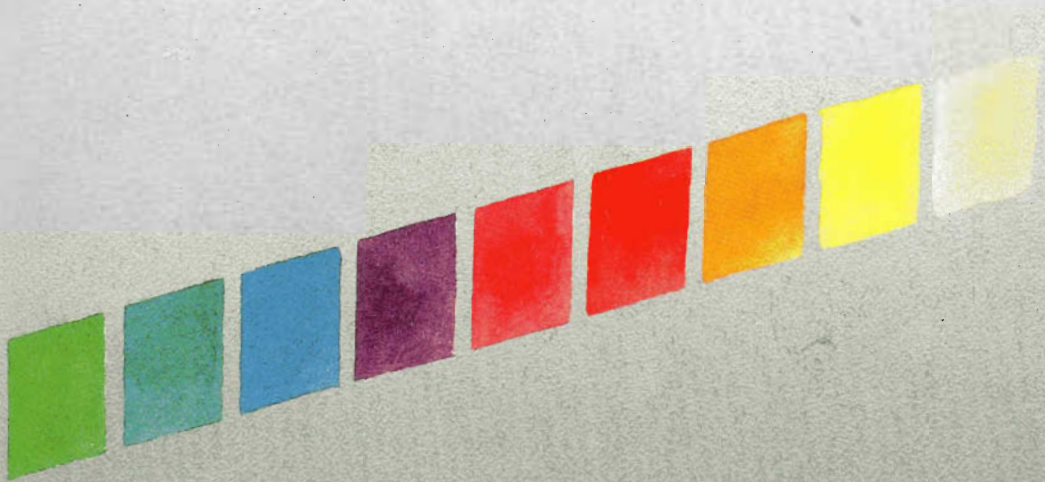
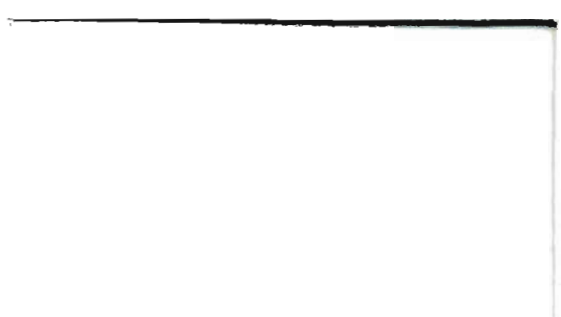


ND-500 SIMULA Reference Manual

ND-60.208.1 EN

Norsk Data



ND-500 SIMULA
Reference Manual

ND-60.208.1 EN

NOTICE

The information in this document is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this document. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

PRINTING RECORD

Printing	Notes
06/85	VERSION 01

ND-500 SIMULA Reference Manual
Publ.No. ND-60.208.1 EN

Norsk Data

Norsk Data A.S
Graphic Center
P.O.Box 25, Bogerud
0621 Oslo 6, Norway

Manuals can be updated in two ways, new versions and revisions. New versions consist of a complete new manual which replaces the old manual. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the Customer Support Information (CSI) and can be ordered as described below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

These forms and comments should be sent to:

Documentation Department
Norsk Data A.S
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Requests for documentation should be sent to the local ND office or (in Norway) to:

Graphic Center
Norsk Data A.S
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Preface:

THE PRODUCT

SIMULA 67 is an object-oriented high-level language for programming digital computers. It is especially suited for solving large and complex problems. This manual describes the language, and the facilities of the following compiler:

ND-500 SIMULA 67 ND 10354, Version A

THE READER

The reader should be engaged in writing SIMULA programs for ND-500 machines.

PREREQUISITE KNOWLEDGE

The reader must at minimum have a basic knowledge of data processing techniques and have some experience in programming some high level language. Some prior familiarity with SIMULA programming is also recommended.

HOW TO USE THE MANUAL

This manual provides a complete formal description of the features and facilities of ND-500 SIMULA. It is intended for reference purposes and is organized in a progressive fashion, i.e. the simpler structural components are explained first. It is not, however, explicitly designed for tutorial use. Thus a general SIMULA textbook which is organized tutorially is recommended to accompany this manual for beginning SIMULA programmers. It should also be noted that this version of the manual is a preliminary one, with some sections concerning more advanced features of the language yet to be written.

RELATED MANUALS

In order to extend the description of the SINTRAN environment in which the SIMULA programs operate, the following manuals are recommended:

ND Relocating Loader	-	ND-60.066
SINTRAN III Reference Manual	-	ND-60.128
ND-500 Loader/Monitor	-	ND-60.182

TABLE OF CONTENTS

Section Page

Section Page

Section Page

Section Page

CHAPTER 1

INTRODUCTION

1 INTRODUCTION

The SIMULA language described in this manual conforms to the SIMULA Standards Group's SIMULA 67, which is proposed as an International Standard. The full language is therefore implemented.

The ND-500 SIMULA implementation is based upon the portable SIMULA system, and is therefore highly compatible with other implementations based upon this system.

SIMULA is a registered trade mark of Simula A.S., Oslo, Norway.

1.1 THE NOTATION

The notation used to specify the syntax of the SIMULA language is based on the Backus-Naur Form. The meanings of the various constructs described in this formalism are given in prose and, in some cases, by equivalent program fragments. In such program fragments some identifiers are printed in upper case; the use of upper case letters signifies that the identifier in question represents some quantity which is inaccessible to a normal program.

Examples are given to illustrate the constructs; both in such examples and in the text the key-words of the language are underlined in order to emphasize their use.

The syntactic specification of the language is given as a set of rules, each defining a specific language construct. The general format of such a rule is

```
left-side
    =  first-alternative
    !  second-alternative
    !  ... etc.
```

Other rules will then define the format of "first-alternative", "second-alternative", etc. Whenever a terminal symbol occurs, it will be quoted in string quotes ("); symbols which are not quoted are

Metasymbol	Meaning
=	is
!	or
[x]	0 or 1 instance of x
(* x *)	0 or more instances of x
(x ! y)	grouping: either x or y
"xyz"	the terminal symbol xyz
meta-identifier	a non-terminal symbol

A meta-identifier is a sequence of letters, digits and hyphens beginning with a letter.

A sequence of terminal and non-terminal symbols in a language rule implies the concatenation of the text that they ultimately represent. Within this chapter the concatenation is direct, i.e. no characters intervene. In all other parts of this standard the concatenation is in accordance with the rules set out on page 34 et seq.

The characters required to form SIMULA programs shall be those implicitly required to form the tokens and separators defined below.

In order to illustrate the use of the syntactic formalism, language rules given in this chapter will be given both in the formal manner and in prose.

1.2 SIMULA CHARACTER SET

The SIMULA character set consists of twenty-six letters, ten digits, nineteen special characters, and some layout characters.

```
letter
  =  "a" ! "b" ! "c" ! "d" ! "e" ! "f" ! "g" ! "h" ! "i" ! "j" ! "k"
  !  "l" ! "m" ! "n" ! "o" ! "p" ! "q" ! "r" ! "s" ! "t" ! "u" ! "v"
  !  "w" ! "x" ! "y" ! "z"
```

This rule means that a letter is the letter "a", or "b", etc, i.e. it is simply one of the twenty-six characters of the English alphabet. Outside strings and character constants SIMULA does not distinguish between upper and lower case letters.

```
digit
  =  "0" ! "1" ! "2" ! "3" ! "4" ! "5" ! "6" ! "7" ! "8" ! "9"
```

This rule means that a digit is one of the ten decimal digits.

Special characters include the following nineteen characters:

```
! " & / ( ) = * ; : _ + ' , . - < > %
```

Note also the reserved use of % as the first character of directives to the compiler or of comment lines.

The layout characters are (ISO code value in parentheses)

blank (32) HT (9) CR (13) LF (10)

The collating sequence used is the ISO sequence. In addition, any printable character may occur within strings or character literals.

Note that the ISO and the ASCII coding are almost the same, apart from the interpretation of control characters, and from the so-called "national use positions" of the ISO code. Thus e.g. square brackets or backslash are not part of the SIMULA (ISO) standard character set.

1.3 SIMULA TERMS AND CONCEPTS

The basic language elements of SIMULA, i.e. lexical tokens, are keywords, special symbols, identifiers and literals. These are all formed from the character set defined previously in this chapter. No layout character may occur within the tokens, except as explicitly noted in this chapter, since layout characters act as delimiters.

The source text is written as a sequence of lines. In contrast to e.g. FORTRAN the line structure does not necessarily follow the structure of the language concepts, with one important exception. Any line which contains a % in column 1 is a directive line. Directive lines are treated in a later section in this chapter.

1.3.1 KEYWORDS

A keyword is a word that is recognised by the compiler. To emphasize their status as such, keywords are underlined throughout this manual. The SIMULA keywords are

ACTIVATE	LABEL
AFTER	LE
AND	LONG
ARRAY	LT
AT	NAME
BEFORE	NE
BEGIN	NEW
BOOLEAN	NONE
CHARACTER	NOT
CLASS	NOTEXT
COMMENT	OR
DELAY	OTHERWISE
DO	PRIOR
ELSE	PROCEDURE
END	PROTECTED
EQ	QUA
EQV	REACTIVATE
EXTERNAL	REAL
FALSE	REF
FOR	SHORT
GE	STEP
GO	SWITCH
GOTO	TEXT
GT	THEN
HIDDEN	THIS
IF	TO
IMP	TRUE
IN	UNTIL
INNER	VALUE
INSPECT	VIRTUAL
INTEGER	WHEN
IS	WHILE

1.3.2 SPECIAL SYMBOLS

The special symbols are formed from the special characters previously described. They are

Special symbol	Meaning
+	plus
-	minus
*	multiplied by
**	to the power of
/	divided by (REAL division)
//	divided by (INTEGER division with truncation)
&	times ten to the power of (REAL literals)
&&	times ten to the power of (LONG REAL literals)
:=	becomes
:-	denotes
<	is less than
<=	is less than or equal to
=	is equal to
>=	is greater than or equal
>	is greater than
<>	is different from
==	reference the same as
=/=	do not reference the same as
!	start of comment / ISO code quote
'	character quote
"	text quote
;	semicolon (1)
:	colon (2)
(	open parenthesis (3)
)	close parenthesis (3)
.	dot (attribute access)
,	comma (3)

Notes:

1) The semicolon is used for terminating statements, declarations, specifications, and comments.

2) The colon may be used for label definitions, and in array declarations.

3) Parentheses are used to group together parameters of a procedure, function, or class, or the indices of a subscripted variable. The comma is used for separating the different parameters or indices. Parentheses are also used to rearrange the order in which expressions are evaluated.

Use of the above representations for special symbols is recommended. In order to provide complete compatability with other implementations some of the special symbols in addition have a keyword representation as shown in the table below.

Recommended	Alternative representation
<	LT
<=	LE
=	EQ
>=	GE
>	GT
<>	NE
!	COMMENT (not as ISO code quote)

1.3.3 IDENTIFIERS

```
identifier
    =  letter  (* letter ! digit ! "_" *)
```

This rule means that an identifier is a sequence of letters, digits, and underscores, the first of which must be a letter. It can be used to identify a program quantity such as

a variable
an array
a function or procedure
a class
an attribute of a class
the kind of an external procedure
a constant

The scope of an identifier is determined by the occurrence of its declaration, as explained elsewhere in this manual (see pages 29 and 42).

The maximum length of an identifier is 72 characters. The keywords defined above cannot be used as identifiers. Note that all the constituent characters are significant, so that A and A_ identify different quantities.

1.3.4 LITERALS AND CONSTANTS

A constant is either a literal, or it is an identifier defined to be constant. It does not change its value during execution. For the definition of identifier constants, see page 43.

1.3.4.1 INTEGER LITERALS

```
unsigned-integer
    =  digit  (* digit *)
```

An integer literal consists of a string of digits. The values must be between 0 and 2147483647 inclusive.

An integer data item is always an exact representation of an integer value.

1.3.4.2 REAL AND LONG REAL LITERALS

```
unsigned-number
    = decimal-number [ exponent-part ]
    ! exponent-part
```

An unsigned-number is a decimal-number, a decimal-number followed by an exponent-part, or just an exponent-part.

```
decimal-number
    = unsigned-integer [ decimal-fraction ]
    ! decimal-fraction
```

A decimal-number is an unsigned-integer (i.e. an integer literal) optionally followed by a decimal-fraction, or it is just a decimal-fraction.

```
decimal-fraction
    = "." unsigned-integer
```

A decimal-fraction is an integer literal preceded by the decimal point.

```
exponent-part
    = ( "&" ! "&&" ) [ "+" ! "-" ] unsigned-integer
```

An exponent part is either the real exponent symbol (&) or the long real exponent symbol (&&), optionally followed by the sign of the exponent, and followed by the integer literal giving the size of the exponent. This is a scale factor expressed as an integral power of 10. The exponent mark used determines the type of the literal defined by the unsigned-number:

1) Unsigned-integers are of type integer.

2) If the unsigned-number contains an exponent-part with a double ampersand ("&&") it is of type long real.

3) All other unsigned-numbers are of type real.

Examples:

```
20                                          - integer value
2&1   2.0&+1   .2&2   20.0   200&-1   - represent same real value
```

```
2.345678&&0                          - long real value
2345.                                - is illegal
                                       (in contrast to FORTRAN)
```

The value of a real literal must be zero or lie between 10**-76 and 10**76 (32 bits representation).

A real data item is in most cases an approximation to the exact value of an expression; the accuracy is approximately 7 decimal digits.

The value of a long real literal must be zero or lie between 10**-76 and 10**76 (64 bits representation). Thus the value range is the same as that of real literals, but the accuracy of the approximation is greater, being approximately 16 decimal digits.

Examples:

```
2&1    2.0&+1    .2&2    20.0    200&-1  - represent same real value
2.345678&&0                              - long real value
```

1.3.4.3 STRINGS (TEXT LITERALS)

```
string
    =  simple-string (* token-separator simple-string *)
```

A string is any number of simple-strings, separated by token-separators (and nothing else), e.g. separated by spaces, end-of-lines, or direct-comments.

```
simple-string
    =  "" (* ISO-code ! non-quote ! text-quote *) ""
```

The rule means that a simple-string is a sequence of ISO-coded characters (see below), non-quotes or text-quotes, bracketed in ". The length of the simple string may be zero (the empty string); a simple string must be wholly contained in one line.

```
ISO-code
    =  "!" unsigned-integer "!"

non-quote
    =  any-printing-character-except-string-quote

text-quote
    =  two-string-quotes-in-sequence
```

Any printing character (including space) except the string quote (") represents itself within a string (or a character literal, see below).

In order to include the complete ISO alphabet any character may be represented within a string (or a character-constant, see below) by its ISO-code surrounded by code quotes (! - exclamation mark). The unsigned-integer given as the code cannot consist of more than three

digits and must be less than 256. If these conditions are not satisfied, the construction is interpreted as a character-sequence.

The string quote may, however, also be represented in simple-strings by two consecutive quotes ("").

Examples:

The string:	represents:
"Ab"!direct-comment;"cde"	Abcde
"AB" <end-of-line> "CDE"	ABCDE
"!2!ABCDE!3!"	ABCDE enclosed in STX and ETX
"AB"" C""DE"	AB" C"DE

Observe the text-quotes and the embedded space in the last example.

1.3.4.4 CHARACTER CONSTANTS

```
character-constant
    = "'" character-designator "'"

character-designator
    = ISO-code
    ! any-printing-character
```

The form of a character literal is a single quote (') followed by either one printing character, or by an ISO-code as explained above, and delimited by a single quote.

A character data item is a byte containing the ISO code of the character value.

1.3.5 TOKEN SEPARATORS

Direct-comments, spaces (except in simple strings and character literals), and the separation of consecutive lines are called token separators. Zero or more token separators may occur between any two consecutive tokens, or before the first token of a program text. There must be at least one token separator between any pair of consecutive tokens made up of identifiers, keywords, simple strings or arithmetic literals. No token separators may occur within tokens.

1.4 DIRECTIVE LINES

Directive lines have a % in column 1. If the immediate following character is a blank, the line is a comment. Otherwise the text of the line (from column 2) is a directive to the ND-500 SIMULA compiler. The relevant directives are listed below.

1.4.1 INCLUDE A SOURCE FILE

This directive will insert the contents of the parameter file at this place in the source input file. The inserted file cannot contain a COPY-directive i.e. insertions cannot be nested.

Format: %COPY file-name

1.4.2 CHANGE LISTING PAGE HEADER

This directive will replace the current page header of the compiler listing file with the string given as the parameter. In addition an implicit %PAGE is performed. Besides the page header given, the listing header will contain an identification of the compiler (SIMULA) and the date and time at which the compilation started.

The page header is initially empty.

Format: %TITLE page-header

1.4.3 CHANGE TO NEW PAGE IN LISTING

This directive will change page in the listing file. If the current page header is non-empty (see above) then it will be printed on top of the new page.

Format: %PAGE

1.4.4 TURN COMPILATION LISTING ON OR OFF

These directives control whether or not to generate a listing of the source program during compilation. Note that the LISTING command supercedes these directives, i.e. the directives are significant only if the listing level is greater than zero (see page 189). Initially listing is ON.

Format: %LIST ON
%SOURCE

These directives will cause the following lines to be listed.

Format: %LIST OFF
%NOSOURCE

These directives will cause listing to be switched off. If listing was in effect the directive line is printed.

1.4.5 CHANGE SIGNIFICANT LINE LENGTH OF SOURCE FILE

Initially the whole source line will be considered significant, i.e. all characters are treated by the compiler. This directive may be used if only the first part of the line shall be used as source text; the last positions may for instance contain a version identification.

Format: %SLENGTH integer literal

The parameter value must be less than or equal to INPLTH (see page 191).

1.5 COMMENT CONVENTIONS

An end-comment is a sequence of characters following the keyword END. The character sequence is terminated by one of the symbols END, ELSE, WHEN, OTHERWISE, or semicolon. Apart from possibly listing it, the compiler will ignore this character sequence (but the embracing symbols are significant).

A direct-comment consists of the exclamation mark, or alternatively the keyword COMMENT, followed by a sequence of characters, and terminated by a semicolon. The compiler will replace the sequence (including the embracing ! and ;) with a space after possibly listing it, but before performing any further analysis. Thus a direct-comment may be used to separate simple strings with the concatenation effect as described above.

Note that neither end- nor direct-comments are recognized within a simple string. Also, the comment structure encountered first takes precedence, i.e. nesting of comments is not allowed.

1.6 PROGRAM EXCHANGE CONSIDERATIONS

Whenever program exchange between different SIMULA implementations is considered, some additional rules should be obeyed in order to avoid trivial problems.

The significant line length of the program should be at most 72 characters. Identifiers should consist of at most 12 characters. A class should be declared before its use as a prefix. The procedures "isorank" and "isochar" should be used, instead of "rank" and "char" respectively.

1.7 STATEMENTS AND DECLARATIONS

The general structure of SIMULA source modules is a sequence of statements and declarations, composed of keywords and other tokens as described above. Statements describe actions to be taken when the program is executed, while declarations in principle describe properties of the objects manipulated through these actions.

A statement may be identified by an identifier (a label) followed by a colon, which precedes the statement. Labels provide a means of changing the otherwise sequential execution of statements. Labels (and other identifiers defined by the program) obey certain scope rules, i.e. their visibility is restricted according to the general rules given on pages 25 to 26, and 32 to 43.

Some statements, namely the so-called blocks, may contain declarations; all declarations are then given as the first part of such a statement.

The reverse is also true, in the case of declarations of procedures and classes. These declarations may contain statements describing the action or actions associated with the entity.

SIMULA is a relatively free-format language, i.e. the statement structure is not tied in with the line structure of the input medium. Thus e.g. a statement may commence anywhere on a line, and it may continue for as many lines as necessary. It is however advisable that, in order to improve readability and understandability of the program, a certain structure be imposed upon the text by the programmer. No recommendations or rules for structured programming will be given in this manual, but hints may be derived from the example programs.

1.8 THE LINE STRUCTURE OF SOURCE MODULES

A source module is either a main program, or it is a declaration of a procedure or a class. It consists of a sequence of lines. The actual length of lines may vary within one module, subject to the restriction that the length of a source line is limited to the value of INPLTH (cf. page 191). Normally the compiler will treat the whole line, it is however possible to restrict the compiler interpretation to only the first part of the line (see page 12 above).

CHAPTER 2

THE S-PORT SIMULA SYSTEM

2 THE S-PORT SIMULA SYSTEM

This chapter contains information about the SIMULA system that is not necessary for normal use of the system. Thus it may be skipped at a first reading.

The ND-500 SIMULA system is based upon the portable system called S-port. It consists of two major parts, a compiler and a run time system. Both of these consists again of two parts, namely a (machine independent) portable part, and a part developed specifically for the ND-500.

2.1 THE COMPILER

The SIMULA compiler performs syntactic and semantic checking of the source module, and produces a loadable NRF-file. It has two major components, the front end compiler (FEC), and the code generator orS-compiler.

The compiler runs in three passes.

2.1.1 THE FRONT END COMPILER (FEC)

FEC is the portable part of the compiling system, and consists of the first two compiler passes. It performs all checking of the syntactical and static semantic structure of the source input, and converts the text into a standardized intermediate form called S-code.

FEC is responsible for almost all messages generated during a compilation, and due to this the wording of error messages and diagnostics will be the same on all S-port based compilers. This is convenient when a programmer moves from one system to another.

Many directive lines are also treated by FEC, thus the use and meaning of at least the directives mentioned in chapter 1 are common to all S-port systems.

2.1.2 THE S-COMPILER

The S-compiler receives the S-code from FEC and translates it into NRF. Due to the nature of the compiler, the possible messages from this part of the compiler are related to machine dependencies such as the actual value range of literals, the capacity of the machine and the compiler, etc. For moderate sized programs no diagnostic messages from the S-compiler should normally occur.

2.2 THE RUN TIME SYSTEM

The ND-500 SIMULA run time system consists of two parts, the machine independent RTS and the ND-500 specific Environment Interface or EI. Both parts consist of many NRF files, collected into a library normally called ND-SIMULA-AR00.

Most messages given by the run time system during the execution of a compiled and linked SIMULA program originate within RTS, i.e. the convenience of homogenous error messages from S-port based implementations is preserved.

CHAPTER 3

DECLARATIONS

3 DECLARATIONS

3.1 DATA TYPES

```
type
    =  value-type
    !  reference-type

value-type
    =  integer-type
    !  real-type
    !  "Boolean"
    !  "character"

integer-type
    =  [ "short" ]  "integer"

real-type
    =  [ "long ]  "real"

reference-type
    =  object-reference
    !  "text"

object-reference
    =  "ref"  "("  qualification  ")"

qualification
    =  class-identifier
```

The various types basically denote properties of values. In addition, a reference-type identifies a value, called the referenced value.

A value is primarily a number, a logical value, a label, an object, a single character, or an ordered sequence of characters.

The value of an array identifier is the ordered set of values of the constituent array components.

3.2 ARITHMETIC TYPES

Arithmetic types are used for representing numerical values. The types are integer-type and real-type. Integer-type is either integer or short integer. Real-type is either real or long real. The types short integer and long real are subject to the following rules.

3.2.1 SHORT INTEGER

The type short integer serves to represent integer values whose value range may be a subset of that of integer. Apart from this, short integer and integer are fully compatible in this language definition.

3.2.2 LONG REAL

Type long real serves to represent real values capable of retaining a higher precision than that of the type real. The relative value range of the respective types is not defined: Apart from this, long real and real are fully compatible in this language definition.

3.3 THE TYPE BOOLEAN

The type Boolean represents logical values. The range of values consists of the values true and false.

3.4 THE TYPE CHARACTER

The type character is used to represent single characters. Such a value is an instance of an "internal character". For any given implementation there is a one-to-one mapping between a subset of internal characters and external ("printable") characters. The character sets (internal and external) are implementation defined.

3.5 REFERENCE TYPES

The reference concept corresponds to the notion of a "name" or a "pointer". It provides a mechanism for referencing values.

3.5.1 OBJECT REFERENCE

Associated with an object there is a unique "object reference" which identifies the object. And for any class C there is an associated reference-type ref (C). A quantity of that type is said to be qualified by the class C. Its value is either an object, or the special value none which represents "no object". The qualification restricts the range of values to objects of classes included in the qualifying class. The range of values includes the value none regardless of the qualification.

3.5.2 THE TYPE TEXT

The type text serves to declare or specify a text variable quantity.

A text value is an ordered sequence, possibly empty, of characters. The number of characters is called the "length" of the text value.

A text frame is a memory device which contains a nonempty text value. A text frame has a fixed length and can only contain text values of this length. A text frame may be "alterable" or "constant". A constant frame always contains the same text value. An alterable text frame may have its contents modified.

A text reference identifies a text frame. The reference is said to possess a value, which is the contents of the identified frame. The special text reference notext identifies "no frame". The value of notext is the empty text value.

Text objects and text frames

A "text object" is conceptually an instance of the following class declaration (cf. page 32 et seq):

```
class TEXTOBJ(SIZE,CONST);
integer SIZE; Boolean CONST;
begin character array MAIN(1:SIZE); end;
```

Any non-empty sequence of consecutive elements of the array attribute MAIN constitutes a text frame. More specifically, any text frame is completely identified by the following pieces of information:

1) a reference to the text object containing the frame,

2) the start position of the frame, being an integer within the subscript bounds of the MAIN attribute of that text object,

3) the length of the frame.

A frame which is completely contained in another frame is said to be a "subframe" of that other frame. The text frame associated with the entire array attribute MAIN is called the "main frame" of the text object. All frames of the text object are subframes of the main frame.

Note: A main frame is a subframe of no frame except itself.

The frames of a text object are either all constant or all variable, as indicated by the attribute CONST. The value of this attribute remains fixed throughout the lifetime of the text object. A constant main frame always corresponds to a string (see page 9).

The attribute SIZE is always positive and remains fixed throughout the lifetime of a text object.

The identifier TEXTOBJ, as well as the three attribute identifiers, are not accessible to the user. Instead, properties of a text object are accessible through text variables.

3.6 TYPE QUALIFICATION

The identifying qualification (a reference type) conforms to the rules of all other identifiers, i.e. the quantity is defined upon block entry and loses its significance at block exit (cf. page 25). By consequence, the type qualification defined upon block entry of a block is not valid in any other block instance, even if the blocks are created from the same (textual) declaration in the program.

Example:

```
class a;
begin class b; ; ref (b) axb; end a;

a class aa;
begin ref (b) aaxb; end aa;

ref (a) a1; ref (aa) a2;
....
inspect a2 do
inspect a1 do   aaxb:- new b;
```

will constitute a runtime error whenever a1 and a2 reference different objects. Thus, replacing .... above with

```
a1:- a2:- new aa;
```

will lead to a valid program, while the replacement

```
a1:- new a;  (or a1:- new aa)
a2:- new aa;
```

is illegal.

3.7 SUBORDINATE TYPES

An object reference is said to be "subordinate" to a second object reference if the qualification of the former is a subclass of the class which qualifies the latter.

A proper procedure is said to be of a universal type. Any type is subordinate to the universal type.

3.8 TYPE TRANSFER

Values may in some cases be transfered (or converted) from one type to another.

Conversion between arithmetic types follows the rules described on page 59. In addition the procedure "entier", converting from real type to integer, is described on page 154.

Conversion between text and arithmetic type values is described on page 126 (text attributes "getint", "putint", "getreal", "putreal", "getfrac", "putfrac").

Conversion between character and text values is described on page 121 (text attributes "getchar", "putchar").

Conversion between character and integer values is described on page 155 ("isorank", "rank", "isochar", "char").

3.9 DECLARATIONS

```
declaration
    =  simple-variable-declaration
    !  array-declaration
    !  switch-declaration
    !  procedure-declaration
    !  class-declaration
    !  external-declaration
```

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes.

Dynamically this implies that at the time of an entry into a block all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers are also defined by other declarations outside the block, they are for the time being given a new significance, which is called "local". Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through end, by a goto-statement, or through a sequencing procedure call) all identifiers which are declared for the block loose their local significance.

Apart from labels, formal-parameters of procedure- and class-declarations, and identifiers declared in the environmental prefix, each identifier appearing in a program must be explicitly declared within the program.

No identifier may be declared either explicitly or implicitly more than once in any one block-head.

For information on external declarations, please refer to pages 131 through 136.

3.10 SIMPLE VARIABLE DECLARATIONS

```
simple-variable-declaration
    = type  type-list

type-list
    = type-list-element  (* "," type-list-element *)

type-list-element
    = identifier
    ! constant-element
```

Type declarations serve to declare certain identifiers to represent simple variables of a given type.

A variable local to a block instance is a memory device whose "contents" is either a value or a reference, according to the type of the variable. A value type variable has a value which is the contents of the variable. A reference type variable is said to have a value which is the one referenced by the contents of the variable. The contents of a variable may be changed by an appropriate assignment operation (see page 69).

3.10.1 VALUE TYPE VARIABLES

Real declared variables may assume positive and negative values including zero.

Integer declared variables may assume positive and negative integral values including zero.

Boolean declared variables may assume the values true and false.

Character declared variables may assume the values previously defined as character values (see page 4).

The type declarations short integer and long real are extensions of the standard arithmetic types. An implementation may choose to support both, neither, or either of them alone.

Apart from the special considerations described below the types short integer and long real are fully compatible with types integer and real respectively, and they can consequently be used in any place where integer or real occur in the language definition outside these paragraphs.

The type declaration short integer serves to declare identifiers representing integer variables whose value range is a subset of that of integer variables. In an expression any position which can be occupied by an integer declared variable may be occupied by a short integer declared variable.

The type declaration long real serves to declare identifiers representing variables capable of retaining a higher precision of floating point values than variables of type real. In an expression any position which can be occupied by a variable or number of type real may be occupied by a variable or number of type long real.

3.11 ARRAY DECLARATION

```
array-declaration
    = array-declarer  array-segment  (* "," array-segment *)

array-declarer
    = [ type ] array

array-segment
    = array-identifier  (* "," array-identifier *)
      "("  bound-pair-list  ")"

bound-pair-list
    = bound-pair  (* "," bound-pair *)

bound-pair
    = lower-bound  ":"  upper-bound

lower-bound
    = arithmetic-expression

upper-bound
    = arithmetic-expression
```

An array-declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables, and gives the dimensions of the arrays, the bounds of the subscripts, and the type of the variables.

The subscript bounds for any array are given in the first subscript brackets following the identifier of this array, in the form of a bound-pair-list. Each item of this list gives the lower- and upper-bounds of a subscript in the form of two arithmetic-expressions separated by ":". The bound-pair-list gives the bounds of all subscripts taken in order from left to right.

The dimensions are given as the number of entries in the bound-pair-lists.

All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type real is understood.

The expressions will be evaluated in the same way as subscript expressions.

The expressions cannot include any identifier that is declared, either explicitly or implicitly, in the same block-head as the array in question.

An array is defined only when the values of all upper-bounds are not smaller than those of the corresponding lower-bounds. If any lower-bound is greater than the corresponding upper-bound, the array has no component.

The expressions will be evaluated once at each entrance into the block through the block-head.

The value of an array-identifier is the ordered set of values of the corresponding array of subscripted variables.

Examples:

```
array a,b,c(7:n,2:m), s(-2:10)
integer array a(2:20)
real array q(-7:if c<0 then 2 else 1)
```

3.12 SWITCH DECLARATION

```
switch-declaration
    = "switch" switch-identifier ":=" switch-list

switch-list
    = designational-expression (* "," designational-expression *)
```

A switch-declaration defines the set of values of the corresponding switch-designators. These values are given one by one as the values of the designational-expressions entered in the switch-list. With each of these designational-expressions there is associated a natural number (1,2, ...) obtained by counting the items in the list from left to right. The value of the switch-designator corresponding to a given value of the subscript expression is the value of the designational-expression in the switch-list having this given value as its associated integer.

An expression in the switch-list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

If a switch-designator occurs outside the scope of a quantity entering into a designational-expression in the switch-list, and an evaluation of this switch-designator selects this designational-expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the

place of the switch-designator will be avoided through suitable systematic changes of the latter identifiers.

Examples:

```
switch  s:=s1,s2,q(m), if v>-5 then s3 else s4
switch  q:=p1,w
```

3.13 PROCEDURE DECLARATION

```
procedure-declaration
    = [ type ] "procedure"  procedure-heading  procedure-body

procedure-heading
    =  procedure-identifier
       [ formal-parameter-part ";"  [ mode-part ]
         specification-part ]

procedure-body
    =  statement

procedure-identifier
    =  identifier
```

A procedure-declaration serves to define the procedure associated with a procedure-identifier. The principal constituent of a procedure-declaration is a statement or a piece of code, the procedure-body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure-declaration appears.

Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal-parameters. Formal parameters in the procedure-body will, whenever the procedure is activated, be replaced by the values of the actual parameters (cf. page 31). Identifiers in the procedure-body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure-declaration appears.

The procedure-body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure-body. In addition, if the identifier of a formal-parameter is declared anew within the procedure-body, it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

No identifier may appear more than once in any one formal-parameter-list, nor may a formal-parameter-list contain the procedure-identifier of the same procedure-heading.

Examples:

```
procedure transpose(a,n);
array a; integer n;
begin real w; integer i,k;
      for i:=1 step 1 until n do
          for k:=1+i step 1 until n do
          begin w:=a(i,k);
                a(i,k):=a(k,i);
                a(k,i):=w
          end
end transpose;

integer procedure factorial(n); integer n;
  factorial:= if n=0 then 1 else n*factorial(n-1);

procedure absmax(a,n,m,y,i,k);
   name i, k, y ; array a; integer n,m,i,k; real y;
comment The absolute greatest element of the matrix a, of size n
by m is transferred to y, and the subscripts of this element to
i and k;
begin integer p,q;
      y:=0; i:=k:=1;
      for p:=1 step 1 until n do
          for q:=1 step 1 until m do
              if abs(a(p,q))>y then
              begin y:=abs(a(p,q));
                    i:=p; k:=q
              end
end absmax;

procedure innerproduct(a,b,k,p,y); name p,y,a,b;
integer k,p; real y,a,b;
begin real s; integer pp;
      s:=0;
      for pp:=1 step 1 until k do
          begin p:= pp; s:=s+a*b; end;
      y:=s
end innerproduct;

text procedure mystrip(t); text t;
mystrip:- if t.sub(t.length,1)=" " then
          mystrip(t.sub(1,t.length-1)) else t;
```

3.13.1 VALUES OF FUNCTION DESIGNATORS

For a procedure-declaration to define the value of a function designator there must, within the procedure-body, occur one or more uses of the procedure-identifier as a destination; at least one of these must be executed, and the type associated with the procedure-identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure-declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure-identifier within the body of the procedure other than

as a destination in an assignment statement denotes activation of the procedure.

If a goto-statement within the procedure, or within any other procedure activated by it, leads to an exit from the procedure, other than through its end, then the execution of all statements that have been started but not yet completed, and which do not contain the label to which the goto-statement leads, is abandoned. The values of all variables that still have significance remain as they were immediately before execution of the goto-statement (cf. page 149).

If a function designator is used as a procedure statement, then the resulting value is discarded, but such a statement may be used, if desired, for the purpose of invoking side effects.

3.13.2 PARAMETER SPECIFICATION

```
formal-parameter-part
    = "(" formal-parameter-list ")"

formal-parameter-list
    = formal-parameter  (* "," formal-parameter *)

formal-parameter
    = identifier

specification-part
    = (* specifier  identifier-list  ";" *)

specifier
    = type
    ! array-declarer
    ! "label"
    ! "switch"
    ! [ type ]  "procedure"
```

The procedure-heading includes a specification-part, giving information about the kinds and types of the formal-parameters. In this part no formal-parameter may occur more than once.

3.13.3 PARAMETER TRANSMISSION MODES

```
mode-part
    = name-part  [ value-part ]
    ! value-part [ name-part ]

name-part
    = "name"  identifier-list  ";"
```

```
value-part
    = "value" identifier-list ";"

identifier-list
    = identifier (* "," identifier *)
```

There are three modes of parameter transmission: "call by value", "call by reference", and "call by name".

The default transmission mode is call by value for value type parameters and call by reference for all other kinds of parameters.

The available transmission modes are shown in figure 3.1 for the different kinds of parameters of procedures.

Figure 3.1: Transmission Modes for Parameters of Procedures

Parameter	Transmission modes		
	by value	by reference	by name
value type	D	I	O
object reference	I	D	O
text	O	D	O
value type array	O	D	O
reference type array	I	D	O
procedure	I	D	O
type procedure	I	D	O
label	I	D	O
switch	I	D	O

D: default mode O: optional mode I: illegal

3.14 CLASS DECLARATION

```
class-declaration
    = [ prefix ] main-part

prefix
    = class-identifier

main-part
    = "class" class-identifier
      [ formal-parameter-part ";" [ value-part ]
        specification-part ]
      [ protection-part] [ virtual-part ]
      class-body
```

```
class-identifier
    = identifier

class-body
    = statement
    ! split-body

split-body
    = initial-operations  inner-part  final-operations

initial-operations
    = ( "begin" ! block-head )  (* statement ";" *)

inner-part
    = (* label ":" *)  "inner"

final-operations
    = "end"
    ! ";"  compound-tail
```

A class-declaration serves to define the class associated with a class-identifier. The class consists of "objects" each of which is a dynamic instance of the class-body.

An object is generated as the result of evaluating an object-generator, which is the analogy of the "call" of a function-designator (see page 51).

A class-body always acts like a block. If it takes the form of a statement which is not an unlabelled block, the class-body is identified with a block of the form:

begin; S end;

where S is the textual body. A split-body acts as a block in which the symbol inner represents a dummy statement.

For a given object the formal-parameters, the quantities specified in the virtual-part, and the quantities declared local to the class-body are called the "attributes" of the object. A declaration or specification of an attribute is called an "attribute definition".

Specification (in the specification-part) is necessary for each formal-parameter. The parameters are treated as variables local to the class-body. They are initialized according to the rules of parameter transmission (see page 40). Call by name is not available for parameters of class-declarations. The following specifiers are accepted:

⟨type⟩, array, and ⟨type⟩ array.

Attributes defined in the virtual-part are called "virtual quantities". They do not occur in the formal-parameter-list. The virtual quantities have some properties which resemble formal-parameters called by name. However, for a given object the environment of the corresponding "actual parameters" is the object itself, rather than that of the generating call.

Identifier conflicts between formal-parameters and other attributes defined in a class-declaration are illegal.

The declaration of an array attribute may, in a constituent subscript bound expression, make reference to the formal-parameters of the class-declaration; thus subscript bound expressions, which refer to attributes other than the formal-parameters of the class-declaration or its prefixes, are illegal.

3.14.1 SUBCLASSES

A class-declaration with the prefix "C" and the class-identifier "D" defines a subclass D of the class C. An object belonging to the subclass consists of a "prefix part", which is itself an object of the class C, and a "main-part" described by the main-part of the class-declaration. The two parts are "concatenated" to form one compound object. The class C may itself have a prefix.

Let C1, C2, ..., Cn be classes such that C1 has no prefix and Ck has the prefix Ck-1 (k = 2,3,....., n). Then C1,C2, ..., Ck-1 is called the "prefix sequence" of Ck (k = 2, 3, ..., n). The subscript k of Ck (k = 1, 2, ..., n) is called the "prefix level" of the class. Ci is said to "include" Cj if i <= j, and Ci is called a "subclass" of Cj if i > j (i, j = 1, 2, ..., n). The prefix level of a class D is said to be "inner" to that of a class C if D is a subclass of C, and "outer" to that of C if C is a subclass of D. Figure 3.2 depicts a class hierarchy consisting of five classes, A, B, C, D and E:

```
  class A .....;
A class B .....;
B class C .....;
B class D .....;
A class E .....;
```

A capital letter denotes a class. The corresponding lower case letter represents the attributes of the main-part of an object belonging to that class. The object structures shown in figure 3.3 indicate the allocation in memory of the values of those attributes which are simple variables.

The following restrictions must be observed in the use of prefixes:

1) A class must not occur in its own prefix sequence.

2) A class may be used as a prefix only at the block level at which it is declared. A system class is considered to be declared in the smallest block enclosing its first textual occurrence. An implementation may restrict the number of different block levels at which such prefixes may be used.

Figure 3.2: A Class Hierarchy

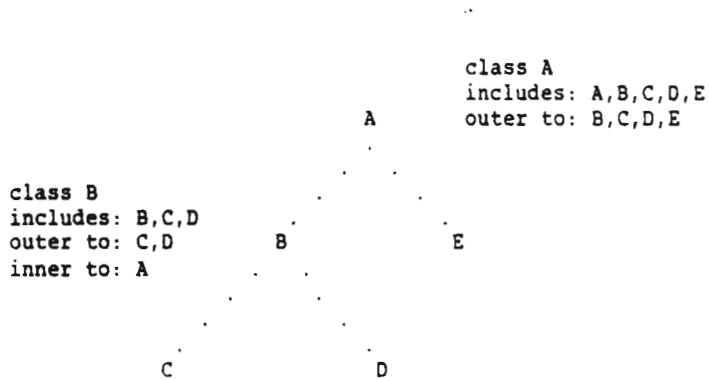
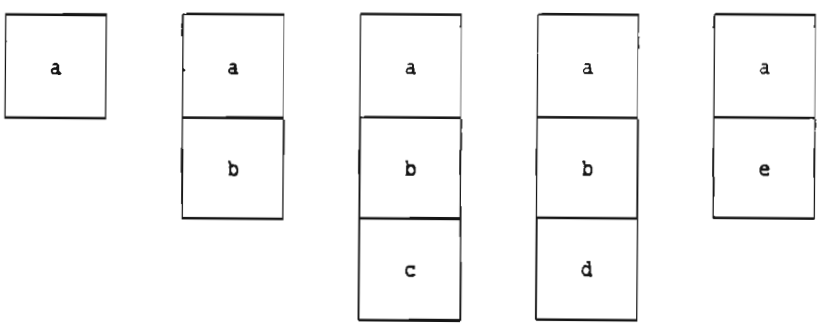


Figure 3.3: Objects of Classes A, B, C, D and E Respectively



Concatenation

Let Cn be a class with the prefix sequence C1, C2, ..., Cn-1, and let X be an object belonging to Cn. Informally, the concatenation mechanism has the following consequences.

1) X has a set of attributes which is the union of those defined in C1, C2, ..., Cn. An attribute defined in Ck (1 <= k <= n) is said to be defined at prefix level k.

2) X has an "operation rule" consisting of statements from the bodies of these classes in a prescribed order. A statement from Ck is said to belong to prefix level k of X.

3) A statement at prefix level k of X has access to all attributes of X defined at prefix levels equal to or outer to k, but not directly to attributes "hidden" by conflicting definitions at levels < k. (These "hidden" attributes may be accessed through use of procedures.)

4) A statement at prefix level k of X has no immediate access to attributes of X defined at prefix levels inner to k, except through virtual quantities (cf. page 38).

5) In a split-body at prefix level k, the symbol inner represents those statements in the operation rule of X which belong to prefix levels inner to k, or a dummy statement if k = n. If none of C1, ..., Cn-1 have a split-body, the statements in the operation rule of X are ordered.

A compound object could be described formally by a "concatenated" class-declaration. The process of concatenation is considered to take place prior to program execution. In order to give a precise description of that process, we need the following definition.

An occurrence of an identifier which is part of a given block is said to be an "uncommitted occurrence in that block", except if it is the attribute identifier of a remote identifier (see page 50), or is part of an inner block in which it is given a local significance. In this context a "block" may be a class-declaration not including its prefix and class-identifier, or a procedure-declaration not including its procedure-identifier. (Observe that an uncommitted identifier occurrence in a block may well have a local significance in that block.)

The class-declarations of a given class hierarchy are processed in order of ascending prefix levels. A class-declaration with a non-empty prefix is replaced by a concatenated class-declaration obtained by first modifying the given one in two steps.

1) If the prefix refers to a concatenated class-declaration, in which identifier substitutions have been carried out, then the same substitutions are effected for uncommitted identifier occurrences within the main-part.

2) If now identifiers of attributes defined within the main-part have uncommitted occurrences within the prefix class, then all uncommitted occurrences within the main-part of these identifiers are systematically changed to avoid name conflicts. Identifiers corresponding to virtual quantities defined in the prefix class are not changed.

The concatenated class-declaration is defined in terms of the given declaration, modified as above, and of the concatenated declaration of the prefix class.

1) Its formal-parameter-list consists of that of the prefix class followed by that of the main-part.

2) Its value-part, specification-part, and virtual-part are the unions (in an informal but obvious sense) of those of the prefix class and those of the main-part. If the resulting virtual-part contains more than one occurrence of some identifier, the virtual-part of the given class-declaration is illegal.

3) Its class-body is obtained from that of the main-part in the following way, assuming the body of the prefix class is a split-body. The begin of the block-head is replaced by a copy of the block-head of the prefix body. A copy of the initial-operations of the prefix body is inserted after the block-head of the main-part, and the end of the compound tail of the main-part is replaced by a copy of the compound tail of the prefix body. If the prefix class-body is not a split-body, it is interpreted as if the symbols ";inner" were inserted in front of the end of its compound tail. If in the resulting class-body two matching declarations for a virtual quantity are given, the one copied from the prefix class-body is deleted.

4) The declaration of a label is its occurrence as the label of a statement.

Examples:

```
class point(x,y); real x,y;
    begin ref (point) procedure plus(P); ref (point) P;
              plus:- new point(x+P.x, y+P.y);
    end point;
```

An object of the class point is a representation of a point in a Cartesian plane. Its attributes are x, y and plus, where plus represents the operation of vector addition.

```
point class polar;
     begin real r,v;
        ref (polar) procedure plus(P); ref (point) P;
           plus :- new polar(x+P.y, y+P.y);
        r:= sqrt(x**2 + y**2);
        v:= arctan(x,y);
    end polar;
```

An object of the class polar is a "point" object with the additional attributes r, v and a redefined plus operation. The values of r and v are computed and assigned at the time of object generation.

3.14.2 VIRTUAL QUANTITIES

```
virtual-part
    = "virtual" ":" specification-part
```

Virtual quantities serve a double purpose:

1) to give access at one prefix level of an object to attributes declared at inner prefix levels, and

2) to make attribute re-declarations at one prefix level valid at outer prefix levels.

The following specifiers are accepted in a virtual-part:

label, switch, procedure and ⟨type⟩ procedure.

A virtual quantity of an object is either "unmatched" or is identified with a "matching" attribute, which is an attribute whose identifier coincides with that of the virtual quantity, declared at the prefix level of the virtual quantity or at an inner one. The matching attribute must be of the same kind as the virtual quantity. At a given prefix level, the type of the matching quantity must coincide with or be subordinate to that of the virtual specification and that of any matching quantity declared at any outer prefix level.

At any given prefix level PL inner or equal to that of a virtual specification, the type of the virtual quantity is:

1) that given in the virtual specification, if there is no match at prefix levels outer or equal to PL.

2) that of the match at the innermost prefix level outer or equal to PL, if there is a match at a prefix level outer or equal to PL.

It is a consequence of the concatenation mechanism that a virtual quantity of a given object can have at most one matching attribute. If matching declarations have been given at more than one prefix level of the class hierarchy, then the one is valid which is given at the innermost prefix level outer or equal to that of the main-part of the object. The match is valid at all prefix levels of the object equal or inner to that of the virtual specification.

Example:

The following class expresses a notion of "hashing", in which the "hash" algorithm itself is a "replaceable part". "Error" is a suitable non-local procedure.

```
class hashing (n); integer n;
virtual: integer procedure hash;
begin integer procedure hash(t); value t; text t;
      begin integer i;
            while t.more do i:= i + rank(t.getchar);
            hash:= mod(i,n);
      end hash;
      text array table (0:n-1); integer entries;
      integer procedure lookup (t,old);
      name old; value t; Boolean old; text t;
      begin integer i,istart; Boolean entered;
            i:= istart:= hash(t);
            while not entered do
            begin if table(i)==notext then
                  begin table(i):- t;
                        entries:= entries + 1;
                        entered:= true;
                  end else if table(i) = t then
                        old:= entered:= true
                        else begin i:= i + 1;
                             if i=istart then
                                error("Table full.");
                             if i = n then i:= 0;
                        end;
            end;
            lookup:= i;
      end lookup;
end hashing;

hashing class ALGOL hash;
begin integer procedure hash(T); value T; text T;
      begin integer i; character c;
         while T.more do
            begin c:= T.getchar;
                  if c <> ' ' then i:= i + rank(c);
            end;
            hash:= mod(i,n);
      end hash;
end ALGOL hash;
```

3.14.3 ATTRIBUTE PROTECTION

```
protection-part
    =  protection-specification  (* protection-specification *)

protection-specification
    =  "hidden"  identifier-list  ";"
    !  "protected"  identifier-list  ";"
    !  "hidden" "protected"  identifier-list  ";"
    !  "protected" "hidden"  identifier-list  ";"
```

The protection-specification makes it possible to restrict the scope of class attribute identifiers.

A class attribute, X, which is specified protected in class C is only accessible:

1) within the body of C or its subclasses, and

2) within the blocks prefixed by C or any subclass of C.

In any other context the meaning of the identifier X is as if the attribute definition of X were absent.

Access to a protected attribute is, subject to the restriction above, legal by remote accessing.

A class attribute may only be specified protected at the prefix level of its definition. Note that a virtual attribute may only be specified protected in the same class heading where the virtual specification is placed.

Attributes of the classes Simset and Simulation are protected.

A class attribute, X, specified hidden in class C is not accessible within subclasses of C or blocks prefixed by C or any subclass of C. In this context the meaning of the identifier X is as if the attribute definition of X were absent.

Observe that specifying a virtual quantity hidden effectively disables further matching at inner levels.

Only a protected attribute may be specified hidden, however the hidden specification may occur at a prefix level inner to the protected specification.

The effect of specifying an attribute hidden protected or protected hidden is identical to that of specifying it as both protected and hidden.

Conflicting or illegal hidden and/or protected specifications constitute a compile time error.

Note that if there are several attributes with the same identifier in the prefix sequence to a hidden specification, and these are previously protected, but not hidden, the innermost accessible attribute will be hidden.

3.14.4 PARAMETER TRANSMISSION MODES

There are two modes of parameter transmission available for classes: "call by value" and "call by reference".

The default transmission mode is call by value for value type parameters and call by reference for all other kinds of parameters.

The available transmission modes for parameters of class-declarations are shown in figure 3.4. For parameters of procedure declarations, see figure 3.1.

Figure 3.4: Transmission Modes for Parameters of Classes

Parameter	Transmission modes	
	by value	by reference
value type	D	I
object reference	I	D
text	O	D
value type array	O	D
reference type array	I	D

D: default mode O: optional mode I: illegal

3.14.5 REMOTE ACCESSING

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

Item 1: the object,

Item 2: a class which is outer to or equal to that of the object, and

Item 3: an attribute identifier defined in that class or in any class belonging to its prefix sequence.

Item 2 is textually defined for any attribute identification. The prefix level of the class is called the "access level" of the attribute identification.

Consider an attribute identification whose item 2 is the class C. Its attribute identifier, item 3, is subjected to the same identifier substitutions as those which would be applied to an uncommitted occurrence of that identifier within the main-part of C, at the time of concatenation. In that way, name conflicts between attributes declared at different prefix levels of an object are resolved by selecting the one defined at the innermost prefix level not inner to the access level of the attribute identification.

An uncommitted occurrence within a given object of the identifier of an attribute of the object is itself a complete attribute identification. In this case items 1 and 2 are implicitly defined, as respectively the given object and the class associated with the prefix level of the identifier occurrence.

If such an identifier occurrence is located in the body of a procedure-declaration (which is part of the object), then, for any dynamic instance of the procedure, the occurrence serves to identify an attribute of the given object, regardless of the context in which the procedure was invoked.

Remote accessing of attributes, i.e. access from outside the object, is either through the mechanism of "remote identifiers" ("dot notation") or through "connection".

A text variable is (itself) a compound structure in the sense that it has attributes accessible through the dot notation (cf. page 50).

3.14.6 STANDARD PROCEDURE ATTRIBUTE "DETACH"

Any class that has no (textually given) prefix will by definition be prefixed by a fictitious class whose only attribute is:

```
procedure detach; ... ;
```

Thus every class object or instance of a prefixed block has this attribute.

3.15 SCOPE OF DECLARED IDENTIFIERS

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid.

Identifiers may be chosen freely; they have no inherent meaning, but serve for the identification of language quantities, i.e. simple variables, arrays, texts, labels, switches, procedures and classes. Identifiers also act as formal-parameters in procedure and class-declarations, in which capacity they may represent a literal value or any language quantity but a class.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes. This rule also applies to the formal-parameters of procedure and class-declarations.

3.16 INITIALIZATION

Any declared variable is initialized at the time of entry into the block to which the variable is local. The initial contents depends on the type of the variable, as follows:

Variable	Initial contents
real	0.0
integer	0
Boolean	false
character	'!0!'
object reference	none
text	notext

3.17 CONSTANT DECLARATIONS

```
constant-element
    = identifier  "="  value-expression
    ! identifier  "="  string
```

An identifier which is declared by means of a constant-element has a fixed value throughout its scope. The evaluation of the expression takes place in the same manner as the evaluation of the bounds of an array. Thus any variables referenced in this expression will contribute their values at the time of their evaluation, and any subsequent change will not affect the constant.

The constant declaration is subject to the following restriction: If the expression contains any identifier that is declared in the same block-head, then this must be a constant that has been defined textually before the referencing constant-element.

CHAPTER 4

EXPRESSIONS

4 EXPRESSIONS

```
expression
    =  value-expression
    !  reference-expression
    !  designational-expression

value-expression
    =  arithmetic-expression
    !  Boolean-expression
    !  character-expression

reference-expression
    =  object-expression
    !  text-expression
```

In SIMULA the primary constituents of the programs describing algorithmic processes are expressions. Constituents of these expressions, except for certain delimiters, are constants, variables, function-designators, labels, switch-designators and elementary operators. Since the syntactic definition of both variables and function-designators (see below) contain expressions, the definition of expressions, and their constituents, is necessarily recursive.

A value-expression is a rule for obtaining a value.

An object-expression is a rule for obtaining an object reference.

A text-expression is a rule for obtaining an identification of a text variable (and thereby a text reference).

A designational-expression is a rule for obtaining a reference to a program point.

Any value-expression or reference-expression has an associated type, which is textually defined.

4.1 VARIABLES

```
variable
    =  simple-variable-1
    !  subscripted-variable

simple-variable-1
    =  identifier-1

subscripted-variable
    =  array-identifier-1  "("  subscript-list  ")"
```

```
array-identifier-1
    = identifier-1

subscript-list
    = subscript-expression  (* "," subscript-expression *)

subscript-expression
    = arithmetic-expression
```

A variable local to a block instance is a memory device whose "contents" is either a value or a reference, according to the type of the variable. A value type variable has a value which is the contents of the variable. A reference type variable is said to have a value which is the one referenced by the contents of the variable. The contents of a variable may be changed by an appropriate assignment operation (see page 69).

The value of an array-identifier is the ordered set of values of the corresponding array of subscripted-variables.

Variables are of two types, corresponding to the values being represented; value type variables and reference type variables.

A "reference" is a piece of information which identifies a value, called the "referenced" value. The distinction between a reference and the referenced value is determined by context.

The reference concept corresponds to the intuitive notion of a "name" or a "pointer". It also reflects the addressing capability of computers: in certain simple cases a reference could be implemented as the memory address of a stored value.

There are two reference type types, object reference type and type text.

Examples:

```
deta
a17
q(7,2)
x(sin(n*pi/2),q(3,n,4))
```

Note: There is no reference concept associated with any value type.

4.1.1 SIMPLE VARIABLES

A simple-variable is any variable which is not a subscripted-variable.

Value type variables represent values of type (short) integer, (long) real, Boolean or character.

A reference type variable has an object as its value (or the value none). Text variables are described below.

4.1.2 TEXT VARIABLES

A text variable is conceptually an instance of a composite structure with four constituent components (attributes):

```
ref (TEXTOBJ) OBJ;
integer START, LENGTH, POS;
```

Let X be a text variable. Then X.OBJ, X.START, X.LENGTH and X.POS denote the components of X, respectively. These four components are not directly accessible to the user. Instead, certain properties of a text variable are represented by procedures accessible through dot notation. These procedures are described in chapter 7.

The components OBJ, START and LENGTH constitute the text reference part of the variable. They identify the frame referenced (see page 23). POS is used for accessing the individual characters of the frame referenced (see page 121).

The components of a text variable always satisfy one of the following two sets of conditions:

1) OBJ =/= none START >= 1 LENGTH >= 1 START + LENGTH <= OBJ.SIZE + 1 1 <= POS <= LENGTH + 1

2) OBJ == none START = 1 LENGTH = 0 POS = 1

The latter alternative defines the contents of a variable which references no frame. Note that this alternative thereby defines the special text reference notext.

4.1.3 SUBSCRIPTED VARIABLES

Subscripted variables designate values which are components of multidimensional arrays. Each arithmetic-expression of the subscript-list occupies one subscript position of the subscripted-variable and is called a subscript. The complete list of subscripts is enclosed in the subscript parentheses ( ). The array component referred to by a subscripted-variable is specified by the actual numerical value of its subscripts.

Each subscript position acts like a variable of type integer and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable. The value of the subscripted-variable is defined only if the value of the subscript-expression is within the subscript bounds of the array.

4.1.4 REMOTE IDENTIFIERS (DOT NOTATION)

```
attribute-identifier
    = identifier

remote-identifier
    = simple-object-expression "." attribute-identifier
    ! simple-text-expression "." attribute-identifier

identifier-1
    = identifier
    ! remote-identifier
```

Let X be a simple-object-expression qualified by the class C, and let A be an appropriate attribute-identifier. Then the remote-identifier "X.A", if valid, is an attribute identification whose object is the value X and whose qualification is C (cf. pages 41 and 63).

The remote-identifier X.A is valid if the following conditions are satisfied:

1) The value X is different from none.

2) The object referenced by X has no class attribute declared at any prefix level equal or outer to that of C.

Note: Condition 1 corresponds to a check which causes an error if the value of X is none.

Condition 2 is an ad hoc rule intended to simplify the language and its implementations.

A remote-identifier of the form

simple-text-expression.attribute-identifier

identifies an attribute of the text variable identified by evaluating the simple-text-expression, provided that the attribute-identifier is one of the procedure-identifiers listed on page 119.

Note: Even if the expression references the value notext, the attribute access is legal (in contrast to object-expressions).

Example:

Let P1 and P2 be variables declared and initialized as in example 2 on page 37. Then the value of the expression

```
P1.plus (P2)
```

is a new "point" object which represents the vector sum of P1 and P2. The value of the expression

```
P1 qua polar.plus (P2)
```

is a new "polar" object representing the same vector sum.

4.2 FUNCTION DESIGNATORS

```
function-designator
    =  procedure-identifier-1  [ actual-parameter-part ]

procedure-identifier-1
    =  identifier-1

actual-parameter-part
    =  "("  actual-parameter-list  ")"

actual-parameter-list
    =  actual-parameter  (* "," actual-parameter *)

actual-parameter
    =  expression
    !  array-identifier-1
    !  switch-identifier
    !  procedure-identifier-1
```

A function-designator defines a value which results through the application of a given set of rules defined by a procedure declaration (see page 29) to a fixed set of actual-parameters. The rules governing specification of actual-parameters are given on page 31.

Note: Not every procedure declaration defines rules for determining the value of a function-designator (cf. page 30).

Examples:

```
sin(a-b)
j(v+s,n)
r
ss(s-5, !Temperature; T, !Pressure; P)
compile ("( := )", !Stack; q)
```

4.3 BOOLEAN EXPRESSIONS

```
Boolean-primary
    =  logical-value
    !  variable
    !  function-designator
    !  relation
    !  "("  Boolean-expression  ")"

Boolean-secondary
    =  [ "not" ]  Boolean-primary

Boolean-factor
    =  Boolean-secondary  (* "and"  Boolean-secondary *)

Boolean-term
    =  Boolean-factor  (* "or"  Boolean-factor *)

implication
    =  Boolean-term  (* "imp"  Boolean term *)

equivalence
    =  implication  (* "eqv"  implication *)

Boolean-tertiary
    =  equivalence  (* "and" "then"  equivalence *)

simple-Boolean
    =  Boolean-tertiary  (* "or" "else"  Boolean-tertiary *)

Boolean-expression
    =  simple-Boolean
    !  if-clause  simple-Boolean  else  Boolean-expression
```

A Boolean-expression is of type Boolean. It is a rule for computing a logical-value. Except for the operators and then and or else (see page 56), the semantics are entirely analogous to those given for arithmetic-expressions.

Variables and function-designators entered as Boolean-expressions must be declared Boolean.

Examples:

```
x = -2
Y>v  or  z<q
a+b> -5  and  z-d>q**2
p  and  not q  or  x<>y
t.more  and then  t.getchar
x == none  or else  x.a>0
```

```
if  k<1  then  s>w  else h<=c
if ( if ( if a then b else c ) then d else f ) then g else h<k
```

4.3.1 RELATIONS

```
relation
    =  arithmetic-relation
    !  character-relation
    !  text-value-relation
    !  object-relation
    !  object-reference-relation
    !  text-reference-relation

value-relational-operator
    =  <  !  <=  !  =  !  >=  !  >  !  <>

reference-comparator
    =  ==  !  =/=
```

The value-relational operators have the conventional meanings. Their specific interpretation is described below in connection with the respective types. The reference comparators have the same priority level as the relational operators.

4.3.1.1 ARITHMETIC RELATIONS

```
arithmetic-relation
    =  simple-arithmetic-expression
       value-relational-operator  simple-arithmetic-expression
```

The relational operators <, <=, =, >=, > and <> have their conventional meaning (less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to). Arithmetic-relations take on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false. If the involved expressions are of different precision types (e.g. long real and real respectively), conversion to the type with the maximum value range is assumed.

4.3.1.2 CHARACTER RELATIONS

```
character-relation
    =  simple-character-expression
       value-relational-operator  simple-character-expression
```

Character values may be compared for equality and inequality and ranked with respect to the (implementation defined) collating sequence. A relation

x rel y,

where x and y are character values, and rel is any value relational operator having the same truth value as the relation

rank(x) rel rank(y).

4.3.1.3 TEXT VALUE RELATIONS

```
text-value-relation
    =  simple-text-expression
       value-relational-operator  simple-text-expression
```

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 and only if they are unequal and one of the following conditions is fulfilled:

1) T is empty.

2) U is equal to T followed by one or more characters.

3) When comparing T and U from left to right the first nonmatching character in T ranks lower than the corresponding character in U.

4.3.1.4 OBJECT RELATIONS

```
object-relation
    =  simple-object-expression  "is"  class-identifier
    !  simple-object-expression  "in"  class-identifier
```

The operators is and in may be used to test the class membership of an object.

The relation "X is C" has the value true if X refers to an object belonging to the class C, otherwise the value is false.

The relation "X in C" has the value true if X refers to an object belonging to a class C or a class inner to C, otherwise the value is false.

4.3.1.5 OBJECT REFERENCE RELATIONS

```
object-reference-relation
    =  simple-object-expression
       reference-comparator  simple-object-expression
```

The reference comparators "==" and "=/=" may be used for the comparison of references (as distinct from the corresponding referenced values). Two object references X and Y are said to be "identical" if they refer to the same object or if they both are none. In those cases the relation "X==Y" has the value true. Otherwise the value is false.

The relation "X=/=Y" is the negation of "X==Y".

4.3.1.6 TEXT REFERENCE RELATIONS

```
text-reference-relation
    =  simple-text-expression
       reference-comparator  simple-text-expression
```

Let T and U be text variables. The relation "T==U" is equivalent to

```
            T.OBJ == U.OBJ
and T.START = U.START
and T.LENGTH = U.LENGTH
```

Note: The POS components are ignored. Also observe that the relations "T=/=U" and "T=U" may both have the value true. (T and U reference different text frames which contain the same text value.)

The following relations are all true:

```
T = notext eqv T == notext
"" == notext
"ABC" =/= "ABC"              (different occurrences)
```

The following example further illustrates this:

```
class C; begin text T; T:- "ABC" end;
```

The relation "new C.T == new C.T" is here true.

4.3.2 THE LOGICAL OPERATORS

The meaning of the logical operators not, and, or, imp, and eqv is given by the following function table:

b1 b2	false false	false true	true false	true true
not b1	true	true	false	false
b1 and b2	false	false	false	true
b1 or b2	false	true	true	true
b1 imp b2	true	true	false	true
b1 eqv b2	true	false	false	true

The operation Boolean-tertiary and then equivalence denotes "conditional and". If the value of the Boolean-tertiary is false the operation will yield the result false, otherwise it will yield the result of evaluating the equivalence.

The operation simple-Boolean or else Boolean-tertiary denotes "conditional or". If the value of the simple-Boolean is true the operator will yield the result true, otherwise it will yield the result of evaluating the Boolean-tertiary.

If A and B are of type Boolean then the value of A and then B is given by the textual substitution of the Boolean-expression (if A then B else false). Similarly, the operation A or else B is defined by the substitution of the expression (if A then true else B).

Note: In both cases these definitions imply that the redundant evaluation of the second operand is suppressed when the evaluation result is evident from the value of the first operand alone.

4.3.3 PRECEDENCE OF OPERATORS

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

The following rules of precedence hold:

first: arithmetic-expressions according to pages 57 to 61
second: < <= = >= > <> == =/= is in
third: not
fourth: and
fifth: or
sixth: imp
seventh: eqv

eight: and then
nine: or else

The use of parentheses will be interpreted in the sense given on page 61.

4.4 ARITHMETIC EXPRESSIONS

```
arithmetic-expression
    = simple-arithmetic-expression
    ! if-clause simple-arithmetic-expression
      'else' arithmetic-expression

if-clause
    = 'if' Boolean-expression 'then'

simple-arithmetic-expression
    = [ adding-operator ] term (* adding-operator term *)

term
    = factor (* multiplying-operator factor *)

factor
    = primary (* '**' primary *)

primary
    = unsigned-number
    ! variable
    ! function-designator
    ! '(' arithmetic-expression ')'

multiplying-operator
    = '*' ! '/' ! '//'

adding-operator
    = '+' ! '-'
```

An arithmetic-expression is a rule for computing a numerical value. In the case of simple-arithmetic-expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail on page 59 below. The actual numerical 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 procedure, when applied to the current values of the procedure parameters given in the expression. Finally, 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, one out of several simple-arithmetic-expressions is selected on the basis of the actual values of the Boolean-expressions (see page 56). This selection is made as follows: The Boolean-expressions of the if-

clauses are evaluated one by one in sequence from left to right until one having the value true is found. The value of the arithmetic-expression is then the value of the first arithmetic-expression following this Boolean (the longest arithmetic-expression found in this position is understood). If none of the Boolean-expressions have the value true, then the value of the arithmetic-expression is the value of the expression following the final else.

In evaluating an arithmetic-expression, it is understood that all the primaries within that expression are evaluated, except those within any arithmetic-expression that is governed by an if-clause but not selected by it. In the special case where an exit is made from a function-designator by means of a goto-statement, the evaluation of the expression is abandoned when the goto-statement is executed. The order of evaluation of primaries within an arithmetic-expression is strictly left to right.

Examples:

Primaries:

```
7.394_604&-8
sum
w(i + 2,8)
cos( y + z*3.141_592_653_589_793_324&&0 )
( a - 3/y + vu**8)
```

Factors:

```
omega
sum ** cos( y + z*3 )
7.394&-8 ** w(i + 2,8) ** ( a - 3/y + vu ** 8 )
```

Terms:

```
u
omega * sum ** cos(y+z*3)/7.394&-8 ** w(i+2,8) ** (a-3/y + vu**8)
```

Simple arithmetic-expression:

```
u - yu + omega*sum**cos(y+z*3)/7.394&-8 ** w(i+2,8)**(a-3/y+vu**8)
```

Arithmetic expressions:

```
w*u - q(s+cu)**2
if  q>0  then  s+3*q//a  else  2*s+3*q
if  a<0  then  u+v  else   if  a*b>17  then  u/v
     else   if  k  >=  y  then  v/u  else  0
a * sin(omega*t)
0.57&12 * a( n*(n-1)//2 ,0 )
( a*arctan(y)+z ) ** (7+Q)
if  q  then  n-1  else  n
if  a<0  then a/b  else   if  a<>0  then  b/a  else  z
```

4.4.1 OPERATORS AND TYPES

Apart from the Boolean-expressions of if-clauses, the constituents of simple-arithmetic-expressions must be of types (long) real or (short) integer. The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

The operators +, -, and * have their conventional meanings (addition, subtraction, and multiplication). The type of the expression will be long real if any of the operands is of type long real. Otherwise, if any operand is of type real, the result will be of type real. Otherwise the result is of type integer.

The procedure "abs" (see page 154) always returns the same type as its argument, i.e. (long) real or (short) integer.

The operators "/" and "//" both denote division. The operations will constitute a run time error if the dividend has the value zero. The type real division (/), is to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence. Thus for example

```
a/b*7/(p-q)*v/s        means      ( ((a/b)*7)/(p-q) ) * v)/s.
```

The operator / is defined for all combinations of type real and type integer, and will always yield results of type real. The operator // is defined only for operands of type integer, and will yield a result of type integer. If a and b are of type integer, then the value of a//b is given by the function:

```
integer procedure DIV(a, b);
(short) integer a, b;
if b=0 then
     error("..." !div by zero;)
else
begin integer q, r;
      q:= 0;  r:= abs(a);
      for  r:= r - abs(b) while r>=0 do q:= q + 1;
      DIV:= if a<0 eqv b>0 then -q else q
end DIV
```

The operation factor**primary denotes exponentiation, where the factor is the base and the primary is the exponent. Thus for example

```
2**n**k        means      (2**n)**k
```

while the alternative evaluation order is achieved by 2**(n**m).
If r is of type real and x is of any arithmetic type, then the value of x**r is given by the function:

```
<real type> procedure  expr(x,r);  ! type returned equal
        to that of r;
real x;   <arithmetic type> r;
if x>0.0 then
        expr:= exp(r*ln(x))
else if x=0.0 and r>0.0 then
        expr:= 0.0
else
        error("..." !expr undefined;)
```

If i and j are both of type integer, then the value of i**j is given by the function:

```
integer procedure expi(i,j);
<integer type> i,j;
if j<0 or i=0 and j=0 then
        error("..." !expi undefined;)
else
begin integer k,result;
      result:= 1;
      for  k:= 1 step 1 until j do
             result:= result*i;
      expi:= result
end expi
```

If n is of type integer and x is of type real, then the value of x**n is given by the function:

```
<real type>  procedure expn(x, n); ! returned type same
      as type of x;
<real type> x; <integer type> n;
if n=0 and x=0.0 then
      error("..." !expn undefined;)
else
begin  <real type> result; <integer type> i;
      result:= 1.0;
      for i:= abs(n) step -1 until 1 do
           result := result*x;
      expn:= if n<0 then 1.0/result else result
end expn
```

It is understood that the finite deviations obtained by using the exponentiation operator may be different from those obtained using the procedures "expr" and "expn".

4.4.2 TYPE OF A CONDITIONAL EXPRESSION

The type of an arithmetic-expression of the form

```
if B then SAE else AE
```

does not depend upon the value of B. The expression is of type (long) real if either SAE or AE is (long) real. Otherwise, if both SAE and AE are of type short integer, the type of the expression will be short integer, else the type will be integer.

4.4.3 OPERATOR PRECEDENCE

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

The following rules of precedence hold:

first:	**
second:	* / //
third:	+ -

The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

4.4.4 ARITHMETICS OF REAL QUANTITIES

Numbers and variables of type real must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic-expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different implementations may evaluate arithmetic-expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

4.5 CHARACTER EXPRESSIONS

```
simple-character-expression
;   =  character-constant
    !   variable
    !   function-designator
    !   "("  character-expression  ")"
character-expression
    =  simple-character-expression
    !  if-clause  simple-character-expression
       "else"  character-expression
```

A character-expression is of type character. It is a rule for obtaining a character value.

4.6 TEXT EXPRESSIONS

```
simple-text-expression
    =  "notext"
    !  string
    !  variable
    !  function-designator
    !  "("  text-expression  ")"

text-expression
    =  simple-text-expression
    !  if-clause  simple-text-expression  "else"  text-expression
```

A text-expression is of type text. It is a rule for obtaining an identification of a text variable.

The result of evaluating

- notext, or an empty string, identifies an anonymous text variable whose contents is defined by the second condition set of page 49..

- a non-empty string identifies an anonymous text variable which references a constant text frame whose value is the internal representation of the external character sequence. This frame is always a main frame. The POS component of the anonymous variable equals 1.

- a text variable identifies the variable itself.

- a text function-designator identifies an anonymous text variable which contains a copy of the final contents of the text variable associated with the procedure-identifier during the execution of the procedure in question.

- a text-expression enclosed in parentheses identifies an anonymous text variable which contains a copy of the contents of the text variable identified when evaluating the same expression without parentheses.

- a conditional text-expression identifies an anonymous text variable which contains a copy of the contents of the text variable, identified by the branch which was selected for evaluation.

For further information on the text concept, see chapter 7.

4.7 OBJECT EXPRESSIONS

```
object-expression
    = simple-object-expression
    ! if-clause simple-object-expression "else" object-
      expression

simple-object-expression
    = "none"
    ! variable
    ! function-designator
    ! object-generator
    ! local-object
    ! qualified-object
    ! "(" object-expression ")"

object-generator
    = "new" class-identifier actual-parameter-part

local-object
    = "this" class-identifier

qualified-object
    = simple-object-expression "qua" class-identifier
```

An object-expression is of type ref (qualification). It is a rule for obtaining a reference to an object. The value of the expression is the referenced object or none. The scope of qualification conforms to the same scope rules as other identifiers.

4.7.1 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 or function-designator is qualified as stated in the declaration (or specification, see below) of the variable, 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 innermost class which includes the qualifications of both alternatives. If there is no such class, the expression is illegal.

5) Any 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 (cf. page 41). The qualification is that of the matching declaration, if any, occurring at the innermost prefix level equal or outer to the access level, or if no such match exists, it is that of the virtual specification.

4.7.2 OBJECT GENERATOR

The value of an object generator is the object generated as the result of its evaluation (see page 83).

4.7.3 LOCAL OBJECTS

A local-object "this C" is a meaningful expression provided that the expression is used within the scope of the class-identifier C and within

1) the class body of C or that of any subclass of C, or

2) a connection block whose block qualification is C or a subclass of C (see pages 32 through 42).

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 (in the given context). For an instance of a procedure or a class body, "textually enclosing" means containing its declaration.

4.7.4 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 is outer to or equal to D or is a subclass of D. Otherwise, i.e. if C and D belong to disjoint prefix sequences, the qualified-object is illegal.

If the value of X is none or is an object belonging to a class outer to C, the evaluation of X qua C constitutes a run time error. Otherwise, the value of X qua C is that of X. The use of instantaneous qualification enables one to restrict or extend the range of attributes of a concatenated class object accessible through inspection or remote accessing (cf. pages 34 et seq and 41).

4.8 DESIGNATIONAL EXPRESSIONS

```
designational-expression
    =  simple-designational-expression
    !  if-clause  simple-designational-expression
       "else"  designational-expression

simple-designational-expression
    =  label
    !  switch-designator
    !  "("  designational-expression  ")"

switch-designator
    =  switch-identifier  "("  subscript-expression  ")"

switch-identifier
    =  identifier

label
    =  identifier
```

A designational-expression is a rule for obtaining a label of a statement. The principle of the evaluation is entirely analogous to that of arithmetic-expressions. In the general case the Boolean-expressions of the if-clauses will select a simple-designational-expression. If this is a label the desired result is already found. A switch-designator refers to the corresponding switch declaration and, by the actual numerical value of its subscript-expression, selects one of the designational-expressions listed in the switch declaration, by counting these from left to right. Since the designational-expression thus selected may again be a switch-designator, this evaluation is obviously a recursive process.

The evaluation of the subscript-expression is analogous to that of subscripted-variables. The value of a switch-designator is defined only if the subscript-expression assumes one of the values 1, 2, ..., n, where n is the number of entries in the switch list.

CHAPTER 5

STATEMENTS

5 STATEMENTS

```
statement
    =  unconditional-statement
    !  conditional-statement
    !  for-statement
    !  connection-statement
    !  while-statement

unconditional-statement
    =  basic-statement
    !  compound-statement
    !  block

basic-statement
    =  (* label ":" *)  unlabelled-basic-statement

unlabelled-basic-statement
    =  assignment-statement
    !  goto-statement
    !  dummy-statement
    !  procedure-statement
    !  activation-statement
    !  object-generator
```

The units of operation within the language are called statements. They will normally be executed consecutively as written. The sequence of operations may be broken by goto-statements, which define their successor explicitly, or by sequencing procedure calls, which define their successor implicitly. It may be changed by conditional-statements, which may cause certain statements to be skipped or repeated. It may be lengthened by for-statements and while-statements which cause certain statements to be repeated.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound-statements and blocks, the definition of statements must necessarily be recursive. Also since declarations, described in chapter 3, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

5.1 ASSIGNMENT STATEMENTS

```
assignment-statement
    =  value-assignment
    !  reference-assignment
```

```
value-assignment
    =  value-left-part  ":="  value-right-part

value-left-part
    =  destination
    !  simple-text-expression

value-right-part
    =  value-expression
    !  text-expression
    !  value-assignment

destination
    =  variable
    !  procedure-identifier

reference-assignment
    =  reference-left-part  ":-"  reference-right-part

reference-left-part
    =  destination

reference-right-part
    =  reference-expression
    !  reference-assignment
```

Assignment-statements serve for assigning the value of an expression to one or several destinations. Assignment to a procedure-identifier may only occur within the body of a procedure defining the value of the function-designator denoted by that identifier. If assignment is made to a subscripted-variable, the values of all the subscripts must lie within the appropriate subscript bounds, otherwise a run-time error will result.

The operator ":=" (read: "becomes") indicates the assignment of a value to the value type variable or value type procedure-identifier which is the left part of the value-assignment or the assignment of a text-value to the text frame referenced by the left part.

The operator ":-" (read: "denotes") indicates the assignment of a reference to the reference type variable or reference type procedure-identifier which is the left part of the reference-assignment.

A procedure-identifier in this context designates a memory device local to the procedure instance. This memory device is initialized upon procedure entry according to page 42.

The value or reference assigned is a suitably transformed representation of the one obtained by evaluating the right part of the assignment. If the right part is itself an assignment, the value or reference obtained is a copy of its constituent left part after that assignment operation has been completed.

The type associated with all destinations of a left part list must be the same unless all destinations are of arithmetic types in which case also the expression must be of arithmetic type. Otherwise the types of all destinations must coincide with that of the expression.

If the type associated with a destination is Boolean, the expression must likewise be Boolean. If the type is character, the expression must likewise be character.

For the description of the text-value-assignment, see page 72. There is no value-assignment operation for objects.

The type of the value or reference obtained by evaluating the right part must coincide with the type of the left part, with the exceptions mentioned in the following sections.

If the left part of an assignment is a formal name parameter, and the type of the corresponding actual parameter does not coincide with that of the formal specification, then the assignment operation is carried out in two steps.

1) An assignment is made to a fictitious variable of the type specified for the formal parameter.

2) An assignment-statement is executed whose left part is the actual parameter and whose right part is the fictitious variable.

The value or reference obtained by evaluating the assignment is, in this case, that of the fictitious variable.

5.1.1 ARITHMETIC ASSIGNMENT

Note that SIMULA is flexible as regards type conversions in multiple arithmetic assignments - it is not required that all destinations be of the same type.

If the type of the arithmetic-expression differs from that associated with the destinations, an appropriate transfer (conversion) function is understood to be automatically invoked. For transfer from type real to integer the transfer function is understood to yield a result which is the largest integral quantity not exceeding E + 0.5 in the mathematical sense (i.e. without rounding error) where E is the value of the expression. It should be noted that E, being of type real, is defined with only finite accuracy. The type associated with a procedure-identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration.

The process will in the general case be understood to take place in three steps as follows:

1) Any subscript expressions occurring in the destinations are evaluated in sequence from left to right. Any expression which is, or is part of, the left part of an assignment is evaluated prior to the evaluation of the right part.

2) The expression of the statement is evaluated.

3) The value of the expression is assigned to all the destinations, with any subscript expressions having values as evaluated in step 1.

The assignment of an integer value to a short integer variable constitutes a run time error if the value being assigned exceeds the permissible range of the short integer type.

The assignment of a long real value to a real variable may constitute a run time error in the case where the range of the long real value exceeds that of the real value. Alternatively, a precision loss may occur at such an assignment. However this should not constitute a run time error.

The assignment of a real value to a long real variable may constitute a run time error in the case where the range of the real value exceeds that of the long real value.

Example:

Consider the statement

```
X:= i:= Y:= F:= 3.14
```

where X and Y are real variables, i is an integer variable, and F is a formal parameter called by name and specified real. If the actual parameter for F is a real variable, then X, i, Y and F are given the values 3, 3, 3.14 and 3.14 respectively. If the actual parameter is an integer variable, the respective values will be 3, 3, 3.14 and 3.

5.1.2 TEXT VALUE-ASSIGNMENT

Let X be the text variable identified as the result of evaluating the left part of a text-value-assignment, and let Y denote the text variable identified by evaluating the corresponding right part: If X references a constant text frame, or X.LENGTH < Y.LENGTH, then the assignment constitutes an error.

Otherwise, the value of Y is conceptually extended to the right by X.LENGTH - Y.LENGTH blank characters, and the resulting text-value is assigned as the new contents of the text frame referenced by X. Note that if X == notext, the assignment is legal if and only if Y == notext.

Note that the effect of the assignment "X:= Y" is equivalent to that of "X:= copy(Y)", regardless of whether or not X and Y overlap.

The position indicators of the left and the right parts are ignored and remain unchanged.

If X and Y are non-overlapping texts of the same length then after the execution of the value-assignment "X:= Y", the relation "X=Y" is true.

A text procedure-identifier occurring as a value-left-part within the procedure-body is interpreted as a text variable. The corresponding assignment-statement will thus imply an assignment to the local procedure-identifier.

5.1.3 TEXT REFERENCE ASSIGNMENT

Let X be the text variable which constitutes the left part of a text reference-assignment, and let Y denote the variable identified by evaluating the corresponding right part. The effect of the assignment is defined as the four component assignment:

```
X.OBJ:- Y.OBJ;
X.START:= Y.START;
X.LENGTH:= Y.LENGTH;
X.POS:= Y.POS;
```

5.1.4 OBJECT REFERENCE ASSIGNMENT

Let the left part of an object reference-assignment be qualified by the class Cl, and let the right part be qualified by Cr. If the right part is itself a reference-assignment, Cr is defined as the qualification of its constituent left part. Let V be the value obtained by evaluating the right part. The legality and effect of the reference-assignment depend on relationships between Cr, Cl and V, as follows:

1) Cl is of the class Cr or outer to Cr: The reference-assignment is legal and the assignment operation is carried out.

2) Cl is inner to Cr: The reference-assignment is legal. The assignment operation is carried out if V is none or is an object belonging to the class Cl or a class inner to Cl. If not, the execution of the reference-assignment constitutes a run time error.

3) Cl and Cr satisfy neither of the above relations: The reference-assignment is illegal.

Similar rules apply to reference-assignments implicit in for-clauses and the transmission of parameters.

Example:

Let "point" and "polar" be the classes declared in the example of page 37.

```
ref (point) p1, p2; ref (polar) p3;
p1:- new polar (3,4); p2:- new point (5,6);
```

Now the statement "p3:- p1" assigns to p3 a reference to the "polar" object which is the value of p1. The statement "p3:- p2" would cause a run time error.

5.2 CONDITIONAL STATEMENTS

```
conditional-statement
    = (* label ":" *) unlabelled-conditional

unlabelled-conditional
    = if-statement [ "else" statement ]
    ! if-clause for-statement

if-statement
    = if-clause  unconditional-statement
    ! if-clause  connection-statement
    ! if-clause  while-statement

if-clause
    = "if"  Boolean-expression  "then"
```

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean-expressions.

An if-statement is of the form

if b then su

where b is a Boolean-expression and su is an unconditional-statement. In execution, b is evaluated; if the result is true, su is executed; if the result is false, su is not executed.

If su contains a label, and a goto-statement leads to the label, then b is not evaluated, and the computation continues with execution of the labelled statement.

Five forms of unlabelled-conditional-statement exist, namely:

```
if b then Su
if b then Sfor
if b then Sconn
if b then Swhile
if b then Su  else  S
```

where Su is an unconditional-statement, Sfor is a for-statement, Sconn is a connection-statement, Swhile is a while-statement and S is a statement.

The second form is equivalent to

```
if B then begin Sfor end
```

The third form is equivalent to

```
if B then begin Sconn end
```

The fourth form is equivalent to

```
if B then begin Swhile end
```

The last form is equivalent to

```
        begin
           if B then begin Su;  goto GAMMA end;
           S;
GAMMA:  end
```

If S is conditional, and also of this form, a different label must be used instead of Gamma in following the same rule.

Note: The effect of a goto-statement leading into a conditional-statement follows directly from the above explanation of the execution of a conditional-statement.

Examples:

```
if x>0 then n:=n+1
if v>u then v: q:=n+m  else   go  to  r
if s<0  or  p < q  then
       aa: begin  if q<v  then
                                a:= v/s
                        else  y:=2*a
           end
      else   if v>s then a:=v-q
                        else   if v>s-1 then  goto  s
```

5.3 WHILE STATEMENT

```
while-statement
    =  (* label ":" *)  "while"  Boolean-expression  "do"
       statement
```

A while-statement causes a statement to be executed zero or more times.

The Boolean-expression is evaluated. When true, the statement following do is executed and control returns to the beginning of the while-statement for a new test of the Boolean-expression.

When the expression is false, control passes to after the while-statement.

5.4 FOR STATEMENTS

```
for-statement
    = (* label ":" *)  for-clause  controlled-statement

for-clause
    = "for"  controlled-variable  for-right-part  "do"

controlled-statement
    = statement

controlled-variable
    = simple-variable

for-right-part
    = ":="  value-for-list
    ! ":-"  reference-for-list

value-for-list
    = value-for-list-element  (* "," value-for-list-element *)

reference-for-list
    = reference-for-list-element
      (* "," reference-for-list-element *)

value-for-list element
    = value-expression  [ "while"  Boolean-expression ]
    ! text-value
    ! arithmetic-expression
      "step"  arithmetic-expression
      "until"  arithmetic-expression

reference-for-list element
    = reference-expression  [ "while"  Boolean-expression ]
```

A for-clause causes the controlled-statement to be executed repeatedly zero or more times. Each execution of the controlled-statement is preceded by an assignment to the controlled-variable and a test to determine whether this particular for-list element is exhausted.

Assignments may change the value of the controlled-variable during execution of the controlled-statement.

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

5.4.1 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
R: reference-expression
A: arithmetic-expression
B: Boolean-expression
S: controlled-statement

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

ALFA, BETA and DELTA are different identifiers which are not used elsewhere in the program. DELTA identifies a nonlocal simple-variable of the same type as A2.

1. V
   = ====

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

2. A1 step A2 until A3
   = ======================

```
C     := A1;
DELTA := A2;
while DELTA*(C-A3) > 0 do begin
   S;
   DELTA := A2;
   C     := C + DELTA;
end;
next for list element
```

3. V while B
   = ============

```
ALFA: C:= V;
      if B then begin
         S;
         goto ALFA; end;
      next for list element
```

```
4.  R
    = ====

       C:- R;
       S;
       next for list element

5.  R while B
    = ============

ALFA:  C:- R;
       if B then begin
          S;
          goto ALFA; end;
       next for list element
```

5.4.2 THE CONTROLLED VARIABLE

The semantics of this section are valid when the controlled-variable is a simple-variable which is neither a formal parameter called by name, nor a procedure-identifier.

To be valid, all for list elements in a for-statement (defined by textual substitution, see page 77) must be semantically and syntactically valid.

In particular, each implied reference-assignment in examples 4 and 5 of page 77 is subject to the rules of page 73, and each text-value-assignment in examples 1 and 3 of page 77 is subject to the rules of page 72.

5.4.3 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.

5.5 GOTO STATEMENTS

```
goto-statement
    = ( "goto" ! "go" "to" ) designational-expression
```

A goto-statement interrupts the normal sequence of operations, by defining its successor explicitly by the value of a designational-expression. Thus the next statement to be executed will be the one having this value as its label.

Since labels are inherently local, no goto-statement can lead from outside into a block. A goto-statement may, however, lead from outside into a compound-statement.

If the designational-expression is a switch-designator whose value is undefined, the execution of the goto-statement constitutes a run-time error.

See also page 149.

Examples:

```
goto  L8
goto  exit(n+1)
go to Town(if y<0 then N else N+1)
goto if Ab<c then L17
       else q(if w<0 then 2 else n)
```

5.6 PROCEDURE STATEMENT

```
procedure-statement
    =  procedure-identifier-1  [ actual-parameter-part ]
```

A procedure-statement serves to invoke (call for) the execution of a procedure-body. Where the procedure-body is a statement written in SIMULA, the effect of this execution is equivalent to performing the following operations (see through page 82 below) on the program at the time of execution of the procedure-statement.

5.6.1 ACTUAL-FORMAL CORRESPONDENCE

The correspondence between the actual parameters of the procedure-statement and the formal parameters of the procedure-heading is established as follows. The actual parameter list of the procedure-statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

The type correspondence of formal and actual parameters is governed by the following rules:

1) Formal parameters of arithmetic type which are not arrays or procedures can have actual parameters of any arithmetic type. The conversion follows the assignment-statement rules for value type parameters and page 60 for name parameters.

2) A proper procedure can have a type procedure as an actual parameter.

3) Exact type correspondence is required for array parameters irrespective of transmission mode.

5.6.2 VALUE ASSIGNMENT (CALL BY VALUE)

A formal parameter called by value designates initially a local copy of the value (or array) obtained by evaluating the corresponding actual parameter.

All formal parameters quoted in the value part of the procedure-heading as well as value type parameters not quoted in the name part are assigned the values of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure-body. The effect is as though an additional block embracing the procedure-body were created in which these assignments were made to variables local to this fictitious block, with types as given in the corresponding specifications. As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block.

A text parameter called by value is a local variable initialized by the statement

```
FP:- copy(AP)
```

where FP is the formal parameter, and AP is the variable identified by evaluating the actual parameter. (":-" is defined on page 70, and "copy" on page 122).

Value specification is redundant for a parameter of type value.

There is no call by value option for object reference parameters and reference type array parameters.

5.6.3 DEFAULT REPLACEMENT (CALL BY REFERENCE)

Any formal parameter which is not of type value, and which is not quoted in the mode part, is said to be called by reference.

A formal parameter called by reference designates initially a local copy of the reference obtained by evaluating the corresponding actual parameter. The evaluation takes place at the time of procedure entry.

A reference type formal parameter is a local variable initialized by a reference-assignment

```
FP:- AP
```

where FP is the formal parameter and AP is the reference obtained by evaluating the actual parameter. The reference-assignment is subject to the rules on page 73. Since in this case the formal parameter is a reference type variable, its contents may be changed by reference-assignments within the procedure-body.

Although array-, procedure-, label- and switch-identifiers do not designate references to values, there is a strong analogy between references in the strict sense and references to entities such as arrays, procedures (i.e. procedure declarations), program points and switches. Therefore a call by reference mechanism is defined in these cases.

An array-, procedure-, label-, or switch-parameter called by reference cannot be changed from within the procedure or class-body; it will thus reference the same entity throughout its scope. However, the contents of an array called by reference may well be changed through appropriate assignments to its elements.

For an array parameter called by reference, the type associated with the actual parameter must coincide with that of the formal specification. For a procedure parameter called by reference, the type associated with the actual parameter must coincide with or be subordinate to that of the formal specification.

5.6.4 NAME REPLACEMENT (CALL BY NAME)

Call by name is an optional transmission mode available for parameters to procedures. It represents a textual replacement.

Any formal parameter quoted in the name part is replaced, throughout the procedure-body, by the corresponding actual parameter, after enclosing this latter in parentheses if it is an expression but not a variable. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure-body will be avoided by suitable systematic changes of the formal or local identifiers involved.

If the actual and formal parameters are of different arithmetic types, then the appropriate type conversion must take place, irrespective of the context of use of the parameter.

For an expression within a procedure-body which is

a) a formal parameter called by name,

b) a subscripted variable whose array identifier is a formal parameter called by name, or

c) a function designator whose procedure-identifier is a formal parameter called by name,

the following rules apply:

1) Its type 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 page 69. The value or reference obtained by the evaluation is the contents of the fictitious variable.

Also, for a formal text parameter called by name, the following rule applies:

If the actual parameter is a string, then all occurrences of the formal parameter evaluate to the same text frame (see page 23).

Pages 69 through 74 define 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.

5.6.5 BODY REPLACEMENT AND EXECUTION

Following parameter replacement the procedure-body, modified as above, is inserted in place of the procedure-statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure-body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure-statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

5.6.6 RESTRICTIONS

For a procedure-statement to be defined it is necessary that the operations on the procedure-body defined in pages 79 through 82 lead to a correct SIMULA statement.

This imposes the restriction on any procedure-statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule, and some additional restrictions, are the following.

A formal parameter which occurs as a destination within the procedure-body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

A formal parameter which is used within the procedure-body as an array identifier can only correspond to an actual parameter which identifies an array of the same dimensions. In addition, if the formal parameter is called by value, the local array created during the call will have the same subscript bounds as the actual array. Similarly the number, kind and type of any parameters of a formal procedure parameter must be compatible with those of the actual parameter.

5.7 OBJECT GENERATOR STATEMENT

```
object-generator
    = "new" class-identifier [ actual-parameter-part ]
```

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, if any, of the object-generator are evaluated. The parameter values and/or references are transmitted.

2) Control enters the object through its initial begin whereby it becomes operating in the "attached" state (see chapter 9). The evaluation of the object-generator is completed:

   case a: whenever the basic procedure "detach" is executed "on behalf of" the generated object (see page 139 et seq), or

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

The state of the object after the evaluation is either "detached" (case a) or "terminated" (case b).

5.7.1 PARAMETER REPLACEMENT

In general the correspondence between actual and formal parameters is the same for classes as for procedures.

The call by name option is not available for classes. Procedure, label and switch parameters cannot be transferred to classes.

For further information on parameter transmission modes, see pages 31 and 40.

5.8 CONNECTION STATEMENT

```
connection-statement
    =  (* label ":" *)  "inspect"  object-expression
                        connection-part  [ otherwise-clause ]
    !  (* label ":" *)  "inspect"  object-expression
                        "do"  connection-block-2
                        [ otherwise-clause ]

connection-part
    =  when-clause  (* when-clause *)

when-clause
    =  "when"  class-identifier  "do"  connection-block-1

otherwise-clause
    =  "otherwise"  statement

connection-block-1
    =  statement

connection-block-2
    =  statement
```

A connection block may itself be or contain a connection-statement. This "inner" connection-statement will then be the largest possible connection-statement. Consider the following:

```
inspect A when A1 do
     inspect B when B1 do S1        *
              when B2 do S2         *
              otherwise S3;         *
```

The inner connection-statement includes the lines that are marked with an asterisk (*).

The purpose of the connection mechanism is to provide implicit definitions to items 1 and 2 on page 41 for certain attribute identifications within connection-blocks.

The execution of a connection-statement may be described as follows:

1) The object-expression of the connection-statement is evaluated. Let its value be X.

2) If when-clauses are present they are considered one after another. If X is an object belonging to a class equal or inner to the one identified by a when-clause, the connection-block-1 of this when-clause is executed, and subsequent when-clauses are skipped. Otherwise the when-clause is skipped.

3) If a connection-block-2 is present it is executed, except if X is none, in which case the connection-block is skipped.

4) The statement of an otherwise-clause is executed if X is none, or if X is an object not belonging to a class included in the one identified by any when-clause. Otherwise it is skipped.

A statement which is a connection-block-1 or a connection-block-2 acts as a block, whether it takes the form of a block or not. It further acts as if enclosed in a second fictitious block, called a "connection block". During the execution of a connection block the object X is said to be "connected". A connection block has an associated "block qualification", which is the preceding class-identifier for a connection-block-1 and the qualification of the preceding object-expression for a connection-block-2.

Let the block qualification of a given connection block be C and let A be an attribute identifier, which is not a label or switch identifier, defined at any prefix level of C. Then any uncommitted occurrence of A within the connection block is given the local significance of being an attribute identification. Its item 1 is the connected object, its item 2 is the block qualification C. It follows that a connection block acts as if its local quantities are those attributes (excluding labels and switches) of the connected object which are defined at prefix levels outer to and including that of C. Name conflicts between attributes defined at different prefix levels of C are resolved by selecting the one defined at the innermost prefix level (see page 41).

Example:

Let "Polar" be the class declared in the example of page 37. Then within the connection-block-2 of the connection-statement

```
inspect new Polar(4,5) do begin ..... end
```

a procedure "plus" is available for vector addition.

5.9 COMPOUND STATEMENT

```
compound-statement
    =  (* label ":" *) unlabelled-compound

unlabelled-compound
    =  "begin"  compound-tail

compound-tail
    =  statement  (* ";" statement *)  "end"
```

This syntax may be illustrated as follows: Denoting arbitrary statements and labels, by the letters S and L, respectively, the syntactic unit takes the form:

```
L:L:... begin S;S;...S;S end
```

It should be kept in mind that each of the statements S may be a complete compound-statement or block.

Example:

```
        begin x:=0;
            for y:=1 step 1 until n do x := x + a(y);
            if x>q then goto stop
            if x>w-2 then goto s;
    aw: st: w:=x+bob
        end
```

5.10 BLOCKS

```
block
    = subblock
    ! prefixed-block

subblock
    = (* label ":" *) unlabelled-block

unlabelled-block
    = block-head ";" compound-tail

block-head
    = "begin" declaration (* ";" declaration *)
```

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the syntactic unit takes the form:

```
L:L:... begin D;D;...D;S;S;...S;S end
```

It should be kept in mind that each of the statements S may be a complete compound-statement or block.

Every block automatically introduces a new level of nomenclature. This is realized as follows. Any identifier occurring within the block may through a suitable declaration be specified to be local to the block in question. This means that

1) the entity represented by this identifier inside the block has no existence outside it, and

2) any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a

block and not being declared to this block will be non-local to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets begin and end enclose that statement.

A label is said to be implicitly declared in this block-head, as distinct from the explicit declaration of all other local identifiers. In this context a procedure-body, or the statement following a for-clause, must be considered as if it were enclosed by begin and end and treated as a block, this block being nested within the fictitious block of page 80 in the case of a procedure with parameters by value. A label that is not within any block of the program (nor within a procedure-body, or the statement following a for-clause) is implicitly declared in the environmental prefix.

Since a statement of a block may again itself be a block the concepts local and non-local to a block must be understood recursively. Thus an identifier which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

Example:

```
Q:  begin integer i, k; real w;
       for i:=1 step -1 until m do
          for k:=i+1 step 1 until m do begin
             w      := A(i,k);
             A(i,k) := A(k,i);
             A(k,i) := w;
          end for i and k
    end  block Q
```

5.10.1 PREFIXED-BLOCKS

```
prefixed-block
    =  (* label ":" *)  unlabelled-prefixed-block

unlabelled-prefixed-block
    =  block-prefix  main-block

block-prefix
    =  class-identifier  [ actual-parameter-part ]

main-block
    =  unlabelled-block
    !  unlabelled-compound
```

An instance of a prefixed-block is a compound object whose prefix part is an object of the class identified by the block-prefix, and whose main part is an instance of the main-block. The formal parameters of the former are initialized as indicated by the actual parameters of the block-prefix. The concatenation is defined by rules similar to those of page 34 et seq.

The following restrictions must be observed:

1) A class in which reference is made to the class itself through use of "this", is an illegal block-prefix.

2) The class-identifier of a block-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 program is enclosed in a prefixed-block.

Example:

Let "hashing" be the class declared in the example of page 38. Then within the prefixed-block,

```
hashing (64) begin integer procedure hash(T);
                   value T; text T; ... ;
                   ...
             end
```

a "lookup" procedure is available which makes use of the "hash" procedure declared within the main-block.

5.11 DUMMY-STATEMENTS

```
dummy-statement
    = empty
```

A dummy-statement executes no operation. It may serve to place a label.

Example:

```
L:
 begin statements; John: end
```

CHAPTER 6

INPUT/OUTPUT STATEMENTS

6 INPUT/OUTPUT STATEMENTS

The semantics of certain I/O facilities will rely on the intuitive notion of "files", which are collections of data external to the the program and organized in a sequential or addressable manner. Actually, a file may in practice be any kind of external device with communication capabilities, such as a terminal, a sensory device, etc. There are two basic modes of file organization, "sequential files" and "direct files".

Examples of sequential files are:

- a system controlled area of a magnetic disk
- a series of printed lines
- input from a keyboard
- data on a tape.

An example of a direct file is a collection of data on a storage medium which allows random access; e.g. a disk or a drum, where each item is identified by a unique integer.

I/O facilities are introduced through block prefixing. For the purpose of this presentation, this collection of facilities will be described by a class called "BASICIO". This class identifier is not explicitly available to the user. The "file" subclasses are available at any block level of a program. An implementation may restrict the number of different block levels where these class identifiers are used for prefixing.

The overall organization of "BASICIO" is as follows:

```
ENVIRONMENT class BASICIO (INPUT_LINELENGTH, OUTPUT_LINELENGTH);
integer INPUT_LINELENGTH, OUTPUT_LINELENGTH;
begin ref (infile) SYSIN; ref (outfile) SYSOUT;
      ref (infile) procedure sysin;  sysin:- SYSIN;
      ref (printfile) procedure sysout;  sysout:- SYSOUT;

      class file ..........................;
      file class imagefile ................;
      file class bytefile .................;
      imagefile class infile ..............;
      imagefile class outfile .............;
      imagefile class directfile ..........;
      outfile   class printfile ...........;
      bytefile class inbytefile ...........;
      bytefile class outbytefile ..........;

      procedure terminate_program;
      begin ......;   go to STOP  end terminate_program;


      SYSIN:- new infile("...");
      SYSOUT:- new printfile("...");
      SYSIN.open(blanks(INPUT_LINELENGTH));
      SYSOUT.open(blanks(OUTPUT_LINELENGTH));
```

```
      inner;
STOP: SYSIN.close;
      SYSOUT.close
end BASICIO;
```

The user's program acts as if it were enclosed by the following block:

```
BASICIO (inlength, outlength) begin
                inspect SYSIN do
                inspect SYSOUT do
                begin  <external head> <program>   end
end prefixed block
```

The files SYSIN and SYSOUT are opened and (if not done explicitly prior to program termination) closed within "BASICIO", i.e. outside the program itself.

In any program execution the unique instance of this prefixed block constitutes the system head of the outermost quasi-parallel system (see page 140).

The actual values of inlength and outlength are device dependent. Their default values are 80 and 132 respectively.

SYSIN and SYSOUT may represent the input and output aspects of an interactive terminal (in which case inlength and outlength are probably equal). In other cases, for example batch runs, SYSIN may represent record-oriented input and SYSOUT may represent line-printer-oriented output.

A program may refer to the corresponding file objects through SYSIN and SYSOUT respectively. Most attributes of these file objects are directly available as the result of the connection blocks enclosing the program.

The procedure "terminate_program" terminates program execution.

6.1 THE CLASS "FILE"

The class "file" is the common prefix class for all input/output classes.

```
class file(FILENAME);  value FILENAME;   text FILENAME;
begin     text procedure filename;
               filename:- copy(FILENAME);

          Boolean OPEN;
          Boolean procedure isopen;     isopen:= OPEN;

          Boolean procedure deletefile;  .....;

          Boolean procedure renamefile(newname);
          text newname;     .......;
```

```
        Boolean procedure setaccess(mode);   text mode;
              .......;

        ........
end file;
```

Within a program, an object of a subclass of "file" is used to represent a file. A file is either open or closed as indicated by the variable "OPEN". The procedure "isopen" returns the current value of "OPEN". A file is initially closed.

Filename

Each "file" object has a text attribute "FILENAME". It is assumed that this text value identifies an external file which, at "open", through an implementation defined mechanism, becomes associated with the "file" object. If the parameter is notext, the system generates a unique file.

Deletefile

The procedure "deletefile" makes, through an implementation defined mechanism, the associated external file inaccessible for future access. A false result indicates that the intended operation failed (e.g. because the access privileges/protection status of the file were such that deletion was not allowed).

Renamefile

The procedure "renamefile" will change the external file identification to the one given. A false result indicates rename failure (cf. "deletefile").

Open and close In all subclasses of "file" there exist (on some level) the procedures "open" and "close" which perform the opening and closing operations on the file. If a specified access mode (see below) cannot be satisfied at open, the "open" function designator will return the value false. The details of these procedures are implementation defined, but they must conform to the following pattern.

```
Boolean procedure open ..............;
if OPEN then !no effect;
else if ...! the file could be opened successfully; then
begin open:= OPEN:= true;
      .......
end open;

Boolean procedure close;
if OPEN then
begin OPEN:= false;
```

```
      close:= true;
      .........
end close;
```

Setaccess

The procedure "setaccess" sets parameters controlling the nature of the file access. Only one mode may be specified in each "setaccess" call. Upper and lower case of the same letter in parameters are considered equal. Unrecognized or irrelevant modes are ignored and "setaccess" then returns the value false. A specific mode will be interpreted either at next "open" or next "close". A mode which is set after "open" ("close") will not have any effect until the next "open" ("close").

Standard access modes are:

Mode	--- Default values --- Files of type --- In-	Out-	Direct-
Effect at "open":			
(NO)SHARED	SHARED	NOSHARED	NOSHARED
(NO)APPEND	Ignored	NOAPPEND	NOAPPEND
(ANY/NO)CREATE	Ignored	ANYCREATE	NOCREATE
(NO)READONLY	Ignored	Ignored	NOREADONLY
(NO)WRITEONLY	Ignored	Ignored	NOWRITEONLY
BYTESIZE:x	-- sets bytesize for bytefile --		
Effect at "close":			
(NO)REWIND	NOREWIND	NOREWIND	Ignored
(NO)PURGE	NOPURGE	NOPURGE	NOPURGE

An implementation may define additional modes. Such modes should start with a percent character '%' in order to avoid conflicts with future access modes.

Explanation of standard modes:

At "open":

The mode NOSHARED implies that the program should be the only program operating on the file (from "open" to "close"). SHARED implies that simultaneous access is acceptable. Several programs using SHARED access on an outfile may produce unpredictable results (cf. procedure "lock", below in this section).

The mode APPEND implies that output should be added to the end of the file. NOAPPEND implies that after "close" the external file will contain only the output produced between "open" and "close". For out-type files, APPEND implies NOSHARED. For direct files, APPEND implies that output is only possible at the (current) end of the file (this enables multiprocessor logging on the same file).

The mode CREATE implies that that an old file with the given (associated external) name must not exist. If it does, "open" will return false. NOCREATE implies that such a file must exist. The mode ANYCREATE may be used to reset this mode (i.e. allowing either of the two states).

For direct files only:

The mode READONLY implies that only input operations are allowed. An attempt to call outimage results in a runtime error. Ignored if WRITEONLY was set.

The mode WRITEONLY implies that only output operations are allowed. The purpose of this mode is to enable programs to deposit mail like messages in a mailbox file where previously deposited mail should be inaccessible to the sender. Ignored if READONLY was set.

For bytefiles only:

The mode BYTESIZE:<integer> defines the byte size for a bytefile (cf. page 111).

At "close":

The mode REWIND indicates that some resetting of the external device should occur. Typically it may cause a magnetic tape to be rewound.

The mode PURGE implies that the file after the "close" statement may be deleted by the file system; it will then be assumed to be inaccessible to future program access.

Direct type file positioning

Characteristic for direct type files is the procedure "locate" for file positioning. Depending on the orientation of the data handled, the parameter will have local interpretations. The procedure "location" returns the current value of the actual position.

Checkpoint

All files producing output (either output type files or direct type files) contain a Boolean procedure "checkpoint". A "checkpoint" call asks the underlying system to attempt to secure the produced output. Depending on the nature of the associated external device, this causes completion of output transfer (i.e. intermediate buffer contents are transferred) or, if this is not possible or meaningful, a dummy operation, in which case the value false is returned.

File locking

```
integer procedure lock(timelimit,loc1,loc2);
real timelimit; integer loc1,loc2;
begin if LOCKED then unlock;
```

```
      ... ! lock indicated part of file;...
end lock;

Boolean procedure unlock;
begin unlocked:= checkpoint;
      if LOCKED then
      begin ... ! release file;...;
            LOCKED:= false
      end
end unlock;

Boolean procedure locked;  locked:= LOCKED;
```

The direct type files contain procedures for control of simultaneous access of the file (cf. access mode SHARED).

The variable "LOCKED" indicates if the file is currently locked by the executing program. The procedure "locked" returns the current value.

The integer procedure "lock" enables the program to get exclusive access to the whole or part of the file. The procedure accepts the following parameters. "Timelimit" is the (real) time in seconds that is the maximum wait time for the resource. If the timelimit is exceeded the procedure returns the value -1. "Timelimit" less than or equal to zero implies immediate return. The next two parameters, "loc1" and "loc2", identify the part of the file to be locked. Depending on host capabilities, the executing program will be given exclusive access to a part of the file which includes the requested region. On some host (file) systems, lockable units may be greater than the requested region and this may even force the entire file to be locked. If the two parameters are both zero, this will imply locking the whole file. A value of zero returned indicates a successful "lock" operation. A negative value less than -1 indicates "lock" failure and its interpretation is implementation defined. The effect of a "lock" call while the file is locked ("LOCKED" is true) is that the previous lock will be immediately released (prior to the requested new locking attempt).

The Boolean procedure "unlock" eliminates the effect of any preceding "lock" call. The Boolean procedure "checkpoint" is called initially. The returned value is that returned by the "checkpoint" call.

Structure of "file" subclasses

There are three predefined subclasses of "file":

```
"imagefile"  -  image (or line) oriented files
"bytefile"  --  character (or "byte") oriented files
"binfile"  ---  binary (unformatted) files.
```

These subclasses have each three subclasses defining the directional mode of operation; input oriented files, output oriented files and random access oriented files (i.e. directfiles). In addition there is defined a subclass to (imagefile) class outfile for line-printer-oriented output, the class printfile.

6.2 IMAGEFILES

Image oriented files operate on logical images, where an image is a representation of sequential characters. An image has a maximum length.

> Note: In this section (Imagefiles), images are referred to as having associated with them "ordinal" numbers, which they are addressable by. Thus, for example, the procedure "inimage" may perform the transfer of the "first", "second", and so forth external file images. These "ordinal" numbers are stored in variables (e.g. "LOC"), and they are accessible by procedures (e.g. "setimage"). It is important to note that these numbers are actually cardinal numbers (e.g. "one", "two", etc., interpreted in an ordinal sense (e.g. "image number 1", "image number 2", etc.).

6.2.1 THE CLASS "IMAGEFILE"

The class "imagefile" defines the common attributes for all image-oriented files.

```
file class imagefile;
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 imagefile;
```

The variable "image" is used to reference a text frame which acts as a "buffer", in the sense that it contains the external file image currently being processed.

The procedures "setpos", "pos", "more" and "length" are introduced for reasons of convenience.

There are four types of imagefiles:

"infile" a sequential file for which input operations (transfer of data from file to program) are available.

"outfile" a sequential file for which output operations (transfer of data from program to file) are available.

"directfile" a direct (random) access file with facilities for both input and output.

"printfile" a sequential file with facilities oriented towards line printers (a subclass of "outfile").

6.2.2 THE CLASS "INFILE"

The class "infile" defines image oriented input operations.

```
imagefile class infile;
begin Boolean ENDFILE;
      integer IMAGECOUNT;
      Boolean procedure open(fileimage);
      text fileimage;
      if OPEN then !no effect;
      else if ...! the file could be opened successfully; then
      begin open:= OPEN:= true;
            ENDFILE:= false;
            image:- fileimage;
            image:= notext;
            IMAGECOUNT:= 0;
            setpos(length+1)
      end open;

      Boolean procedure close;
      if OPEN then
      begin .......;
            image:- notext;
            OPEN:= false;
            close:= ENDFILE:= true
      end close;

      Boolean procedure setimage(i);   integer i;
      if i > 0 then
      begin if i < IMAGECOUNT then
            begin if ... ! Successful "reset" ; then
                  begin IMAGECOUNT:= 0;
                        setimage:= true
                  end
            end else setimage:= true;
            while IMAGECOUNT < i and not ENDFILE do
            inimage
      end setimage;

      Boolean procedure endfile;  endfile:= ENDFILE;
```

```
integer procedure imagecount;  imagecount:= IMAGECOUNT;

procedure inimage;
if not OPEN then
error("..." !Inimage on closed file ;) else
if ENDFILE then
error("..." !Inimage on exhausted file ;) else
begin ... ! attempt to store external image in "image";...;
      if ... ! "image" too short; then
          error("..."
                !Inimage on too long external image ;)
      else
      begin if ... ! there was no more to read; then
            begin ENDFILE:= true;
                  image:= "!25!"
            end else
            begin.. ! pad image with blank(s);...;
                  IMAGECOUNT:= IMAGECOUNT+1
            end
      end;
      setpos(1)
end inimage;

Boolean procedure inrecord;
if not OPEN then
error("..." !Inrecord on closed file ;) else
if ENDFILE then
error("..." !Inrecord on exhausted file ;) else
begin ... ! attempt to store external image in "image";...;
      if ... ! no more to read; ... then
      begin ENDFILE:= true;
            image:= "!25!";
            n:= 1
      end else
      begin n:= ... ! number of characters transferred; ...;
            inrecord:= not
            ...! whole external record received?;...;
            IMAGECOUNT:= IMAGECOUNT+1
      end;
      ! Note, no blanking of rest of image;
      setpos(n+1)
end inrecord;

character procedure inchar;
begin if not more then inimage;
      inchar:= image.getchar
end inchar;

Boolean procedure lastitem;
begin character c;
      c:= ' ';
      while not ENDFILE and then
          (c = ' ' or else c = '!9!') do
                c:= inchar;
      lastitem:= ENDFILE;
      if c <> ' ' then setpos(pos-1)
end lastitem;
```

```
      integer procedure inint;
      begin text t;
            if lastitem then error("..." ! Inint: End of file ;);
            t:- image.sub(pos,length-pos+1);
            inint:= t.getint;
            setpos(pos+t.pos-1)
      end inint;

      long real procedure inreal; .....;

      integer procedure infrac; .......;

      text procedure intext(w); integer w;
      begin text t;
            intext:- t:- blanks(w);
            while t.more do t.putchar(inchar)
      end intext;
      .......;
      ENDFILE:= true
end infile;
```

An object of the class "infile" is used to represent a sequentially organized input file.

The procedure "inimage" performs the transfer of an external file image into the text "image". A runtime error occurs if the text is notext or if otherwise "image" is too short to contain the external image. If "image" is longer than the external image, the latter is left-justified and the remainder of the text is filled with <blank> characters. The position indicator is set to one.

The procedure "inrecord" is similar to "inimage" with the following exceptions. Whenever the number of characters accessible in the input (physical) image is less than "length", the rest of image is left unchanged. The part of the image that was changed is from pos 1 upto (but not including) resulting value of "POS". Moreover, if the external image is too long, only the "length" first characters are input; the remaining characters are input at the subsequent "inrecord" (or possibly "inimage") statement. The fact that there remain characters in the external image is indicated by the returned value true. Otherwise, if the input of the external image was completed, the value false is returned.

If an "end of file" is encountered, the text value "!25!" is assigned to the text "image" and the variable "ENDFILE" is given the value true. A call on "inimage" or "inrecord" when "ENDFILE" already has the value true constitutes a runtime error.

The procedure "open" gives "ENDFILE" the value false and fills "image" with blanks.

The procedure "endfile" gives access to the value of the variable "ENDFILE".

The variable "IMAGECOUNT" represents the number of images that have been read. It is accessible through the procedure "imagecount".

The procedure "setimage" makes subsequent input start from the image indicated by the parameter. The images in the file are associated with ordinal numbers, starting with one. A non-positive parameter value is ignored. On some external devices, for which "setimage" is not meaningful, the returned value is false (in which case there is no action performed), otherwise the value true is returned. IMAGECOUNT is adjusted accordingly.

The procedure "inchar" gives access to and scans past the next character.

The procedure "lastitem" has the purpose of skipping past all <blank> and <tab> (ISO code 9) characters. The process of scanning may involve the transfer of several successive external images until the first one containing a non-<blank>, non-<tab> character is encountered. The position indicator is set to designate this character and the returned result is false. If the file contains no further non-<blank>, non-<tab> characters the value true is returned.

The expression "intext(n)" where n is a positive integer is a reference to a new alterable main frame of length "n" containing the next "n" characters of the file. "POS" is set to the position of the following character. The expression "intext(0)" references notext. In contrast to the item-oriented procedures (see below), "intext" operates on a continuous stream of characters, reading several images if necessary.

The procedures "inchar", "intext", "inimage" and "inrecord" may all give access to the contents of the image which corresponds to an "end of file".

The remaining procedures provide mechanisms for "item oriented" input. They will skip past any intermediate sequences of <blank>s and <tab>s by calling "lastitem".

The procedures "inint", "inreal" and "infrac" are defined in terms of the corresponding de-editing procedures of "image". These three procedures will scan past and convert a numeric item starting with the first non-<blank>, non-<tab> character and contained in one image.

6.2.3 THE CLASS "OUTFILE"

The class "outfile" defines image oriented output operations.

```
file class outfile;
begin integer LINE;

      Boolean procedure open(fileimage);
      text fileimage;
      if OPEN then !No effect;
      else if ...! the file could be opened successfully; then
```

```
begin open:= OPEN:= true;
      image:- fileimage;
      setpos(1);
      LINE:= 0
end open;

Boolean procedure close;
if OPEN then
begin if pos <> 1 then outimage;
      OPEN:= false;
      close:= true;
      image:- notext;
      ........
end close;

Boolean procedure checkpoint; ......;

procedure outimage;
begin if not OPEN then
      error("..." ! Outimage on closed file. ;);
      ........
      image:= notext;
      LINE:= LINE+1;
      setpos(1)
end outimage;

procedure outrecord;
begin if not OPEN then
      error("..." ! Outrecord on closed file. ;);
      ! transfer image.sub(1,pos-1); ...;
      ! Note, no blanking of image;
      setpos(1)
end outrecord;

integer procedure line;
      line:= LINE;

procedure outchar(c); character c;
begin if not more then outimage;
      image.putchar(c)
end outchar;

text procedure field(w); integer w;
begin if w<0 or w>length then
        error("..." ! Output item out of field. ;);
      if pos+w-1 > length then outimage;
      field:- image.sub(pos,w);
      setpos(pos+w)
end field;

procedure outint(i,w); integer i,w;
begin if w = 0 then field(...).putint(i)
      else
      if w < 0 then
      begin text f;
            f:- field(-w);
            f:= notext;
```

```
                f.sub(1,...).putint(i)
            end else
            field(w).putint(i);
        end outint;

        procedure outfix(r,n,w); long real r;
                                 integer n,w;
        begin ........
            field(w).putfix(r,n)
        end outfix;

        procedure outreal(r,n,w); long real r;
                                  integer n,w;
        begin ........
            field(w).putreal(r,n)
        end outreal;

        procedure outfrac(i,n,w); integer i,n,w;
        begin ........
            field(w).putfrac(i,n)
        end outfrac;

        procedure outtext(t); text t;
        begin if pos > 1 and then t.length > length-pos+1 then outimage;
            t.setpos(1);
            while t.more do outchar(t.getchar);
        end outtext;

        procedure breakoutimage;
        begin if not OPEN then
            error("..." ! Breakoutimage on closed file ;);
            ... ! output image.sub(1,pos-1);...;
            image:= notext;
            setpos(1)
        end breakoutimage;

        ......
end outfile;
```

An object of the class "outfile" is used to represent a sequentially organized output file.

The transfer of an image from the text "image" to the file is performed by the procedure "outimage". The procedure reacts in an implementation defined way if the image length is not appropriate for the external file. (Depending on file type and host system, the external file does not necessarily store trailing blanks from the image.) After the transfer, "image" is cleared to blanks and the position indicator is set to 1. The variable "LINE" is incremented by one. The procedure "line" gives access to current value of "LINE".

The procedure "outrecord" transfers to the file only that part of image which precedes "POS". The contents are not blanked after the transfer, although "POS" is set to one.

The procedure "breakoutimage" outputs the part of image that precedes "POS", suppressing any line terminators. After transfer the image is blanked and "POS" is set to one. On some physical media this operation is not possible and will then have an effect identical to "outrecord" followed by blanking of "image".

The procedure "close" calls "outimage" if the position indicator is different from 1.

The procedure "checkpoint" is described on page 95.

The procedure "outchar" stores a character in the "POS" position of image; if "more" is false, "outimage" is called first.

The remaining procedures provide facilities for "item-oriented" output. Each item is edited into a subtext of "image" of a specified width. The first character of the subtext is the one identified by the position indicator of "image". The position indicator is advanced correspondingly. If an item would extend beyond the last character of "image", the procedure "outimage" is called implicitly prior to the editing operation.

With the exception of "outtext", a runtime error will occur if an item cannot be contained within the full length of "image". Procedure "outtext" will always transfer the complete text parameter contents to the file.

The procedures "outint", "outfix", "outreal" and "outfrac" are defined in terms of the corresponding editing procedures of "image". They include an additional integer parameter which specifies the width of the subtext into which the item is to be edited. A positive width signifies right-justification. A zero width parameter edits the item into the image with no leading blanks. A negative parameter implies using a width equal to the absolute parameter value and the item is left-justified within the field.

6.2.4 THE CLASS "DIRECTFILE"

The class directfile defines image oriented operations on direct access files.

```
imagefile class directfile;
begin integer LOC, MAXLOC;  Boolean ENDFILE, LOCKED;
      integer procedure location;  location:= LOC;

      procedure locate(i); integer i;
      begin if i < 1 then error("..." ! Locate parm. < 1 ;);
            if i > MAXLOC then
                  error("..." ! Locate parm. > MAXLOC ;);
            LOC:= i;
            ......
      end locate;

      integer procedure lastloc;
```

```
begin if not OPEN then
        error("..." ! Lastloc on closed file; );
      lastloc:= ......
end lastloc;

integer procedure maxloc;
begin if not OPEN then
        error("..." ! Maxloc on closed file; );
      maxloc:= ......
end maxloc;


Boolean procedure locked;   locked:= LOCKED;

Boolean procedure open(fileimage);   text fileimage;
if OPEN then !No effect; else
if ...! the file could be opened successfully; then
begin open:= OPEN:= true;
      MAXLOC:= if ! fixed allocation; then
               ... ! fixed value;
               else maxint-1;
      image:- fileimage;
      setpos(1);
      locate(1)
end open;

Boolean procedure close .....;
if OPEN then
begin image:- notext;
      OPEN:= false;
      if LOCKED then unlock;
      close:= ENDFILE:= true;
      MAXLOC:= 0;
      ......
end close;

Boolean procedure checkpoint; ......;

Boolean procedure endfile;   endfile:= ENDFILE;

procedure inimage;
begin setpos(1);
      ENDFILE:= LOC > lastloc;
      if ENDFILE then image:= "!25!" else
      if ... ! external image # LOC already exists ; then
         ... ! transfer to image;...
      else
      begin
            while more do image.putchar('!0!')
            ! Note that pos is now = length+1;
      end not written;
      locate(LOC+1)   ! Location for *next* image;
end inimage;

procedure outimage;
if LOC > MAXLOC then
      error("..." !Outimage, file overflow ;) else
```

```
      begin if not OPEN then
            error("..." ! Outimage on closed file;);
            ... ! output "image" to external image # LOC; ...;
            locate(LOC+1);
            image:= notext;
            setpos(1)
      end outimage;

      Boolean procedure deleteimage;
      begin if ... ! image LOC was written; then
            begin if ... ! delete operation successful; then
                  begin deleteimage:= true;
                        locate(LOC+1);
                  end successful
            end
      end deleteimage;

      integer procedure lock .......;

      Boolean procedure unlock ......;

      character procedure inchar;
      begin while not more do inimage;
            inchar:= image.getchar
      end inchar;

      Boolean procedure lastitem .......;
      integer procedure inint ..........;
      long real procedure inreal .......;
      integer procedure infrac .........;
      text procedure intext ............;
      procedure outchar ................;
      text procedure field .............;
      procedure outint .................;
      procedure outfix .................;
      procedure outreal ................;
      procedure outfrac ................;
      procedure outtext ................;

      .......;
      ENDFILE:= true
end directfile;
```

An object of the class "directfile" is used to represent an external file in which the individual images are addressable by ordinal numbers. The variable "LOC" contains the current ordinal number. When the file is closed, the value of "LOC" is zero. The procedure "location" gives access to the current value of "LOC". The procedure "locate" may be used to assign a given value to the variable. There are no visible side effects, although the assignment may be accompanied by implementation defined checks and (possibly asynchronous) instructions to an external memory device associated with the file. A parameter to "locate" less than one or greater than "MAXLOC" constitutes a runtime error.

The procedure "lastloc" indicates the location of the (so far) highest written image. For a new file the returned value is zero.

The Boolean procedure "endfile" returns true when the file is closed or when an image with location greater than "lastloc" has been input (through "inimage"). It will be set after each "inimage" statement.

The variable "MAXLOC" indicates the highest permitted value of "LOC". On some systems this value corresponds to the size of a preallocated file, while on other systems which allow the file to be dynamically extended, this variable is assigned the value "maxint"-1. The actual value is accessible through procedure "maxloc".

The procedure "checkpoint" is described on page 95.

The procedure "open" locates the first image of the file. The length of image must, at all "inimage" and "outimage" statements, be identical to the length of image at the "open" call.

The procedure "inimage" transfers into the text "image" a copy of the external image as currently identified by the variable "LOC". If the file does not contain an image with an ordinal number equal to the value of "LOC", the effect of the procedure "inimage" is as follows. If the indicated location is greater than "lastloc", then ENDFILE is set to true and the end of file text ("!25!") is assigned to "image". Otherwise, if the image is a non-written image (where there exists at least one written image whose LOC is greater than current LOC), then the image is filled with <null> (ISO code 0) characters and the position indicator is set to "length"+1 (i.e. "more" becomes false). Finally the value of "LOC" is incremented by one through a "locate" call.

The procedure "outimage" transfers a copy of the text value "image" to the external image, thereby storing in the file an external image whose ordinal number is equal to the current value of "LOC". If the file contains another image with the same ordinal number, that image is overwritten. The value of "LOC" is then incremented by one through a "locate" call.

The Boolean procedure "deleteimage" makes the image with current "LOC" effectively un-written. Irrespective of any physical differences on the external medium between never-written images and deleted ones, there is no difference from the program's point of view. Note that this means that "deleteimage" may decrement the value returned by "lastloc" (in case "LOC" was equal to "lastloc").

Outputting images completely filled with <null> (ISO code 0) at the end of the file will not necessarily decrement the "lastloc" value; explicit writing (outimage) of such images should be avoided.

The procedures "lock" and "unlock" (see page 95), provide locking mechanisms. The last two parameters of "lock" indicate the minimum range of locations to be locked (inclusive).

The remaining procedures ("lastitem" to "intext" and "outchar" to "outtext") are defined in accordance with the corresponding procedures in "infile" and "outfile" respectively.

6.2.5 THE CLASS "PRINTFILE"

The class "printfile" defines a class for line printer-oriented output.

```
outfile class printfile;
begin integer LINES_PER_PAGE, SPACING, PAGE;

      procedure linesperpage(n); integer n;
      begin
            LINES_PER_PAGE:=
            if n < 0 then maxint else
            if n = 0 then ... ! default value;...
            else n
      end linesperpage;

      procedure spacing(n); integer n;
      if n < 0 then error("..." !Spacing parm. < 0 ;) else
      if n > LINES_PER_PAGE then
            error("..." !Spacing parm. > LINES_PER_PAGE ;)
      else SPACING:= n;

      integer procedure page;
            page:= PAGE;

      procedure eject(n); integer n;
      begin if not OPEN then
            error("..." ! Eject on closed file ;);
            if n <= 0 then
               error("..." ! Eject parm. <= 0 ;);
            if n > LINES_PER_PAGE then n:= 1;
            if n <= LINE then
            begin ... ! new page;...;
                  PAGE:= PAGE + 1
            end;
            ... ! move to line "n";
            LINE:= n
      end eject;

      Boolean procedure open(fileimage);   text fileimage;
      if OPEN then !No effect;
      else if ...! the file could be opened successfully; then
      begin open:= OPEN:= true;
            image:- fileimage;
            setpos(1);
            PAGE:= 0;
            LINE:= 1;
            eject(1)
      end open;
```

```
      Boolean procedure close;
      if OPEN then
      begin ......
            if pos <> 1 then outimage;
            eject(LINES_PER_PAGE);
            OPEN:= false;
            close:= true;
            LINE:= 0;
            SPACING:= 1;
            LINES_PER_PAGE:= .....;
            image:- notext
      end close;

      procedure outimage;
      begin if not OPEN then
            error("..." ! Outimage on closed file ;);
            if LINE > LINES_PER_PAGE then eject(1);
            ... ! output the image on the line
                  indicated by LINE;
            LINE:= LINE + SPACING;
            image:= notext;
            setpos(1)
      end outimage;

      procedure outrecord;
      begin if not OPEN then
            error("..." ! Outrecord on closed file ;);
            if LINE > LINES_PER_PAGE then eject(1);
            ... ! output image.sub(1,pos-1) on the line
                  indicated by LINE;
            LINE:= LINE + SPACING;
            setpos(1)
      end outrecord;

        ..........;
      SPACING:= 1;
      LINES_PER_PAGE:= .......

end printfile;
```

An object of the class "printfile" is used to represent a line printer-oriented output file. The class is a subclass of "outfile". A file image represents a line on the printed page.

The variable "LINES_PER_PAGE" indicates the maximum number of physical lines that may be printed on each page, including intervening blank lines. An implementation defined value is assigned to the variable at the time of object generation, and when the printfile is closed. The procedure "linesperpage" may be used to change the value. If the parameter to "linesperpage" is zero, "LINES_PER_PAGE" is reset to the original value (assigned at object generation). A parameter value less than zero may be used to indicate an 'infinite' value of LINES_PER_PAGE thus avoiding any automatic calls on "eject".

The variable "SPACING" represents the value by which the variable "LINE" is incremented after the next printing operation. Its value may be changed by the procedure "spacing". A call on the procedure "spacing" with a parameter less than zero or greater than "LINES_PER_PAGE" constitutes an error. The effect of a parameter to "spacing" which is equal to zero may be defined to force successive printing operations on the same physical line. Note however, that on some physical media this may not be possible, in which case spacing(0) has the same effect as spacing(1) (i.e. no overprinting).

The variable "LINE" indicates the ordinal number of the next line to be printed (on the current page), provided that no implicit or explicit "eject" statement occurs. Its value is accessible through the procedure "line". Note that the value of "LINE" may be greater than "LINES_PER_PAGE".

The procedure "eject" is used to position to a certain line identified by the parameter, n. The variable "PAGE" is incremented by one each time an explicit or implicit "eject" implies a new page.

The following cases can be distinguished:

```
n <= 0      : ERROR
n >  LINES_PER_PAGE: Equivalent to eject (1)
n <= LINE   : Position to line number n on the next page
n >  LINE   : Position to line number n on the current page
```

The tests above are performed in the given sequence.

The procedure "page" gives access to the current value of "PAGECOUNT".

The procedures "outimage" and "outrecord" operate according to the rules for "outfiles" (cf. page 101 et seq). In addition, they update the variable "LINE".

The procedures "open" and "close" conform to the rules of page 93. In addition, "close" will output the current value of "image" if "POS" is different from 1 and set LINE to zero.

It is a property of the system defined class "printfile" that "outfile" attributes, which are redeclared at "printfile" level, are not accessible to the user's program through explicit qualification (qua). Thus these "outfile" procedures ("open", "close", "outimage", "outrecord") may be envisaged to include the following initial code:

```
procedure X...;
inspect this outfile when printfile do X...
otherwise ...;
```

6.3 BYTEFILES

6.3.1 THE CLASS "BYTEFILE"

The class bytefile is the common prefix class for all byte oriented files.

```
file class bytefile;
begin short integer BYTESIZE;

      short integer procedure bytesize;
            bytesize:= BYTESIZE;

      procedure open;
      if OPEN then !No effect; else
      if ... ! the file could be opened successfully; then
      begin open:= OPEN:= true;
            BYTESIZE:= .....
      end open;

end bytefile;
```

Bytefiles read and write files as continuous streams of bytes. The access mode "BYTESIZE:x" to class "bytefile" defines the size of transferred bytes (in number of bits). The actual value for the file is accessible through procedure "bytesize". The default value will be implementation defined; this value will be used if x equals "0" (zero).

A successful "open" call requires that an acceptable BYTESIZE was defined at object generation. Otherwise the value false is returned.

Byte values are represented as integers in the range (0:2**BYTESIZE-1)

There are two files of type bytefile:

"inbytefile"	representing a sequential file for which input operations are available.
"outbytefile"	representing a sequential file for which output operations are available.

6.3.2 THE CLASS "INBYTEFILE"

The class "inbytefile" defines input operations on byte oriented files.

```
bytefile class inbytefile;
  begin Boolean ENDFILE;
      Boolean procedure open;
      if OPEN then !no effect; else
      if ... ! the file could be opened successfully; then
      begin open:= OPEN:= true;
            ENDFILE:= false;
            BYTESIZE:= ....
      end open;

      Boolean procedure close;
      if OPEN then
      begin OPEN:= false;
            close:= ENDFILE:= true;
            ..........
      end close;

      Boolean procedure endfile;
            endfile:= ENDFILE;

      short integer procedure inbyte;
      begin if ENDFILE then error("..." ! End of file ;);
            if ... ! no more bytes to read;
            then ENDFILE:= true
            else inbyte:= ...! next byte of size BYTESIZE;...
      end inbyte;

      text procedure intext(t);   text t;
      begin t.setpos(1);
            while t.more and not ENDFILE do
               t.putchar(char(inbyte));
            if ENDFILE then t.setpos(t.pos-1);
            intext:- t.sub(1,t.pos-1)
      end intext;

      ..........;
      ENDFILE:= true
  end inbytefile;
```

An object of the class "inbytefile" is used to represent a byte oriented, sequentially organized input file.

The procedure "inbyte" returns the integer value corresponding to the input byte. If there are no more bytes to read, a zero result is returned. If, prior to an "inbyte" call, "ENDFILE" is true, a runtime error will occur.

The procedure "intext" fills the frame of the parameter "t" with successive input bytes.

The procedure "endfile" returns the value true if there are no more bytes to read.

6.3.3 THE CLASS "OUTBYTEFILE"

The class "outbytefile" defines output operations on byte oriented files.

```
bytefile class outbytefile;
begin short integer BYTEBASE;
      Boolean procedure open;
      if OPEN then !no effect; else
      if ...! the file could be opened successfully; then
      begin open:= OPEN:= true;
            BYTEBASE:= 2**BYTESIZE
      end open;

      Boolean procedure close;
      if OPEN then
      begin OPEN:= false;
            close:= true;
            .......
      end close;

      Boolean procedure checkpoint; ......;

      procedure outbyte(x);    short integer x;
      begin if not OPEN then
            error("..." ! Outbyte on closed file ;);
            if x < 0 or else x >= BYTEBASE then
            error("..." ! Outbyte, illegal byte value ;);
            ... ! output of x;
      end outbyte;

      procedure outtext(t);   text t;
      begin t.setpos(1);
            while t.more do
                  outbyte(rank(t.getchar))
      end outtext;

end outbytefile;
```

An object of the class "outbytefile" is used to represent a sequentially organized output file of bytes.

The procedure "outbyte" outputs a byte corresponding to the parameter value. If the parameter value is less than zero or exceeds the maximum permitted value, as defined by current BYTESIZE, a runtime error occurs. If the file is not open, a runtime error occurs.

The procedure "outtext" outputs all characters in the parameter "t" as bytes.

6.4 FILE NAMING IN THE SINTRAN ENVIRONMENT

In addition to the usual SINTRAN III file name convention, elements of partitioned data sets may also be specified.

Data set specifications have the general form:

```
file-spec  [ specifier ] ....

file-spec
  ::= [ ( [ dir : ] user ) ]  filename
  ::= # unit-number

specifier
  ::= : type
  ::= @ access  [ / connect-strategy ]
  ::= . elementname  [  / D ]
```

Filename and elementname are sequences of letters, digits and minuses, starting with a letter, i.e. standard ND-500 file names.

The connect-strategy is an integer giving information about the file:

```
= 0  -- file contains initial data
= 1  -- uninitialized, empty file
= 2  -- primarily sequential file
= 3  -- combination of 1 and 2
```

Examples:

```
(PACK-ONE:XLIB)QUICK-SORT:SYMB
(ND-SIMULA-AROO)SCREEN-IO:NRF
#100.SCRATCH-1/D:DATA
```

6.5 FILE OPENING

User SYSTEM is never searched implicitly when a file is searched. The search strategy and the result depends on the type of the file being opened as follows.

Infile or Inbytefile:

- If the file is found in user directory -- OK
  otherwise try all library directories : if not found -- ERROR

- If element-specifier is present:
  Check that SINTRAN-file is a SIMULA library file.
  If it is not -- ERROR

Outfile or Outbytefile:

- If the file is found in user directory -- OK
  otherwise try to create a new file in user directory

- If element-specifier is present:
  If SINTRAN-file is a SIMULA library file -- OK
  otherwise if SINTRAN-file is empty -- OK
  otherwise -- ERROR

Printfile or Directfile:

- If the file is found in user directory -- OK
  otherwise try to create a new file in user directory

- If element-specifier is present -- ERROR

SYSIN, SYSOUT or SYSTRACE:

- SYSIN, SYSOUT and SYSTRACE are attached to the job's primary input and output files resp., which will normally be input/output on a terminal.

CHAPTER 7

TEXT HANDLING

7 TEXT HANDLING

7.1 TEXT ATTRIBUTES

The following procedures are attributes of any text variable.

Boolean procedure constant	(see below)
integer procedure start	(see below)
integer procedure length	(see below)
text procedure main	(see below)
integer procedure pos	(cf. page 121)
procedure setpos	(cf. page 121)
Boolean procedure more	(cf. page 121)
character procedure getchar	(cf. page 121)
procedure putchar	(cf. page 121)
text procedure sub	(cf. page 122)
text procedure strip	(cf. page 123)
integer procedure getint	(cf. page 126)
long real procedure getreal	(cf. page 126)
integer procedure getfrac	(cf. page 127)
procedure putint	(cf. page 125)
procedure putfix	(cf. page 125)
procedure putreal	(cf. page 125)
procedure putfrac	(cf. page 125)

They may be accessed by remote identifiers of the dot notation form (cf. page 50), i.e.

simple-text-expression . procedure-identifier

In the following section "X" denotes a text variable unless otherwise specified.

7.2 "CONSTANT", "START", "LENGTH" AND "MAIN"

The following code defines the procedures "constant", "start", "length" and "main.

```
Boolean procedure constant;
constant:= OBJ == none or else OBJ.CONST;

integer procedure start; start:= START;

integer procedure length; length:= LENGTH;

text procedure main;
if OBJ == none then main:- notext else
```

```
begin text T; T.OBJ:- OBJ;
      T.START:= 1;
      T.LENGTH:= OBJ.SIZE;
      T.POS:= 1;
      main:- T;
end;
```

"X.main" is a reference to the main frame which contains the frame referenced by X.

The following relations are true for any text variable X:

```
X.main.length >= X.length
X.main.main == X.main
notext.main == notext
"ABC".main = "ABC"
```

Examples:

```
Boolean procedure overlapping(X,Y); text X,Y;
if X.main == Y.main then
overlapping:= if X.start <= Y.start then
                 X.start + X.length > Y.start
              else
                 Y.start + Y.length > X.start;
```

"overlapping(X,Y)" is true if and only if X and Y reference text frames which overlap each other.

```
Boolean procedure subtext(X,Y); text X,Y;
subtext:= X.main == Y.main
      and then X.start >= Y.start
      and then X.start + X.length <= Y.start + Y.length;
```

"subtext(X,Y)" is true if and only if X references a subframe of Y, or if both reference notext.

7.3 CHARACTER ACCESS

The characters of a text are accessible one at a time. Any text variable contains a "position indicator", which identifies the currently accessible character, if any, of the reference text frame. The position indicator of a given text variable X is an integer in the range (1,X.length+1).

The position indicator of a given text variable may be altered by the procedures "setpos", "getchar", and "putchar" of the text variable. Also any of the procedures defined on pages 124 to 127 may alter the position indicator of the text variable which contains the procedure.

Position indicators are ignored and left unaltered by text reference relations, text value relations and text value assignments.

The following procedures are facilities available for character accessing. They are oriented towards sequential access.

```
integer procedure pos; pos:= POS;

procedure setpos(i); integer i;
POS:= if i < 1 or i > LENGTH + 1 then LENGTH + 1 else i;

Boolean procedure more;
more:= POS <= LENGTH;

character procedure getchar;
if POS > LENGTH then error("..." ! Pos out of range;) else
begin getchar:= OBJ.MAIN(START + POS - 1);
      POS:= POS + 1
end getchar;

procedure putchar(c); character c;
if OBJ == none then error("..." ! Notext error;) else
if OBJ.CONST then error("..." ! Text constant error;) else
if POS > LENGTH then error("..." ! Pos out of range;) else
begin OBJ.MAIN(START + POS - 1):= c;
      POS:= POS + 1
end putchar;
```

Note: The implicit modification of POS is lost immediately if "setpos", "getchar" or "putchar" is successfully applied to a text-expression which is not a variable (cf. page 23).

Example:

```
procedure compact(T); text T;
begin text U; character c;
      T.setpos(1);
      U:- T;
      while U.more do
      begin c:=U.getchar;
            if c <> ' ' then T.putchar(c)
      end;
      while T.more do T.putchar(' ')
end compact;
```

The procedure will rearrange the characters of the text frame referenced by its parameter. The non-blank characters are collected in the leftmost part of the text frame and the remainder, if any, is filled with the blank characters. Since the parameter is called by reference, its position indicator is not altered. The character constant ' ' represents a blank character value.

7.4 TEXT GENERATION

The following standard procedures are available for text frame generation:

```
text procedure blanks(n); integer n;
if n < 0 then error("..." ! Parm. to blank < 0;) else
if n = 0 then blanks:- notext else
begin text T;
      T.OBJ:- new TEXTOBJ(n,false);
      T.START:= 1;
      T.LENGTH:= n;
      T.POS:= 1;
      T:= notext;
      blanks:- T
end blanks;
```

"Blanks(n)", with n > 0, references a new alterable main frame of length n, containing only blank characters. "Blanks(0)" references notext. Observe that the statement "T:= notext" effectively fills the text frame with blank characters.

```
text procedure copy(T); text T;
if T == notext then copy:- notext else
begin text U;
      U.OBJ:- new TEXTOBJ(T.LENGTH,false);
      U.START:= 1;
      U.LENGTH:= T.LENGTH;
      U.POS:= 1;
      U:= T;
      copy:- U
end copy;
```

"Copy(T)", with T =/= notext, references a new alterable main frame which contains the same text value as T.

7.5 SUBTEXTS

Two procedures are available for referencing subtexts (subframes).

```
text procedure sub(i,n); integer i,n;
if i < 0 or n < 0 or i + n > LENGTH + 1 then
error("..." ! Sub out of frame;) else
if n = 0 then sub:- notext else
begin text T;
      T.OBJ:- OBJ;
      T.START:= START + i - 1;
      T.LENGTH:= n;
      T.POS:= 1;
      sub:- T
end;
```

If legal, "X.sub(i,n)" references that subframe of X whose first character is character number i of X, and which contains n consecutive characters. The POS attribute of the expression defines a local numbering of the characters within the subframe. If n = 0, the expression references notext.

If legal, the following Boolean expressions are true for any text variable X:

```
X.sub(i,n).sub(j,m) == X.sub(i+j-1,m)

n <> 0 imp X.main == X.sub(i,n).main

X.main.sub(X.start,X.length) == X
```

text procedure strip;

The expression "X.strip" is equivalent to "X.sub(1,n)", where n indicates the position of the last non-blank character in X. If X does not contain any non-blank character, notext is returned.

Let X and Y be text variables. Then after the value assignment "X := Y", if legal, the relation

```
X.strip = Y.strip
```

has the value true, while the value of X = Y will be true only if X.length = Y.length.

7.6 NUMERIC TEXT VALUES

Note that the names of the syntactic units in this section are in upper case to indicate that these rules concerns syntax for data and not for program text.

```
NUMERIC-ITEM   =  REAL-ITEM  !  GROUPED-ITEM

REAL-ITEM      =  DECIMAL-ITEM  [ EXPONENT ]
                  !  SIGN-PART  EXPONENT

GROUPED-ITEM   =  SIGN-PART  GROUPS  [ DECIMALMARK  GROUPS ]
                  !  SIGN-PART  DECIMALMARK  GROUPS

DECIMAL-ITEM   =  INTEGER-ITEM  [ FRACTION ]
                  !  SIGN-PART  FRACTION

INTEGER-ITEM   =  SIGN-PART  DIGITS
```

```
SIGN-PART       =  BLANKS  SIGN  BLANKS

EXPONENT        =  LOWTEN-CHARACTER  INTEGER-ITEM

GROUPS          =  DIGITS  (* BLANK  DIGITS *)

SIGN            =  EMPTY  !  +  !  -

DIGITS          =  DIGIT  (* DIGIT *)

DIGIT           =  0  !  1  !  2  !  3  !  4
                   !  5  !  6  !  7  !  8  !  9

BLANKS          =  (*  BLANK  !  TAB  *)

BLANK           =  ISO code 32

TAB             =  ISO code 9

EMPTY           =

LOWTEN-CHARACTER  ...initial value...

DECIMALMARK     =  .
```

The syntax applies to sequences of characters, i.e. to text values.

A numeric item is a character sequence which may be derived from NUMERIC-ITEM. "Editing" and "de-editing" procedures are available for the conversion between arithmetic values and text values which are numeric items, and vice versa.

7.7 EDITING PROCEDURES

Editing procedures of a given text variable X serve to convert arithmetic values to numeric items. After an editing operation, the numeric item obtained, if any, is right adjusted in the text frame referenced by X and preceded by as many blanks as necessary to fill the text frame. The final value of the position indicator of X is equal to X.length+1. Note that this increment is lost immediately if X does not correspond to a variable.

A positive number is edited without a sign, a negative number is edited with a minus sign immediately preceding the most significant character. Leading nonsignificant zeros are suppressed, except possibly in an EXPONENT.

If X references a constant text frame or notext, an error is caused. Otherwise if the text frame is too short to contain the resulting numeric item, the text frame into which the number was to be edited, is filled with asterisks.

The following editing procedures are available.

procedure putint(i); integer i;

The value of the parameter is converted to an INTEGER-ITEM which designates an integer equal to that value.

procedure putfix(r,n); long real r; integer n;

The resulting numeric item is an INTEGER-ITEM if n=0 or a DECIMAL-ITEM with a FRACTION of n digits if n>0. It designates a number equal to the value of r or an approximation to the value of r, correctly rounded to n decimal places. If n<0, a runtime error is caused.

procedure putreal(r,n); long real r; integer n;

The resulting numeric item is a REAL-ITEM containing an EXPONENT with a fixed implementation defined number of characters. The EXPONENT is preceded by a SIGN-PART if n=0, or by an INTEGER-ITEM with one digit if n=1, or if n>1, by a DECIMAL-ITEM with an INTEGER-ITEM of 1 digit only, and a fraction of n-1 digits. If n<0 a runtime error is caused.

In putfix and putreal, the numeric item designates that number of the specified form which differs by the smallest possible amount from the value of r or from the approximation to the value of r.

If the parameters to putfix (putreal) are such that some of the printed digits will be without significance, zeros are substituted for these digits (and no error condition is raised).

procedure putfrac(i,n); integer i,n;

The resulting numeric item is a GROUPED-ITEM with no decimal mark if n<=0, and with a decimal mark followed by total of n digits if n>0. Each digit group consists of 3 digits, except possibly the first one, and possibly the last one following a decimal mark. The numeric item is an exact representation of the number i * 10**(-n).

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(40,10);
```

```
.....................
if type = "order" then
begin pay:= amount.getint * price.getfrac;
      total:= total + pay;
      payment.putfrac(pay,2)
end
```

If tr initially holds the text

```
"order                1200    155.75                ..."
```

it will after editing contain

```
"order                1200    155.75     18 690.00 ...".
```

7.8 "DE-EDITING" PROCEDURES

A de-editing procedure of a given text variable X operates in the following way:

1) The longest numeric item, if any, of a given form is located, which is contained in X and contains the first character of X. (Notice that leading blanks and tabs are accepted as part of any numeric item.)

2) If no such numeric item is found, a runtime error is caused.

3) Otherwise the numeric item is interpreted as a number.

4) If that number is outside a relevant implementation defined range, a runtime error is caused.

5) Otherwise an arithmetic value is computed, which is equal to or approximates that number.

6) The position indicator of X is made one greater than the position of the last character of the numeric item. Note that this increment is lost immediately if X does not correspond to a variable.

The following de-editing procedures are available.

integer procedure getint;

The procedure locates an INTEGER-ITEM. The function value is equal to the corresponding integer.

long real procedure getreal;

The procedure locates a REAL-ITEM. The function value is equal to or approximates the corresponding number. An INTEGER-ITEM exceeding a certain implementation defined range may lose precision when converted to real.

integer procedure getfrac;

The procedure locates a GROUPED-ITEM. In its interpretation of the GROUPED-ITEM the procedure will ignore any BLANKS and a possible decimal point.

The function value is equal to the resulting integer.

Note: Thus "getfrac" is able to de-edit more general patterns than those generated by "putfrac".

CHAPTER 8

SEPARATE COMPILATION

8 SEPARATE COMPILATION

```
SIMULA-source-module
   = { external-head }
     ( program ! procedure-declaration ! class-declaration )
```

Program modules include compilable programs, procedure declarations, and class declarations.

Example:

```
external class b, c;  ! external head of class e;
b class e(f); ref (c)f;
begin external class d;
      external procedure aproc;
      ref (d) dref;
      dref:- new d;
      aproc(dref);
end class e;
```

In SIMULA it is possible to compile procedure and class declarations separately. This allows a modular programming style which is especially important when a large system is developed by a group of people. Equally important is the facility called compatible recompilation which allows for prototyping by means of programming stubs.

By compiling modules one by one, it is also possible to construct extremely large programs that would otherwise exceed the compilers capacity. Since only the attributes of e.g. an external class will be seen by the compiler, when it is compiling a program that references that class, the possibility of table overflow is reduced.

8.1 INTRODUCTION TO SEPARATE COMPILATION

A source module may commence with a sequence of external declarations, constituting the external head of that module. If the compiler finds an external declaration of A, and A had an external head when it was compiled, then the declarations of this external head are treated as if they occurred immediately before the external declaration of A. It may happen that an external declaration occurs more than once during this implicit insertion; in that case the duplicates are checked by the compiler to see whether these really describe exactly the same procedure or class. If not, the compiler will report an error in the module, otherwise the declaration is accepted (but naturally it is treated only once).

Notice that this insertion of external heads is done recursively to any depth.

8.1.1 THE ATTRIBUTE FILE

During compilation of a procedure (class), the compiler will generate two files: an object file containing the executable result of the compilation (in NRF), and an attribute file which gives certain characteristics of the procedure or class.

The attribute file contains information which makes it possible to check occurrences in other modules of the procedure (class) against the declaration given, without having to process the complete source text of the module each time an external declaration referring to it occurs.

This includes information on the externally visible attributes of the entity. That is information about the parameters (the sequence, type, and transmission mode), and for procedures the possible resulting type. For classes all additional attributes also occur, specified in the same manner. Thus all information necessary to check e.g. a call of an external procedure is at hand.

In addition certain internal information is stored; most important is the check code. This code is a unique identification of the module generated by the compiler, i.e. no two check codes will ever be equal. When an external declaration occurs explicitly in a source module, the check code is read from the attribute file, and recorded in both the object file and the attribute file of the current compilation.

When an external declaration is brought into the compilation implicitly, as described above, the check code is read from the attribute file and compared to the code previously recorded; these two check codes must be the same. If this is not the case, the attribute file has been changed e.g. by an incompatible recompilation.

The following example should demonstrate some of the points made above:

Compilation 1, to attribute file CATTR:

```
CLASS c; BEGIN ... END;
```

Compilation 2, to attribute file DATTR:

```
EXTERNAL CLASS c="CATTR";
c CLASS d; BEGIN ... END;
```

Compilation 3, to attribute file EATTR:

```
EXTERNAL CLASS d="DATTR";
! Class c is implicitely brought in.  It would,
  however, do no harm if c was also declared.  ;
d CLASS e; BEGIN ... END;
```

Compilation 4, to attribute file FATTR:

```
EXTERNAL CLASS c="CATTR";
c CLASS f; BEGIN ... END;
```

Compilation 5, a main program:

```
BEGIN
   EXTERNAL CLASS e="EATTR";
   EXTERNAL CLASS f="FATTR";
   ! classes c and d are implicitely brought in.
     c is brought in "twice", but these are
     considered as the same class.  ;
   REF(c) rc;
   rc:-NEW e;  rc:-NEW f;   ! are both legal;
END;
```

If C is recompiled (e.g. because it was changed), then all other modules may have to be recompiled (see next section). If, however, D is recompiled, then only class E and the program are candidates for recompilation.

8.1.2 COMPATIBLE RECOMPILATION

The SIMULA Compiler may be instructed to recompile a procedure (class) by the command: RECOMPILE. If the recompilation shows that the procedure (class) is compatible with the procedure (class) of the previous compilation, then other classes/procedures or programs referring to it by an external declaration need not be recompiled.

The following compatibility rules apply to two versions, here called old-CP and new-CP, of a procedure(class) CP:

1) New-CP and old-CP are compatible if they have the same name (CP), the same type (if procedure) and the same prefix chain (if class). Otherwise they are incompatible.

2) New-CP and old-CP are compatible if they have (textually) the same external head. Otherwise they are incompatible.

3) New-CP and old-CP are compatible if they have the same parameters and the same local attributes: procedures local to new-CP and old-CP must have the same names and the same parameters, and classes local to new-CP and old-CP must have the same name, the same parameters, and the same attributes. "Same" means here: the same names, kinds, and types, and in the same order. Otherwise they are incompatible.

   Observe that a label local to a class body is an attribute of the class. This means that new-CP and old-CP (and possible local classes) must have the same labels to be compatible. The labels within one class must appear in the same order, but they need not designate the same statements. Note that quantities local to sub-blocks, prefixed blocks and connection blocks need not be the

same.

4) New-CP and old-CP are compatible if they have the same nesting of FOR-statements in the procedure(class) body. In other words, the maximum nesting depth at the outermost level cannot have been changed between old-CP and new-CP. Otherwise they are incompatible. FOR-statements within local subblocks, prefixed blocks and connection-blocks need not have the same nesting depth.

   A way to avoid incompatibilities is to enclose each FOR-statement in a dummy block or to enclose the statement part of the class body in a dummy block. The last way can be used only when the labels in the statement part of the class are not used as attributes of the class.

5) Exactly the same rule applies to nested INSPECT-statements as applies to nested FOR-statements (i.e the maximum nesting depth at the outermost level cannot have been changed between old-CP and new-CP). The same means of avoiding incompatibilities apply as well.

6) The occurrence of 'this-CP' affects the compatibility of old-CP and new-CP as follows:

This-CP occurs in:		The two versions are:
old-CP	new-CP	
yes	yes	compatible
yes	no	compatible
no	yes	incompatible
no	no	compatible

   If 'THIS CP' does not occur in old-CP then it may not occur in new-CP. The reason for this is that CP may be used as a prefix to a block. Thus introducing 'THIS CP' in new-CP, without its occurring in old-CP, would make it an illegal block prefix.

If recompilation is compatible, the previous attribute file is left unchanged and is used for the recompiled procedure (class). No new attribute file is produced, and thus no new check code is generated.

If a recompilation is incompatible, then the cause for this is reported. No output file will be generated. (The criterion used to determine whether two modules are compatible or not is that they have the same attribute file contents.)

8.2 EXTERNAL DECLARATIONS

```
external-head
    = external-declaration  (* external-declaration *)

external-declaration
    = external-procedure-declaration
    ! external-class-declaration
```

An external declaration is either an external procedure declaration, treated on page 135, or it is an external class declaration, treated on page 136.

The external declaration is a substitute for a complete introduction of the corresponding source module, including its external head. In the case where multiple but indentical external declarations occur as a consequence of this rule, this declaration will be incorporated only once.

The occurrence of a standard identifier within a source module refers to the declaration of that identifier within the class ENVIRONMENT, implicitly prefixing the main program.

Note: If a class identifier is referenced before the body of a separately compiled procedure or class declaration, or in a program block prefix, then this identifier must be declared in the external head.

8.3 EXTERNAL PROCEDURE DECLARATION

```
external-procedure-declaration
      = "external"  [ kind ]  [ type ]  "procedure"
        external-list
      ! "external"  kind  "procedure"
        external-item  external-specification

kind
      = identifier

external-specification
      = "is"  procedure-declaration
```

where "kind" is an identifier that indicates the source language in which the procedure referred to is programmed, and "type" is the type of the procedure referred to.

If "kind" is omitted, the source language of the procedure is SIMULA.

If a procedure declaration occurs (preceded by the keyword IS), its procedure body must be empty, since it is given in a separate (non-SIMULA) module.

The procedure heading will determine the procedure (or function) identifier to be used within the source module in which the external declaration occurs, as well as the type, order and transmission mode of the parameters. This information is derived automatically by the compiler from the file system in case the first form above is given.

A non-SIMULA procedure cannot be used as an actual parameter corresponding to a formal procedure.

8.4 EXTERNAL CLASS DECLARATION

```
external-class-declaration
      = "external" "class" external-list
```

An external declaration of a separately compiled class implicitly declares all classes in its prefix chain (since these will be declared in the external head of the class in question).

8.5 MODULE IDENTIFICATION

```
external-list
      = external-item  (* "," external-item *)

external-item
      = identifier  [ "=" external-identification ]

external-identification
      = string
```

The identifier given in the external-item must be the same as the identifier of the corresponding class or procedure. If the optional string is given, it is the name of the SINTRAN file on which the module resides.

> Note: At this point ND-500 SIMULA deviates from the standard in that an external non-SIMULA procedure declaration (i.e. where <kind> is non-empty) cannot occur in an external head, but only within modules.

CHAPTER 9

SEQUENCING

9 SEQUENCING

9.1 BLOCK INSTANCES AND STATES OF EXECUTION

The constituent parts of a program execution are dynamic instances of blocks, i.e. sub-blocks, prefixed blocks, connection blocks and class bodies.

A block instance is said to be "local to" the block which (directly) contains its describing text. E.g. an object of a given class is local to the block instance which contains the class declaration. The instance of the outermost block is local to no block instance.

At any time, the "program sequence control", PSC, refers to that program point within a block instance which is currently being executed. For brevity we say that the PSC is "positioned" at the program point and is "contained" in the block instance.

The entry into any block invokes the generation of an instance of that block, whereupon the PSC enters the block instance at its first executable statement. If and when the PSC reaches the final end of a non-class block instance (i.e. an instance of a prefixed block, a sub-block, a procedure body or a connection block) the PSC returns to the program point immediately following the statement or expression which caused the generation of the block instance.

A block instance is at any time in one of four states of execution: "attached", "detached", "resumed" or "terminated".

A non-class block instance is always in the state attached. The instance is said to be "attached to" the block instance which caused its generation. Thus, an instance of a procedure-body is attached to the block instance containing the corresponding procedure-statement or function-designator. A non-class, non-procedure block instance is attached to the block instance to which it is local. The outermost block instance is attached to no block instance. If and when the PSC leaves a non-class block instance through its final end, or through a goto-statement, the block instance ceases to exist.

A class object is initially in the attached state and said to be attached to the block instance containing the corresponding object generator. It may enter the detached state through the execution of a detach statement (see page 146). The object may reenter the attached state through the execution of a call statement (see page 147), whereby it becomes attached to the block instance containing the call statement. A detached object may enter the resumed state through the execution of a resume statement (see page 148). If and when the PSC leaves the object through its final end or through a goto statement, the object enters the terminated state. No block instance is attached to a terminated class object.

The execution of a program which makes no use of detach, call or resume statements is a simple nested structure of attached block instances.

Whenever a block instance ceases to exist, all block instances local or attached to it also cease to exist. The dynamic scope of an object is thus limited by that of its class declaration.

The dynamic scope of an array declaration may extend beyond that of the block instance containing the declaration, due to the call by reference parameter transmission mode being applicable to arrays.

9.2 QUASI-PARALLEL SYSTEMS

A quasi-parallel system is identified by any instance of a sub-block or a prefixed block, containing a local class declaration. The block instance which identifies a system is called the "system head".

The outermost block instance identifies a system referred to as the "outermost system".

A quasi-parallel system consists of "components". In each system one of the components is referred to as the "main component" of the system. The other components are called "object components".

A component is a nested structure of block instances one of which, called the "component head", identifies the component. The head of the main component of a system coincides with the system head. The heads of the object components of a system are exactly those detached or resumed objects which are local to the system head.

At any time exactly one of the components of a system is said to be "operative". A non-operative component has an associated "reactivation point" which identifies the program point where execution will continue if and when the component is activated.

The head of an object component is in the resumed state if and only if the component is operative. Note that the head of the main component of a system is always in the attached state.

In addition to system components, a program execution may contain "independent object components" which belong to no particular system. The head of any such component is a detached object which is local to a class object or an instance of a procedure body, i.e. which is not local to a system head. By definition, independent components are always non-operative.

The sequencing of components is governed by the detach, call and resume statements, defined on page 146 et seq. All three statements operate with respect to an explicitly or implicitly specified object. The following two sections serve as an informal outline of the effects of these statements.

9.2.1 SEMI-SYMMETRIC SEQUENCING: DETACH - CALL

In this section the concept of a quasi-parallel system is irrelevant. Consequently we only consider object components, making no distinction between components which belong to a system and those which are independent.

An object component is created through the execution of a detach statement with respect to an attached object, whereby the PSC returns to the block instance to which the object is attached. The object enters the detached state and becomes the head of a new non-operative component whose reactivation point is positioned immediately after the detach statement.

The component may be reactivated through the execution of a call statement with respect to its detached head, whereby the PSC is moved to its reactivation point. The head reenters the attached state and becomes attached to the block instance containing the call statement. Formally, the component thereby loses its status as a component.

9.2.2 SYMMETRIC COMPONENT SEQUENCING: DETACH - RESUME

In this section we only consider components which belong to a quasi-parallel system.

Initially, i.e. upon the generation of a system head the main component is the operative and only component of the system.

Non-operative object components of the system are created as described in the previous section, i.e. by detach statements with respect to attached objects local to the system head.

Non-operative object components of the system may be activated by call-statements, whereby they lose their component status, as described in the previous section.

A non-operative object component of the system may also be reactivated through the execution of a resume statement with respect to its detached head, whereby the PSC is moved to its reactivation point. The head of the component enters the resumed state and the component becomes operative. The previously operative component of the system becomes non-operative, its reactivation point positioned immediately after the resume statement. If this component is an object component its head enters the detached state.

The main component of the system regains operative status through the execution of a detach statement with respect to the resumed head of the currently operative object component, whereby the PSC is moved to the reactivation point of the main component. The previously operative component becomes non-operative, its reactivation point positioned immediately after the detach statement. The head of this component enters the detached state.

Observe the symmetric relationship between a resumer and its resumee, in contrast to that between a caller and its callee.

9.2.3 DYNAMIC ENCLOSURE AND THE OPERATING CHAIN

A block instance X is said to be "dynamically enclosed" by a block instance Y if and only if there exists a sequence of block instances

X = Z0, Z1, ..., Zn = Y (n>=0)

such that for i= 1,2,...,n:

a) Zi-1 is attached to Zi, or

b) Zi-1 is a resumed object whose associated system head is attached to Zi.

Note that a terminated or detached object is dynamically enclosed by no block instance except itself.

The sequence of block instances dynamically enclosing the block instance currently containing the PSC is called the "operating chain". A block instance on the operating chain is said to be "operating". The outermost block instance is always operating.

A component is said to be operating if the component head is operating.

A system is said to be operating if one of its components is operating. At any time, at most one of the components of a system can be operating. Note that the head of an operating system may be non-operating.

An operating component is always operative. If the operative component of a system is non-operating, then the system is also non-operating. In such a system, the operative component is that component which was operating at the time when the system became non-operating, and the one which will be operating if and when the system again becomes operating.

Consider a non-operative component C whose reactivation point is contained in the block instance X. Then the following is true:

1) X is dynamically enclosed by the head of C.

2) X dynamically encloses no block instance other than itself.

The sequence of block instances dynamically enclosed by the head of C is referred to as the "reactivation chain" of C. All component heads on this chain, except the head of C, identify (non-operating) operative components. If and when C becomes operating, all block instances on its reactivation chain also become operating.

Example:

```
 1  begin comment S1;
 2        ref(C1) X1;
 3        class C1;
 4        begin procedure P1; detach;
 5              P1
 6        end C1;
 7        ref(C2) X2;
 8        class C2;
 9        begin procedure P2;
10              begin detach;
11                    *)
12              end P2;
13              begin comment system S2;
14                    ref(C3) X3;
15                    class C3;
16                    begin detach;
17                          P2
18                    end C3;
19                    X3:- new C3;
20                    resume(X3)
21              end S2
22        end C2;
23        X1:- new C1;
24        X2:- new C2;
25        call(X2)
26  end S1;
```

The execution of this program is explained below. In the figures, system heads are indicated by squares and other block instances by circles. Vertical bars connect the component heads of a system, and left arrows indicate attachment. Reactivation point is abbreviated RP.

Just before, and just after the execution of the detach statement in line 4, the situations are:

Figure 9.1:

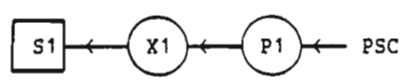
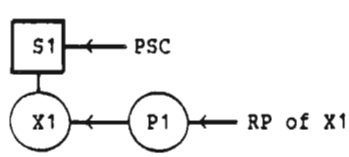


Figure 9.2:



Before and after the detach in line 16:

Figure 9.3:

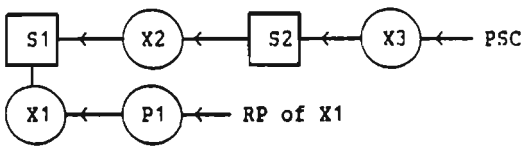


Figure 9.4:

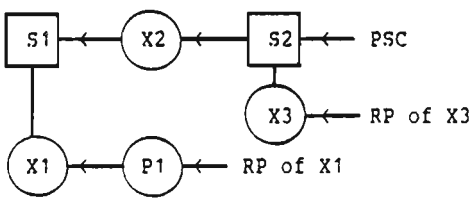
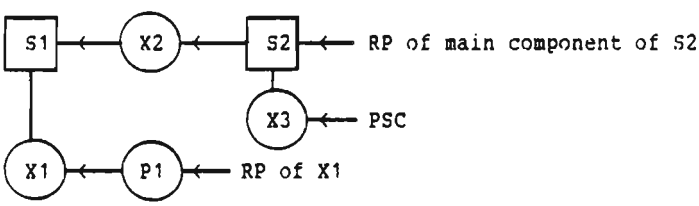


Figure 9.4 also shows the situation before the resume in line 20.
After this resume:

Figure 9.5:



Before and after the detach in line 10:

Figure 9.6:

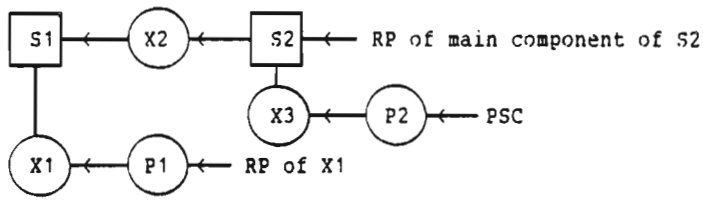
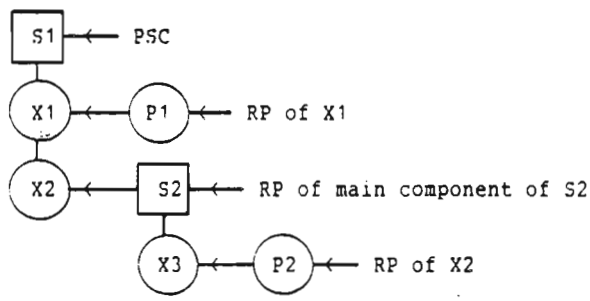
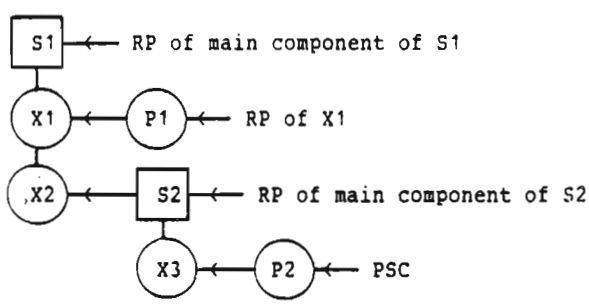


Figure 9.7:



Note that X3 is still the operative component of S2 and does not have a reactivation point of its own. Figure 9.7 also shows the situation before the call in line 25. After this call, the situation in figure 9.6 is re-established. If, however, the call in line 25 is replaced by a "resume(X2)" the following situation arises:

Figure 9.8:



If now a "resume(X1)" is executed at * in line 11, the PSC is moved to the "RP of X1" in figure 9.8, leaving an "RP of X2" at the former PSC. If instead a "detach" is executed, figure 9.8 leads back to figure 9.7.

9.3 QUASI-PARALLEL SEQUENCING

A quasi-parallel system is created through entry into a sub-block or a prefixed block, which contains a local class declaration, whereby the generated instance becomes the head of the new system. Initially, the main component is the operative and only component of the system.

9.3.1 THE DETACH STATEMENT

Any class that has no (textually given) prefix will by definition be prefixed by a fictitious class whose only attribute is:

```
procedure detach; ... ;
```

Thus, every class object or instance of a prefixed block has this attribute. Consider the effect of an invocation of the detach attribute of such a block instance X:

If X is an instance of a prefixed block the detach statement has no effect.

If X is a class object, the following cases arise:

1) X is an attached object. If X is not operating the detach statement constitutes an error.

   If X is operating, the effect of the detach statement is:

   a) X becomes detached and thereby (the head of) a new non-operative object component, its reactivation point positioned immediately after the detach statement. As a consequence, that part of the operating chain which is dynamically enclosed by X becomes the (non-operating) reactivation chain of X.

   b) The PSC returns to the block instance to which X was attached and execution continues immediately after the associated object generator or call statement (see page 147).

   If X is local to a system head, the new component becomes a member of the associated system. It is a consequence of the language definition that, prior to the execution of the detach statement, X was dynamically enclosed by the head of the operative component of this system. The operative component remains operative.

2) X is a detached object. The detach statement then constitutes an error.

3) X is a resumed object. X is then (the head of) an operative system component. Let S be the associated system. It is a consequence of the language definition that X must be operating. The effect of the detach statement is:

   a) X enters the detached state and becomes non-operative, its reactivation point positioned immediately after the detach statement. As a consequence, that part of the operating chain which is dynamically enclosed by X becomes the (non-operating) reactivation chain of X.

   b) The PSC is moved to the current reactivation point of the main component of S, whereby this main component becomes operative and operating. As a consequence, all block instances on the reactivation chain of the main component also become operating.

4) X is a terminated object. The detach statement then constitutes an error.

9.3.2 THE CALL STATEMENT

```
procedure call(x);      ref (...) x;    ......;
```

"Call" is formally a procedure with one object reference parameter qualified by a fictitious class including all classes. Let Y denote the object referenced by a call statement.

If Y is terminated, attached or resumed, or Y == none, the call statement constitutes an error.

If Y is a detached object, the effect of the call statement is:

1) Y becomes attached to the block instance containing the call statement, whereby Y loses its status as a component head. As a consequence the system to which Y belongs (if any) loses the associated component.

2) The PSC is moved to the (former) reactivation point of Y. As a consequence, all block instances on the reactivation chain of Y become operating.

9.3.3 THE RESUME STATEMENT

```
procedure resume(x);   ref (...) x;   ......;
```

"Resume" is formally a procedure with one object reference parameter qualified by a fictitious class including all classes. Let Y denote the object referenced by a resume statement.

If Y is not local to a system head, i.e. if Y is local to a class object or an instance of a procedure-body, the resume statement constitutes an error.

If Y is terminated or attached, or Y==none, the resume statement constitutes an error.

If Y is a resumed object, the resume statement has no effect. (It is a consequence of the language definition that Y must then be operating.)

Assume Y is a detached object being (the head of) a non-operative system component. Let S be the associated system and let X denote (the head of) the current operative component of S. It is a consequence of the language definition that X must be operating, and that X is either the main component of S or local to the head of S. The effect of the resume statement is:

1) X becomes non-operative, its reactivation point positioned immediately after the resume statement. As a consequence, that part of the operating chain which is dynamically enclosed by X becomes the (non-operating) reactivation chain of X. If X is an object component its head enters the detached state.

2) The PSC is moved to the reactivation point of Y, whereby Y enters the resumed state and becomes operative and operating. As a consequence, all block instances on the reactivation chain of Y also become operating.

9.3.4 OBJECT "END"

The effect of the PSC passing through the final end of a class object is the same as that of a detach with respect to that object, except that the object becomes terminated, not detached. As a consequence it attains no reactivation point and loses its status as a component head (if it has such status).

9.3.5 GOTO STATEMENTS

A designational-expression defines a program point within a block instance.

Let P denote the program point identified by evaluating the designational-expression of a goto-statement. Let X be the block instance containing P, and let Y denote the block instance currently containing the PSC.

Consider the execution of the goto-statement:

1) If X equals Y the PSC is moved to P.

2) Otherwise, if Y is the outermost block instance the goto-statement constitutes an error.

3) Otherwise the effect is that of the PSC passing through the final end of Y, after which the process is immediately repeated from 1.

CHAPTER 10

THE STANDARD ENVIRONMENT

10 THE STANDARD ENVIRONMENT

The purpose of the environmental class is to collect all procedures and classes available to all programs. It contains procedures for mathematical functions, text generation, random drawing etc.

General structure:

```
class ENVIRONMENT;
begin character CURRENTLOWTEN, CURRENTDECIMALMARK;

      ! Basic operations ------------------------------;
      ! Procedures mod, rem, abs, sign, entier, epsilon. ;

      ! Text utilities ---------------------------------;
      ! Procedures copy, blanks, char, isochar, rank,    ;
      !   isorank, digit, letter, upc, lowc, lowten,     ;
      !   decimalmark, upcase, lowcase.                  ;
        (last 2 only if not attr)

      ! Scheduling -------------------------------------;
      ! Procedures call, resume                          ;

      ! Mathematical functions -------------------------;
      ! Procedures sqrt, sin, cos, tan, cotan, arcsin,   ;
      !   arccos, arctan, arctan2, sinh, cosh, tanh, ln, ;
      !   log10, exp, pi.                                ;

      ! Extremum functions -----------------------------;
      ! Procedures max, min, maximum, minimum.           ;

      ! Environmental enquiries ------------------------;
      ! Procedures maxrank, maxint, maxshortint, minint, ;
      !   minshortint, maxlongreal, maxreal, minreal,    ;
      !   minlongreal, simulaid, systemid, runid,        ;
      !   sourceline.                                    ;

      ! Error control ----------------------------------;
      ! Procedures enterdebug, error.                    ;

      ! Array quantities -------------------------------;
      ! Procedures dimensions, upperbound, lowerbound.   ;

      ! Random drawing ---------------------------------;
      ! Procedures draw, randint, uniform, normal,       ;
      !   negexp, Poisson, Erlang, discrete, linear,     ;
      !   histd, nextrandom, unique.                     ;

      ! Calendar and timing utilities ------------------;
      ! Procedures date_time, cptime, clocktime.         ;

      ! Miscellaneous utilities ------------------------;
      ! Procedures histo, simple.                        ;
```

```
      ! System class for list handling ------------------;
      ! Class simset.                                     ;

      ! System class for discrete event modelling -------;
      ! Class simulation.                                 ;

      CURRENTDECIMALMARK:= '.';
      CURRENTLOWTEN:= 'E'

end ENVIRONMENT;
```

The class "ENVIRONMENT" defines basic, general purpose facilities available to all programs. The identifier "ENVIRONMENT" is not accessible to the user's program.

10.1 BASIC OPERATIONS

```
integer procedure mod(i,j);   integer i,j;
begin integer res;
      res:= i - (i//j)*j;
      mod:= if res = 0 then 0 else
            if sign(res) <> sign(j) then res+j
            else res
end mod;

integer procedure rem(i,j);   integer i,j;
rem:= i - (i//j)*j;

<type> procedure abs(i);   <type> i;
abs:= if i >= 0 then i else -i;

integer procedure sign(i);   <type> i;
sign:= if i > 0 then  1 else
       if i < 0 then -1 else 0;

integer procedure entier(r);   real r;
begin integer j;
      j:= r;
      entier:= if j > r then j-1 else j
end entier;

<type> procedure epsilon(e,up);   <type> e;   Boolean up;
........;
```

The procedure "mod" returns the mathematical modulo value of its parameters. The procedure "rem" returns the remainder of an integer division.

The procedure "abs" returns the absolute value and the result is of the same <type> as its parameter (e.g. short integer, integer, real or long real).

The procedure "sign" returns an integer zero value if the parameter is zero, minus one for negative parameters, otherwise plus one.

The procedure "entier" returns the integer part of a real item, the value always being less than or equal to the parameter. Thus, "entier(1.8)" returns the value 1, while "entier(-1.8)" returns -2.

The procedure "epsilon(x,true)" returns:

if x is >= 0	the smallest (long) real value which is greater than x,
if x is < 0	the greatest (long) real value which is less than x.

Similarly, "epislon(x,false)" returns:

if x is > 0	the greatest (long) real value which is less than x,
if x is <= 0	the smallest (long) real value which is greater than x.

10.2 TEXT UTILITIES

```
text procedure copy(t);   text t; ......;

text procedure blanks(w);    integer w; .....;

character procedure char(i);     integer i;   ....;
character procedure isochar(i); integer i;   ....;
integer procedure rank(c);       character c; ....;
integer procedure isorank(c);    character c; ....;
Boolean procedure digit(c);      character c; ....;
Boolean procedure letter(c);     character c; ....;
character procedure upc(c);      character c; ....;
character procedure lowc(c);     character c; ....;

character procedure lowten(c);   character c;
if ... ! illegal character;... then
      error("..." ! Lowten error ;) else
begin lowten:= CURRENTLOWTEN;
      CURRENTLOWTEN:= c
end lowten;

character procedure decimalmark(c);   character c;
if c <> '.' and then c <> ',' then
```

```
      error("..." ! Decimalmark error ;) else
begin decimalmark:= CURRENTDECIMALMARK;
      CURRENTDECIMALMARK:= c
end decimalmark;
```

```
text procedure upcase(t);   text t;
begin text s;
      t.setpos(1);   upcase:- s:- t;
      while t.more do t.putchar(upc(s.getchar))
end upcase;

text procedure lowcase(t);   text t;
begin text s;
      t.setpos(1);   lowcase:- s:- t;
      while t.more do t.putchar(lowc(s.getchar))
end lowcase;
```

only if not accepted as attribute

Procedures copy and blanks are described on page 122.

The procedure "char" accepts a parameter in the range (0,maxrank), returning the corresponding internal character value. Other parameter values constitute a run time error.

The procedure "isochar" returns the internal character value corresponding to the indicated character in the ISO 646-1973 set.

Conversely, the procedure "rank" accepts a character parameter returning its corresponding internal integer value.

Likewise, the procedure "isorank" returns the ordinal number in the ISO 646-1973 character set.

The procedure "digit" returns the value true if the parameter is a digit (0..9).

The procedure "letter" returns the value true if the parameter is a letter (a..z or A..Z).

The procedure "upc" ("lowc") returns the given letter character in its upper (lower) case equivalent.

The procedure "lowten" changes the value of the current lowten character to that of the parameter. The previous value is returned. Illegal parameter values will constitute a runtime error. Illegal values are digits, plus-sign (+), minus-sign (-), non printable characters (<= isochar(<blank>)), <del> (ISO code 127), period ('.') and comma (',').

Correspondingly the procedure "decimalmark" changes the current value of the decimal point used in (de-)editing procedures (cf. pages 124 to 127). Legal parameter values are '.' and ','.

10.3 MATHEMATICAL FUNCTIONS

```
real-type procedure sqrt(r);       real-type r; ...;
real-type procedure sin(r);        real-type r; ...;
real-type procedure cos(r);        real-type r; ...;
real-type procedure tan(r);        real-type r; ...;
real-type procedure cotan(r);      real-type r; ...;
real-type procedure arcsin(r);     real-type r; ...;
real-type procedure arccos(r);     real-type r; ...;
real-type procedure arctan(r);     real-type r; ...;
real-type procedure arctan2(y,x); real-type y,x;..;
real-type procedure sinh(r);       real-type r; ...;
real-type procedure cosh(r);       real-type r; ...;
real-type procedure tanh(r);       real-type r; ...;
real-type procedure ln(r);         real-type r; ...;
real-type procedure log10(r);      real-type r; ...;
real-type procedure exp(r);        real-type r; ...;

long real procedure pi;  pi:= 3.14159_26535_89793....&&0;
```

These procedures return long real results whenever a parameter is of this type. Otherwise a real type result is returned.

All procedures return floating point approximations to the associated mathematical functions. Their exact definitions (concerning precision, allowed parameter values etc.) are implementation defined. It should be expected, however, that the procedures should return good approximations to the exact mathematical results.

The procedure "sqrt" returns the square root of the parameter value. Parameter values less than zero constitute a run time error.

The trigonometric functions deal with angles expressed in radians.

The procedure "arctan" returns the arctan value of the parameter (i.e. the function inverse to "tan"). The result is in the range (0,pi/2) for non-negative parameters and in the range (-pi/2,0) for negative parameters.

The procedure "arctan2" accepts two parameters, y and x. The result is in the range(-pi,pi) and a negative value is returned whenever y is negative. Positive y values always result in a postive result, while a zero value returns zero if x is positive and pi if x is negative. If both y and x are zero, a runtime error occurs. (For positive x, in most cases arctan2(y,x) equals arctan(y/x).)

In addition, the procedure "pi" returns the long real (approximation) value of the mathematical constant "pi".

10.4 EXTREMUM FUNCTIONS

```
<type> procedure max(i1,i2);   <type> i1,i2; ......;

<type> procedure min(i1,i2);   <type> i1,i2; ......;

integer procedure maximum(arr);   <type> array arr; ......;

integer procedure minimum(arr);   <type> array arr; ......;
```

The procedure "max" ("min") returns the maximum (minimum) value of the two parameters. Legal types are character, text, real-type and integer-type. The resulting type conforms to the rules on page 59.

The procedure "maximum" returns the index of the first element in the one-dimensional array parameter which is greater than or equal to all other elements in the array. It is considered an error if the array has more than one dimension or if the array type is not any of the types (long) real, (short) integer, character or text.

Correspondingly, the procedure "minimum" returns the index of the first element that is less than or equal to all other elements of the array parameter.

10.5 ENVIRONMENTAL ENQUIRIES

```
integer procedure maxrank; .......;
integer procedure maxint; ..........;
short integer procedure maxshortint; .....;
integer procedure minint; ..........;
short integer procedure minshortint; .....;
long real procedure maxlongreal; .....;
real procedure maxreal; .........;
real procedure minreal; .........;
long real procedure minlongreal; .....;

text procedure simulaid;   .......;
text procedure systemid;   .......;
text procedure runid;      .......;

integer procedure sourceline;  ......;
```

The procedures "max<type>" and "min<type>" return the maximum and minimum possible value for each type, respectively.

<type>	Max	Min
short integer	maxshortint	minshortint
integer	maxint	minint
long real	maxlongreal	minlongreal
real	maxreal	minreal

The procedure "maxrank" returns the maximum allowed argument to the procedure "rank".

The procedure "simulaid" returns a reference to a new constant text frame with the following contents:

"<SIMULA system identification>".

The text is intended to contain information on the version number and date for the currently executing SIMULA implementation.

The procedure "systemid" returns a reference to a new constant text frame with the contents:

"<site identification>!!!<name of OS>!!!<CPU manufacturer and type>".

The first part is intended to be the identification of the installation (typically an organizational name), the second part the name of the currently executing operating system and the third part the make and type name of the currently executing CPU (e.g. "IBM-370/158", "DEC-KL1091"). It is recommended that the manufacturer's name be followed by a minus sign.

The procedure "runid" returns a constant text frame with contents:

"<user id>!!!<job id>!!!<account id>!!!<program id>".

The <user id> part contains the user's name on the system (it may on some systems be identical to the <account id>). The <job id> part is the current job number (session number). On single-user systems this part may be empty. The <account id> part should contain the current account identification. The <program id> part should contain the name of the executing program (on some systems this may be equivalent to the identification of the external file in which the program is stored.

The procedure "sourceline" returns an integer indicating the source line number where the procedure was called. The interpretation of this number is implementation defined; it will normally indicate the source line number as defined within the source file used when the calling code was compiled.

10.6 ERROR CONTROL

```
procedure error(t);   text t;
begin ... display text "t" and stop program...
end error;

Boolean procedure enterdebug(continue);
Boolean continue; .....;
```

The procedure "error" stops the execution of the program as if a runtime error has occurred and present the contents of the text parameter on the diagnostic channel (normally the controlling terminal).

The procedure "enterdebug" invokes a system utility for interactive examination of the program. If the parameter "continue" is true the user will be allowed to continue program execution afterwards. The procedure returns the result false if the invocation failed.

10.7 ARRAY QUANTITIES

```
integer procedure dimensions(a);   <type> array a;
dimensions:= .......;

integer procedure upperbound(a,i);   <type> array a;
integer i;   upperbound:= .......;

integer procedure lowerbound(a,i);   <type> array a;
integer i;   lowerbound:= .......;
```

The procedure "dimensions" returns the number of indices of the actual array parameter.

The procedures "upperbound" and "lowerbound" return the upper and lower bounds respectively of the dimensions of the given array corresponding to the given index. An index value less than one or greater than "dimensions(..)" constitutes a runtime error.

10.8 RANDOM DRAWING

10.8.1 PSEUDO-RANDOM NUMBER STREAMS

All random drawing procedures of SIMULA 67 are based on the technique of obtaining "basic drawings" from the uniform distribution in the interval <0,1>.

A basic drawing replaces the value of a specified integer variable, say U, by a new value according to an implementation defined algorithm.

As an example, the following algorithm may be suitable for binary computers:

```
U(i+1) = remainder ((U(i) * 5**(2*p+1)) // 2**n)
```

where U(i) is the i'th value of U, n is an integer related to the size of a computer word and p is a positive integer. It can be proved that, if U(0) is a positive odd integer, the same is true for all U(i) and the sequence U(0), U(1), U(2), ... is cyclic with period 2**n-2. (The last two bits of U remain constant, while the other n-2 take on all possible combinations). Thus there are two sequences - one in the range (1:2**n-3) and the other in (3:2**n-1).

It is a property of this algorithm that any successor to a stream number U(i), e.g. U(i+m), can be computed using modular arithmetic in 2*log(m) steps.

The real numbers u(i) = U(i) * 2**(-n) are fractions in the range <0,1>. The sequence u(1), u(2), ... is called a "stream" of pseudo-random numbers, and u(i) (i = 1,2, ...) is the result of the i'th basic drawing in the stream U. A stream is completely determined by the initial value U(0) of the corresponding integer variable. Nevertheless, it is a "good approximation" to a sequence of truly random drawings.

By reversing the sign of the non-zero initial value U(0) of a stream variable, the antithetic drawings 1-u(1), 1-u(2), ... should be obtained. In certain situations it can be proved that means obtained from samples based on antithetic drawings have a smaller variance than those obtained from uncorrelated streams. This can be used to reduce the sample size required to obtain reliable estimates.

10.8.2 RANDOM DRAWING PROCEDURES

The following procedures all perform a random drawing of some kind. Procedures "draw", "randint", "uniform", "negexp", "discrete", "linear" and "histd" always perform the operation by means of one single basic drawing, i.e. the procedure has the side effect of advancing the specified stream by one step. The necessary type conversions are effected for the actual parameters, with the exception of the last one. The latter must always be an integer variable specifying a pseudo-random number stream. Note, that it must not be a procedure parameter transferred by name.

1) Boolean procedure draw (a,U); name U; real a; integer U;

   The value is true with the probability a, false with the probability 1 - a. It is always true if a >= 1 and always false if a <= 0.

2) integer procedure randint (a,b,U); name U; integer a,b,U;

   The value is one of the integers a, a+1, ..., b-1, b with equal probability. If b < a, the call constitutes an error.

3) real procedure uniform (a,b,U); name U; real a,b; integer U;

   The value is uniformly distributed in the interval a <= u < b. If b < a, the call constitutes an error.

4) 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 formula may be used for the normal distribution function.

5) real procedure negexp (a,U); name U; real a; integer U;

   The value is a drawing from the negative exponential distribution with mean 1/a, defined by -ln(u)/a, where u is a basic drawing. This is the same as a random "waiting time" in a Poisson distributed arrival pattern with expected number of arrivals per time unit equal to a. If a is non-positive, a runtime error occurs.

6) integer procedure Poisson (a,U); name U; real a; integer U;

   The value is a drawing from the Poisson distribution with parameter a. It is obtained by n+1 basic drawings, u(i), where n is the function value. n is defined as the smallest non-negative integer for which

$$\sum_{i=0}^{n} u(i) < e^{**}(-a)$$

The validity of the formula follows from the equivalent condition

$$\sum_{i=0}^{n} -\ln(u(i))/a > 1$$

where the left hand side is seen to be a sum of "waiting times" drawn from the corresponding negative exponential distribution.

When the parameter a is greater than some implementation defined value, for instance 20.0, the value may be approximated by entier(normal(a,sqrt(a),U) + 0.5) or, when this is negative, by zero.

7) real procedure Erlang (a,b,U); name U; real a,b; integer U;

The value is a drawing from the Erlang distribution with mean 1/a and standard deviation 1/(a*sqrt(b)). It is defined by b basic drawings u(i), if b is an integer value,

$$-\sum_{i=1}^{b} \ln(u(i))/(a*b)$$

and by c+1 basic drawings u(i) otherwise, where c is equal to entier(b),

$$-\left(\sum_{i=1}^{c} \ln(u(i))/(a*b)\right) - (b-c)*\ln(u(c+1))/(a*b)$$

Both a and b must be greater than zero.

The last formula represents an approximation.

8) integer procedure discrete (A,U); name U; real array A; integer U;

The one-dimensional array A, augmented by the element 1 to the right, is interpreted as a step function of the subscript, defining a discrete (cumulative) distribution function. The array is assumed to be of type real.

The function value is an integer in the range (lsb, usb+1), where lsb and usb are the lower and upper subscript bounds of the array. It is defined as the smallest i such that A(i) > u, where u is a basic drawing and A(usb+1) = 1.

9) 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 of the same length, that the first and last elements of A are equal to 0 and 1 respectively and that A(i) >= A(j) and B(i) > B(j) for i > j. If any of these conditions are not satisfied, the effect is implementation defined.

The steps in the function evaluation are:

i) Draw a uniform <0,1> random number, u.

ii) Determine the lowest value of i, for which

A(i-1) <= u <= A(i)

iii) Compute D = A(i) - A(i-1)

iv) if D = 0: linear = B(i-1)

if D <> 0: linear = B(i-1) + (B(i)
- B(i-1))*(u-A(i-1))/D

10) integer procedure histd (A,U); name U; real array A; integer U;

The value is an integer in the range (lsb,usb), where lsb and usb are the lower and upper subscript bounds of the one-dimensional array A. The latter is interpreted as a histogram defining the relative frequencies of the values.

10.8.3 SUPPLEMENTARY PROCEDURES

```
integer procedure nextrandom(n,u);   integer n,u; .......;

integer procedure unique;   unique:= .......;
```

The procedure statement "nextrandom(n,u)" returns the "n"'th successor to the stream number "u". Negative values for "n" are permitted, implying the corresponding predecessor.

Procedure "unique" returns some "unpredictable" non-negative integer value, to be used for example in simulation programs where varying starting values are required for random generator seeds. (It may in practice be some reading of the internal machine clock or something similar.)

10.9 CALENDAR AND TIMING UTILITIES

```
text procedure date_time;    date_time:- Copy("....");

real procedure cptime;       cptime:= .............;

real procedure clocktime;    clocktime:= ..........;
```

The procedure "date_time" returns a new constant text frame containing the current date and time in the form YYYY-MM-DD HH:MM:SS.sss.... The number of decimals in the field for seconds is implementation dependent.

The procedure "cptime" returns the number of processor seconds spent by the currently executing program.

The procedure "clocktime" returns the number of seconds since midnight.

10.10 MISCELLANEOUS UTILITIES

```
procedure histo(a,b,c,d);   real array a,b;
real c,d;  ......;

Boolean procedure simple(x); name x; <type> x;
simple:= ... is actual parameter "x" simple? ...;
```

Procedure statement "histo(A,B,c,d)" updates a histogram defined by the one-dimensional arrays A and B according to the observation c with the weight d. A(lba+i) is increased by d, where i is the smallest integer such that c <= B(lbb+i) and lba and lbb are the lower bounds of A and B respectively. If the length of A is not one greater than that of B the effect is implementation defined. The last element of A corresponds to those observations which are greater than all elements of B.

The procedure "simple" returns the Boolean value true only if the indicated procedure actual parameter transferred by name is legal as the left hand side of an assignment. In the case of a text parameter, the result is true only if the actual parameter is a text variable (i.e. a text reference assignment would be legal - a text value

assignment may still be invalid; e.g. if the referenced value is a constant).

10.11 SYSTEM CLASSES FOR LIST HANDLING (SIMSET) AND DISCRETE EVENT MODELLING (SIMULATION)

```
class simset; .......;

simset class simulation; ........;
```

The two classes "simset" and "simulation" are available at any block level of a program. An uncommitted occurrence of the identifier "simset" or "simulation" will act as if an appropriate declaration of the corresponding system class were part of the block head of the smallest textually enclosing block. An implementation may restrict the number of different block levels at which such implicit declarations may occur in any one program.

The classes "simset" and "simulation" are described in greater detail in chapters 11 and 12 respectively.

CHAPTER 11

THE CLASS SIMSET

11 THE CLASS SIMSET

The class "simset" contains facilities for the manipulation of circular two-way lists, called "sets".

11.1 GENERAL STRUCTURE

```
class simset;
begin class linkage; ......;
      linkage class link; ......;
      linkage class head; ......;
end simset;
```

The reference variables and procedures necessary for set handling are introduced in standard classes declared within the class "simset". Using these classes as prefixes, their relevant data and other properties are made parts of the object themselves.

Both sets and objects which may acquire set membership have references to a successor and a predecessor. Consequently they are made subclasses of the "linkage" class.

The sets are represented by objects belonging to a subclass "head" of "linkage". Objects which may be set members belong to subclasses of "link" which is itself another subclass of "linkage".

11.2 CLASS "LINKAGE"

```
class linkage;
begin ref (linkage) SUC, PRED;
      ref (link) procedure suc;
          suc:- if SUC in link then SUC else none;

      ref (link) procedure pred;
          pred:- if PRED in link then PRED else none;

      ref (linkage) procedure prev;
          prev:- PRED;

end linkage;
```

The class "linkage" is the common denominator for "set heads" and "set members".

"SUC" is a reference to the successor of the linkage object in the set; "PRED" is a reference to the predecessor.

The value of "SUC" and "PRED" may be obtained through the procedures "suc" and "pred". These procedures will give the value none if the designated object is not a "set" member, i.e. of class "link" or a subclass of "link".

The attributes "SUC" and "PRED" may only be modified through the use of procedures defined within "link" and "head". This protects the user against certain kinds of programming errors.

The procedure "prev" enables a user to access a set head from its first member.

11.3 CLASS "LINK"

```
linkage class link;
begin procedure out;
      if SUC =/= none then
      begin SUC.PRED:- PRED;
            PRED.SUC:- SUC;
            SUC:- PRED:- none
      end out;

      procedure follow(X); ref (linkage) X;
      begin out;
            if X =/= none and then X.SUC =/= none then
            begin PRED:- X;
                  SUC:- X.SUC;
                  SUC.PRED:- X.SUC:- this linkage
            end
      end follow;

      procedure precede(X); ref (linkage) X;
      begin out;
            if X =/= none and then X.SUC =/= none then
            begin SUC:- X;
                  PRED:- X.PRED;
                  PRED.SUC:- X.PRED:- this linkage
            end
      end precede;

      procedure into(S); ref (head) S;
            precede(S);

end link;
```

Objects belonging to subclasses of the class "link" may acquire set membership. An object may only be a member of one set at a given instant.

In addition to the procedures "suc" and "pred", there are four procedures associated with each "link" object: "out", "follow", "precede" and "into".

The procedure "out" will remove the object from the set (if any) of which it is a member. The procedure call will have no effect if the object has no set membership.

The procedures "follow" and "precede" will remove the object from the set (if any) of which it is a member and insert it in a set at a given position. The set and the position are indicated by a parameter which is inner to "linkage". The procedure call will have the same effect as "out" (except for possible side effects from evaluation of the parameter) if the parameter is none or if it has no set membership and is not a set head. Otherwise the object will be inserted immediately after ("follow") or before ("precede") the "linkage" object designated by the parameter.

The procedure "into" will remove the object from the set (if any) of which it is a member and insert it as the last member of the set designated by the parameter. The procedure call will have the same effect as "out" if the parameter has the value none (except for possible side effects from evaluation of the actual parameter).

11.4 CLASS "HEAD"

```
linkage class head;
begin ref (link) procedure first; first:- suc;
      ref (link) procedure last; last:- pred;

      Boolean procedure empty;
            empty:= SUC == this linkage;

      integer procedure cardinal;
      begin integer I;
            ref (link) X;
            X:- first;
            while X =/= none do
            begin I:= I+1;
                  X:- X.suc
            end;
            cardinal:= I
      end cardinal;

      procedure clear;
            while first =/= none do first.out;

      SUC:- PRED:- this linkage
end head;
```

An object of the class "head", or a subclass of "head" is used to represent a set. "Head" objects may not acquire set membership. Thus, a unique "head" is defined for each set.

The procedure "first" may be used to obtain a reference to the first member of the set, while the procedure "last" may be used to obtain a reference to the last member.

The Boolean procedure "empty" will give the value true only if the set has no members.

The integer procedure "cardinal" may be used to count the number of members in a set.

The procedure "clear" may be used to remove all members from the set.

The references "SUC" and "PRED" will initially point to the "head" itself, which thereby represents an empty set.

CHAPTER 12

THE CLASS SIMULATION

12 THE CLASS SIMULATION

The system class "simulation" may be considered an "application package" oriented towards simulation problems. It has the class "simset" as prefix, and set-handling facilities are thus immediately available.

In the following sections the concepts defined in "simulation" are explained with respect to a prefixed block, whose prefix part is an instance of the body of "simulation" or of a subclass. The prefixed block instance will act as the head of a quasi-parallel system which may represent a "discrete-event" simulation model.

12.1 GENERAL STRUCTURE

```
simset class simulation;
begin link class EVENT_NOTICE (EVTIME, PROC);
          long real EVTIME; ref (process) PROC;
      begin ref (EVENT_NOTICE) procedure suc;
                suc:- if SUC is EVENT_NOTICE then SUC
                                                 else none;

          ref (EVENT_NOTICE) procedure pred;
                pred:- PRED;

          procedure RANK_IN_SQS (AFORE); Boolean AFORE;
          begin ref (EVENT_NOTICE) P;
                P:- SQS.last;
                while P.EVTIME > EVTIME do
                      P:- P.pred;
                if AFORE then
                while P.EVTIME = EVTIME do
                      P:- P.pred;
                follow(P)
          end RANK_IN_SQS;
      end EVENT_NOTICE;

      link class process;
      begin ref (EVENT_NOTICE) EVENT;
            ......
      end process;

      ref (head) SQS;

      ref (EVENT_NOTICE) procedure firstev;
            FIRSTEV:- SQS.first;
      ref (process) procedure current;
            current:- FIRSTEV.PROC;
      long real procedure time;
            time:= FIRSTEV.EVTIME;
      procedure hold ..............;
```

```
      procedure passivate .........;
      procedure wait ..............;
      procedure cancel ............;
      procedure activat ...........;
      procedure accum .............;
      process class MAIN_PROGRAM .......;
      ref (MAIN_PROGRAM) main;
      SQS:- new head;
      main:- new MAIN_PROGRAM;
      main.EVENT:- new EVENT_NOTICE(0,main);
      main.EVENT.into(SQS)
end simulation;
```

When used as a prefix to a block or a class, "simulation" introduces simulation-oriented features through the class "process" and associated procedures.

The variable "SQS" refers to a "set" which is called the "sequencing set", and serves to represent the system time axis. The members of the sequencing set are event notices ranked according to increasing values of the attribute "EVTIME". An event notice refers through its attribute "PROC" to a "process" object, and represents an event which is the next active phase of that object, scheduled to take place at system time EVTIME. There may be at most one event notice referencing any given process object.

The event notice at the "lower" end of the sequencing set refers to the currently active process object. The object can be referenced through the procedure "current". The value of EVTIME for this event notice is identified as the current value of system time. It may be accessed through the procedure "time".

12.2 CLASS "PROCESS"

```
link class process;
begin ref (EVENT_NOTICE) EVENT;
      Boolean TERMINATED;
      Boolean procedure idle; idle:= EVENT == none;

      Boolean procedure terminated;
            terminated:= TERMINATED;

      long real procedure evtime;
            if idle then
                  error("..." ! No Evtime for idle process)
            else evtime:= EVENT.EVTIME;

      ref (process) procedure nextev;
            nextev:- if idle then none else
                     if EVENT.suc == none then none
                     else EVENT.suc.PROC;
```

```
        detach;
        inner;
        TERMINATED:= true;
        passivate;
        error("..." ! Terminated process;)
end process;
```

An object of a class prefixed by "process" will be called a process object. A process object has the properties of "link" and, in addition, the capability to be represented in the sequencing set and to be manipulated by certain sequencing statements which may modify its "process state". The possible process states are: active, suspended, passive and terminated.

When a process object is generated it immediately becomes detached, its reactivation point positioned in front of the first statement of its user-defined operation rule. The process object remains detached throughout its dynamic scope.

The procedure "idle" has the value true if the process object is not currently represented in the sequencing set. It is said to be in the passive or terminated state depending on the value of the procedure "terminated". An idle process object is passive if its reactivation point is at a user defined prefix level. If and when the PSC passes through the final end of the user-defined part of the body, it proceeds to the final operations at the prefix level of the class "process", and the value of the procedure "terminated" becomes true. Although the process state "terminated" is not strictly equivalent to the corresponding basic concept defined above (cf. pages 83 and 139), an implementation may treat a terminated process object as terminated in the strict sense. A process object currently represented in the sequencing set is said to be "suspended", except if it is represented by the event notice at the lower end of the sequencing set. In the latter case it is active. A suspended process is scheduled to become active at the system time indicated by the attribute EVTIME of its event notice. This time value may be accessed through the procedure "evtime". The procedure "nextev" will reference the process object, if any, represented by the next event notice in the sequencing set.

12.3 ACTIVATION STATEMENT

```
activation-statement
    = activation-clause  scheduling-clause

activation-clause
    = activator  object-expression

activator
    = "activate"
    ! "reactivate"
```

```
scheduling-clause
    = empty
    ! timing-clause
    ! ( "before" ! "after" ) object-expression

timing-clause
    = simple-timing-clause [ "prior" ]

simple-timing-clause
    = ( "at" ! "delay" ) arithmetic-expression
```

An activation statement is only valid within an object of a class included in "simulation", or within a prefixed block whose prefix part is such an object.

The effect of an activation statement is defined as being that of a call on the sequencing procedure "activat" local to "simulation", i.e.

```
procedure activat (REAC, X, CODE, T, Y, PRIO);
value CODE; ref (process) X, Y; Boolean REAC, PRIO;
text CODE; long real T;
```

The actual parameter list is determined from the form of the activation statement, by the following rules:

1) The actual parameter corresponding to "REAC" is true if the activator is reactivate, false otherwise.

2) The actual parameter corresponding to "X" is the object expression of the activation clause.

3) The actual parameter corresponding to "T" is the arithmetic expression of the simple timing clause if present, otherwise it is zero.

4) The actual parameter corresponding to "PRIO" is true if prior is in the timing clause, false if it is not used or if there is no timing clause.

5) The actual parameter corresponding to "Y" is the object expression of the scheduling clause if present, otherwise it is none.

6) The actual parameter corresponding to "CODE" is defined from the scheduling clause as follows:

scheduling clause	actual text parameter
empty	"direct"
at arithmetic expression	"at"
delay arithmetic expression	"delay"
before object expression	"before"
after object expression	"after"

12.4 SEQUENCING PROCEDURES

```
procedure hold(T); long real T;
inspect FIRSTEV do
begin if T > 0 then EVTIME:= EVTIME + T;
      if suc =/= none and then suc.EVTIME <= EVTIME then
      begin out; RANK_IN_SQS(false);
            resume(current)
      end
end hold;

procedure passivate;
begin inspect current do
      begin EVENT.out; EVENT:- none
      end;
      if SQS.empty then error("..." !SQS empty;)
      else resume(current)
end passivate;

procedure wait(S); ref (head) S;
begin current.into(S);
      passivate
end wait;

procedure cancel(X); ref (process) X;
if X == current then passivate else
inspect X do if EVENT =/= none then
begin EVENT.out;
      EVENT:- none
end cancel;

procedure activat(REAC, X, CODE, T, Y, PRIO);
value CODE; ref (process) X, Y; Boolean REAC, PRIO;
text CODE; long real T;
inspect X do if not TERMINATED then
begin ref (process) z;
      ref (EVENT_NOTICE) EV;
      if REAC then EV:- EVENT
      else if EVENT =/= none then goto exit;
      z:- current;
      if CODE = "direct" then
direct:
      begin EVENT:- new EVENT_NOTICE(time,X);
            EVENT.precede(FIRSTEV)
      end direct
      else if CODE = "delay" then
      begin T:= T + time;
            goto at_
      end delay
      else if CODE = "at" then
at_: begin if T < time then T:= time;
```

```
            if T = time and PRIO then goto direct;
            EVENT:- new EVENT_NOTICE(T, X);
            EVENT.RANK_IN_SQS(PRIO)
      end at

      else if Y == none or else Y.EVENT == none
           then EVENT:- none else
      begin if X == Y then goto exit;
            comment reactivate X before/after X;
            EVENT:- new EVENT_NOTICE(Y.EVENT.EVTIME, X);
            if CODE = "before" then EVENT.precede(Y.EVENT)
                               else EVENT.follow(Y.EVENT)
      end before or after;
      if EV =/= none then
      begin EV.out;
            if SQS.empty then error("..." !SQS empty;)
      end;
      if z =/= current then resume(current);
exit:
end activat;
```

The sequencing procedures serve to organize the quasi-parallel operation of process objects in a simulation model. Explicit use of the basic sequencing facilities (call, detach, resume) should be made only after thorough consideration of their effects.

The statement "hold(T)", where T is a real number greater than or equal to zero, will halt the active phase of the currently active process object, and schedule its next active phase at the system time "time + T". The statement thus represents an inactive period of duration T. During the inactive period the reactivation point is positioned within the "hold" statement. The process object becomes suspended.

The statement "passivate" will stop the active phase of the currently active process object and delete its event notice. The process object becomes passive. Its next active phase must be scheduled from outside the process object. The statement thus represents an inactive period of indefinite duration. The reactivation point of the process object is positioned within the "passivate" statement.

The procedure "wait" will include the currently active process object in a referenced set, and then call the procedure "passivate".

The statement "cancel(X)", where X is a reference to a process object, will delete the corresponding event notice, if any. If the process object is currently active or suspended, it becomes passive. Otherwise the statement has no effect. The statement "cancel(current)" is equivalent to "passivate".

The procedure "activat" represents an activation statement, as described in 12.3. The effects of a call on the procedure are described in terms of the corresponding activation statement. The purpose of an activation statement is to schedule an active phase of a process object.

Let X be the value of the object expression of the activation clause. If the activator is activate the statement will have no effect (beyond that of evaluating its constituent expressions) unless X is a passive process object. If the activator is reactivate and X is a suspended or active process object, the corresponding event notice is deleted (after the subsequent scheduling operation) and, in the latter case, the current active phase is terminated. The statement otherwise operates as an activate statement.

The scheduling takes place by generating an event notice for X and inserting it in the sequencing set. The type of scheduling is determined by the scheduling clause.

An empty scheduling clause indicates direct activation, whereby an active phase of X is initiated immediately. The event notice is inserted in front of the one currently at the lower end of the sequencing set and X becomes active.The system time remains unchanged. The formerly active process object becomes suspended.

A timing clause may be used to specify the system time of the scheduled active phase. The clause "delay T", where T is an arithmetic expression, is equivalent to "at time + T". The event notice is inserted into the sequencing set using the specified system time as ranking criterion. It is normally inserted after any event notice with the same system time; the symbol "prior" may, however, be used to specify insertion in front of any event notice with the same system time.

Let Y be a reference to an active or suspended process object. Then the clause "before Y" or "after Y" may be used to insert the event notice in a position defined relation to (before or after) the event notice of Y. The generated event notice is given the same system time as that of Y. If Y is not an active or suspended process object, no scheduling will take place.

Examples:

The statements

```
activate X
activate X before current
activate X delay 0 prior
activate X at time prior
```

are equivalent. They all specify direct activation.

The statement

```
reactivate current delay T
```

is equivalent to "hold(T)".

12.5 THE MAIN (SIMULATION) PROGRAM

```
process class MAIN_PROGRAM;
begin
      while true do detach
end MAIN PROGRAM;
```

It is desirable that the main component of a simulation model, i.e. the "simulation" block instance, should respond to the sequencing procedures of page 179 as if it were itself a process object. This is accomplished by having a process object of the class "MAIN_PROGRAM" as a permanent component of the quasi-parallel system.

The process object will represent the main component with respect to the sequencing procedures. Whenever it becomes operative, the PSC (and OSC - the outer sequence control) will immediately enter the main component as a result of the "detach" statement (cf. page 146). The procedure "current" will reference this process object whenever the main component is active.

A simulation model is initialized by generating the MAIN_PROGRAM object and scheduling an active phase for it at system time zero. Then the PSC proceeds to the first user-defined statement of the "simulation" block.

12.6 THE PROCEDURE "ACCUM"

```
procedure accum (a,b,c,d); name a,b,c; long real a,b,c,d;
begin
    a:= a+c * (time-b);  b:= time;  c:= c + d
end accum;
```

A statement of the form "accum (A,B,C,D)" may be used to accumulate the "system time integral" of the variable C, interpreted as a step function of system time. The integral is accumulated in the variable A. The variable B contains the system time at which the variables were last updated. The value of D is the current increment of the step function.

CHAPTER 13

COMPILING A SIMULA PROGRAM

13 COMPILING A SIMULA PROGRAM

The SIMULA compiler is normally activated by:

```
@ND-500  (ND-SIMULA-AR00)SIMULA
```

The compiler will then be loaded, and after initialization it will respond with the ND-500 SIMULA prompt:

```
SIM:
```

At this point the compiler is in monitor mode, and it will then accept one of the commands listed below:

CC	RECOMPILE
COMPILE	S-COMPILE
EXIT	SAVE
HELP	SET
LIBRARY	STATUS
LISTING	

These commands are described in separate sections later in this chapter.

13.1 THE HELP FUNCTION

Whenever the ND-500 SIMULA system is in monitor mode the help function may be activated, simply by typing HELP in answer to the prompt, i.e.:

```
SIM: HELP
```

Currently this results in a list of the available commands being displayed.

13.2 COMPILATION OF SOURCE MODULES

Format: SIM: COMPILE <source-file> <listing-file> <object-file>
SIM: RECOMPILE <source-file> <listing-file> <object-file>

These commands are used to initiate compilation of a SIMULA source module. The COMPILE command is used to compile a source module (i.e. a main program, a procedure or a class declaration) from scratch. The RECOMPILE command is used to perform a compatible recompilation of a previously compiled procedure or class.

The parameters of the COMPILE and RECOMPILE commands are:

<SOURCE-FILE> Specification of the main source file.

If this parameter is not given, then the input will be taken from the default source file (SYSIN or as redefined by a SET SOURCE command). The default file type for the source file is :SYMB.

(The input may be combined from different source files by means of the %COPY directive; see page 11.)

If source input is taken from SYSIN, then any occurrence of the character @ in column 1 is interpreted as an End-of-File, and the rest of the line is ignored.

<LISTING-FILE> Specification of the output listing file.

If this parameter is non-empty, it implies a listing command with listing level 2, i.e. a direct listing of the source text with line numbers on the specified file.

<OBJECT-FILE> Specification of the output object file.

If this parameter is not given, then the output will be written to the default object file (file-name: <module-ident>:NRF, or as redefined by a SET NRFCODE command). The default file type for the object file is :NRF.

The <module-ident> is the identifier of the separately compiled procedure or class, or SIMULA-PROGRAM in the case of compiling a main program.

Compatible recompilation is treated on page 133 et seq.

13.2.1 PHYSICAL LIMITATIONS

Storage Space

When running the SIMULA Compiler, it will need space for storing information about the program. The need for space will depend on the program compiled, and mainly on its structure of blocks and declarations.

If you get the message "Storage request cannot be met", this implies that you have to specify a larger work space. Space requirements are specified through the SET STRG command.

Other Physical Limitations

The maximum depth of nested block levels that is accepted is 50.

The maximum number of BEGIN-END pairs is 900.

The number of indices to arrays cannot exceed 10.

The number of parameters to classes (accumulated through the prefix chain) and to procedures is limited to 65535.

The number of characters in a text object cannot exceed 65535.

The total number of elements in an array object cannot exceed 65535.

The sum of the number of different constants and the number of different identifiers used in the program has a limit of about 2700. Here visible identifiers in external classes or procedures should also be counted.

The limits for legal constants of the types integer, real and long real are as follows:

integer	32 bits
real	32 bits
long real	64 bits

13.2.2 DIAGNOSTICS FROM THE COMPILER

The source text input to the compiler consists of compiler directive lines and program lines. The compiler directive lines are treated in a special way and special messages will be given in case of errors in the directive. The sequence of program lines constitutes the program text and this is checked for syntactic and static semantic errors.

The compiler will recover from a syntactic error in order to check as much as possible of the remaining program text. Recovery entails skipping part of the program text after the point of detection of the error. The skipped text will thus not be checked for syntactic/semantic errors.

In case of a semantic error the compiler will recover so that no (or at least a minimum of) errors will be caused by the error and so that an optimal checking of the remaining program may be performed.

When syntactic or semantic errors are detected no code will be produced.

Messages concerning syntactic errors will be output during FEC pass 1, while semantic errors are reported from FEC pass 2.

Syntactic error messages will normally appear in the listing of the program. They will appear under the line where they were detected, with an '*' in the position where the error seems to start (This can sometimes be misleading!). A short message explaining the error will occur under this indicator. More than one error may be noted under each line.

If listing is given on another file than the messages, or if no listing is given at all, then those lines containing syntactic errors will be listed above the error messages concerning this line. These lines will be preceeded by line numbers.

The semantic error messages will be given after the listing of the program is finished. Each will start with a line number, but they will not be fully sorted. Errors concerning the consistency of declarations (e.g. double declarations or illegal qualifications), will in some cases occur out of order. The number appearing in parentheses after an identifier gives the line number where the identifier is declared, or if it is not declared, it gives the line number where this identifier was seen the first time.

Some of the messages will be warnings and are marked as such. A warning does not affect the generation of code. They may be suppressed by suitable setting of parameters.

In some cases the compiler detects situations which, although in accordance with the language definition, never the less are sufficiently unusual to be worth bringing to the programmer's attention. Examples are Boolean expressions which always evaluate to a constant value. In this case the compiler will issue a message called a note. It will not affect generation of code. Notes are suppressed together with the suppression of warnings.

13.2.3 DEVIATIONS FROM THE SIMULA STANDARD

According to the standard there must be space between colon and a minus sign in array bounds. In this implementation a space is not needed.

According to the standard, the procedures "putfix", "putreal", "outfix" and "outreal" must have long real formal parameters. In this implementation these are, however, overloaded, so that there exist a real and a long real version of each. The procedures "getreal" and "inreal" are always long real procedures.

The following constitutes a complete list of implementation omissions:

1) SWITCHes cannot be transmitted as parameter.

2) SWITCH elements have to be simple labels that are directly visible.

3) Attribute protection is not implemented (but is syntactically checked).

4) FOR-statements with value assignment, where the controlled variable is of type text, and FOR-statements with controlled variable of type character are not properly implemented.

5) Labels and switches 'seen' through inspection are currently illegal, and not invisible as the standard prescribes.

6) It is not legal to make subclasses of the predefined file classes.

7) The last actual parameter to the random drawing procedures (corresponding to an integer name formal parameter) should always be a simple integer variable.

8) Separately compiled modules are always compiled to block level 1 (i.e. local to the program block).

9) External non-SIMULA procedures have to appear with a binding and only non-SIMULA external procedures may have a binding. External non-SIMULA procedures are not allowed to occur in an external head (see page 136).

13.3 LIBRARY SPECIFICATION COMMAND

Format: SIM: LIBRARY <Dir:User-id>

The LIBRARY command is used to define additional library directories. This is particulary useful when a SIMULA module contains references to separately compiled procedures and classes, and the results of these compilations reside on files contained under another user.

During file opening a file is searched under the current user, or under user SYSTEM, in that order. If the file is not found under either user, additional libraries may be searched as specified in LIBRARY commands. The order in which the search take place is the reverse of the order in which the LIBRARY commands are given.

13.4 THE LISTING COMMAND

Format: SIM: LISTING <listing-level> <listing-file>

The LISTING command is used to specify the amount of information contained in the compiler listing. In addition it may be used to direct the listing output from the compiler to a specified file. Initially listing is off.

Parameters to the LISTING command:

<LISTING-LEVEL> An integer specifying the listing format:

0 - No listing is produced
* 1 - Direct listing without line numbers is produced
2 - Direct listing with line numbers is produced
* 3 - A pretty-print listing is produced

*) currently not implemented

If this parameter is not given, 2 is assumed.

<LISTING-FILE> Specification of the output listing file.

If this parameter is not given, then the listing will be written to the default listing file (normally SYSOUT or as redefined by a SET LISTFIL command). The default file type for the listing file is :SYMB.

13.5 CREATE SIMULA INIT FILE

Format: SIM: SAVE < RT ! CT >

The SAVE command is used to save the currently defined default parameter values in one of two initialization files: SIMULA-RT:INIT or SIMULA-CT:INIT under the user directory. If RT (i.e. Run Time) is specified, the file SIMULA-RT:INIT is used, otherwise the file SIMULA-CT:INIT is used.

When the SIMULA compiler is loaded, the default parameter values are read from the file (<user>)SIMULA-CT:INIT. On the other hand, whenever a user program is activated, SIMULA-RT:INIT is read. Both files are formatted suitable for editing by PED or 7-bit NOTIS.

The default values may be changed through application of the SET command (see below).

13.6 DISPLAY CURRENT PARAMETER VALUES

Format: SIM: STATUS <detail-level>

The STATUS command is used to display certain internal information. If no argument is given, or if the argument is 0, the currently defined parameter values are displayed.

13.7 CHANGE PARAMETERS AND SWITCHES

Format: SIM: SET <mnemonic> <value>

The SET command is used to alter the default value of certain system parameters or to set or reset trace and debugging switches. As the initial default values are read from the file SIMULA-CT:INIT, use the STATUS command to check these values.

Normally these parameters are set during system generation and should be of no concern to the user. It may however be necessary in exceptional cases to alter some of the capacity parameters to enable the compiler to process a particular program. But it should be noted that this will influence both the storage space used by the compiler and the compilation speed.

The changes obtained by application of the SET command are temporary, the default values of the ND-500 SIMULA system will not be changed unless the SAVE command is used.

13.7.1 USER PARAMETER CHANGE

The following arguments to the SET command are meant for the user:

SOURCE <source-file>	Alters the default source input file to the one given as the second argument. The default type for the source file is :SYMB.
LISTFIL <listing-file>	Alters the default listing file to the one given as the second argument. The default type for the listing file is :SYMB.
SCODE <S-code-file>	Alters the default intermediate S-code file to the one given as the second argument. The default type for the S-code file is :SCOD.
NRFCODE <object-file>	Alters the default object file to the one given as the second argument. The default type for the object file is :NRF.
INPLTH <decimal number>	Alters the default image length of SYSIN to the value given. For the compiler, this determines the longest source line that can be read.
OUTLTH <decimal number>	Alters the default image length of SYSOUT to the value given.
LPP <decimal number>	Alters the default value used for all PRINTFILEs of the attribute LINES_PER_PAGE.

MAXERR <value>	Sets the number of error messages that will be accepted before the compilation is terminated.
STRG <decimal number>	Sets the default size (in bytes) of the working storage used by the compiler.
SCRATCH-1 <file-name>	Alters the default name of scratch file 1. The default type for scratch file 1 is :DATA.
SCRATCH-2 <file-name>	Alters the default name of scratch file 2. The default type for scratch file 2 is :DATA.
SCRATCH-3 <file-name>	Alters the default name of scratch file 3. The default type for scratch file 3 is :DATA.

13.7.2 DEBUGGING AND TESTING PARAMETERS

> Note: The following parameters of the SET command are intended for maintenance use only. They are typically used during system generation, or during debugging of the SIMULA system, and are not recommended for general use.

ATRFIL <attribute-file-prefix>	No details given here.
FECOPT <option-string>	This is a string of options for debugging purposes etc. for the front end compiler. They should be properly set in certain initiation runs (see Installation Guide for the FEC).The general user should skip this section. Further details are not given here.
PREDEF <attribute-file>	Alters the attribute file for the predefined procedures and classes.
SIMSET <attribute-file>	Alters the attribute file for class simset.
SIMLTN <attribute-file>	Alters the attribute file for class simulation.

```
DEBUG   <value>   S-Compiler: Debugging level
                              (0: skip all debug info)
STKLNG  <value>   S-Compiler: Total runtime stack length
SYSGEN  <value>   S-Compiler: System generation
                        1: Generation of Runtime System
                        2: Generation of S-Compiler
                        3: Generation of Environment
LINTAB  <value>   >0: Produce line-number-table
SK1LIN  <value>   S-Compiler-Trace - Pass 1 starting line
```

```
SK1TRC  <value>   Pass 1 Trace value: OMTI , where
                         O = 0..9 Output trace level
                         M = 0..9 Module input/output trace level
                         T = 0..9 Trace-mode level
                         I = 0..9 Input trace level
SK2LIN  <value>   S-Compiler-Trace - Pass 2 starting line
SK2TRC  <value>   Pass 2 Trace value: OMTI (as SK1TRC)
MAXTAG  <value>   S-Compiler: Max number of tags
MAXXTG  <value>   S-Compiler: Max number of ext-tags in
                              global module
MAXMOD  <value>   S-Compiler: Max number of inserted modules
MAXFIX  <value>   S-Compiler: Max number of fixups
MAXSMB  <value>   S-Compiler: Max number of symbols in symbol-table
MAXREF  <value>   S-Compiler: Max number of undefined symbols
MAXDEF  <value>   S-Compiler: Max number of entry points
TRACE   <value>   SIMULA-Monitor: Trace level
OPTION  <option-list>         Sets debug-options.
```

Each element in the OPTION-LIST is a letter, possibly followed by a number. The following option letters are currently defined (not all of these are as of yet implemented):

```
B   FEC: Include begin/end counters in program listing
C   Environment call trace level
D   FEC: Include statements counters
E   Environment error trace level
F   Environment file trace level
G   GARB: Trace level
N   FEC: Do not print warning messages
O   FEC: Include information for SIMOB
P   FEC: Include processor usage measurements
Q   S-Compiler: Make main program a routine
R   FEC: Reduce runtime checking (array and none)
T   Runtime-Trace level: Passed on to the RTS
                         as a value 0..10
V   Environment mode (0:user,1:FEC,2:FEC,3:S-Compiler)
W   Runtime Work space (initially): 2**W addressing units
    will be allocated at definition time
X   FEC: Produce cross-reference listing
Y   FEC: Start/stop information level (1,2,3)
Z   Let FEC/S-Compiler handle parameters on its own
```

13.8 SEPARATE ACTIVATION OF THE S-COMPILER

Format: SIM: S-COMPILE <S-code-file> <object-file>

This command is not meant for the ordinary user. It is described here for completeness only. When used, the S-COMPILE command will initiate S-Compilation of a previously saved S-code file.

Parameters to the S-COMPILE commands:

<S-code-file> Specification of the input S-code file. If this parameter is not given, then the input will be taken from the default S-code file (set by a SET SCODE command). The default file type for S-code files is :SCOD.

<object-file> Specification of the output object file. If this parameter is not given, then the output will be written to the default object file (SIMULA-PROGRAM:NRF or as redefined by a SET NRFCODE command). The default file type for the object file is :NRF.

13.9 SPECIAL MAINTENANCE DIRECTIVES

The directives listed in this section are intended for maintenance purposes and are not recommended for general use. Consequently only a very short explanation is given for each directive; further information may be found in the system documentation. Note that the resulting output may be very voluminous!

13.9.1 S-COMPILER DIRECTIVES

The following directives will cause the S-Compiler to set certain internal switches. Only nonnegative integers less than 256 are accepted as values for 'val'.

```
%SETSWITCH 1 val  -- S-Compiler Input Listing
%SETSWITCH 2 val  -- S-Compiler Trace-mode
%SETSWITCH 3 val  -- S-Compiler Module-Trace-mode
%SETSWITCH 4 val  -- S-Compiler Output Listing
```

13.9.2 FEC DIRECTIVES

%SETOPT <test-option change indicator>

This directive will change the set of options used for test output from the front-end compiler. The test-option change indicator is a string of characters. If the first of these characters is '+' then the options corresponding to the characters in the rest of the string are switched on. If the first character is '-' then the corresponding options are switched off. If the string starts with any other character, then all options corresponding to characters in the string will be switched on, and all other options are switched off.

%COMPCALL <integer index> <any string>

This directive is handled as if it was unknown to the front end compiler in the sense that it is sent out as INFO-instruction in the S-code. However, it has the extra effect that at the place it is encountered, each of the passes of the front end compiler will call the procedure "give textinfo" with the two parameters of the directive.

CHAPTER 14

LINK-LOADING OF SIMULA PROGRAMS

14 LINK-LOADING OF SIMULA PROGRAMS

14.1 SINGLE SEGMENT LOAD

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <domain-ident>
DELETE-DOMAIN <domain-ident>
SET-DOMAIN "<domain-ident>"
OPEN-SEGMENT "<domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AR00)RTS-AP02
LOAD-SEGMENT <program-ident-1>
.....
LOAD-SEGMENT <program-ident-n>
CLOSE-SEGMENT
EXIT
```

14.2 SEVERAL SEGMENT LOAD

14.2.1 PREPARING A LIBRARY SEGMENT

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <library-domain-ident>
DELETE-DOMAIN <library-domain-ident>
SET-DOMAIN "<library-domain-ident>"
SET-SEGMENT-NUMBER <segment number, not 1 or 2>
OPEN-SEGMENT "<library-domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AR00)RTS-AP02
LOAD-SEGMENT <program-ident-1>
.....
LOAD-SEGMENT <program-ident-n>
CLOSE-SEGMENT
EXIT
```

14.2.2 USING A PREPARED LIBRARY SEGMENT

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <domain-ident>
DELETE-DOMAIN <domain-ident>
SET-DOMAIN "<domain-ident>"
OPEN-SEGMENT "<domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AROO)RTS-APO2
FORCE-SEGMENT-LINK <library-domain-ident>
LOAD-SEGMENT <program-ident-1>
.....
LOAD-SEGMENT <program-ident-n>
CLOSE-SEGMENT
EXIT
```

14 LINK-LOADING OF SIMULA PROGRAMS

14.1 SINGLE SEGMENT LOAD

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <domain-ident>
DELETE-DOMAIN <domain-ident>
SET-DOMAIN "<domain-ident>"
OPEN-SEGMENT "<domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AR00)RTS-AP02
LOAD-SEGMENT <program-ident-1>
.....
LOAD-SEGMENT <program-ident-n>
CLOSE-SEGMENT
EXIT
```

14.2 SEVERAL SEGMENT LOAD

14.2.1 PREPARING A LIBRARY SEGMENT

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <library-domain-ident>
DELETE-DOMAIN <library-domain-ident>
SET-DOMAIN "<library-domain-ident>"
SET-SEGMENT-NUMBER <segment number, not 1 or 2>
OPEN-SEGMENT "<library-domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AR00)RTS-AP02
LOAD-SEGMENT <program-ident-1>
.....
LOAD-SEGMENT <program-ident-n>
CLOSE-SEGMENT
EXIT
```

14.2.2 USING A PREPARED LIBRARY SEGMENT

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <domain-ident>
DELETE-DOMAIN <domain-ident>
SET-DOMAIN "<domain-ident>"
OPEN-SEGMENT "<domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AROO)RTS-APO2
FORCE-SEGMENT-LINK <library-domain-ident>
LOAD-SEGMENT <program-ident-1>
.....
LOAD-SEGMENT <program-ident-n>
CLOSE-SEGMENT
EXIT
```

CHAPTER 15

RUNNING A SIMULA PROGRAM

15 RUNNING A SIMULA PROGRAM

15.1 ACTIVATING THE PROGRAM

A SIMULA program is activated by entering

```
@ND-500  (dir:user)program-name [ Wxx ]
```

in the SINTRAN environment, where "program-name" is the implementation-specific SIMULA program-name, and where the optional parameter "Wxx" specifies, if given, the size of the working swap area. W must be given as shown, xx is a two-digit decimal number specifying the swap size in 2**xx bytes, e.g. W10 means 1024 bytes.

This option is useful whenever the swap space should be changed for one specific program. If the default area size generally is too small for your program, the compiler command SETSTRG should be used followed by the command SAVE RT.

15.2 SIMPLE PROGRAM COMPILE-LOAD-AND-GO IN BATCH

```
@ND-500  (ND-SIMULA-AR00)SIMULA
COMPILE
begin
       .
       .  SIMULA Source Text
       .
end;
@EOF
EXIT
@PERFORM (ND-SIMULA-AR00)LOAD LOAD
SIMULA-PROGRAM
@ND-500  SIMULA-PROGRAM
       .
       . Data Images to Sysin
       .
@EOF
```

15.3 PRECOMPILED PROGRAMS WITH LOAD-AND-GO

Step 1. Precompiling a class and a procedure:

```
@ND-500  (ND-SIMULA-AR00)SIMULA
COMPILE
    class c;
    begin .......
      .
      .  SIMULA Source Text
      .
    end;
@EOF
COMPILE
    procedure P;
    begin .......
      .
      .  SIMULA Source Text
      .
    end;
@EOF
EXIT
```

Step 2. Compiling the main program:

```
@ND-500  (ND-SIMULA-AR00)SIMULA
COMPILE
    begin
          external class c;
          external procedure P;
        c begin
            .
            .  SIMULA Source Text
            .
          end;
    end;
@EOF
EXIT
```

Step 3. Recompiling the class:

```
@ND-500  (ND-SIMULA-AR00)SIMULA
RECOMPILE
    class c;
    begin .......
      .
      .  Modified SIMULA Source Text
      .
    end;
@EOF
EXIT
```

Step 4. Performing a load-and-go:

```
@ND LINKAGE-LOADER
RELEASE-DOMAIN <domain-ident>
DELETE-DOMAIN <domain-ident>
SET-DOMAIN "<domain-ident>"
OPEN-SEGMENT "<domain-ident>" CWP
SUPPRESS-DEBUG-INFO OFF
LINK-SEGMENT (ND-SIMULA-AR00)RTS-AP02
LOAD-SEGMENT C
LOAD-SEGMENT SIMULA-PROGRAM
CLOSE-SEGMENT
EXIT

@ND-500  <domain-ident>
       .
       . Data Lines to Sysin
       .
@EOF
```