
SIMPLE-TEXT-EXPRESSION 2:6.1, 2:6.4, 2:6.8
SIMPLE-VALUE EXPRESSION 2:6.4
SIMPLE-VARIABLE 2:6.1
SIMSET 3:2.0
SIMSET,EXAMPLE ON THE USE OF 3:2.0
SIMSET,PREFIX TO SIMULATION 3:3.0
SIMULATION OF A QUEUING SITUATION 1:4.0
SOURCE PROGRAM 1: .0
SPACE (␣) 2:2.1
SPACING 3:5.0
SPECIAL CHARACTER 2:1.0, 2:2.1
SPECIFICATORS 2:2.2
SPEC-PART 2:5.4
SPLIT-BODY 2:5.5
SQUARE BRACKETS 2:1.6
STACK 1:3.0
STACK OF ATTACHED BLOCKS 3:1.0
STACK OF BLOCKS 3:1.0

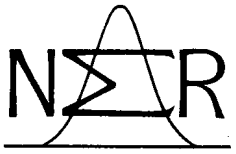
STANDARD SYSTEM PROCEDURES 1:2.0
STATEMENT 1:1.0, 2:5.4, 2:5.5 2:7.1, 2:7.2, 2:7.2
STATEMENTS 2:7.2
STATEMENT-BRACKETS 2:2.2
STEP 2:2.2, 2:7.2
STRIP 3:4.0
STRUCTURE OF A QUANTITY 2:5.0
SUB 3:4.0
SUBCLASS 2:5.1, 2:5.5
SUBSCRIPT 2:6.1
SUBSCRIPT BOUND 2:5.2
SUBSCRIPT BOUNDS 2:6.1
SUBSCRIPTED VARIABLE 2:5.2
SUBSCRIPTED-VARIABLE 2:6.1
SUBSCRIPT-LIST 2:6.1
SUBSCRIPT,SWITCH 2:6.6
SUBTEXTS 3:4.0
SUBTRACTION SIGN (-) 2:2.2
SUC 3:2.0
SWITCH 2:2.2, 2:5.3, 2:5.4 2:5.5
SWITCH DECLARATIONS 2:5.3
SWITCH-DECLARATION 2:5.0
SWITCH-DESIGNATOR 2:5.3, 2:6.6
SWITCH-IDENTIFIER 2:5.3, 2:6.2, 2:6.6
SWITCH-LIST 2:5.3
SYNTACTIC UNIT 2:1.0, 2:1.3
SYNTACTIC VARIABLE 2:1.0, 2:1.1
SYNTAX 2:1.0
SYSIN 3:5.0
SYSOUT 3:5.0
TERMINATED 3:3.0
TERMINATED STATE 2:6.7, 3:1.0
TEXT 2:2.2, 2:4.1, 2:5.1 2:5.5, 3:4.0, 3:5.0
TEXT ASSIGNMENT 3:4.0
TEXT EDITING AND DE-EDITING 3:4.0
TEXT EXPRESSIONS AND TEXT VALUES 2:6.8
TEXT GENERATION 3:4.0
TEXT HANDLING FACILITIES 3:4.0
TEXT HANDLING 3:5.0
TEXT OBJECT 2:6.8, 3:4.0, 3:5.0
TEXT QUOTE (Δ) 2:2.1, 2:4.2
TEXT REFERENCE 2:6.4

TEXT VALUES,COMPARISON 2:6.4
TEXTUAL LINK 1:2.0
TEXTUALLY ENCLOSING BLOCK 2:3.0
TEXT- 3:4.0
TEXT-CONSTANT 2:4.2, 2:4.2, 2:6.8
TEXT-EXPRESSION 2:6.0, 2:6.8
TEXT-FUNCTION-DESIGNATOR 2:6.8, 2:7.2
TEXT-REFERENCE-ASSIGNMENT 3:4.0
TEXT-VALUE 2:6.0, 2:6.8, 3:4.0
TEXT-VALUE-ASSIGNMENT 2:7.2, 3:4.0
TEXT-VARIABLE 2:6.8, 3:4.0
THE CONTROLLED VARIABLE 2:7.2
THE HIGHEST MARK 1:1.0
THE SYNTAX AND SEMANTICS OF SIMULA 2: .0
THE SYSTEM CLASS BASICIO 3:5.0
THE SYSTEM CLASS SIMSET 3:2.0
THE SYSTEM CLASS SIMULATION 3:3.0
THE USE OF BLANKS 2:2.4
THEN 2:2.2, 2:6.0, 2:7.2
THIS 2:2.2, 2:6.7, 2:7.1
THREE DIMENSIONAL ARRAY 2:5.2
THREE DOTS 2:1.7
TIME 3:3.0, 3:3.0
TIME AXIS 3:3.0
TIMING CLAUSE 3:3.0
TO 2:2.2
TRACE OF A MODEL 1:4.0
TRUE 2:2.2, 2:4.2, 2:6.4
TWO DIMENSIONAL ARRAY 2:5.2
TWO WAY LIST 3:2.0
TYING A DDNAME TO A FILE 3:5.0
TYPE 2:4.0, 2:4.1, 2:5.2
2:5.4, 2:5.5, 2:5.5

TYPE ARRAY 2:5.5
TYPE CONVERSION (ARITHMETIC) 2:5.1
TYPE DECLARATIONS 2:5.1
TYPE OF A QUANTITY 2:5.0
TYPE OF ARITHMETIC EXPRESSIONS 2:6.3
TYPES 2:4.1, 2:7.2
TYPES AND CONSTANTS 2:4.0
TYPE-DECLARATION 2:5.0, 2:5.1
UNDERSCORE (_) 2:2.1, 2:3.0
UNTIL 2:2.2, 2:7.2
UPPER SUBSCRIPT BOUND 2:5.2
UPPER-BOUND 2:5.2

VALUE 1:3.0, 2:2.2, 2:4.0 2:5.4, 2:5.5
VALUE COMPARATORS 2:6.4
VALUE OF CONTROLLED VARIABLE ON EXIT 2:7.2
VALUE-ASSIGNMENT 2:7.2
VALUE-ELEMENT 2:7.2
VALUE-EXPRESSION 2:7.2, 2:7.2
VALUE-TYPE 2:4.1, 2:5.1
VARIABLE 2:4.0, 2:6.0, 2:6.1 2:6.7, 2:7.2
VARIABLES 2:6.1
VERTICAL STROKE 2:1.5
VIRTUAL 2:2.2, 2:5.5
VIRTUAL ATTRIBUTES OF FILES 3:5.0
VIRTUAL-PART 2:5.5
VIRTUAL-QUANTITIES 2:5.5
VIRTUAL,ATTRIBUTE REDEFINITION 2:5.5
VIRTUAL,INNER ACCESSIBILITY 2:5.5
VIRTUAL,MATCHING ATTRIBUTE 2:5.5
VIRTUAL,VALIDITY OF A MATCH 2:5.5
WAIT 3:3.0
WHEN 2:2.2, 2:7.2
WHILE 2:2.2, 2:7.2
WHILE STATEMENTS 2:7.2
WHILE-STATEMENT 2:7.2

NORSK REGNESENTRAL
NORWEGIAN COMPUTING CENTER



Revised pages

SIMULA®

USERS GUIDE
IBM System/360

UNIVERSITETSFORLAGET
Oslo - Bergen - Tromsø

® SIMULA is registered trademark of the Norwegian Computing Center

To make the use of subscripted variables very flexible, one is allowed to write arithmetic expressions in the subscript positions, so that if I = 2, for example, then LONGSIDE(2*I) and LONGSIDE(4) refer to the same quantity.

A CHARACTER ARRAY is used to represent the crossword. A blank is represented by the CHARACTER constant '␣' and a blocked out square by '*'. The declaration of the ARRAY is:

```
CHARACTER ARRAY CWORD (0:N+1, 0:N+1)
```

representing the crossword (dimensions 1 through N), and the rim (dimensions 0 and N+1). This will make available $(N+2)^2$ variables of type CHARACTER each initialised to the initial value '^', binary zero (CHAR(0)). N.B. several CHARACTERS are not visible when printed - this is one of them. It is not the same CHARACTER as blank. To distinguish, blank is written '␣'.

The strategy is (in words):

1) N, the number of rows and columns of the crossword itself is given on the first card

2) Read in the data from cards - each row being represented on a fresh card

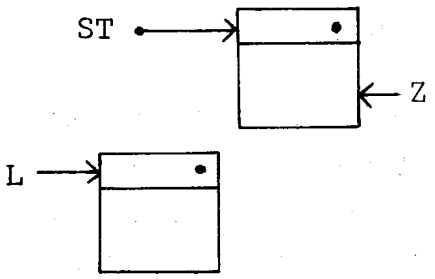
   Characters may be read in one by one by successive calls on INCHAR.

The coding is:

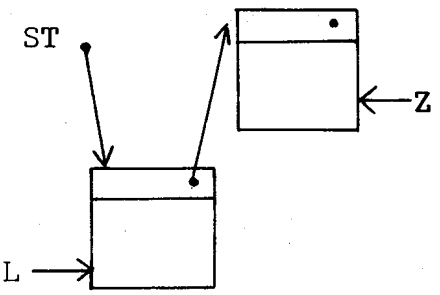
```
L.NEXT :- ST;
ST :- L;
COMMENT ***NOW L IS INSERTED*** ;
C.NEXT :- ST;
ST :- C;
COMMENT ***NOW C IS INSERTED*** ;
```

Snapshots before and after the insertion of L are:

before



after



The coding is a repeat except for a parameter saying which object is to be inserted. The PROCEDURE concept enables us to give CLASS STACKABLE procedure attribute "INTO", and then we can write

```
L.INTO (ST)
C.INTO (ST)
```

for the same effect. This is not only more concise, but more readable than before. This time we have to be rather careful

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)

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

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

Where the strings "OF TRUE PART" and "OF FALSE 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;

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|_]...

Examples:

valid identifiers

X
SIMULA_67
A15
MORGAN_PLUS_4
APPLE-

invalid identifiers

END	reserved for use as a keyword
SYM␣BOL	contains a blank
3C	does not begin with a letter

(SHORT)INTEGER-constant

decimal-digits

The range of values is the set of whole numbers from 0 through $2^{31}-1$ (= 2147483447). 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

TEXT-constants have the form

"{any sequence of members of the data character set}"

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

If one wishes to represent a text quote inside a TEXT-constant, it is represented by "" - two adjacent text quotes. Notice that """" has length 1, """""" has length 2 etc.

Each variable declared in a type declaration has an initial value (given in the table below). Thereafter the value of a variable is the one last assigned to it, or if no assignment has yet been made, the initial value.

Type	Initial value	Assignable range
INTEGER	0	Whole number in the range -2^{31} through $2^{31}-1$.
SHORT INTEGER	0	Whole number in the range -2^{15} through $2^{15}-1$.
REAL	0.0	Number in the range $\pm(0 \rightarrow 10^{75}$ approx.) to ∿7 decimal places.
LONG REAL	0.0	Number in the range $\pm(0 \rightarrow 10^{75}$ approx.) to ∿16 decimal places.
BOOLEAN	FALSE	TRUE, FALSE.
CHARACTER	CHAR(0)	CHAR(0), CHAR(1), ... CHAR(255).
REF(CLASS-identifier)	NONE	NONE or any object of the qualifying class or included in the qualifying class.
TEXT	NOTEXT	NOTEXT or any string of characters from the data character set of length 0 through $(2^{15}-20)$ characters.

function-declaration

```
REF(POINT) PROCEDURE ADD(Q); REF(POINT)Q;
   IF Q =/= NONE THEN ADD :- NEW POINT(X+Q.X,Y+Q.Y)

REAL PROCEDURE NORM(A,N); REAL ARRAY A; INTEGER N;
BEGIN  REAL T; INTEGER I;
       FOR I := 1 STEP 1 UNTIL N DO
           T := T + A(I)**2;
       NORM := SQRT(T)
END ***NORM***

INTEGER PROCEDURE FACTORIAL(N); INTEGER N;
IF N < 0 THEN ERROR ELSE
   IF N < 2 THEN FACTORIAL := 1
            ELSE FACTORIAL := N*FACTORIAL(N-1)
```

A function returns a value of the type indicated in its declaration, and may be used wherever a value of that type is legal. (It may also be used as a statement in which case the function value is ignored);

```
P :- R.ADD(S)

X := NORM(MATRIX, 10)

IF NORM (MATRIX, 10) < &-6 THEN
   OUTTEXT ("ELEMENTS␣ALL␣ZERO")
```

Call by name

Call by name is an optional transmission mode available for parameters to procedures, but not to classes. It represents a textual replacement in that the formal-parameter may be considered replaced throughout the PROCEDURE-body by the corresponding actual-parameter.

Whereas call by value and call by reference operate on variables local to the PROCEDURE-body itself, call by name operates on non-local quantities and can alter global quantities. It is therefore especially useful for the controlled alteration of several variables external to the PROCEDURE-body.

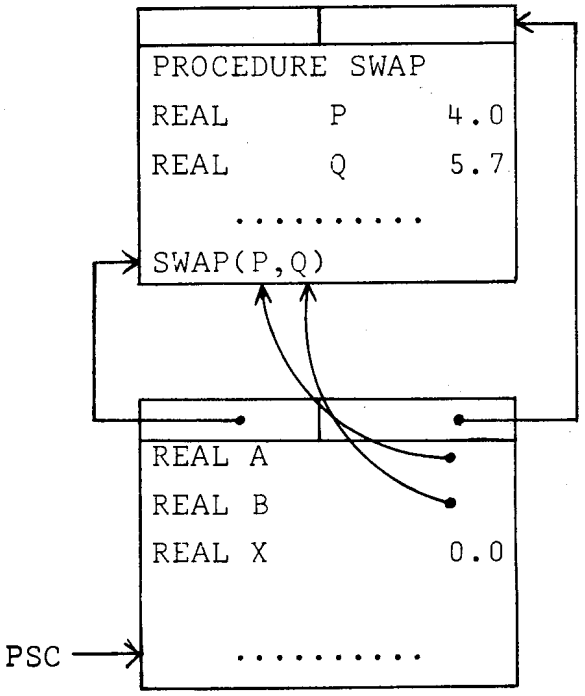
The following rules apply:

1) the type of a name parameter is that prescribed by the corresponding formal specification.

2) if the type of the actual-parameter does not coincide with that of the formal specification, then an evaluation of the expression is followed by an assignment of the value or reference obtained to a fictitious variable of the latter type. This assignment is subject to the rules of section 2.7.2. The value or reference obtained by the evaluation is the contents of the fictitious variable.

Section 2.7.2 defines the meaning of an assignment to a variable which is a formal-parameter called by name, or is a subscripted variable whose array identifier is a formal-parameter called by name, if the type of the actual parameter does not coincide with that of the formal specification.

A snapshot at the procedure call is:



Program Sequence Control (PSC) references the current statement

No local copies are made. Every occurrence of A or B in the PROCEDURE-body means a re-evaluation of the actual-parameter. Notice that the actual-parameters are evaluated in the context of the procedure call.

The result of executing the call on SWAP is to set the value of P to 5.7 and the value of Q to 4.0.

The available transmission modes for legal parameters to classes are shown in the following table:

CLASS PARAMETERS

Parameter	Transmission	
	call by value	call by reference
value-type	D	X
reference-type	X	D
TEXT	O	D
value-type ARRAY	O	D
reference-type ARRAY	X	D

D : default mode
O : optional
X : illegal

The transmission modes "call by value" and "call by reference" are explained in section 2.5.4.

The discussion of CLASS-declarations begins by considering two selected examples of increasing scope.

The expression

NEW A(...)

creates an object of the class A (the order and number of the actual-parameters must correspond with the order and number of the formal-parameters) and commences execution of its actions. The execution continues until the final END of the class body is encountered, when the execution is terminated. However this may be interrupted in four ways - by a GOTO-statement (which leads out of the object), or by calls on the system procedures "DETACH", "RESUME" or "CALL". A GOTO exit will leave the object in the terminated state. "Detach" suspends the actions of the class body and names it an independent component of the program. Its actions may be continued later by a call on "resume". The object is "attached" when entered on generation by execution of NEW or when it is the subject of a call on CALL.

5.6 EXTERNAL DECLARATIONS

External declarations allow the inclusion in a program of previously compiled procedures and classes. By judicious use of this feature a programmer can build up a library of building blocks on which to base future programs.

External FORTRAN and Assembly procedure declarations may also be important for the following reasons:

1) efficiency. Minimal coding may be achieved in critical regions of a program.

2) features of a machine which are hidden by SIMULA may be accessed.

3) routines (already written) in other languages may be incorporated into SIMULA programs.

external-declaration

EXTERNAL { [FORTRAN] [ASSEMBLY] [type] PROCEDURE } id-list
{ CLASS }

Examples

```
EXTERNAL ASSEMBLY INTEGER PROCEDURE BITS_ON
EXTERNAL REAL PROCEDURE MIN, MAX
EXTERNAL PROCEDURE READ, WRITE
EXTERNAL FORTRAN PROCEDURE FORMAT
EXTERNAL CLASS DRAUGHTING
```

Preparation of external procedures

In order to prepare external procedures written in SIMULA, they should be compiled one at a time by compiling with PARM = EXTERN or PARM = 'EXTERN = P'. Note that the syntax of a separately compiled SIMULA procedure is identical with that of a procedure declaration, i.e. the program does not start with BEGIN! Separately compiled procedures are placed in a user specified library (see "SIMULA Programmer's Guide IBM System 360", section 2.4, p. 30 for examples).

An external procedure may be saved as an object module or as a load module.

Note the following restrictions:

1) FORTRAN, ASSEMBLY procedures

Parameter checking is the responsibility of the writer of the external procedure since it cannot be wholly done by the compiler. In FORTRAN procedures, only FORTRAN parameters may be transmitted.

In ASSEMBLY procedures, only literals, simple or array variables are allowed as actual parameters. For details concerning parameter retrieval and other aspects relevant to the writing of FORTRAN and ASSEMBLY procedures, consult Programmer's Guide, Appendix G.

2) External procedures may not contain local classes and object reference-variables.

Preparation of external classes

In order to prepare external classes, they should be compiled one at a time by compiling with PARM = 'EXTERN = C'. Note that the syntax of a separately compiled class is identical to that of a class declaration. Separately compiled classes are placed in a user specified source module library with DD-name SYSPUNCH.

Note the following restriction:

A separately compiled class must not contain either COPY code, or other external class declarations.

Use of externally compiled procedures and classes

Retrieving an externally compiled class is achieved by specifying the library containing it as a data set with DDNAME SYSLIB to the SIMULA compiler.

The retrieval of an externally compiled procedure is achieved by specifying the library containing it as a data set concatenated with SYSLIB of the loader or linkage editor.

Let X be a simple object expression qualified by class C, and A an attribute identifier.

The X.A is valid if X =/= NONE and class C has at least one attribute identified by A. If there is only one attribute with identifier A contained in class C or in its prefix chain, then it is designated by X.A. If class C contains more than one attribute with identifier A, then X.A designates the attribute with identifier A at the innermost prefix level.

N.B. The main part of any class declaration can contain at most one attribute with that identifier.

Example:

```
    REF(C1)X1; REF(C2)X2; REF(C3)X3; REF(C4)X4

    CLASS C1(A); REAL A;
    BEGIN ..................... END;
C1  CLASS C2;
    BEGIN  TEXT A; ... END;
C2  CLASS C3;
    BEGIN  ..................... END;
C3  CLASS C4;
    BEGIN  BOOLEAN A; ........ END;
X1 :- X2 :- X3 :- X4 :- NEW C4(6.0);
```

6.2 FUNCTION DESIGNATORS

A function designator denotes the value obtained by evaluating the associated procedure body when supplied with the associated actual parameter list (if any).

function-designator

{identifier / remote-identifier}[actual-parameter-list]

actual-parameter-list

(actual-parameter[, actual-parameter]...)

actual-parameter

{expression / ARRAY-identifier / SWITCH-identifier / PROCEDURE-identifier / LABEL-identifier}

Examples:

```
ININT
INTEXT(20)
SIN(A+B)
H.FIRST
CURRENT.NEXTEV.SUC
POINT(5).MODULUS
```

Two text values are equal if they are both empty or if they are both instances of the same character sequence. Otherwise they are unequal, and then a text value T ranks lower than a text value U if one of the following conditions is fulfilled.

1) T is empty
2) U is equal to T followed by one or more characters
3) If the first i-1 characters of T and U are the same, and the ith character of T ranks lower than the ith character of U.

Examples:

```
NOTEXT = ""          TRUE
"0"  <  "9"          TRUE
"ABCDE"="ABCDEF"     FALSE
"+12" =/= "=12"      TRUE
"ABC"  =  "ABC␣"     FALSE
```

When a block is prefixed, the identifiers declared in the corresponding CLASS are made available. Nevertheless, an identifier in the CLASS may be redefined in the main-block or compound-statement.

The execution of a block is as follows:

step 1: if the block is prefixed then the actual parameters if any are evaluated.

2: if the declarations of the block contain array bounds then these are evaluated. (They may make reference to parameters of the prefix).

3: Execution of the statements of the block body begins with the first statement of the prefix, if any, otherwise with the first statement of the main block. After execution of the block body (unless it is a GOTO statement) a block exit occurs and the statement textually following the entire block is executed.

A CLASS identifier possibly followed by an actual parameter list can prefix a main-block or compound-statement. This results in concatenating the object of the stated class with that main-block or compound-statement, which means that the capabilities of the stated class and its including classes are available within that main-block or compound statement.

When an instance of a prefixed block is generated, the formal parameters of the class are initialised as indicated by the actual parameters of the block prefix. A virtual quantity is identified by the quantity defined by a matching declaration in the block head of the main-block or compound-statement, or by the matching definition at the innermost prefix level of the prefix sequence. The operation rule of the concatenated object is defined by principles similar to those given in section 5.5.

reference-assignment

{variable :- / PROCEDURE-identifier :-} ... object-expression / TEXT-expression

Examples:

```
P.X[1] :- NONE
P :- Q :- THIS POINT.SUC
ADD :- NEW POINT
THIS LINE.P :- NONE
T :- BLANKS(80)
S :- R :- T.SUB(1,20)
```

Consider the object-reference-assignment

V :- object-expression

Both the left part and the right part have a qualification. Let these be Q_ℓ and Q_r respectively - note that NONE is here considered as having a universal qualification which is inner to every other qualification. The situations that can arise are illustrated in the context:

```
REF(CHAIN)C;      CLASS CHAIN.....;
REF(MEMBER)M;     CLASS MEMBER.....;
REF(POINT)P;      MEMBER CLASS POINT....;
REF(LINE)L;       MEMBER CLASS LINE.....;
```

Case 1: Q_ℓ is equal or outer to Q_r

e.g.

```
A)    M :- NEW MEMBER;
B)    M :- NEW POINT;
C)    M :- P;
D)    M :- NONE;
```

The assignment is legal. In B), the attributes of the referenced POINT-object may be accessed by use of QUA as in

```
M QUA POINT.X
```

or by use of INSPECT as in either of

```
INSPECT M QUA POINT DO ... X ...
INSPECT M WHEN POINT DO ... X ...
```

Case 2: Q_ℓ is inner to Q_r

e.g.

```
A)      P :- NEW MEMBER;
B)      P :- M;
```

The assignment may be legal. A) is clearly not, but B) is legal if M is currently referencing a POINT object or an object of a class inner to POINT, e.g. after

```
M :- NEW POINT...;
```

This must be checked at runtime.

In cases 1 and 2, the qualifications Q_ℓ and Q_r are said to be compatible. Notice that compatibility is decided at compile time.

Case 3: Q_ℓ and Q_r are not compatible.

e.g.

```
A)      P :- NEW LINE.....;
B)      P :- NEW CHAIN;
C)      M :- C;
```

The assignment is illegal.

Similar rules apply to the object-reference assignments implicit in the passing of parameters (FP :- AP) and assignments to the result variable in a function procedure body.

Multiple assignments take the form

v_1 :- v_2 :- .... :- v_n :- object-expression

and are equivalent to

v_n :- object expression,

v_{n-1} :- v_n;

.....

v_1 :- v_2;

The considerations above apply at each step.

The fact that an object-reference-assignment is always checked for legality (mainly at compile time) has the following implication. For any object-reference-variable or indeed, object-expression whose value is V and with qualification Q_v, the following is always true under program execution:

(V == NONE) OR (V IN Q_v)

Example:

The program below prints out the first three verses of "The Twelve Days of Christmas". The program logic is built around a SWITCH.

```
BEGIN  SWITCH CASE := LINE1, LINE2, LINE3;
       INTEGER VERSE;
       TEXT ARRAY T(1:3);
       T(1) :- COPY("FIRST");
       T(2) :- COPY("SECOND");
       T(3) :- COPY("THIRD");

IF_WE_USE_FOR_HERE_THE_LABELS_BECOME_INVISIBLE:
       VERSE := VERSE + 1;
       OUTTEXT("ON␣THE␣"); OUTTEXT(T(VERSE));
       OUTTEXT("␣DAY␣OF␣CHRISTMAS"); OUTIMAGE;
       OUTTEXT("MY␣TRUE␣LOVE␣SENT␣TO␣ME"); OUTIMAGE;
       GOTO CASE(VERSE);
LINE3: OUTTEXT("THREE␣FRENCH␣HENS"); OUTIMAGE;
LINE2: OUTTEXT("TWO␣TURTLE␣DOVES,␣AND"); OUTIMAGE;
LINE1: OUTTEXT("A␣PARTRIDGE␣IN␣A␣PEAR␣TREE"); OUTIMAGE;
       EJECT(LINE + 2);
       IF VERSE < 3 THEN
          GOTO IF_WE_USE_FOR_HERE_THE_LABELS_BECOME_INVISIBLE;
END
```

DUMMY STATEMENTS

A dummy-statement executes no actions. Its main use is to place a label before an END

dummy-statement

Examples:

```
IF X > 0 THEN ELSE X := -X;

BEGIN ..... LAB : END;
```

α, β, σ are different identifiers which are not used elsewhere in the program. σ identifies a non-local simple variable of the same type as A_2.

1. V

```
    C := V;
    S;
    next for list element
```

2. A_1 STEP A_2 UNTIL A_3

```
    C := A1;
    σ := A2;
α : IF   σ*(C-A3) > 0 THEN GOTO β;
    S;
    σ := A2;
    C := C + σ;
    GOTO α;
β : next for list element
```

3. V WHILE B

```
α : C := V;
    IF ¬ B THEN GOTO β;
    S;
    GOTO α;
β : next for list element
```

4. 0

```
    C :- 0;
    S;
    next for list element
```

5. 0 WHILE B

```
α : C :- 0;
    IF ¬ B THEN GOTO β;
    S;
    GOTO α;
β : next for list element
```

PROCEDURE STATEMENTS

A procedure-statement calls for the execution of the PROCEDURE-body. Before execution, the formal-parameters of the procedure are replaced by the actual-parameters.

PROCEDURE-statement

[simple-object-expression.
 simple-TEXT-expression.] PROCEDURE-identifier [(expression
[, expression]...)]

Examples:

```
INTO(H)
OUTTEXT("***0***")
SYSIN.INIMAGE
```

The procedure statement must have the same number of actual parameters in the same order as the formal-parameters of the procedure heading.

Restrictions

1) An actual-parameter corresponding to a formal-parameter called by NAME which is assigned to within the PROCEDURE-body must be a variable.

2) If the formal-parameter is an ARRAY (PROCEDURE), then the number of dimensions (actual-parameters) used within the PROCEDURE-body must correspond to the number of dimensions (actual-parameters) of the actual ARRAY (PROCEDURE).

Connection statement

```
                                        statement
INSPECT object-reference DO {[WHEN CLASS-identifier DO statement]...}
                                    [OTHERWISE statement]
```

Examples:

```
INSPECT SYSOUT DO
BEGIN  OUTTEXT("TITLE");
       OUTIMAGE;
       EJECT(LINE+10);
END;

INSPECT X WHEN A DO OUTTEXT("X IN A")
          WHEN B DO OUTTEXT("X IN B")
          OTHERWISE OUTTEXT("ERROR");

INSPECT X DO
  INSPECT Y DO P := Q
            OTHERWISE OUTTEXT("Y==NONE");
```

To avoid ambiguity, an OTHERWISE refers back to the nearest INSPECT.

The remote accessing of objects of classes may be accomplished by the dot notation or by connection. In most cases the methods are interchangeable, but if the object contains a class attribute at a certain level, then attributes at that level and levels inner to it can only be accessed by connection.

In addition to sequencing PROCEDURES, "DETACH" and "RESUME", there is the PROCEDURE CALL which has one reference parameter which must be a reference to a detached object.

The execution of CALL(Y) from within a block X, will "attach" the detached object Y to X and continue execution of the actions of Y.

The detailed description of program sequencing given in the "SIMULA 67 Common Base Language" is not repeated here. Further enquires are directed to that document §9.

```
CLASS HAND;
BEGIN  PROCEDURE PLACE(C); REF(CARD)C;
       BEGIN  REF(HEAD)S; REF(CARD)X;
              S :- SUIT(C.COLOUR);
              IF ¬ S.EMPTY THEN
              BEGIN  X :- S.FIRST QUA CARD;
                     WHILE X =/= NONE DO
                     BEGIN  IF X.RANK > C.RANK THEN
                                BEGIN  C.PRECEDE(X);
                                       GOTO L;
                                END;
                            X :- X.SUC;
                     END;
              END;
              COMMENT ***WE ENTER HERE IF S IS EMPTY OR
                         IF C.RANK IS THE HIGHEST MET
                         SO FAR*** ;
              C.INTO(S);
   L:  END ***PLACE*** ;

       REF(HEAD) ARRAY SUIT(1:4);
       SUIT(1) :- NEW HEAD;
       SUIT(2) :- NEW HEAD;
       SUIT(3) :- NEW HEAD;
       SUIT(4) :- NEW HEAD;
END ***HAND*** ;
```

actions are executed. When the active phase is over, that PROCESS may be rescheduled for a later active phase (for example, by REACTIVATE or HOLD) or removed from the timing tree (by PASSIVATE or WAIT). It is apparent that RESUME is too primitive for this purpose as it involves rescheduling or removing EVENT NOTICES as well as switching the PSC from one PROCESS object to another.

However RESUME and DETACH do form the basis for the scheduling procedures. To prevent the user from destroying system security, event notices may not be explicitly referenced by the user - he must use the system procedures for scheduling or rescheduling. In addition, it is strongly recommended that explicit use of "DETACH", "RESUME" and "CALL" be avoided within a SIMULATION block.

There is one special PROCESS object which plays a key role in any SIMULATION - one referenced by MAIN. Whenever MAIN becomes CURRENT, it causes the actions of the SIMULATION block itself to be continued. The corresponding event notice can then be rescheduled (typically by a call on HOLD) and then the action switches from the SIMULATION block to the new CURRENT. Thus the SIMULATION block is itself treated as a program component during the SIMULATION.

PROCEDURE CANCEL(X); REF(PROCESS)X;.....;

CANCEL(P) where P is a reference to a PROCESS object will delete the corresponding event notice if any. If P is currently active or suspended, it thus becomes passive. If P is a reference to a passive or terminated PROCESS object or NONE, CANCEL(P) has no effect. Thus CANCEL(CURRENT) is equivalent to PASSIVATE.

PROCEDURE ACTIVATE

For user convenience, calls on the procedure ACTIVATE are written in terms of the corresponding activation-statements.

activation-statement

```
{ ACTIVATE   }
{ REACTIVATE } PROCESS-expression1 [{AT|DELAY} time [PRIOR]                    ]
                                   [{BEFORE|AFTER} PROCESS-expression2         ]
```

Let X be the value of PROCESS-expression1. If the activator ACTIVATE is used, then the activation-statement will have no effect (other than evaluating X) unless X is passive. If the activator REACTIVATE is used, then X may be active, suspended, or passive (in which latter case, the activation-statement acts as an ACTIVATE statement).

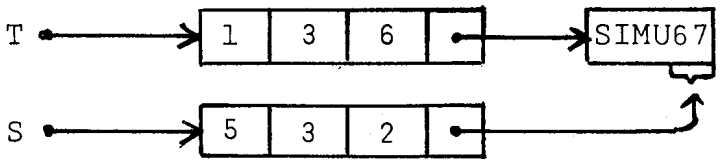
The type of scheduling is determined by the scheduling clause.

A complete description of the program now follows:

```
BEGIN  INTEGER POPULATION, LENGTH, CONTACTS, INCUBATION,
               U1, U2, U3, U4, UNINFECTED;
       REAL PRINF, PROBMASS, SIMPERIOD;
       COMMENT ***THE RANDOM STREAM NUMBERS ARE READ IN*** ;
       U1 := ININT; U2 := ININT;
       U3 := ININT; U4 := ININT;
       UNINFECTED := POPULATION := ININT;
       INCUBATION := ININT; LENGTH := ININT;
       CONTACTS := ININT;
       SIMPERIOD := INREAL;
       PRINF := INREAL; PROBMASS := INREAL;
SIMULATION BEGIN  REAL ARRAY PROBTREAT(1:LENGTH);
                  PROCESS CLASS SICKP;
                  BEGIN  INTEGER DAY;
                         BOOLEAN SYMPTOMS;
                         REF(HEAD)ENV;
                         PROCEDURE INFECT(N); INTEGER N;
                         BEGIN  INTEGER I;
                                FOR J := 1 STEP 1 UNTIL N DO
                                    IF DRAW(PRINF*UNINFECTED/
                                            POPULATION,U3) THEN
                                    BEGIN  NEW SICKP.INTO(ENV);
                                           ACTIVATE ENV.LAST;
                                    END;
                         END ***INFECT*** ;
```

After S.PUTCHAR('6');
S.PUTCHAR('7')

the snapshot is



Note that the value of T has been changed. The CP of S is now out of range. A further call

S.PUTCHAR
or S.GETCHAR

will result in a run time error. To provide a check, a BOOLEAN PROCEDURE MORE is provided which returns FALSE if the CP is out of range and TRUE otherwise. Currently,

T.MORE = TRUE S.MORE = FALSE

Other useful system defined procedures are:

LENGTH which returns the length of the currently referenced value
(T.LENGTH = 6
S.LENGTH = 2)

POS which returns the value of the CP
(T.POS = 3
S.POS = 3)

CHARACTER PROCEDURE GETCHAR;

The value of X.GETCHAR is a copy of the currently accessible CHARACTER of X provided X.MORE is TRUE. In addition, the position indicator of X is then increased by one. A run time error results if X.MORE is FALSE.

PROCEDURE PUTCHAR(C); CHARACTER C;

The effect of X.PUTCHAR(C) is to replace the currently accessible CHARACTER of X by the value of C provided that X.MORE is TRUE. In addition the position indicator of X is then increased by one. If X.MORE is FALSE, a run time error results.

Example:

The PROCEDURE COMPRESS rearranges the CHARACTERS of the TEXT object referenced by the actual parameter by collecting non-blank CHARACTERS in the leftmost part of the TEXT object and filling in the remainder, if any, with blanks. Since the parameter is called by reference (and not by name), its position indicator is unaltered.

```
PROCEDURE COMPRESS(T); TEXT T;
BEGIN  TEXT U; CHARACTER C;
       T.SETPOS(1); U :- T;
MOVELEFT:  WHILE U.MORE DO
           BEGIN  C := U.GETCHAR;
                  IF C ¬= '␣' THEN T.PUTCHAR(C);
           END;
           COMMENT ***WE NOW FILL IN THE RIGHT WITH
                      BLANKS*** ;
           T.SUB(T.POS,T.LENGTH-T.POS+1) := NOTEXT;
END ***COMPRESS***
```

b) TEXT-value-assignment

Consider the value assignment

```
T := P;
```

let the length of T be L1, and the length of the right part be a TEXT value of length Lr. There are three cases to consider:

L1 = Lr:	the character contents of the right part TEXT are copied to the left part TEXT
L1 > Lr:	the character contents of the left part are copied into the leftmost Lr characters of the left part TEXT, whose remaining L1-Lr CHARACTERS are filled with blanks.
L1 < Lr:	a run time error results.

After
```
T :- COPY("EIGHT␣CHARS");
T := "WRONG:11";
```

then
```
T = "WRONG:11␣␣␣␣"
```

Note that

```
T := NOTEXT;
```

would set all the character positions of T to blanks.

In a multiple TEXT value assignment

```
T1 := T2 := ..... TN := P;
```

then

T_J.LENGTH >= T_{J+1}.LENGTH

for J = 1,2,...,N-1

De-editing prccedures

A de-editing procedure operating on a given TEXT reference X operates in the following way:

1) the longest numeric item of the given form is located, contained within X and containing the first character of X. If such an item can be found, a run time error results.

2) the numeric item is interpreted as a number. If it is outside the accepted range (see PART 2, section 5.1 a run time error results.

3) the position indicator of X is made one greater than the last character of the numeric item.

N.B. Unless otherwise stated, the de-editing procedures are illustrated in the context:

```
T :- COPY("1234.5+7.3&4AB");
S :- T.SUB(7,6);
R :- T.SUB(5,2);
```

INTEGER PROCEDURE GETINT;

Locates an integer-item.

```
T.GETINT = 1234
S.GETINT = 7
R.GETINT   causes a run time error
```

Editing procedures

Editing procedures in a given text reference X convert arithmetic values to numeric items. After an editing operation, the numeric item obtained is right adjusted in the TEXT X preceded by padding blanks. The final value of the position indicator is X.LENGTH+1.

A positive number is edited with no sign. If X == NOTEXT then a run time error results, otherwise if X is too short to contain the numeric item, an edit overflow is caused (X is filled with asterisks) and a warning message is given at the end of program execution.

```
Let     T = BLANKS(10);
```

PROCEDURE PUTINT(I); INTEGER I;

T.PUTINT(VAL) converts the value of the parameter to an integer-item of the designated value.

```
T.PUTINT(-37)        ␣␣␣␣␣␣␣-37
T.PUTINT(118.8)      ␣␣␣␣␣␣␣119
```

PROCEDURE PUTFIX(R,N); REAL R; INTEGER N;

T.PUTFIX(VAL,M) results in an integer-item of M=0, or a real-item (with no exponent) if M>1 with M digits after the decimal point. It designates a number equal in value to VAL rounded to M decimal places. A run time error results if M<0.

```
T.PUTFIX(18,0)       ␣␣␣␣␣␣␣␣18
T.PUTFIX(-1375.4,3)  ␣-1375.400
```

The PROCEDURES OPEN and CLOSE, which are specified as VIRTUAL but have no matching declaration at the "file" level, complete the definition of CLASS file. The matching PROCEDURES declared in the subclasses of "file" conform to the patterns below with possible minor variations depending upon the subclass. The variations are listed in the appropriate following sub-sections.

The PROCEDURE outlines are:

```
PROCEDURE OPEN(BUF); TEXT BUF;
BEGIN  IF "open" THEN ERROR;
       IMAGE :- BUF;
END
PROCEDURE CLOSE;
BEGIN .........
       IMAGE :- NOTEXT;
END
```

No information can be processed through a "file" object until it has not only been generated but also opened. This can only be achieved by a call on the PROCEDURE OPEN whose actual parameter is assigned to IMAGE and acts as the buffer. A call on OPEN when a "file" is already open gives a run time error.

The PROCEDURE CLOSE closes a file and releases the buffer (by the assignment IMAGE :- NOTEXT). No information may be transmitted through a closed "file" object, but it may be opened again by a further call on OPEN.

than IMAGE, it is left adjusted and the remainder of IMAGE is filled with blanks. Finally the position indicator of IMAGE is set to 1. When the last record has been read in, and INIMAGE is called again, a call on ENDFILE will return TRUE. Any further call on INIMAGE, INCHAR, INTEXT, ININT, INREAL or INFRAC will result in a run time error.

BOOLEAN PROCEDURE LASTITEM;

returns FALSE only if the external file contains more information (non-blank CHARACTERS). It scans past all blank CHARACTERS (calling INIMAGE if need be). If LASTITEM returns FALSE then the currently accessible CHARACTER of IMAGE is the first non-blank CHARACTER. If ENDFILE returns TRUE, a call on LASTITEM also returns TRUE.

CHARACTER PROCEDURE INCHAR;

gives access to the next available CHARACTER and scans past it. If IMAGE.MORE is FALSE, then INIMAGE is called once and the value of the call is the first CHARACTER of the new image. INCHAR gives a run time error if an attempt is made to read past the last record in the file.

TEXT PROCEDURE INTEXT(W); INTEGER W;

INTEXT(M) creates a copy of the next M CHARACTERS (which may be spread over several records) and returns a reference to this copy. If M < 0 or M > 2^{15}-20 then a run time error results. A run time error will also result if the file does not contain M more CHARACTERS, i.e. an attempt is made to read past the last record.

The remaining PROCEDURES treat the file as a continuous stream of records. They scan past any number of blanks (calling INIMAGE if need be) and then de-edit a numeric item lying in one image. This is done by calling LASTITEM (which scans past the blanks)

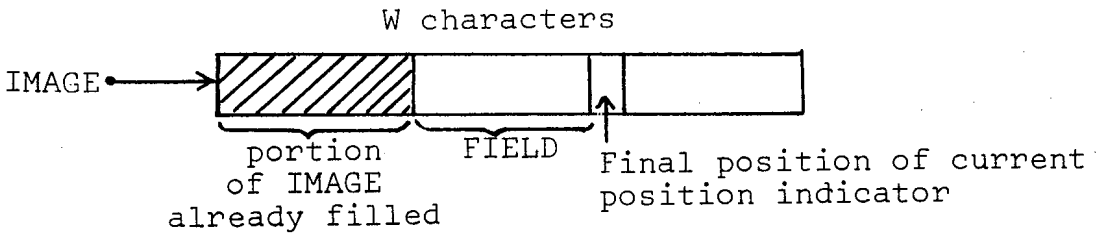
PROCEDURE OUTTEXT(T); VALUE T; TEXT T;

A copy of the CHARACTER sequence represented by the actual parameter is edited into IMAGE from the current position. If the remaining length of IMAGE is insufficient, OUTIMAGE is called and the editing process proceeds. Thus the TEXT value may be split over several external records.

PROCEDURE OUTCHAR(C); CHARACTER C;

Outputs the value of C into the current position of IMAGE (if MORE = FALSE, then OUTIMAGE is called first). In either case, the current position indicator is then incremented.

The remaining PROCEDURES are all based upon the PUT-PROCEDURES local to TEXTs. The corresponding OUT-PROCEDURES are augmented by an extra parameter W which specifies the field within.



The editing PROCEDURE commences by establishing a temporary TEXT reference (FIELD) to the next sequence of W CHARACTERS lying in one IMAGE. If the current IMAGE has not enough space left, OUTIMAGE is called. Then the value is edited by calling FIELD."PUT***" where "PUT***" is the PUT-PROCEDURE corresponding to the OUT-PROCEDURE. Finally the current position indicator is increased by W to reference past FIELD - past the just-edited field.

PROCEDURE EJECT(N); INTEGER N;

This PROCEDURE skips to a certain line on the page - (it avoids calling OUTIMAGE several times). EJECT(L) will position to line L on this page if this is further down the current page (if L > LINE), or else skip to LINE L of the next page if L <= LINE.

A run time error occurs if L <=0. If L > LINESPERPAGE, EJECT(L) is equivalent to EJECT(1).

INTEGER PROCEDURE LINE;

This PROCEDURE returns the INTEGER value of the line number which indicates the next line to be printed. Thus EJECT(LINE+3) will skip three lines and not alter spacing. After each call on OUTIMAGE, the line number is incremented by the current spacing.

PROCEDURE OUTIMAGE;

This PROCEDURE acts like the OUTIMAGE of OUTFILE but in addition increments the line number by spacing, and will position to the top of the next page if the current page is filled.

CLASS BASICIO

The system defined file facilities are grouped together in the CLASS BASICIO whose skeleton reads:

```
CLASS BASICIO(linelength); INTEGER linelength;
BEGIN  CLASS file.....................;
       file CLASS INFILE.............;
       file CLASS OUTFILE............;
       file CLASS DIRECTFILE.........;
       file CLASS PRINTFILE..........;
       REF(INFILE)sysin;
       REF(PRINTFILE)sysout;
       REF(INFILE) PROCEDURE SYSIN; SYSIN :- sysin;
       REF(PRINTFILE) PROCEDURE SYSOUT; SYSOUT :- sysout;
       sysin :- NEW INFILE("SYSIN");
       sysin.OPEN(BLANKS(80));
       sysout :- NEW PRINTFILE("SYSOUT");
       sysout.OPEN(BLANKS(linelength));
       INNER;
       sysin.CLOSE; sysout.CLOSE;
END ***BASICIO***
```

When the actions of the user defined program are exhausted, control returns to the prefix level of the BASICIO object and continues after the INNER. The following three statements close the three system generated files.

The inspect statements enclosing the program allow the user to write ININT, INIMAGE,..... instead of SYSIN.ININT, SYSIN.IMAGE and OUTREAL, OUTIMAGE,.... instead of SYSOUT.OUTREAL, SYSOUT.OUTIMAGE. There are attribute name clashes

OPEN, CLOSE } which should never be used for SYSIN or SYSOUT

IMAGE
SETPOS
POS
MORE
LENGTH

When these occur they are naturally bound to SYSOUT and the corresponding attributes of SYSIN may be obtained by writing SYSIN.SETPOS, SYSIN.IMAGE etc. Alternatively, an input section may be written as

```
INSPECT SYSIN DO
BEGIN
        input - in this block occurrences IMAGE, SETPOS,
        POS, MORE and LENGTH are bound to SYSIN
END;
```

Arithmetic functions

Certain identifiers, expressed as procedures are defined by the Simula system for standard arithmetic functions.

ABS(E)	modulus (absolute value) of E
ARCCOS(E) ARCSIN(E) ARCTAN(E)	return the principal values of the arc-cosine, arc-sine, arc-tangent of E (E is measured in radians)
COS(E) SIN(E) TAN(E)	return the cosine, sine, tangent of E (E is measured in radians)
COSH(E) SINH(E) TANH(E)	return the hyperbolic cosine, hyperbolic sine, hyperbolic tangent of E (E is measured in radians)
EXP(E)	exponential function of E (e^E)
LN(E)	natural logarithm of E ($\log_e E$, or ln E). If E <= 0, a run time error results.
SQRT(E)	returns the square root of E if E >= 0. If E < 0, a run time error results.

The above 13 functions operate on arithmetic arguments. If the type of E is [SHORT]INTEGER or REAL, then the function value is of type REAL. If the type of E is LONG REAL, then the function value is of type LONG REAL.

MOD(M,N) M modulo N, that is

M - ((M//N)*N)

e.g. MOD (7,3) is 1
MOD (-48,5) is 2

The function operates on [SHORT]INTEGER arguments, LONG REAL arguments being rounded. The result is [SHORT]INTEGER.

Utility procedures

PROCEDURE HISTO(A,B,C,D);{INTEGER / REAL}ARRAY A,B;
{INTEGER / REAL}C,D;

A call on HISTO updates a histogram defined by the one dimensional ARRAYS (INTEGER or REAL) A,B according to observation C with weight D. A(I) is incremented by D, where I is the smallest INTEGER such that C <= B(I). It is assumed that the length of A is one greater than the length of B. The last element of A corresponds to those observations which are greater than all the elements of B.

PROCEDURE LOWTEN(C); CHARACTER C;

Without use of LOWTEN, the CHARACTER 'E' represents the exponent sign in any numeric item to be edited or de-edited.
A call on "LOWTEN" with actual parameter "EXPSIGN" will replace 'E' by the value of EXPSIGN in future editing and de-editing.

SEQUENCING PROCEDURES

```
PROCEDURE CALL(X); REF(anyclass)X;
PROCEDURE DETACH;
PROCEDURE RESUME(Y); REF(anyclass)X;
```

RANDOM DRAWING PROCEDURES

```
INTEGER PROCEDURE DISCRETE(A,U); NAME U; ARRAY A; INTEGER U;
BOOLEAN PROCEDURE DRAW(A,U); NAME U; REAL A; INTEGER U;
   REAL PROCEDURE ERLANG(A,B,U); NAME U; REAL A,B; INTEGER U;
INTEGER PROCEDURE HISTD(A,U); NAME U; ARRAY A; INTEGER U;
   REAL PROCEDURE LINEAR(A,B,U); NAME U; ARRAY A,B; INTEGER U;
   REAL PROCEDURE NEGEXP(A,U); NAME U; REAL A; INTEGER U;
   REAL PROCEDURE NORMAL(A,B,U); NAME U; REAL A,B; INTEGER U;
INTEGER PROCEDURE POISSON(A,U); NAME U; REAL A; INTEGER U;
INTEGER PROCEDURE RANDINT(A,B,U); NAME U; INTEGER A,B,U;
   REAL PROCEDURE UNIFORM(A,B,U); NAME U; REAL A,B; INTEGER U;
```

UTILITY PROCEDURES

```
PROCEDURE HISTO(A,B,C,D); ARRAY A,B; REAL C,D;
PROCEDURE LOWTEN(C); CHARACTER C;
```