SIMULA INFORMATION

NCR

743

COMMON BASE LANGUAGE

by

Ole-Johan Dahl, Bjørn Myhrhaug

and

Kristen Nygaard

NORWEGIAN COMPUTING CENTER
NORSK REGNESENTRAL
NR

B—No.	Res. inst. name	NTNF-group	Availability
	NR		OPEN

Title

SIMULA 67
Common Base Language

Reprint 5 Edition

Report no.	ISBN
743	82-539-0225-5

Authors	No. of pages
Ole-Johan Dahl, Bjørn Myhrhaug & Kristen Nygaard	181
	Date
	10. Feb.1984

Project client/Sponsor

Abstract

This document contains the definition of the programming language SIMULA 67, excluding the Algol language. The language definition is supervized by the SIMULA Standards Group.

The document is a revised version of the SIMULA 67 Common Base Language of 1982, and includes the modifications and clarifications passed by the SIMULA Standards Group at their meeting in September 1983.

KEY WORDS	KEY WORDS IN NORWEGIAN	CR No.
SIMULA 67	SIMULA 67	
Programming languages	Programmeringsspråk	
Coroutines	Korutiner	
Simulation language	Simuleringsspråk	

SUBJECT GR.

February 1984

A complete definition encomprising the Modified
Report on the Alghorithmic Language ALGOL 60 and
the Common Base Language authorised
as the SIMULA 67 programming language definition,
by the SIMULA Standards Group in October 1970 and
subsequently modified by recommendations made
by the SSG at the annual meetings
Jan 1973, Sept 1973, Oct 1974, Sept 1976, Sept 1977,
Sept 1978 Sept 1979 ,Aug 1980, Sept 1981,
Sept 1982 and Sept 1983.

COMMON BASE LANGUAGE

by
Ole-Johan Dahl, Bjørn Myhrhaug
and
Kristen Nygaard

PREFACE TO THE 1968 COMMON BASE EDITION

SIMULA 67 is a general purpose programming language developed by the authors at the Norwegian Computing Center. Compilers for this language are now implemented on a number of different computers.

The Norwegian Computing Center regards the SIMULA 67 language as its own property. The implementations have taken place under contracts with the NCC for professional assistance.

A main characteristic of SIMULA is that it is easily structured towards specialised problem areas, and hence can be used as a basis for Special Application Languages.

This report is a reference document for the SIMULA 67 Common Base. The Common Base comprises the language features required in every SIMULA 67 compiler. The "Introduction" highlights some features of the language. The following sections are intended as a precise language definition. Users manuals and textbooks will appear later.

During our development of SIMULA 67 we have benefited from ideas and suggestions from a number of colleagues. First of all we should like to mention C.A.R. Hoare whose ideas on referencing have been used and extended, S. Kubosch who has been an important source of useful comments and criticism and Mrs. I. Siguenza whose help in the typing of this report has been indispensable.

We should also like to express our gratitude to D. Belsnes, P. Blunden, J. Buxton, J.V. Garwick, Ø. Hjartøy, Ø. Hope, P.M. Kjeldaas, D. Knuth, J. Laski, A. Lunde, J. Newey, T. Noodt, K.S. Skog, C. Strachey and N. Wirth, as well as SIMULA I users who have given advice based upon their experience. Finally, the authors feel that they benefited very much from the SIMULA 67 Common Base Conference in Oslo, June 1967, and would like to thank the participants.

Oslo, May 1968

Ole-Johan Dahl Bjørn Myhrhaug Kristen Nygaard

PREFACE TO THE 1970 COMMON BASE EDITION

This revised version contains the modifications and clarifications passed by the SIMULA Standards Group at their meeting in May 1970. Several minor errors have also been corrected.

Compilers are now available on a wide range of computers, including CD 3300, CD 3600, CD 6600, UNIVAC 1108 and IBM 360/370.

The authors would like to thank the members of the SIMULA Standards Group for their interest and help, and the typing pool and printing shop of the Norwegian Computing Center for their efficient work.

We would also like to extend our list of acknowledgements in the original preface by K. Babcicky, G.M. Birtwistle, R. Kerr and M. Woodger.

Oslo, October 1970

Ole-Johan Dahl Bjørn Myhrhaug Kristen Nygaard

PREFACE TO THE 1982 COMMON BASE EDITION

This revised version contains the modifications and clarifications passed by the SIMULA Standards Group at their meetings January 1973, September 1973, October 1974, September 1976, September 1977, September 1978, September 1979, August 1980, September 1981 and September 1982. Several minor errors have been corrected.

Compilers are now available on a wide range of computers, including CD 3300, CD 3600, CD 6600, CYBER, UNIVAC 1100, IBM 360/370, DEC-10, DEC-20, Siemens 7000, ICL 2900, Data General Eclipse, Nord 100, BESM 6, VAX 11, PRIME and CII HB DPS8.

NCC has developed, in cooperation with the Program Library Unit and the Regional Computing Center at the University of Edinburgh, a SIMULA implementation package that significantly will reduce the effort to bring up a new SIMULA system. The first adaptions for this system have been for VAX 11, CII HB DPS8, PRIME and Nord 500.

Oslo, December 1982

PREFACE TO THE 1984 EDITION

This revised version contains the modifications and clarifications passed by the SIMULA Standards Group at its meeting September 1983.

Oslo, February 1984

Contents:

1 Introduction

1.1 General purpose programming languages

High level languages, like FORTRAN, ALGOL 60 and COBOL were originally regarded as useful for two purposes:

- to provide concepts and statements allowing a precise formal description of computing processes and also making communication between programmers easier.

- to provide the non-specialist with a tool making it possible for him to solve small and medium-sized problems without specialist help.

High level languages have succeeded in these respects. However, strong new support for these languages is developing from a fresh group: those who are confronted with the task of organizing and implementing very complex, highly interactive programs, e.g. large simulation programs.

These tasks put new requirements on a language:

- in order to decompose the problem into natural, easily conceived components, each part should be describable as an individual program. The language should provide for this and also contain means for describing the joint interactive execution of these sub-programs.

- in order to relate and operate a collection of programs, the language should have the necessary powerful list processing and sequencing capabilities.

- in order to reduce the already excessive amount of debugging trouble associated with present day methods, the language should give "reference security". That is, the language and its compiler should spot and prohibit execution involving invalid use of data through data referencing based on wrong assumptions.

Even if the organizational aspects of complex programming are becoming more and more important, the computational aspects must, of course, be taken care of at least as well as in the current high-level languages.

It is also evident that such a general language should be oriented towards a very wide area of use. The market cannot for long accommodate the present proliferation of languages.

1.2 Special application languages

Until now, the computer has been a powerful but frightening tool to most people. This should be changed in the years to come, and the computer should be regarded as an obvious part of the human environment. More and more people should get their capabilities increased through the availability of the "know-how" and data they need.

A condition for this development is that the demands on the computer user are reduced, which implies that communication between man and computer is made easier.

Know-how is today to a large extent made operative through "application packages" covering various fields of knowledge and methods. But these packages are in general not sufficiently flexible and expandable, and also often require specialist assistance for their use.

The future seems to be "application languages" which are problem-oriented, perhaps in the extreme. Such languages may provide the basic concepts and methods associated with the field in question and allow the user to formulate his specific problem in accordance with his own earlier training.

At the same time, such languages should be flexible in the sense that new knowledge acquired should be easily incorporated, even by the individual user.

The need for application languages is apparently in conflict with the desire for the non-proliferation of languages and for general purpose programming languages.

A solution is to design a general purpose programming language to serve as a "substrate" for the application languages by making it easy to orient towards specialized fields, and to augment it by the introduction of additional aggregated concepts useful as "building blocks" for programming.

By making the general purpose language highly standardized and available on many types of computers, the application languages also become easily transferable, and at the same time the software development costs for the computer manufacturers may be retarded from the present rapid increase.

1.3 The basic characteristics of SIMULA 67

1.3.1 Algorithmic capability

SIMULA 67 contains most features of the general algoritmic language ALGOL 60 as a subset. The reason for choosing ALGOL 60 as a starting point was that its basic structure lent itself to extension. It was felt that it would be impractical for the users to base SIMULA 67 on yet another new algorithmic language, and ALGOL 60 already had a user basis, mainly in Europe.

1.3.2 Decomposition

In dealing with problems and systems containing a large number of details, decomposition is of prime importance. The human mind must concentrate; it is a requirement for precise and coherent thinking that the number of concepts involved is small. By decomposing a large problem, one can obtain component problems of manageable size to be dealt with one at a time, and each containing a limited number of details. Suitable decomposition is an absolute requirement if more than one person takes part in the analysis and programming.

The fundamental mechanism for decomposition in ALGOL 60 is the block concept. As far as local quantities are concerned, a block is completely independent of the rest of the program. The locality principle ensures that any reference to a local quantity is correctly interpreted regardless of the environment of the block.

The block concept corresponds to the intuitive notion of "sub-problem" or "sub-algorithm" which is a useful unit of decomposition in orthodox application areas.

A block is a formal description, or "pattern", of an aggregated data structure and associated algorithms and actions. When a block is executed, a dynamic "instance" of the block is generated. In a computer a block instance may take the form of a memory area containing the necessary dynamic block information and including space for holding the contents of variables local to the block. A block instance can be thought of as a textual copy of its formal description, in which local variables indentify parts of memory allocated to the block instance. Any inner block of a block instance is still a "pattern", in which occurrences of non-local identifiers, however, identify items local to textually enclosing block instances. Such "bindings" of identifiers non-local to an inner block remain valid for any subsequent dynamic instance of that inner block.

The notion of block instances leads to the possibility of generating several instances of a given block which may co-exist and interact, such as, for example, instances of a recursive procedure. This further leads to the concept of a block as a "class" of "objects", each being a dynamic instance of the block, and therefore conforming to the same pattern.

An extended block concept is introduced through a "class" declara tion and associated interaction mechanism such as "object references" (pointers), "remote accessing", "quasi-parallel" operation, and block "concatenation".

Whereas ALGOL 60 program execution consists of a sequence of dynamically nested block instances, block instances in SIMULA 67 may form arbitrary list structures. The interaction mechanisms which are introduced, serve to increase the power of the block concept as a means for decomposition and classification.

1.3.3 Classes

A central new concept in SIMULA 67 is the "object". An object is a self-contained program (program instance), having its own local data and actions defined by a "class declaration". The class declaration defines a program (data and action) pattern, and objects conforming to that pattern are said to "belong to the same class".

If no actions are specified in the class declaration, a class of pure data structures is defined.

Example:

```
class order(number); integer number;
   begin integer number_of_units, arrival_date;
         real processing_time;
   end;
```

A new object belonging to the class "order" is generated by an expression such as

"new order(103)"

and as many "orders" may be introduced as desired.

The need for manipulating objects and relating objects to each other makes it necessary to introduce list processing facilities (as described below).

A class may be used as "prefix" to another class declaration, thereby building the properties defined by the prefix into the objects defined by the new class declaration.

Example:

```
order class batch_order;
   begin integer batch_size;
         real setup_time;
   end;

order class single_order;
   begin real setup_time, finishing_time, weight; end;

single_order class plate;
   begin real length, width; end;
```

New objects belonging to the "sub-classes" - "batch order", "single order" and "plate" all have the data defined for "order", plus the additional data defined in the various class declarations. Objects belonging to the class "plate" will, for example, comprise the following pieces of information: "number", "number_of_units", "arrival_date", "processing_time", "setup_time", "finishing_time", "weight", "length" and "width".

If actions are defined in a class declaration, actions conforming to this pattern may be executed by all objects belonging to that class. The actions belonging to one object may all be executed in sequence, as for a procedure. But these actions may also be executed as a series of separate subsequences, or "active phases". Between two active phases of a given object, any number of active phases of other objects may occur.

SIMULA 67 contains basic features necessary for organizing the total program execution as a sequence of active phases belonging to objects. These basic features may be the foundation for aggregated sequencing principles, of which the class SIMULATION is an example.

1.3.4 Application language capability

SIMULA 67 may be oriented towards a special application area by defining a suitable class containing the necessary problem-oriented concepts. This class can then be used as prefix to the program by the user interested in this problem area.

The unsophisticated user may restrict himself to using the aggregated, problem-oriented and familiar concepts as constituent "building blocks" in his programming. He may not need to know the full SIMULA 67 language, whereas the experienced programmer at the same time has the general language available, and he may extend the "application language" by new concepts defined by himself.

As an example, in discrete event system simulation, the concept of "simulated system time" is commonly used. SIMULA 67 is turned into a simulation language by providing the class "SIMULATION" as a part of the language. (In this case provided with the compilers)

In the class declaration

```
class SIMULATION;
   begin ..... end;
```

a "time axis" is defined, as well as two-way lists (which may serve as queues), and also the class "process" which gives an object the property of having its active phases organized through the "time axis".

A user wanting to write a simulation program starts his program by

```
SIMULATION begin ..................
```

in order to make all the simulation capabilities available in his program. If he himself wants to generate a special-purpose simulation language to be used in job-shop analysis, he may write:

```
SIMULATION class JOBSHOP;
   begin ........................... end;
```

and between "begin" and "end" define the building blocks he needs, such as

```
PROCESS class CRANE;
   begin ........................... end;

PROCESS class MACHINE;
   begin procedure DATACOLLECTION; .....
   ......................
end;
```

etc.

The programmer now compiles this class, and whenever he or his colleagues want to use SIMULA 67 for jobshop simulation, they may write in their program

```
JOBSHOP begin ..............
```

thereby making available the concepts of both "SIMULATION" and "JOBSHOP".

This facility requires that a mechanism for the incorporation of separately compiled classes is available in the compiler (see section 15).

1.3.5 List processing capability

When many objects belonging to various classes do co-exist as parts of the same total program, it is necessary to be able to assign names to individual objects, and also to relate objects to each other, e.g. through binary trees and various other types of list structures. A system class, "SIMSET", introducing circular two-way lists is a part of the language.

Hence basic new types, "references", are introduced. References are "qualified", which implies that a given reference only may refer to objects belonging to the class mentioned in the qualification (or belonging to subclasses of the qualifying class).

Example:

```
ref (order) next, previous;
```

The operation of making a reference denote a specified object is written ":-" and read "denotes".

Example:

```
next:- new order(101); previous:- next;
```

or (also valid since "plate" is a subclass of "order")

```
next:- new plate(50);
```

Data belonging to other objects may be referred to and used by "remote accessing", utilizing a special "dot notation".

Example:

```
if next.number > previous.number then .....;
```

comparing the "number" of the "order" named "next" with the "number" of the "order" named "previous".

The "dot notation" gives access to individual pieces of information. "Group access" is achieved through "connection statements".

Example:

```
inspect next when plate do begin ..... end;
```

In the statement between begin and end all pieces of information contained in the "plate" referenced by "next" may be referred to directly.

1.3.6 String handling

SIMULA 67 contains the new basic type "character". The representation of characters is implementation defined.

In order to provide the desired flexibility in string handling, a compound type called "text" is introduced. The "text" concept is closely associated with input/output facilities.

1.3.7 Input/output

ALGOL 60 has been seriously affected by the lack of standardized input/output and string handling. Clearly a general purpose programming language should have great flexibility in these areas. Consequently, input/output are defined and made a standardized part of SIMULA 67.

1.4 Standardization

For a general purpose programming language it is of paramount importance that while the language is uniquely defined and at the same time under strict control, it may be extended in the future.

This is achieved by the SIMULA Standard Group, consisting of representatives for firms and organizations having responsibility for SIMULA 67 compilers. The statutes lay down rigid rules to provide for both standardization and future extensions.

The SIMULA definition which is required to be a part of any SIMULA 67 system is named the "SIMULA 67 Common Base Definition".

1.5 Language definition

The language definition given in the following sections must be supplemented by the formal definition of ALGOL 60 (1). The syntactic definitions given in this report are to be understood in the following way.

1) Syntactic classes referred to, but not defined in this report, refer to syntactic definitions given in (1).

2) Definitions in this report of syntactic classes defined in (1) replace the crrresponding definitions given in (1).

3) Any construction of the form

   <ALGOL some syntactic class>

   stands for the list of alternative direct productions of <some syntactic class> according to the definition given in (1).

1.6 Comment conventions

For the purpose of including explanatory texts among the symbols of a program the following comment convention holds:

The sequence:	is equivalent to:
comment <any sequence of printable characters not containing ;>;	space
end <any sequence of printable characters not containing end, else, when, otherwise or ;>	end
! <any sequence of printable characters not containing ;>;	space

Note: This comment convention is an extension of the Algol 60 convention since it permits comments wherever a space may be recognised.

Also note that since comments are not symbols of the SIMULA language they are permitted to extend over more than one line record.

1.7 Hardware representation language

A SIMULA program text must be represented according to ISO standard, Ref. No. ISO 646-1973, and consists of a sequence of line records. If a line record consists of more than 72 positions, an implementation may treat as significant only the first 72 positions. There are two kinds of line records: Compiler directives (see 1.7.1) and Program lines (see 1.7.2). Apart from 1.7.1 this document is only concerned with Program lines.

1.7.1 Compiler directive lines

Compiler directives are identified by having a % character (ISO code 37) in the first position of the line record. Such lines are always taken as compiler control lines and the interpretation of their contents is implementation dependent. If a line starts with a % character followed by a blank, this should be treated as a line of comment.

1.7.2 Program lines

Line records which do not have a % character in the first position are taken as Program lines. In the remaining part of this document, the words "program text" mean the sequence of Program lines excluding the Compiler directives.

The program text consists of a sequence of symbols, comments and spaces. Comments are treated in detail in section 1.6. The symbols are:

- Identifiers
- Keywords
- Constants
- Special symbols

A symbol must be contained in a single line record, i.e. it cannot be continued from one line to the next. The extent of a symbol is decided by a left-to-right scan of a Program line beginning at position 1 or at the first position containing a non-space character following an already recognized symbol or comment, trying to recognize the largest possible string of characters which fits the syntax of a symbol.

As a consequence of this rule there must be at least one space separating colon and minus sign in an array declaration.

1.7.3 Representation of identifiers and keywords

Identifiers and keywords must conform to the following syntax rule

```
<id or keyword>
      ::= <letter>
       !  <id or keyword> <letter>
       !  <id or keyword> <digit>
       !  <id or keyword> <underline>
```

i.e. a sequence of letters, digits and underlines starting with a letter. Note that spaces are not permitted inside identifiers or keywords.

1.7.3.1 Identifiers

A symbol is recognized as an identifier in the scan mentioned in section 1.7.1.2 if it conforms to the syntax rule above and it is not found in table 1.

Note that the length of an identifier is restricted to at most 72 characters, and all characters (including underlines) are significant.

1.7.3.2 Keywords

A symbol is recognized as a keyword if it conforms to the syntax rule above and is found in table 1. These keywords are reserved and cannot be used as identifiers. An implementation may not reserve symbols other than those mentioned in table 1.

1.7.4 Representation of special symbols

The set of special symbols which is a part of this recommendation is given in table 2. Some of the special symbols of the reference language have only a reserved word representation in the hardware representation. These are:

Hardware representation	Reference language
AND	$\wedge$
EQV	$\equiv$
IMP	$\supset$
NOT	$\neg$
OR	$\vee$

Some of the special symbols of the reference language have an alternative reserved word representation in the hardware representation. These are:

Alternative hardware representation	Hardware representation	Reference language
EQ	=	=
GE	>=	$\geq$
GT	>	>
LE	<=	$\leq$
LT	<	<
NE	<>	$\neq$

Hardware representation	Reference language	Hardware representation	Reference language
ACTIVATE	activate	LABEL	label
AFTER	after	LE	$\leq$
AND	$\wedge$	LONG	long
ARRAY	array	LT	<
AT	at	NAME	name
BEFORE	before	NE	$\neq$
BEGIN	begin	NEW	new
BOOLEAN	boolean	NONE	none
CHARACTER	character	NOT	$\neg$
CLASS	class	NOTEXT	notext
COMMENT	comment	OR	$\vee$
DELAY	delay	OTHERWISE	otherwise
DO	do	PRIOR	prior
ELSE	else	PROCEDURE	procedure
END	end	PROTECTED	protected
EQ	=	QUA	qua
EQV	$\equiv$	REACTIVATE	reactivate
EXTERNAL	external	REAL	real
FALSE	false	REF	ref
FOR	for	SHORT	short
GE	$\geq$	STEP	step
GO	go	SWITCH	switch
GOTO	goto	TEXT	text
GT	>	THEN	then
HIDDEN	hidden	THIS	this
IF	if	TO	to
IMP	$\supset$	TRUE	true
IN	in	UNTIL	until
INNER	inner	VALUE	value
INSPECT	inspect	VIRTUAL	virtual
INTEGER	integer	WHEN	when
IS	is	WHILE	while

Table 1. Reserved words and their meanings

Only the upper case versions are given - all combinations of upper and lower case letters that spell a word given here are also considered reserved words.

Reference Language	Hardware Representation	Alternative Hardw. Repr.	Name
+	+		plus
-	-		minus
*	*		times
/	/		divide (real)
÷	//		divide (integer)
↑	**		exponentiate
>	>	GT	greater than
≥	>=	GE	greater or equal
<	<	LT	less than
≤	<=	LE	less or equal
=	=	EQ	equal
≠	<>	NE	not equal
==	==		reference equal
=/=	=/=		ref. not equal
.	.		dot
,	,		comma
:	:		colon
;	;		semicolon
	&		basis of ten
(	(		left parenthesis
)	)		right par.
[	(		left bracket
]	)		right bracket
:=	:=		becomes
:-	:-		denotes
'	'		single quote
"	"		double quote
comment	COMMENT	!	comment

Table 2. Hardware representation of special symbols.

2 Class declarations

2.1 Syntax

```
<declaration >
      ::= <ALGOL declaration>
       ! <class declaration>
       ! <external declaration>

<class identifier>
      ::= <identifier>

<prefix>
      ::= <empty>
       ! <class identifier>

<virtual part>
      ::= <empty>
       ! virtual: <specification part>

<class body>
      ::= <statement>
       ! <split body>

<initial operations>
      ::= begin
       ! <block head>;
       ! <initial operations> <statement>;

<final operations>
      ::= end
       ! ; <compound tail>

<split body>
      ::= <initial operations>
          <inner part> <final operations>

<class declaration>
      ::= <prefix> <main part>

<main part>
      ::= class <class identifier>
          <formal parameter part>;
          <value part> <specification part>
          <protection part>
          <virtual part> <class body>
```

<inner part>
::= inner
! <label>: <inner part>

<protection part>
::= <empty>
! <protection part> <protection specification>

<protection specification>
::= hidden <identifier list>;
! protected <identifier list>;
! hidden protected <identifier list>;
! protected hidden <identifier list>;

2.2 Semantics

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 section 4.3.2.2.

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 section 8.2). 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. See section 2.2.3.

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 expression which refer to attributes other than the formal parameters of the class declaration (or its prefixes, see 2.2.1) are illegal.

Example:

The following class declaration expresses the notion of "n-point Gauss integration" as an aggregated concept.

```
class Gauss (n); integer n;
    begin array W, X(1:n);
        real procedure integral(F,a,b); real procedure F;
                                        real a,b;
            begin real sum, range; integer i;
                range:= (b-a)*0.5;
                for i:= step 1 until n do
                sum:= sum + F(a+range*(X(i)+1))*W(i);
                integral:= range * sum;
            end integral;
        comment compute the values of the elements of
                W and X as functions of n;
        ........
    end Gauss;
```

The optimum weights W and abscissae X can be computed as functions of n. By making the algorithm part of the class body, the evaluation and assignment of these values can be performed at the time of object generation. Several "Gauss" objects with different values of n may co-exist. Each object has a local procedure "integral" for the evaluation of the corresponding n-point formula. See also examples of section 6.1.2.2 and section 7.1.2.

2.2.1 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 if D is a subclass of C, and "outer" to that of C if C is a subclass of D. The figure 2.1 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 .....;
```

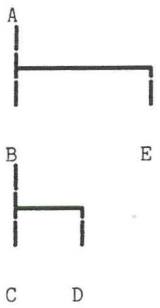
A capital letter denotes a class. The corresponding lower case letter represents the attributes of the main part of an object belonging to that class. In an implementation of the language, the object structures shown in Fig. 2.2 indicate the allocation in memory of the values of those attributes which 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 can be used as 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 (See section 11, 14 and 15).

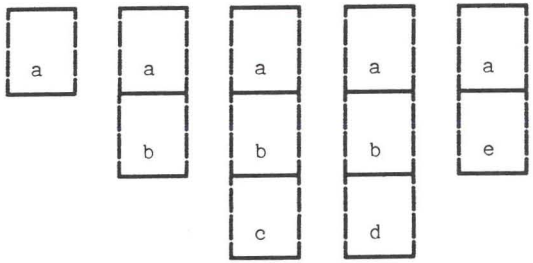
An implementation may restrict the language by demanding that a class should be defined textually before all its subclasses.



includes: A,B,C,D,E
outer to: B,C,D,E

includes: B,C,D
outer to: C,D
inner to: A

Fig. 2.1



Objects of classes A, B, C, D and E respectively.

Fig. 2.2

2.2.2 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 or this.)

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 (See section 2.2.3).

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 has a split body the statements in operation rule of X are ordered according to ascending prefix levels.

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 section 7.1), 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. (Notice 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 an 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 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 (see section 2.2.3), the one copied from the prefix class body is deleted.

   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:= arctg(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. ("arctg" is a suitable non-local procedure.)

2.2.3 Virtual quantities

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 permit attribute redeclarations 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 (see Section 3.2.7) 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

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

- if there is a match at prefix level outer or equal to PL, then that of the match at the innermost 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:= i - (i//n*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; Boolean entered;
                    i:= 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:= true
                          else begin i:= i + 1;
                               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:= i - (i//n*n);
         end hash;
   end ALGOL_hash;
```

2.3 Attribute protection

2.3.1 Syntax

The definition of the syntactic unit <main part> in the Common Base text, section 2.1 may be replaced by the following:

```
<main part>
      ::= class <class identifier>
          <formal parameter part>;
          <value part> <specification part>
          <protection part>
          <virtual part> <class body>
```

where <protection part> is defined as follows:

```
<protection part>
      ::= <empty>
       ! <protection part> <protection specification>

<protection specification>
      ::= hidden <identifier list>;
       ! protected <identifier list>;
       ! hidden protected <identifier list>;
       ! protected hidden <identifier list>;
```

2.3.2 Semantics

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

2.3.2.1 Protected

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

1) within the body of C or its subclasses
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 (see Common Base 2.2) were absent.

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

A class attribute may be specified protected only 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.

2.3.2.2 Hidden

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 (see Common Base 2.2) 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.

2.3.2.3 Combination of protected and hidden

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 protected, but not hidden, the innermost accessible attribute will be hidden.

3 Types and variables

3.1 Syntax

```
<type declaration>
      ::= <type> <type list>

<array declaration>
      ::=  array <array list>
       !  <type> array <array list>

<type>
      ::= <value type>
       !  <reference type>

<value type>
      ::=  integer
       !   short integer
       !   real
       !   long real
       !   Boolean
       !   character

<reference type>
      ::= <object reference>
       !   text

<object reference>
      ::=  ref ( <qualification> )

<qualification>
      ::= <class identifier>

<type list> ::= <type list element> !
                <type list> , <type list element>

<type list element> ::= <identifier> !
                        <constant element>

<constant element> ::= <identifier> = <value expression> !
                       <identifier> = <string>
```

3.2.2 Characters

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

3.2.2.1 Collating sequence

The set of internal characters is ordered according to an implementation defined collating sequence. The collating sequence defines a one-to-one mapping between internal characters and integers expressed by the function procedures:

```
integer procedure rank(c); character c;
```

whose value is in the range (0,N-1), where N is the number of internal characters, and

```
character procedure char(n); integer n;
```

The parameter value must be in the range (0,N-1), otherwise a run time error is caused.

Example:

Most character codes are such that the digits (0-9) are character values which are consecutive and in ascending order with respect to the collating sequence. Under this assumption, the expressions

"rank(c) - rank('0')" and "char(rank('0') + i)"

provide implementation independent conversion between digits and their arithmetic values.

3.2.2.2 Character subsets

Two character subsets are defined by the standard non-local procedures:

```
Boolean procedure digit(c); character c;
```

which is true if c is a digit, and

```
Boolean procedure letter(c); character c;
```

which is true if c is a letter of the English alphabet, i.e. a-z, A-Z (but not for additional national letters like æ,ø,å).

3.2.3 Text

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 "variable" or "constant". A constant frame always contains the same text value. A variable 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 frame it identifies. The special text reference notext identifies "no frame". The value of notext is the empty text value.

A text variable is a memory device which contains a complete structure consisting of a text reference and a "position indicator". A text variable is said to reference the text frame (or no frame) identified by its contents. The position indicator is used for accessing the individual characters of the frame referenced.

A text variable may reference both variable and constant text frames.

A text variable is said to possess a value, which is the value possessed by the text reference it contains.

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

3.2.3.1 Text objects and text frames

A "text object" is conceptually an instance of the following class declaration:

```
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 piece 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 that 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 4.4.2.1).

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.

3.2.3.2 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 section 10.

The components OBJ, START and LENGTH constitutes the text reference part of the variable. They identify the frame referenced (see 3.2.3.1). POS is used for accessing the individual characters of the frame referenced (see section 10.3).

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.

3.2.4 Representation of constants

3.2.4.1 Strings

```
<string>
      ::= <simple string>
       ! <string> <string separator> <simple string>
<simple string>
      ::= "<character sequence>"
<character sequence>
      ::= <empty>
       ! <character sequence> <ISO code>
       ! <character sequence> <non-quote>
       ! <character sequence> <text quote>
<ISO code>
      ::= !<unsigned integer>!
<non-quote>
      ::= <any printing character except ">
<text quote>
      ::= ""
<string separators>
      ::= <space>
       ! <end of line>
       ! <string separators> <space>
       ! <string separators> <end of line>
```

A <simple string> must be contained in one Program line. Long strings are included as a sequence of <simple string>s separated by <string separators>. The <end of line> marks the end of every line record.

Any printing character except the string quote represents itself. In order to include the complete ISO alphabet any character may be represented inside a string (or a character constant) by its ISO code according to table 4 and surrounded by code quotes (! - ISO code 33). The <unsigned integer> 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 strings as two consequtive quotes ("").

Examples:

string:	represents:
"AB" "CDE"	ABCDE
"AB" "CDE"	ABCDE
"!2!ABCDE!3!"	ABCDE enclosed in the STX and ETX characters
"AB""C""DE"	AB"C"DE

3.2.4.2 Character constants

<character designation>
::= <ISO code>
! <any printing character>

A character constant is either a single printing character (ISO code 32-126) itself or it is an <ISO code> (cf. 3.2.4.1) representing the corresponding character - in both cases surrounded by character quotes (´ - ISO code 39).

3.2.4.3 Numeric constants

In real constants the ampersand character (& - ISO code 38) represents the ´basis of ten´ of the reference language.

E.g. 2&12 stands for 2 12

A double ampersand must be used if and only if the constant is to be taken as a literal of type long real.

E.g. 0.3141592&&1, 2&&-1 etc.

Any such constant will be treated as real in case the type long real is not supported.

3.2.5 Extended arithmetic types

The type declarations short integer and long real are recommended extensions of the standard arithmetic types. An implementation may choose to support both, none or either of them alone but the following rules must be observed:

- If type short integer is supported the effect of any such declaration shall be that described in 3.2.5.1, otherwise short integer declared identifiers shall be treated as though they were declared integer.

- If type long real is supported the effect of any such declaration shall be that described in 3.2.5.2, otherwise long real declared identifiers shall be treated as though they were declared real.

Apart from the special considerations described in 3.2.5.1 and 3.2.5.2 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.

3.2.5.1 Short integers

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.

- There is no apparent difference in evaluation of an arithmetic expression with short integer constituents as compared with one without such constituents.

- 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 for the short integer type.

3.2.5.2 Long reals

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. The following rules govern the evaluation of expressions containing long real constituents:

- The operations +, -, * and / with at least one long real operand have the conventional meaning and are understood to be carried out with a higher arithmetic precision than if there were no long real constituents present. The result of any such operation is of type long real.

- The operation // is not defined for long real operands.

- If at least one of the branches of a conditional arithmetic expression evaluates to a long real value the whole expression is understood to be of type long real.

- The comparison operation in a relation involving at least one long real expression is to be carried out with a higher arithmetic precision than if there were no long real constituents present.

- A standard mathematical function of type real yields value of type long real in case that the argument is also of type long real. Furthermore the standard procedures TIME and EVTIME return long real results and finally the first formal parameter to the standard procedures PUTFIX, PUTREAL, OUTFIX and OUTREAL is assumed to be of type long real.

- The standard procedures GETREAL and INREAL should have type long real when it is not evident from the context which type it should be.

- Random drawing procedures involving real quantities work in single precision. It is permitted to restrict some of the other standard procedures to single precision only.

- The assignment of a long real value to a real variable may constitute a run time error in case that the range of the long real values exceeds that of real values. Alternatively a precision loss may occur at such 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 case that the range of the real values exceeds that of long real values.

3.2.6 Constant declarations

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>.

The constant elements of a block head are evaluated from left to right.

3.2.7 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.

real	0.0
integer	0
Boolean	false
character	implementation defined
object reference	none
text	notext

3.2.8 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 "type universal". Any type is subordinate to the universal type (Cf. sections 2.2.3, 8.2.2 and 8.2.3).

4 Expressions

4.1 Value and reference expressions

4.1.1 Syntax

```
<label>
      ::= <identifier>

<expression>
      ::= <value expression>
       ! <reference expression>
       ! <designational expression>

<value expression>
      ::= <arithmetic expression>
       ! <Boolean expression>
       ! <character expression>

<reference expression>
      ::= <object expression>
       ! <text expression>
```

4.1.2 Semantics

The syntax for label represents a restriction compared with ALGOL 60.

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. The type of an arithmetic expression is that of its value. The following deviations from ALGOL 60 are introduced;

1) An expression of the form

   <factor>**<primary>
   is of type real.

2) A conditional arithmetic expression is of type integer if both alternatives are of type integer, otherwise its type is (long) real. If necessary, a conversion of the value of the selected alternative is invoked.

4.2 Character expressions

4.2.1 Syntax

```
<simple character expression>
      ::= '<character designation>'
       ! <variable>
       ! <function designator>
       !  ( <character expression> )

<character expression>
      ::= <simple character expression>
       ! <if clause> <simple character expression>
           else <character expression>
```

4.2.2 Semantics

A character expression is of type character. It is a rule for obtaining a character value. A character designation is either an external character or another implementation defined representation of an internal character.

4.3 Object expressions

4.3.1 Syntax

```
<simple object expression>
      ::= none
       ! <variable>
       ! <function designator>
       ! <object generator>
       ! <local object>
       ! <qualified object>
       !  ( <object expression> )

<object expression>
      ::= <simple object expression>
       ! <if clause><simple object expression>
           else <object expression>

<object generator>
      ::= new <class identifier> <actual parameter part>

<local object>
      ::= this <class identifier>

<qualified object>
      ::= <simple object expression>
          qua <class identifier>
```

4.3.2 Semantics

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

4.3.2.1 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 or array or procedure in question.

3) An object generator, local object, or qualified object is qualified by the class of the identifier following the symbol "new", "this", or "qua" respectively.

4) A conditional object expression is qualified by the 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 (see section 7). 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.

4.3.2.2 Object generators

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. (For parameter transmission modes, see section 8).

2) Control enters the object through its initial begin whereby it becomes operating in the "attached" state (see section 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 section 9.1), or

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

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

4.3.2.3 Local objects

A local object "this C" is a meaningful expression provided, of course 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 section 7.2).

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 procedure or class body "textually enclosing" means containing its declaration.

4.3.2.4 Instantaneous qualifications

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 (See also section 7).

4.4 Text expressions

4.4.1 Syntax

```
<simple text expression> ::= notext !
                             <string> !
                             <variable> !
                             <function designator> !
                             (<text expression>)

<text expression> ::= <simple text expression> !

                      <if clause> <simple text expression>
                       else <text expression>
```

4.4.2 Semantics

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

The result of evaluating

- notext, or an empty string, identifies an anonymous text variable whose contents are defined by the conditions (2) of section 3.2.3.2.

- 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 ideentifies 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 section 10.

5.2 Text value relations

5.2.1 Syntax

```
<text value relation>
      ::= <simple text expression>
          <relational operator> <simple text expression>
```

5.2.2 Semantics

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.

5.3 Object relations

5.3.1 Syntax

```
<object relation>
      ::= <simple object expression>
          is <class identifier>
       ! <simple object expression>
          in <class identifier>
```

5.3.2 Semantics

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.

5.4 Reference relations

5.4.1 Syntax

```
<reference comparator>
      ::= == ! =/=

<reference relation>
      ::= <object reference relation>
       ! <text reference relation>

<object reference relation>
      ::= <simple object expression>
          <reference comparator>
          <simple object expression>

<text reference relation>
      ::= <simple text expression>
          <reference comparator>
          <simple text expression>
```

5.4.2 Semantics

The reference comparators "==" and "=/=" may be used for the comparison of references (as distinct from the corresponding referenced values). Two object (text) references X and Y are said to be "identical" if they refer to the same object (text frame) or if they both are none (notext). 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".

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
```

Observe that 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 (cf. section 4.4.2.1).

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

The following example further illustrates the definitions of section 4.4.2.1.

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

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

Reference comparators have the same priority level as the relational operators.

6 Statements

```
<statement>
      ::= <ALGOL unconditional statement>
       ! <conditional statement>
       ! <for statement>
       ! <connection statement>
       ! <while statement>

<unlabelled basic statement>
      ::= <assignment statement>
       ! <go to statement>
       ! <dummy statement>
       ! <procedure statement>
       ! <activation statement>
       ! <object generator>

<conditional statement>
      ::= <ALGOL conditional statement>
       ! <if clause><connection statement>
       ! <if clause><while statement>
```

For <connection statement> see section 7.2.
For <activation statement> see section 14.2.3.

6.1 Assignment statements

6.1.1 Syntax

```
<assignment statement>
      ::= <value assignment>
       ! <reference assignment>

<value left part>
      ::= <variable>
       ! <procedure identifier>
       ! <simple text expression>

<value right part>
      ::= <value expression>
       ! <text expression>
       ! <value assignment>

<value assignment>
      ::= <value left part> := <value right part>

<reference left part>
      ::= <variable>
       ! <procedure identifier>

<reference right part>
      ::= <reference expression>
       ! <reference assignment>

<reference assignment>
      ::= <reference left part> :- <reference right part>
```


6.1.2 Semantics

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. A text procedure identifier 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.

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 section 3.2.6.

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.

Any expression which is, or is part of, the left part of an assignment is evaluated prior to the evaluation of the evaluation of the right part.

For a detailed description of the text value assignment, see section 10.6. 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.

For text reference assignment see section 10.5.

6.1.2.1 Arithmetic value assignment

In accordance with ALGOL 60, any arithmetic value may be assigned to a left part of type real or integer. If necessary, an appropriate transfer function is invoked.

Example:

Consider the statement (not legal in ALGOL 60):

```
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.

6.1.2.2 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.

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

Case 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.

Case 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 1:

Let "Gauss" be the class declared in the example of the section 2.2.

```
ref (Gauss) G5, G10;
G5:- new Gauss(5); G10:- new Gauss(10);
```

The values of G5 and G10 are now Gauss objects. See also example 1 of section 7.1.1.

Example 2:

Let "point" and "polar" be the classes declared in the example of section 2.2.2.

```
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.

6.2 For statements

6.2.1 Syntax

```
<controlled variable>
      ::= <simple variable>

<controlled statement>
      ::= <statement>

<for statement>
      ::= <for clause><controlled statement>
       !  <label> : <for statement>

<for clause>
      ::=  for <controlled variable>
           <for right part> do

<for right part>
      ::=  := <value for list>
       !   :- <reference for list>

<value for list>
      ::= <value for list element>
       !  <value for list> ,
          <value for list element>

<reference for list>
      ::= <reference for list element>
       !  <reference for list> ,
          <reference for list element>

<value for list element>
      ::= <value expression>
       !  <text value>
       !  <arithmetic expression>
           step <arithmetic expression>
           until <arithmetic expression>
       !  <value expression>
           while <Boolean expression>

<reference for list element>
      ::= <reference expression>
       !  <reference expression>
           while <Boolean expression>
```


6.2.2 Semantics

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.

6.2.3 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 non-local 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;
Alfa: if Delta*(C-A3) > 0 then go to Beta;
      S;
      Delta:= A2;
      C:= C + Delta;
      go to Alfa;
Beta: next for list element
```


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

```
Alfa: C:= V;
      if not B then go to Beta;
      S;
      go to Alfa;
Beta: next for list element
```

4. R
=====

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

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

```
Alfa: C:- R;
      if not B then go to Beta;
      S;
      go to Alfa;
Beta: next for list element
```

6.2.4 The controlled variable

The semantics of this section (6.2) is valid when the controlled variable is a simple variable which is not a formal parameter called by name, or a procedure identifier.

To be valid, all for list elements in a for statement (defined by textual substitution, section 6.2.3) must be semantically and syntactically valid.

In particular each implied reference assignment in cases 4 and 5 of section 6.2.3 is subject to the rules of section 6.1.2.2, and section 10.5, and each text value assignment in cases 1 and 3 of section 6.2.3 is subject to the rules of section 10.6.

6.2.5 The value of the controlled variable upon exit

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

6.2.6 Labels local to the controlled statement

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

6.3 While statement

6.3.1 Syntax

```
<while statement>
      ::= while <Boolean expression>
          do <statement>
       ! <label> : <while statement>
```

6.3.2 Semantics

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.

6.4 Prefixed blocks

6.4.1 Syntax

```
<block>
      ::= <ALGOL block>
       !  <prefixed block>

<block prefix>
      ::= <class identifier> <actual parameter part>

<main block>
      ::= <unlabelled block>
       !  <unlabelled compound>

<unlabelled prefixed block>
      ::= <block prefix> <main block>

<prefixed block>
      ::= <unlabelled prefixed block>
       !  <label> : <prefixed block>
```

6.4.2 Semantics

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 section 2.2.2.

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 (cf. section 11).

Example:

Let "hashing" be the class declared in the example of section 2.2.3. 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.

7 Remote accessing

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

1) the object,

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

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". The former is an adaptation of a technique proposed in (3), the latter corresponds to the connection mechanism of SIMULA I (1).

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

7.1 Remote identifiers

7.1.1 Syntax

```
<attribute identifier>
      ::= <identifier>

<remote identifier>
      ::= <simple object expression>.<attribute identifier>
       ! <simple text expression>.<attribute identifier>

<identifier 1>
      ::= <identifier>
       ! <remote identifier>

<variable identifier 1>
      ::= <identifier 1>

<simple variable 1>
      ::= <variable identifier 1>

<array identifier 1>
      ::= <identifier 1>

<variable>
      ::= <simple variable 1>
       ! <array identifier 1> ( <subscript list> )

<procedure identifier 1>
      ::= <identifier 1>

<function designator>
      ::= <procedure identifier 1> <actual parameter part>

<procedure statement>
      ::= <procedure identifier 1> <actual parameter part>

<actual parameter>
      ::= <expression>
       ! <array identifier 1>
       ! <switch identifier>
       ! <procedure identifier 1>
```


7.1.2 Semantics

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 item 1 is the value X and whose item 2 is C.

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.

Condition 1 corresponds to a run time check which causes a run time error if thevalue 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 in section 10.1.

Example 1:

Let G5 and G10 be variables declared and initialized as in example 1 of section 6.1.2.2. Then an expression of the form

```
G5.integral(.....) or G10.integral(.....)
```

is an approximation to a definite integral obtained by applying respectively a 5 point or a 10 point Gauss formula.

Example 2:

Let P1 and P2 be variables declared and initialized as in example 2 of section 6.1.2.2. 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.

7.2 Connection

7.2.1 Syntax

```
<connection block 1>
      ::= <statement>

<connection block 2>
      ::= <statement>

<when clause>
      ::=  when <class identifier> do <connection block 1>

<otherwise clause>
      ::= <empty>
       !   otherwise <statement>

<connection part>
      ::= <when clause>
       !  <connection part> <when clause>

<connection statement>
      ::=  inspect <object expression>
           <connection part> <otherwise clause>
       !   inspect <object expression> do
           <connection block 2> <otherwise clause>
       !  <label> : <connection statement>
```

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

Example:

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 (*).

7.2.2 Semantics

The purpose of the connection mechanism is to provide implicit definitions of the above items 1 and 2 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 idenfification. 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.)

Example:

Let "Gauss" be the class declared in the example of section 2.2. Then within the connection block 2 of the connection statement

```
inspect new Gauss(5) do begin ..... end
```

a procedure "integral" is available for numeric integration by means of a 5 point Gauss formula.

8 Procedures and parameter transmission

8.1 Syntax

```
<procedure heading>
      ::= <procedure identifier> <formal parameter part> ;
          <mode part><specification part>

<mode part>
      ::= <value part> <name part>
       ! <name part> <value part>

<name part>
      ::= name <identifier list>;
       ! <empty>

<specifier>
      ::= <type>
       ! array
       ! <type> array
       ! label
       ! switch
       ! procedure
       ! <type> procedure

<parameter delimiter>
      ::= ,
```

For actual parameter see section 7.1.1.

8.2 Semantics

With respect to procedures, SIMULA 67 deviates from ALGOL 60 on the following points:

1) Specification is required for each formal parameter.

2) The ALGOL specifier "string" is replaced by "text".

3) A "name part" is introduced as an optional part of a procedure heading to identify parameters called by name. Call by name is not the default parameter transmission mode.

4) Call by name is redefined in the case that the type of actual parameter does not coincide with that of the formal specification.

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

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 fig. 8.1 for the different kinds of parameters to procedures. The upper left subtable defines transmission modes available for parameters of class declarations.

	Transmission modes		
Parameter	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

fig. 8.1 Transmission modes

8.2.1 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. The evaluation takes place at the time of procedure entry or object generation.

The call by value of value type and value type array parameters is as in ALGOL 60.

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 in section 10.5, and "copy" in section 10.4).

Value specification is redundant for a parameter of value type.

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

8.2.2 Call 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 or object generation.

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 of section 6.1.2.2. Since in this case the formal parameter is a reference type variable, its contents may be changed by reference assignments within the procedure body, or within or without (by remote accessing) a class 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.

8.2.3 Call by name

Call by name is an optional transmission mode available for parameters to procedures. It represents a textual replacement as in ALGOL 60.

However, for an expression within a procedure body which is

1) a formal parameter called by name,

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

3) a function dseignator 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 lattper type. This assignment is subject to the rules of section 6.1.2. 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 section 4.4.2.1).

Section 6.1.2 defines the meaning of an assignment to a variable which is a formal parameter called by name, or is a subscripted variable whose array identifier is a formal parameter called by name, if the type of the actual parameter does not coincide with that of the formal specification.

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

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

9 Sequencing

9.1 Block instances and states of execution

The constituent parts of a program execution are dynamic instances of blocks, being sub-blocks, prefixed blocks, connection blocks and class bodies.

A block instance is said to be "local to" the one 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 (see section 11) 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 proghram point immediately following the statement or expression which caused the generation of the block instance. For sequencing of class objects see sections 9.2 and 9.3.

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 (see section 11) 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 section 9.3.1). The object may reenter the attached state through the execution of a call statement (see section 9.3.2), 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 section 9.3.3). 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 (see section 11) 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 in section 9.3. 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 such.

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 desribed 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:

- Zi-1 is attached to Zi, or
- 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:

- X is dynamically enclosed by the head of C.
- 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 operative (non-operating) 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.

Just before, and just after the execution of the detach statement in line 4, the situations are:

Fig. 9.1

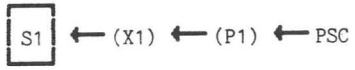


Fig. 9.2

Before and after the detach in line 16:

Fig. 9.3

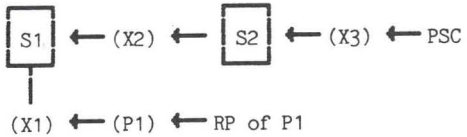


Fig. 9.4

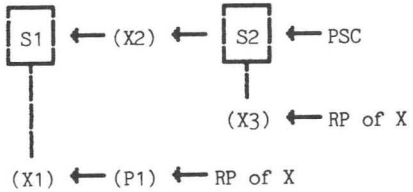
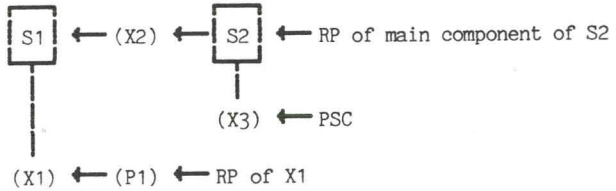


Fig. 9.4 also shows the situation before the resume in line 20. After this resume:

Fig. 9.5



Before and after the detach in line 10:

Fig. 9.6

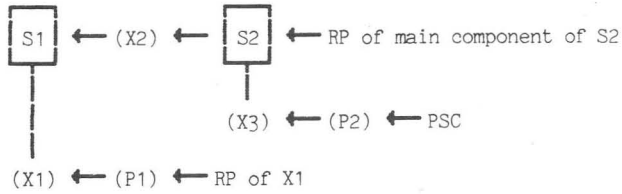
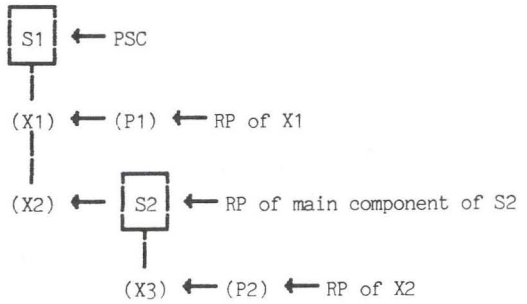
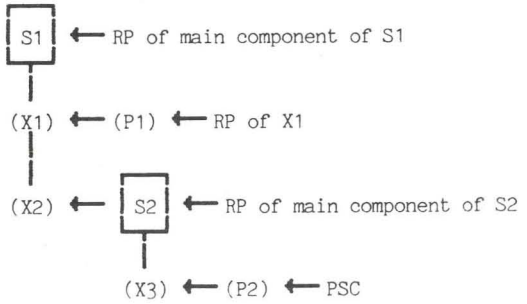


Fig. 9.7



Note that X3 is still the operative component of S2 and does not have a reactivation point of its own. Fig. 9.7 also shows the situation before the call in line 25. After this call, the situation in fig. 9.6 is reestablished. If, however, the call in line 25 is replaced by a "resume(X2)" the following situation arises:

Fig. 9.8



If now a "resume(X1)" is executed at * in line 11, the PSC is moved to the "RP of X1" in fig. 9.8, leaving an "RP of X2" at the former PSC. If instead a "detach" is executed, fig. 9.8 leads back to fig. 9.7.

9.3 Quasi-parallel sequencing

A quasi-parallel system is created through the 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 invokation of the detach attribute of such a block instance X:

If X is an instance of a prefixed block the detach stament has no effect.

Assume that 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.

   Assume X is operating. The effect of the detach statement is:

   - 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.

   - 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 section 9.3.2).

   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:

   - 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.

   - 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

"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.

Assume Y is a detached object. The effect of the call statement is:

- 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.

- 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

"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:

- 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.

- 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, and let X be the block instance containing P. Consider the execution of the goto statement:

1) Let Y denote the block instance currently containing the PSC.

2) If X equals Y the PSC i moved to P.

3) Othewrwise, if Y is the outermost block instance the goto statement constitutes an error.

4) Otherwise the effect is that of the PSC passing through the final end of Y (see section 9.3.4) whereafter the process is immediately repeated from 1).

10 The type "text"

Cf. sections 3.2.3, 4.4.2, 5.2 and 5.4.

10.1 Text attributes

The following procedures are attributes of any text variable. They may be accessed by remote identifiers of the form

<simple text expression>.<procedure identifier>

Boolean procedure constant	(cf. 10.2)
integer procedure start	(cf. 10.2)
integer procedure length	(cf. 10.2)
text procedure main	(cf. 10.2)
integer procedure pos	(cf. 10.3)
procedure setpos	(cf. 10.3)
boolean procedure more	(cf. 10.3)
character procedure getchar	(cf. 10.3)
procedure putchar	(cf. 10.3)
text procedure sub	(cf. 10.7)
text procedure strip	(cf. 10.7)
integer procedure getint	(cf. 10.9)
long real procedure getreal	(cf. 10.9)
integer procedure getfrac	(cf. 10.9)
procedure putint	(cf. 10.10)
procedure putfix	(cf. 10.10)
procedure putreal	(cf. 10.10)
procedure putfrac	(cf. 10.10)

In the following section "X" denotes a text variable unless otherwise is specified.

10.2 "constant", "start", "length" and "main"

```
Boolean procedure constant;
constant:= if OBJ == none then true 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;
overlapping:= if X.main =/= Y.main then false else
              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 X.start >= Y.start
      and 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.

10.3 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 in sections 10.9 and 10.10 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 else
begin getchar:= OBJ.MAIN(START + POS - 1);
      POS:= POS + 1
end;

procedure putchar(c); character c;
if OBJ == none then error else
if OBJ.CONST then error else
if POS > LENGTH then error else
begin OBJ.MAIN(START + POS - 1):= c;
      POS:= POS + 1
end;
```

Note that 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> (see section 4.4.2).

Example:

```
procedure compress(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 compress;
```

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.

10.4 Text generation

The following standard procedures are available for text frame generation:

```
text procedure blanks(n); integer n;
if n < 0 then error 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(n)", with n > 0, references a new variable 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 (see section 10.6).

```
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(T)", with T =/= notext, references a new variable main frame which contains the same text value as T.

10.5 Text reference assignment

Syntax, see section 6.1.1.

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 (see section 6.1.2). The effect of the assignment is defined as the four component assignments:

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

10.6 Text value assignment

Syntax, see section 6.1.1.

Let X be the text variable identified as the result of evaluating the left part (see section 4.4.2) of a text value assignment, and let Y denote the text variable identified by evaluating the corresponding right part (see section 6.1.2): 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 eqiuvalent 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.

10.7 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 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 is the smallest integer such that the remaining characters of X, if any, are blanks.

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.

10.8 Numeric text values

10.8.1 Syntax

```
<EMPTY>
      ::=

<DIGIT>
      ::= 0 ! 1 ! 2 ! 3 ! 4
       ! 5 ! 6 ! 7 ! 8 ! 9

<DIGITS>
      ::= <DIGIT>
       ! <DIGITS> <DIGIT>

<BLANKS>
      ::= <EMPTY>
       ! <BLANKS> <BLANK>

<SIGN>
      ::= <EMPTY> ! + ! -

<SIGN PART>
      ::= <BLANKS> <SIGN> <BLANKS>

<INTEGER ITEM>
      ::= <SIGN PART> <DIGITS>

<FRACTION>
      ::= . <DIGITS>

<DECIMAL ITEM>
      ::= <INTEGER ITEM>
       ! <SIGN PART> <FRACTION>
       ! <INTEGER ITEM> <FRACTION>

<EXPONENT>
      ::= <LOWTEN CHARACTER> <INTEGER ITEM>

<REAL ITEM>
      ::= <DECIMAL ITEM>
       ! <SIGN PART> <EXPONENT>
       ! <DECIMAL ITEM> <EXPONENT>

<GROUPS>
      ::= <DIGITS>
       ! <GROUPS> <BLANK> <DIGITS>
```



```
<GROUPED ITEM>
      ::= <SIGN PART> <GROUPS>
       ! <SIGN PART> . <GROUPS>
       ! <SIGN PART> <GROUPS> . <GROUPS>

<NUMERIC ITEM>
      ::= <REAL ITEM>
       ! <GROUPED ITEM>

<LOWTEN CHARACTER>
      ::=
```

10.8.2 Semantics

The syntax applies to sequences of characters, i.e. to text values. <BLANK> stands for a blank character.

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.

10.9 "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 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>, (see section 4.4.2).

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. If the number is an integer within an implementation defined range, the conversion is exact.

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.

10.10 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 X.length+1. Note that this increment is lost immediately if X does not correspond to a <variable>, (see section 4.4.2).

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.

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, zeroes 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 point if n<=0, and with a decimal point 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 point. 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(1,10);
amount:- Tr.sub(20,5);
price:- Tr.sub(30,6);
payment:- Tr.sub(60,10);
.....................
if type.strip = "order" then
begin pay:= amount.getint * price.getfrac;
      total:= total + pay;
      payment.putfrac(pay,2)
end
```


11 Input-output

The semantics of certain I/O facilities will rely on the intuitive notion of "files" ("data sets") which are collections of data external to the the program and organized in a sequential or addressable manner. We shall speak of a "sequential file" or a "direct file" according to the method of organization.

Examples of sequential files are:

- a batch of cards
- a series of printed lines
- input from a keyboard
- data on a tape

An example of a direct file is a collection of data items on a drum, or a disc, with each item identified by a unique index.

The individual logical unit in a file will be called an "image". Each "image" is an ordered sequence of characters.

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". The class is not explicitly available in any users program.

The program acts as if it were enclosed in the following block:

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

where outlength is an integer constant representing the length of a printed line as defined for the particular implementation.

In any program execution the unique instance of this prefixed block constitutes the head of the outermost quasi-parallel system (see section 9.2).

The parameter INPUT_LINELENGTH may be omitted; in this case all occurrences elsewhere of it are replaced by 80. The actual values of INPUT_LINELENGTH and OUTPUT_LINELENGTH are device dependent, their default values are 80 and 132 respectively.

Within the definition of the I/O semantics, identifiers in CAPITAL LETTERS represent quantities which are not accessible in a user program. A series of dots is used to indicate that actual coding is either found elsewhere, described informally, or implementation defined.

The overall organization of "BASICIO" is as follows:

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

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

"SYSIN" and "SYSOUT" represent a card-oriented standard input unit and a printer-oriented standard output unit. A program may refer to the corresponding file objects through "sysin" and "sysout" respectively. Most attributes of these file objects are directly available as a result of the implied connection blocks enclosing the program.

The files "SYSIN" and "SYSOUT" will be opened and closed within "BASICIO", i.e. outside the program itself.

11.1 The class "FILE"

11.1.1 Definition

```
class FILE(FILENAME,.....); value FILENAME; text FILENAME; ...
      virtual: procedure open, close;
      begin text image;
            Boolean OPEN;
            procedure setpos(i); integer i;
                      image.setpos(i);
            integer procedure pos;
                      pos:= image.pos;
            Boolean procedure more;
                      more:= image.more;
            integer procedure length;
                      length:= image.length;
            ........
      end FILE;
```

11.1.2 Semantics

Within a program, an object of a subclass of "FILE" is used to represent a file. The following four types are predefined:

"infile"	representing a sequential file where input operations (transfer of data from file to program) are available.
"outfile"	representing a sequential file where output operations (transfer of data from program to file) are available.
"directfile"	representing a direct file with facilities for both input and output.
"printfile"	(a subclass of outfile) representing a sequential file with certain facilities oriented towards line printers.

An implementation may restrict, in any way, the use of these classes for prefixing or block prefixing. System defined subclasses may, however, be provided in an implementation.

Each FILE object has a text attribute "FILENAME". It is assumed that this text value identifies an external file which, through an implementation defined mechanism, remains associated with the FILE object. The effect of several file objects representing the same (external) file is implementation defined.

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. An implementation may require that "image", at the time of an input or output of an image, refers to a main frame.

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

A file is either "open" or "closed", as indicated by the variable "OPEN". Input or output of images may only take place on an open file. A file is initially closed (except SYSIN and SYSOUT as seen from the program).

The procedures "open" and "close" perform the opening and closing operations on a file. Since the procedures are virtual quantities they may be redefined completely (i.e. at all access levels) for objects belonging to special purpose subclasses of infile, outfile, etc.

These procedures will be implementation defined, but they must conform to the following pattern.

```
boolean procedure open (fileimage,....);
text fileimage; ......
if OPEN then ERROR
else if ... ! the file was opened sucessfully; then
begin open:= OPEN:= true;
      image:- fileimage;
      .......
end open;

procedure close (.......);
begin OPEN:= false;
      ........
      image:- notext;
end close;
```

The procedures may have additional parameters and additional effects.

11.2 The class "infile"

11.2.1 Definition

```
FILE class infile; virtual: Boolean procedure endfile;
                            procedure inimage;
      begin boolean procedure open(fileimage,...);
            text fileimage;
            if OPEN then ERROR
            else if !the file was opened successfully; then
            begin open:= OPEN:= true
                  ENDFILE:= false;
                  image:- fileimage;
                  image:= notext;
                  setpos(length + 1);
            end open;

            procedure close .....;
            begin .......;
                  ENDFILE:= true;
            end close;

            Boolean ENDFILE;

            Boolean procedure endfile;
                  endfile:= ENDFILE;

            procedure inimage;
            begin .......;
                  if ENDFILE then image:= "!25!";
                  setpos(1);
            end inimage;

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

```
        Boolean procedure lastitem;
        begin while not ENDFILE do
              begin while more do
                          if inchar <> ' ' then
                          begin setpos(pos-1);
                                goto exit;
                          end;
                    inimage;
              end;
              lastitem:= true;
        exit:
        end lastitem;

        integer procedure inint;
        begin text T;
              if lastitem then ERROR;
              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; integer m;
              intext:- T:- blanks(w);
              for m:= 1 step 1 until w do
                    T.putchar(inchar);
        end intext;
        ........
        ENDFILE:= true;
        ........
end infile;
```


11.2.2 Semantics

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 is too short to contain the external image. If it is longer than the external image, the latter is left adjusted and the remainder of the text is blank filled. The position indicator is set to one.

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" when ENDFILE has the value true is a runtime error.

The procedure "open" will give ENDFILE the value false and set "image" to blanks. Otherwise it conforms to the pattern of section 11.1.2.

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

The remaining procedures provide mechanisms for "item oriented" input, which treat the file as a "continuous" stream of characters with a "position indicator" (pos) which is relative to the first character of the current image.

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

If the remainder of the file contains one or more non-blank characters, "lastitem" has the value false, and the position indicator of the file is set to the first non-blank character.

The procedures "inreal" and "infrac" are defined in terms of the corresponding de-editing procedures of "image". Otherwise the definition of either procedure is analogous to that of "inint". These three procedures will scan past and convert a numeric item containing the first non-blank character and contained in one image, excepting an arbitrary number of leading blanks.

The expression "intext(n)" where n is a positive integer is a reference to a new varable main frame of length n containing the next n characters of the file. "pos" is moved to the following character. The expression "intext(0)" references notext.

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

Example:

The following piece of program will input a matrix by columns. It is assumed that consecutive elements are separated by blanks or contained in different images. The last element of each column should be followed immediately by an asterisk.

```
begin array a(1:n,1:m);
   integer i,j;
   procedure error; ........;
   for j:= 1 step 1 until m do
   begin for i:= 1 step 1 until n-1 do
      begin a(i,j):= inreal;
         if (if sysin.more then inchar <> ' '
                                 else false)
         then error
         end;
         a(n,j):= inreal;
         if inchar <> '*' then error;
      end;
end
```


11.3 The class "outfile"

11.3.1 Definition

```
FILE class outfile; virtual: procedure outimage;
      begin boolean procedure open(fileimage,...);
            text fileimage;
            if OPEN then ERROR
            else if ... !the file was opened successfully; then
            begin open:= OPEN:= true;
                  image:- fileimage;
                  setpos(1);
            end open;

            procedure close ..........;
            begin ..... if pos <> 1 then outimage; end;

            procedure outimage;
            begin if not OPEN then ERROR;
                  ........
                  image:= notext;
                  setpos(1)
            end outimage;

            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;
                  if pos+w-1 > length then outimage;
                  FIELD:- image.sub(pos,w);
                  setpos(pos+w)
            end FIELD;

            procedure outint(i,w); integer i,w;
                  FIELD(w).putint(i);

            procedure outfix(r,n,w); long real r; integer n,w;
                  FIELD(w).putfix(r,n);

            procedure outreal(r,n,w); long real r;
                                      integer n,w;
                  FIELD(w).putreal(r,n);
```

```
        procedure outfrac(i,n,w); integer i,n,w;
            FIELD(w).putfrac(i,n);

        procedure outtext(T); text T;
            FIELD(T.length):= T;
        ......
    end outfile;
```

11.3.2 Semantics

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 will react in an implementation defined way if the image length is not appropriate for the external file. The text is cleared to blanks and the position indicator is set to 1, after the transfer.

The procedure "close" will call "outimage" once if the position indicator is different from 1. Otherwise it conforms to the pattern of section 11.1.2.

The procedure "outchar" treats the file as a "continuous" stream of characters.

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

The procedures "outint", "outfix", "outreal" and "outfrac" are defined in terms of the corresponding editing procedures of "image". They have an additional integer parameter which specifies the width of the subtext into which the item will be edited.

For the procedure "outtext", the item width is equal to the length of the text parameter.

11.4 The class "directfile"

Note: The definition of "directfile" is presently under study by the SIMULA Developement Group.

11.4.1 Definition

```
FILE class directfile;
        virtual: Boolean procedure endfile;
                 procedure locate,inimage,outimage;
    begin integer LOC;
        integer procedure location;
            location:= LOC;

        procedure locate(i); integer i;
        begin if not OPEN then ERROR;
            ......
            LOC:= i;
        end locate;

        boolean procedure open(fileimage,...);
        text fileimage;
        if OPEN then ERROR
        else if ... !the file was opened successfully; then
        begin open:= OPEN:= true;
            image:- fileimage;
            setpos(1);
            locate(1);
        end open;

        procedure close .....; .....;

        Boolean procedure endfile; .......;

        procedure inimage;
        begin ......
            locate(LOC+1);
            setpos(1)
        end inimage;

        procedure outimage;
        begin ......
            locate(LOC+1);
            image:= notext;
            setpos(1)
        end outimage;
```

```
        character procedure 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 ................;
end directfile;
```


11.4.2 Semantics

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" normally contains the ordinal number of an external image. 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. The assignment may be accompanied by implementation defined checks and possibly by instructions to an external memory device associated with the given file.

The procedure "open" will locate the first image of the file. Otherwise it conforms to the rules of section 11.1.2.

The procedure "endfile" may have the value true only if the current value of LOC does not identify an image of the external file. The procedure is implementation defined.

The procedure "inimage" will transfer into the text "image" a copy of the external image currently identified by the variable LOC if there is one. Then the value of LOC is increased by one through a "locate" statement. 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 implementation defined. The procedure is otherwise analogous to that of section 11.2.

The procedure "outimage" will transfer a copy of the text value "image" to the external file, thereby adding to the file an external image whose ordinal number is equal to the current value of LOC. A runtime error occurs if the file cannot be made to contain the image. If the file contains another image with the same ordinal number, that image is deleted. The value of LOC is then increased by one through a "locate" statement. The procedure "outimage" is otherwise analogous to that of section 11.3.

The remaining procedures are analogous to the corresponding procedures of section 11.2 and 11.3.

11.5 The class "printfile"

11.5.1 Definition

```
outfile class printfile;
      begin integer LINES_PER_PAGE, SPACING, LINE;

            integer procedure line; line:= LINE;

            procedure linesperpage(n); integer n;
                  LINES_PER_PAGE:= n;

            procedure spacing(n); integer n; SPACING:= n;

            procedure eject(n); integer n;
            begin if not OPEN then ERROR;
                  if n > LINES_PER_PAGE then n:= 1;
                  if LINE > LINES_PER_PAGE then LINE:= 0;
                  ......
                  LINE:= n;
            end eject;

            boolean procedure open(fileimage,...);
            text fileimage;
            if OPEN then ERROR
            else if ... !the file was opened successfully; then
            begin open:= OPEN:= true;
                  image:- fileimage;
                  setpos(1);
                  eject(1)
            end open;

            procedure close ....;
            begin ......
                  if pos <> 1 then outimage;
                  SPACING:= 1;
                  eject(LINES_PER_PAGE);
                  LINES_PER_PAGE:= ......;
                  LINE:= 0;
            end close;
```



```
            procedure outimage;
            begin if not OPEN or image == notext
                              then ERROR;
                  if LINE > LINES_PER_PAGE then eject(1);
                  comment output the image on the line
                          denoted by LINE;
                  LINE:= LINE + SPACING;
                  image:= notext;
                  setpos(1);
            end outimage;

            LINES_PER_PAGE:= ......;
            SPACING:= 1;
      end printfile;
```

11.5.2 Semantics

An object of the class "printfile" is used to represent a 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 will 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 "lines per page" is zero, "LINES_PER_PAGE" is reset to the same implementation defined value as at the time of object generation. The effect is implementation defined if the parameter is less than zero.

The variable "SPACING" represents the value by which the variable "LINE" will be incremented after the next printing operation. The variable is set equal to 1 at the time of object generation and when the printfile is closed. Its value may be changed by the procedure "spacing". A call on the procedure "spacing" with 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 by an implementation either to mean successive printing operations on the same physical line, or to be an error.

The variable "LINE" indicates the ordinal number of the next line to be printed, 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 value of "LINE" is zero when the file is not open.

The procedure "eject" is used to position to a certain line identified by the parameter, n.

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 "outimage" operates according to the rules of section 11.3. In addition, it will update the variable "LINE".

The procedure "open" and "close" conform to the rules of section 11.1. In addition, "open" will position to the top of a page, and "close" will output the current value of "image" if "pos" is different from 1 and reset "LINE", "SPACING" and "LINES_PER_PAGE".

12 Random drawing

12.1 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 will replace 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**(2p+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).

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 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.

12.2 Random drawing procedures

The following procedures all perform a random drawing of some kind. Unless it is explicitly stated otherwise the drawing is effected 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.

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,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.

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

$$\prod_{i=0}^{n} u(i) < e^{**}(-a)$$

   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),

$$-\left(\sum_{i=1}^{c} \ln(u(i))/(a*b)\right) - \left((b-c)*\ln(u(c+1))/(a*b)\right)$$

   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:

1. draw a uniform <0,1> random number, u.

2. determine the lowest value of i, for which

A(i-1) <= u <= A(i)

3. compute D = A(i) - A(i-1)

4. 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.

13 Utility procedures

The following procedures are defined:

procedure histo (A,B,c,d); real array A,B; real c,d;

It will update 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.

procedure terminate_program;
! terminate program; ;

A call on this procedure will terminate program execution as if control passed through the final end.

14 System classes

Two additional system-defined classes are available:

```
class SIMSET; ...;
```

and

```
SIMSET class SIMULATION; ...;
```

The class SIMSET introduces list processing facilities corresponding to the "set" concept of SIMULA I (2). The class SIMULATION further defines facilities analogous to the "process" concept and sequencing facilities of SIMULA I.

The two classes 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 block levels at which such implicit declarations may occur in any one program.

In the following definitions, identifiers in capital letters, except "SIMSET" and "SIMULATION", represent quantities not accessible to the user. A series of dots is used to indicate that the actual coding is found in another section.

14.1 The class "SIMSET"

The class "SIMSET" contains facilities for the manipulation of circular two-way lists, called "sets".

14.1.1 General structure

14.1.1.1 Definition

```
class SIMSET;
    begin class linkage; ......;
          linkage class head; ......;
          linkage class link; ......;
    end SIMSET;
```

14.1.1.2 Semantics

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".

14.1.2 The class "linkage"

14.1.2.1 Definition

```
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;
```

14.1.2.2 Semantics

The class "linkage" is the common denominator for "set heads" and "set members".

"SUC" is a reference to the successor of this 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.

14.1.3 The class "link"

14.1.3.1 Definition

```
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 then
            begin if X.SUC =/= none then
                  begin PRED:- X;
                        SUC:- X.SUC;
                        SUC.PRED:- X.SUC:- this linkage;
                  end
            end
      end follow;

      procedure precede(X); ref (linkage) X;
      begin out;
            if X =/= none then
            begin if X.SUC =/= none then
                  begin SUC:- X;
                        PRED:- X.PRED;
                        PRED.SUC:- X.PRED:- this linkage;
                  end
            end
      end precede;

      procedure into(S); ref (head) S;
            precede(S);
end link;
```


14.1.3.2 Semantics

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).

14.1.4 The class "head"

14.1.4.1 Definition

```
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;
```

14.1.4.2 Semantics

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.

14.2 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.

The definition of "SIMULATION" which follows is only one of many possible schemes of organization of the class. An implementation may choose any other scheme which is equivalent from the point of view of any user's program.

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.

14.2.1 General structure

14.2.1.1 Definition

```
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 (BEFORE); Boolean BEFORE;
            begin ref (EVENT_NOTICE) P;
                  P:- SQS.last;
                  while P.EVTIME > EVTIME do
                        P:- P.pred;
                  if BEFORE then
                  while P.EVTIME = EVTIME do
                        P:- P.pred;
                  follow(P);
            end RANK;
      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 ACTIVATE ..........;
      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;
```


14.2.1.2 Semantics

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".

14.2.2 The class "process"

14.2.2.1 Definition

```
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
                    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
end process;
```


14.2.2.2 Semantics

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 userdefined 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 in section 9, 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.

14.2.3 Activation statements

14.2.3.1 Syntax

```
<activator>
      ::= activate
       ! reactivate

<activation clause>
      ::= <activator><object expression>

<simple timing clause>
      ::= at <arithmetic expression>
       ! delay <arithmetic expression>

<timing clause>
      ::= <simple timing clause>
       ! <simple timing clause> prior

<scheduling clause>
      ::= <empty>
       ! <timing clause>
       ! before <object expression>
       ! after <object expression>

<activation statement>
      ::= <activation clause> <scheduling clause>
```

14.2.3.2 Semantics

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 "ACTIVATE" local to SIMULATION.

```
procedure ACTIVATE (REAC, X, CODE, T, Y, PRIOR);
      value CODE; ref (process) X, Y; Boolean REAC, PRIOR;
      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 "PRIOR" is true if prior is in the timing clause, false if it is not used or 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"

14.2.4 Sequencing procedures

14.2.4.1 Definitions

```
procedure hold(T); long real T;
inspect FIRSTEV do
begin if T > 0 then EVTIME:= EVTIME + T;
      if suc =/= none then
      begin if suc.EVTIME <= EVTIME then
            begin out; RANK(false);
                  resume(current)
            end
      end
end hold;

procedure passivate;
begin inspect current do
      begin EVENT.out; EVENT:- none
      end;
      if SQS.empty then ERROR 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 ACTIVATE(REAC, X, CODE, T, Y, PRIOR);
value CODE; ref (process) X, Y; Boolean REAC, PRIOR;
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 go to 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;
            go to at
      end delay
      else if CODE = "at" then
at:   begin if T < time then T:= time;
            if T = time and PRIOR then go to direct;
            EVENT:- new EVENT_NOTICE(T, X);
            EVENT.RANK(PRIOR)
      end at
      else if (if Y == none then true
                              else Y.EVENT == none)
          then EVENT:- none else
      begin if X == Y then go to 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
      end;
      if z =/= current then resume(current);
exit:
end ACTIVATE;
```

14.2.4.2 Semantics

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 avoided within SIMULATION blocks.

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 proces 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 "ACTIVATE" represents an activation statement, as described in section 14.2.3. 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 the 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.

Example:

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)".

14.2.5 The main program

14.2.5.1 Definition

```
process class MAIN_PROGRAM;
begin
      while true do detach;
end MAIN PROGRAM;
```

14.2.5.2 Semantics

It is desirable that the main component of a simulation model, i.e. the SIMULATION block instance, should respond to the sequencing procedures of section 14.2.4 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 will immediately enter the main component as a result of the "detach" statement (cf. section 9.3.1). 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 userdefined statement of the SIMULATION block.

14.2.6 Utility procedures

14.2.6.1 Definition

```
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;
```

14.2.6.2 Semantics

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.

15 Separate compilation

A user defined procedure or class declaration can be compiled as a separate module. Other programs and/or separate modules can make references to a separately compiled module by means of an <external declaration>.

15.1 Syntax

```
<SIMULA source module>
      ::= <external head> <program>
       ! <external head> <procedure declaration>
       ! <external head> <class declaration>

<external head>
      ::= <empty>
       ! <external head> <external declaration>

<external declaration>
      ::= <external procedure declaration>
       ! <external class declaration>

<external procedure declaration>
      ::= external <kind> <type>
          procedure <external list>
       ! external <kind> procedure <external list>
       ! external <kind> procedure <external item>
          <procedure binding>

<kind>
      ::= <empty>
       ! <identifier>

<external class declaration>
      ::= external class <external list>

<external list>
      ::= <external item>
       ! <external list> , <external item>

<external item>
      ::= <identifier>
       ! <identifier> = <external identification>
```

<external identification>
::= <string>

<external binding>
::= is <procedure declaration>

15.2 Semantics

An <external declaration> is a substitute for a complete declaration of the corresponding class or procedure. An uncommitted occurrence of a standard identifier within a separately compiled class or procedure refers to the implicit declaration of the identifier at the outermost level of the program. Note, however, that the use of the standard procedure detach in a separately compiled class matches the implicit declaration within that class.

The <kind> of an <external procedure declaration> may indicate the source language in which the separately compiled procedure is written (e.g ASSEMBLY, COBOL, FORTRAN, PL1 etc.). The <kind> must be empty if this language is SIMULA. A non-SIMULA procedure cannot be used as an actual parameter corresponding to a formal procedure.

The <identifier> of an <external item> must be identical to the <identifier> of the corresponding separately compiled procedure or class. An <external item> may also introduce an <external identification> to identify the separately compiled module with respect to an operating system.

If the <identifier> of an <external class declaration> is referenced before the body of the separately compiled procedure or class or in a <program> block prefix, then this identifier must be declared in the <external head>.

In the case that an external procedure declaration contains a <procedure binding>, the procedure body must be empty. The dummy declaration of the procedure given in the <procedure binding> will define the usage of the procedure within the Simula source module, thus the external procedure is fully declared with respect to type and order of the parameters while the algorithm of the procedure body is given in a separate (non-Simula) module.

Example:

```
external class B, C;
B class E(F); ref (C) F;
begin external class D;
      external procedure A;
      ref (D) G;
      G:- new D;
      A(G);
      ......
end;
```

As a consequence of paragraph 2.2.1 all classes belonging to the prefix chain of a separately compiled class must be declared in the same block as this class. However, this need not be done explicitly; an <external declaration> of a separately compiled class may serve as an implicit declaration of all classes in its prefix chain. Possible conflicts between explicit and implicit declarations are automatically resolved by the system provided that the external items in question refer to the same separately compiled procedure or class, otherwise such conflict constitutes an error.

An implementation may restrict the number of block levels at which an <external class declaration> may occur.

16 Extensions to SIMULA

This section contains extensions to SIMULA which implementors are recommended to include. Extensions other than those listed here are allowed only if the following conditions are fulfilled:

a) The implementor provides a translator program, which takes any SIMULA program accepted by that SIMULA implementation and translates it into a readable Common Base program in recommended hardware representation. The resulting Common Base program may contain a minimum of calls to non-SIMULA procedures in cases where this is absolutely necessary due to a lack of facilities in SIMULA (e.g. TIME and DATE procedures).

b) Each SIMULA implementation has a switch which must be set to make the compiler accept programs with extensions not recommended in the Common Base SIMULA.

   A SIMULA implementation which allows extensions not in the Common Base, shall give warning messages for the use of such extensions.

c) All such non-SIMULA extensions should be reported to the SIMULA Standards Group, which will send such reports to the SSG members for comments. Responses from SSG members will be sent to the originator through the SSG.

17 Features being investigated

The SIMULA Standards Group and the SIMULA Developement Group are studying this definition in order to clarify possible obscure points or to recommend extensions to the language. At the time of print, the following features are under investigation:

- class directfile

- transplantation

18 References

1. P. Naur (Ed.): Revised Report on the Algorithmic Language ALGOL 60. CACM., vol. 6, No. 1, 1963, pp 1-17.

2. O-J. Dahl, K. Nygaard: "SIMULA - A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual." Norwegian Computing Center, Oslo.

3. C.A.R. Hoare: "Record Handling." Lectures delivered at the NATO Summer School, Villard-de-Lans, September 1966 (Academic Press.)

4. M. Abramowitz & I. A. Stegun (ed): Handbook of Mathematical Functions, National Bureau of Standard Applied Mathematics Series No. 55, p. 952 and C. Hastings formula (26.2.23) on p. 933.).

5. "Minutes from Annual Meeting of SIMULA Standards Group May 1970". Publication No. S-18, July 1970, Norwegian Computing Center, Oslo.

6. "Minutes from the Annual Meeting of the SIMULA Standards Group" from the following dates (NCC publication no):

January, 1973	(S-45)
September, 1973	(S-66)
October, 1974	(S-70)
September, 1976	(S-82)
September, 1977	(S-611)
September, 1978	(S-625)
September, 1979	(S-664)
August, 1980	(S-686)
September, 1981	(S-704)
September, 1982	(S-726)
September, 1983	(S-741)

19 Alphabetic index of syntactical units

For each syntactical unit, the section of definition is given. AR indicates that the definition is found in the "revised" ALGOL report (1). The numbers of the sections in this document where the syntactical unit is referenced are also indicated.The metalanguage brackets < and > have been removed from the syntactic units.