SIS – Standardiseringskommissionen i Sverige

Standarden utarbetad av

SIS STANDARDISERINGSGRUPP

SVENSK STANDARD SS 63 61 14

Första giltighetsdag	Utgåva	Sida
1987 - 05 - 20	1	1 (178)

Databehandling – Programspråk – SIMULA®

Denna standard utgörs av en specifikation av programspråket SIMULA, som är utarbetad inom SIMULA Standards Group (SSG), se bilaga D.

SIMULA är ett registrerat varumärke som ägs av Simula A.S.. OSLO. Norge. Rätt att använda benämningen tillkommer alla tillämpningar av programspråket SIMULA under förutsättning av att Simula A.S. nämns som varumärkesinnehavare och att tillämpningen är i överensstämmelse med denna standard.

Underhåll av SIMULA handhas av SIMULA Standards Group, som godkände specifikationen den 25 augusti 1986.

I standarden hänvisas till:

ISO 646–1983, som oförändrad överförts till svensk standard SS 63 61 29, Datarepresentation – Internationell 7-bits teckenkod för datautbyte,

ISO 1538–1984, Programming languages – ALGOL 60, för vilken ingen motsvarande svensk standard finns,

NCC 743–1984, Common Base Language av O–J Dahl, B Myhrhaug och K Nygaard, Norsk Regnesentral 1984 (ISBN 82–539–0225–5).

Data processing – Programming languages – SIMULA

This standard consists of specification of the programming language SIMULA elaborated within the SIMULA Standards Group (SSG), see Annex D.

SIMULA is a registered trade mark owned by Simula A.S., OSLO. Norway. The right to use the name is due to all implementations of the SIMULA programming language provided that Simula A.S. is stated as the owner of the trade mark. and that the implementation conforms to this standard.

The maintenance of SIMULA is handled by the SIMULA Standards Group, which approved the specification on August 25, 1986.

In the standard reference is made to:

ISO 646–1983. Information processing – ISO 7-bit coded character set for information interchange, which is fully adopted as Swedish standard SS 63 61 29,

ISO 1538–1984. Programming languages – ALGOL 60, for which no corresponding Swedish standard exists,

NCC 743–1984, Common Base Language by O–J Dahl, B Myhrhaug and K Nygaard, Norwegian Computing Center 1984 (ISBN 82–539–0225–5).

SIS TRYCKERI, STOCKHOLM
ITG/HK–01–636114–01–X

UDK 681.3.06:800.92

Beteckningen SS för svensk standard infördes 1978. Svensk standard med beteckning SEN, SIS eller SMS får beteckningen SS vid revidering.

TABLE OF CONTENTS

0 GENERAL INTRODUCTION

0.1 INTRODUCTION

SIMULA is a general purpose programming language. It inherits the algorithmic properties of ALGOL 60 and introduces methods for structuring data. The main characteristic of SIMULA is the possibility for convenient adaption towards specialized problem areas. Hence SIMULA can be used as a basis for special application languages.

In this standard the name SIMULA is considered synonymous with SIMULA 67. Although there exists a predecessor, SIMULA I, this latter language has achieved limited use. It is recommended that the language defined in this standard be referred to as STANDARD SIMULA.

SIMULA includes most of the ALGOL 60 language. Wherever ALGOL is referred to in this standard it relates to the the STANDARD ALGOL 60 definition (ISO 1538).

0.2 SCOPE AND FIELD OF APPLICATION

This standard establishes the definition of SIMULA and specifies conformity rules to related products, such as programs and processors. Its purpose is to facilitate interchange and promote portability of SIMULA programs between data processing systems.

This standard specifies:

a) the syntax, semantics and representation of SIMULA,

b) characteristics of processors (see 4.2.1) and their accompanying documents, and of SIMULA programs, required for conformity to this standard,

c) what is left to the discretion of the implementor, or to be specified for each implementation.

This standard does not specify:

a) results or issues that are explicitly left undefined or said to be undefined,

b) how non-valid programs are to be rejected and how this will be reported,

c) the relationship of the hypothetical computer, used to explain the actions which constitute the elaboration of a program, to an actual data processing system.

0.3 REFERENCES

ISO 646-1983: Information processing - ISO 7-bit coded character set for information interchange.

ISO 1538-1984: Programming languages - ALGOL 60.

NCC 743-1984: "Common Base Language" by O.-J. Dahl, B. Myhrhaug and K. Nygaard Norwegian Computing Center 1984. (ISBN 82-539-0225-5)

0.4 DEFINITIONS

For the purpose of this standard the following definitions apply.

Note: Several terms used in this standard are explained at the appropriate place in chapters 1 to 12. For convenience some of these have been included here as well, at times with a simplified definition. It is understood, however, that no difference of meaning is considered to exist, and all definitions of a term are equivalent.

0.4.1 Programs

0.4.1.1 Potential program:
A text, that is a sequence of characters or typographical marks, meant to be a sequence of tokens constituting a SIMULA program.

0.4.1.2 Valid program:
A potential program that is a program according to the rules in this standard.

0.4.1.3 Non-valid program:
A potential program that is not a program but can be turned into one by deleting or inserting a number of symbols.

0.4.1.4 Elaboration of a program:
A sequence of actions specified by the semantics to be carried out.

0.4.1.5 SIMULA program:
A valid program whose elaboration is defined by this standard for an indicated class of input data.

0.4.2 SIMULA Processors

0.4.2.1 Processor:
A translator, compiler or interpreter in combination with a data processing system, that accepts a potential program, transcribed in a form that can be processed by that data processing system, reports whether the potential program is valid or not, and if so requested is able to execute it, if it has not rejected it.

0.4.3 SIMULA Implementations

0.4.3.1 Implementation:
A well-documented processor is said to establish an implementation of SIMULA.

0.4.3.2 Implemented language:
The version of the language defined by the implementation.

0.4.3.3 Extension:
A rule in the implemented language that

a) is not given in this standard,

b) does not cause any ambiguity when added to this standard (but may serve to remove a restriction),

c) is within the scope of this standard.

0.4.3.4 Implementation-defined:
What is to be specified for each implementation.

0.4.3.5 Implementation-dependent:
What is left to the discretion of the implementor.

0.5 CONFORMITY

0.5.1 Requirements

0.5.1.1 Conforming programs

Conformity to this standard requires for a program that

a) it shall be a SIMULA program

b) a set of input data shall be given for which it has a defined meaning.

0.5.1.2 Conforming processors

Conformity to this standard requires for a processor that

a) it shall accept valid programs as being valid,

b) it shall reject non-valid programs as being non-valid,

c) it shall not elaborate a SIMULA program differently from what is defined in this standard,

d) it shall be accompanied by documents complying with the requirements below.

0.5.1.3 Documentation

It is required for the documents accompanying a conforming processor that these shall describe clearly

a) its purpose and the environment (hardware and software) in which it will work,

b) its intended properties, including

- the actions taken when results or issues occur, left undefined in this standard,
- conventions for issues said to be implementation-defined,
- what is provided for issues declared to be implementation-dependent,

c) all differences between the implemented language and this standard,

d) its logical structure,

e) the way to put it into use.

0.5.1.4 Conforming implementations

A conforming implementation shall comply with the above requirements for a processor and its accompanying documents.

0.5.2 Quantitative restrictions

The requirements specified in 0.5.1 shall allow for quantitative restrictions to rules stated or implied as having no such restriction in this standard, but only if they are fully described in the documents with the implementation. These restrictions are to be considered implementation-defined in as far as they are not dependent of any momentary resource restraint during execution of a program.

0.5.3 Extensions

An implementation that allows for extensions in the implemented language is considered to conform to this standard, notwithstanding 0.5.1 if

a) it would be conforming when the extensions were omitted,

b) those extensions are clearly described with the implementation,

c) while accepting programs that are non-valid according to the rules given in this standard, it provides means for indicating which part, or parts, of a program would have led to its rejection, had no extensions been allowed,

d) the implemented language is a super-language of SIMULA.

Extensions are allowed only if the following conditions are fulfilled:

a) The implementor provides a translator program, which takes any program accepted by that implementation and translates it into a valid program. The resulting 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 the language.

b) Each implementation has a switch which must be set to make the compiler or interpreter accept programs with extensions.

An implementation which allows extensions, shall give warning messages for the use of such extensions.

Valid programs using extensions shall be described as "conforming to the SIMULA Standard but for the following indicated parts".

0.5.4 Subsets

This standard does not include subsets.

0.6 TESTS

Whether an implementation is a conforming implementation may possibly be detected by a suite of test programs. If there is any uncertainty or doubt regarding acceptance of these programs then the conclusion drawn from the actual behaviour of the processor will prevail over those derived from its accompanying documents.

0.7 STATUS OF ANNEXES

In all parts of this standard, the annexes do not form an integral part of the standard but are included to provide extra information and explanation.

1 LEXICAL TOKENS

DEFINITIONAL CONVENTIONS

The meta language used in this standard to specify the syntax of the constructs is based on the Backus-Naur Form. The meanings of the various meta symbols are listed in the table below. Further (semantic) specifications of the constructs are given in prose and, in some cases, by equivalent program fragments. In such program fragments some identifiers introduced by declarations are printed in upper case. The use of upper case letters signifies that the identifier in question represents some quantity which is inaccessible to a program. An example of this convention is the identifier EVENT_NOTICE of chapter 12. Any other identifier that is defined elsewhere in the standard will denote the corresponding entity by its occurrence in such a program fragment.

Note: The use of program fragments as described above, as well as the description of standard facilities (see chapters 8-12) by algorithmic means should be taken as definitive only as far as their effects are concerned. An actual implementation should seek to produce these effects in as efficient a manner as practicable. Furthermore, where arithmetic of real type quantities is concerned, even the effects must be regarded as defined with only a finite degree of accuracy (see 3.5.3).

Examples are sometimes given to illustrate the constructs. Following the principles laid down in 1.1 the language keywords (see 1.3) are printed in a special manner in such examples.

Metalanguage Symbols

Metasymbol	*Meaning*
=	is defined to be
|	alternatively
[x]	0 or 1 instance of x
{ x }	0 or more instances of x
⟨ x | y ⟩	grouping: either x or y
xyz	the terminal symbol **xyz**
meta-identifier	a non-terminal symbol
...	see below

A meta-identifier is a sequence of letters, digits and hyphens beginning with a letter. The identifier has intentionally been chosen to convey a hint of its meaning to the reader. The exact meaning is, however, defined by its (single) occurrence on the left hand side of a production. When used outside productions these identifiers are generally written with spaces instead of hyphens, except in cases where possible ambiguities might result.

A few productions contain the ellipsis (...) as a right hand side. In such cases a prose explanation is given immediately below the production.

A sequence of terminal and non-terminal symbols in a production implies concatenation of the text that they ultimately represent. Within chapter 1 this concatenation is direct; no characters may intervene. In the remainder of the standard the concatenation is in accordance with the rules set out in this chapter.

The characters required to form SIMULA programs are those explicitly classified as "basic" in the table given in section 1.2. Additional characters of that table may be employed as described in that section.

A SIMULA source module consists of directive lines and program lines. Apart from 1.1 this standard is not concerned with directive lines. The lexical tokens used to construct program lines are classified into special symbols, identifiers, unsigned numbers, simple strings and character constants.

No lexical token may consist of more than 72 characters.

letter
= A | B | C | D | E | F | G | H | I
| J | K | L | M | N | O | P | Q | R
| S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i
| j | k | l | m | n | o | p | q | r
| s | t | u | v | w | x | y | z

The representation of any letter (upper or lower case, differences in font, etc.) occurring anywhere other than in a simple string or a character constant has no significance in that occurrence for the meaning of the program.

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

space
= SP

SP is the space (blank) character (ISO 646 code 2/0).

1.1 Directive lines

If the first character of a line is "%" (percent) the line as a whole is a directive line.

A directive line serves to communicate information to the processor and consequently its meaning is entirely implementation-dependent, with the following single exception. If the second character is a space, the line has no significance; it may be used for annotation purposes.

Note: The interpretation of a directive line takes precedence over the treatment of subsequent lines. The interpretation by the processor may cause inclusion of lines not present in the module, or deletion of some lines actually following the directive in question.

The language defined in the following defines the resulting program text after all directive lines have been interpreted and thereafter deleted.

1.2 The character set

The standard presupposes an 8-bit internal representation of characters. Thus the isocode facility allows for inclusion of characters with "isorank" value greater than 127 in simple strings and character constants. An implementation may restrict this possibility as well as the character set given below, as long as the "basic" characters of the table are included.

The standard character set is defined by the table below. For each character its "isorank" (see 9.2), name or printable representation and the classification of the character as a program text constituent are given.

basic:	Significant in all contexts.
skip:	Skipped in all contexts.
graphic:	Significant inside comments, inside simple strings, and inside character constants; illegal outside these constructs.
national:	Reserved for national alphabet extension; treated as "graphic".
format:	Format effector, see 1.9.

0	NUL	skip	32	SP	basic	64	@	national	96	`	national
1	SOH	illegal	33	!	basic	65	A	basic	97	a	basic
2	STX	illegal	34	"	basic	66	B	basic	98	b	basic
3	ETX	illegal	35	#	graphic	67	C	basic	99	c	basic
4	EOT	illegal	36	$	graphic	68	D	basic	100	d	basic
5	ENQ	illegal	37	%	graphic	69	E	basic	101	e	basic
6	ACK	illegal	38	&	basic	70	F	basic	102	f	basic
7	BEL	illegal	39	'	basic	71	G	basic	103	g	basic
8	BS	format	40	(	basic	72	H	basic	104	h	basic
9	HT	format	41	)	basic	73	I	basic	105	i	basic
10	LF	format	42	*	basic	74	J	basic	106	j	basic
11	VT	format	43	+	basic	75	K	basic	107	k	basic
12	FF	format	44	,	basic	76	L	basic	108	l	basic
13	CR	format	45	–	basic	77	M	basic	109	m	basic
14	SO	illegal	46	.	basic	78	N	basic	110	n	basic
15	SI	illegal	47	/	basic	79	O	basic	111	o	basic
16	DLE	illegal	48	0	basic	80	P	basic	112	p	basic
17	DC1	illegal	49	1	basic	81	Q	basic	113	q	basic
18	DC2	illegal	50	2	basic	82	R	basic	114	r	basic
19	DC3	illegal	51	3	basic	83	S	basic	115	s	basic
20	DC4	illegal	52	4	basic	84	T	basic	116	t	basic
21	NAK	illegal	53	5	basic	85	U	basic	117	u	basic
22	SYN	illegal	54	6	basic	86	V	basic	118	v	basic
23	ETB	illegal	55	7	basic	87	W	basic	119	w	basic
24	CAN	illegal	56	8	basic	88	X	basic	120	x	basic
25	EM	illegal	57	9	basic	89	Y	basic	121	y	basic
26	SUB	illegal	58	:	basic	90	Z	basic	122	z	basic
27	ESC	illegal	59	;	basic	91	[	national	123	{	national
28	FS	illegal	60	<	basic	92	\	national	124	|	national
29	GS	illegal	61	=	basic	93	]	national	125	}	national
30	RS	illegal	62	>	basic	94	^	national	126	~	national
31	US	illegal	63	?	graphic	95	_	basic	127	DEL	skip

Table 1.1. **Standard character set** (International Reference Version)

1.3 Special symbols

Symbol	Meaning
+ − * / // **	Arithmetic operators
&	Text concatenation operator, or exponent mark in **real** numbers
&&	Exponent mark in **long real** numbers
:= :-	Assignment operators
< <= = >= > <>	Value relational operators
== =/=	Reference relational operators
'	Character quote
"	String quote
!	Code quote, or comment
;	Statement separator, or declaration or specification delimiter
:	Array bound separator, or label definition or **virtual** delimiter
( )	Parameter or expression grouping, or array bounds delimiter
.	Remote indicator ("dot"), or decimal mark in real numbers
,	Parameter or expression separator

Table 1.2. **Special symbols, excluding keywords**

Normally the syntax of the language assumes that all syntactical units are recognized as being the longest possible string of characters which fits the syntax of a symbol. However, in an array declaration the symbol ":" is always a bound separator, even if it is immediately followed by a minus sign.

activate	**false**	**name**	**short**
after	**for**	**ne**	**step**
and		**new**	**switch**
array	**ge**	**none**	
at	**go**	**not**	**text**
	goto	**notext**	**then**
before	**gt**		**this**
begin		**or**	**to**
Boolean	**hidden**	**otherwise**	**true**
character	**if**	**prior**	**until**
class	**imp**	**procedure**	
comment	**in**	**protected**	**value**
	inner		**virtual**
delay	**inspect**	**qua**	
do	**integer**		**when**
	is	**reactivate**	**while**
else		**real**	
end	**label**	**ref**	
eq	**le**		
eqv	**long**		
external	**lt**		

Table 1.3. **SIMULA keywords**

1.4 Identifiers

identifier
= *letter* { *letter* | *digit* | _ }

No identifier can have the same spelling as any keyword. Apart from this, identifiers may be chosen freely. They have no inherent meaning, but serve for the identification of language quantities i.e. simple variables, arrays, texts, labels, switches, procedures, classes and class attributes. Within a procedure declaration identifiers also act as formal parameters, in which capacity they may represent a literal value or any language quantity except a class. All constituent characters are significant in distinguishing between identifiers.

1.5 Numbers

unsigned–number
= *decimal–number* [*exponent–part*]
| *exponent–part*

decimal–number
= *unsigned–integer* [*decimal–fraction*]
| *decimal–fraction*

decimal–fraction
= . *unsigned–integer*

exponent–part
= ⟨ & | && ⟩ [+ | –] *unsigned–integer*

unsigned–integer
= *digit* { *digit* | _ *digit* }
| *radix* R *radix–digit* { *radix–digit* | _ *radix–digit* }

radix
= 2 | 4 | 8 | 16

radix–digit
= *digit* | A | B | C | D | E | F

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

Unsigned integers are normally expressed in decimal digits. Unsigned integers of radix 2, 4, 8, or 16 may be expressed as shown. The radix digits "A" through "F" express radix 16 digits 10 through 15 (decimal). The radix determines the legality and the interpretation of a radix digit in an obvious manner.

An unsigned number which is an unsigned integer is of type **integer**. Otherwise, if an unsigned number contains an exponent part with a double ampersand ("&&") it is of type **long real**, else it is of type **real**.

Examples

```
2&1   2.0&+1   .2&2   20.0   200&–1          - represent same real value
2.345_678&&0                                 - long real value
```

1.6 Strings

string
= *simple–string* { *string–separator* *simple–string* }

string–separator
= *token–separator* { *token–separator* }

simple–string
= " { *isocode* | *non–quote–character* | "" } "

isocode
= ! *digit* [*digit*] [*digit*] !

non–quote–character
= ...

A non-quote-character is any printing character (incl. space) except the string quote ". Such a character represents itself.

A simple string must be contained within a single program line. Long strings are included as a sequence of simple strings separated by token separators.

In order to include a complete 8-bit coded character set, any character may be represented within a string by an integer, its isocode, corresponding to its bit combination. An isocode cannot consist of more than three digits, and it must be less than 256. If these conditions are not satisfied, the construction is interpreted as a character sequence. The string quote may, however, also be represented in simple strings by two consecutive quotes (see the last example below). Observe that, as a consequence of the definitional conventions given earlier in this chapter, no spaces may intervene between such a pair of string quotes.

Examples

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

1.7 Character constants

character–constant
= ' *character–designator* '

character–designator
= *isocode*
| *non–quote–character*
| ""

A character constant is either a single printing character or it is an isocode - in both cases surrounded by character quotes (' - ISO 646 code 2/7).

Within the data processing system, characters are represented by values according to some implementation-defined code. This code also defines the collating sequence used when comparing character (and text) values by means of relational operators.

1.8 Comment convention

For the purpose of annotating the program proper comments may be included in a program. The substitution of **end** for an end-comment, or a space for a direct comment does not alter the meaning of a program.

Note: As a consequence of 1.8.1 and 1.8.2 comments cannot be nested. It is understood that the comment structure encountered first in a program when reading from left to right has precedence in being replaced over later structures contained by the sequence.

1.8.1 End comment

The keyword **end** may be followed by any sequence of characters and separation of lines not containing any of the special symbols **end, else, when, otherwise,** or ";". This sequence (excluding the delimiting special symbol, but including the initial **end**) constitutes an end-comment.

1.8.2 Direct comment

The special symbol "!" (exclamation mark) followed by any sequence of characters or separation of lines not containing ";" (semicolon), and delimited by semicolon, is treated as a comment if the exclamation mark does not occur within a character constant or a simple string (in which cases it may either represent itself or act as a code quote), or within a comment.

Note: The delimiting semicolon is considered part of a direct comment and thus takes part in the substitution.

Example

```
if B then begin ... end  !then;  else ...
```

is not valid since the "!" is part of an end-comment. Thus ";" will act as a statement separator (and no statement can start with **else**).

1.9 Token separators

format–effector
= BS | HT | LF | VT | FF | CR

BS, HT, LF, VT, FF, and CR represent the characters thus named in table 1.1. A format effector in general acts as a space. In addition, an implementation may define some additional action to be taken (such as tabulation when listing the program); such action has no significance for the meaning of the program.

token–separator
= ...

A token-separator is

- a direct comment, or
- a space (except in simple strings and character constants), or
- a format effector (except as noted for spaces), or
- the separation of consecutive lines.

Zero or more token separators may occur between any two consecutive tokens, or before the first token of a program text. At least one token separator must occur between any pair of consecutive tokens made up of identifiers, keywords, simple strings or unsigned numbers. No token separators may occur within tokens.

1.10 Program interchange and lexical alternatives

In order to ease portability of SIMULA programs, a common representation has been adopted for the language. This representation is used throughout this standard except for the following conventions adopted for typographical reasons:

- In order to emphasize the use of the language keywords, these are printed in a special manner.
- Identifiers printed in upper case within this document represent quantities which are inaccessible to the user. Such identifiers are used for definitional purposes, they may or may not have actual counterparts in an implementation of the language.
- Program fragments may contain the ellipsis (...) instead of valid constructs, where it is either obvious from the context what the construct should be or the intended meaning cannot be expressed in a simple manner within the language.

1.10.1 Alternate representation of some symbols

The representation for lexical tokens and separators given in 1.2 to 1.9 constitutes a standard representation for these tokens and separators. This standard representation is recommended for program interchange.

For historical reasons the following alternatives have been defined. All processors that have the required characters in their character set must provide both the standard and the alternate representations, and there is no distinction made between corresponding tokens or separators.

The alternate representations for the tokens are

standard token	alternate representation
<	lt
<=	le
=	eq
>=	ge
>	gt
<>	ne
!	comment

2 TYPES, VALUES AND REFERENCES

type
= *value–type*
| *reference–type*

value–type
= *arithmetic–type*
| **Boolean**
| **character**

arithmetic–type
= *integer–type*
| *real–type*

integer–type
= [**short**] **integer**

real–type
= [**long**] **real**

reference–type
= *object–reference–type*
| **text**

object–reference–type
= **ref** (*qualification*)

qualification
= *class–identifier*

The various types basically denote properties of values. A "value" is a piece of information interpreted at run time to represent itself. A "reference" is a piece of information which identifies a value, called the "referenced" value. The distinction between a reference and its referenced value is determined by context.

A value is primarily a number, a logical value, a program point, an object, a single character or an ordered sequence of characters (a string).

The values of expressions and their constituents are defined in chapter 3.

Value types are characterized by being directly associated with a set of possible values (the "value range" of the type). With the exception of type **Boolean** these associated values for each value type constitute an ordered set.

The reference concept corresponds to the intuitive notion of a "name" or a "pointer". It provides a mechanism for referencing values. It also reflects the addressing possibilities of the machine. In certain simple cases a reference could be implemented as the memory address of a stored value. There are two reference types, object reference type and text reference.

Note: There is no reference concept associated with value types.

2.1 Arithmetic types

Arithmetic types are used for representing numerical values. The types are integer type and real type. The integer type is either **integer** or **short integer**. The real type is either **real** or **long real**.

2.1.1 The type short integer

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

An implementation may choose to implement **short integer** exactly as **integer**, i.e. ignoring the keyword **short**.

Note: All (integer) arithmetic operations upon integer type values are performed as **integer** operations.

2.1.2 The type long real

Type **long real** serves to represent real values capable of retaining a higher precision than that of the type **real**. The relative value range of the respective types is implementation-defined. Apart from this, **long real** and **real** are fully compatible in this language definition.

An implementation may choose to implement **long real** exactly as **real**, i.e. ignoring the keyword **long** and the extra "&" in an exponent part.

2.2 The type Boolean

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

2.3 The type character

The type **character** is used to represent single characters. Such a value is an instance of an "internal character". For any given implementation there is a one-to-one mapping between a subset of internal characters and external ("printable") characters. The internal character set is implementation-defined. The external character set is defined in 1.2.

2.4 Object reference

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

2.4.1 Qualification

The qualification of an object reference is a class identifier and is thus subject to the scope and visibility rules of identifiers given in 5.6.

The qualification is associated with an instance of the block in which the class declaration referred to is local. This implies that certain validity checks on the qualification cannot be performed on the basis of the program text alone. Such tests must then be made during the execution of the program.

Consider the following example.

Example

```
class a;  begin class b;  ;  end *** class a;

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

ref (a) a1;   ref (aa) a2;

a1:– a2:– new aa;
if inint=1 then a1:– new a;
inspect a2 do
inspect a1 do   aaxb:– new b;
```

The reference assignment in the last line is valid only if the qualification of "aaxb" is the same as that of "**new** b". This is the case only when the then-branch of the conditional statement is not taken, i.e. when a1 and a2 refer to the same object. Thus a qualification check must be performed during execution.

2.4.2 Subordinate types

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

A proper procedure is said to be of a universal type. Any type is subordinate to the universal type (cf. 4.6.1 and 5.5.3).

2.5 The type text

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

A text value is a string, i.e. an ordered sequence (possibly empty) of characters. The number of characters is called the "length" of the text value.

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

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

2.5.1 *Text objects*

A "text object" is conceptually an instance of the following class declaration (cf. 5.5):

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

2.5.2 *Text frames*

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 information:

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

2) the start position of the frame, being an ordinal number less than or equal to SIZE,

3) the length of the frame.

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

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

The frames of a text object are either all constant or all variable, as indicated by the attribute CONST. The value of this attribute remains fixed throughout the lifetime of the text object. A constant main frame always corresponds to a string (see 1.6).

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

Note: The identifier TEXTOBJ and its three attribute identifiers are not accessible to the user. Instead, properties of a text object are accessible through text variables, using the dot notation.

2.6 Type conversion

Values may in some cases be converted from one type to another.

Implicit conversion between arithmetic type values follows the rules described elsewhere (see 3.3.1, 3.5.1, 4.1.1). In addition, the procedure "entier", used to convert values of real type to **integer**, is described in 9.1.

Conversion between **text** and arithmetic type values is described in 8.6 and 8.7 (text attributes "getint", "putint", "getreal", "putreal", "putfix", "getfrac", "putfrac").

Conversion between **character** and **text** values is described in 8.2 (text attributes "getchar", "putchar").

Conversion between **character** and **integer** values is described in 9.2 ("isorank", "rank", "isochar", "char").

3 EXPRESSIONS

expression
= *value–expression*
| *reference–expression*
| *designational–expression*

value–expression
= *arithmetic–expression*
| *Boolean–expression*
| *character–expression*

reference–expression
= *object–expression*
| *text–expression*

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

A value expression is a rule for obtaining a value.

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

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

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

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

3.1 Variables

variable
= *simple–variable–1*
| *subscripted–variable*

simple–variable–1
= *identifier–1*

subscripted–variable
= *array–identifier–1* (*subscript–list*)

array–identifier–1
= *identifier–1*

subscript–list
= *subscript–expression* { , *subscript–expression* }

subscript–expression
= *arithmetic–expression*

A variable local to a block instance is a memory device whose "contents" are either a value or a reference, according to the type of the variable. The contents of a variable may be changed by an appropriate assignment operation, see 4.1.

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

A value type variable has a value which is the contents of the variable. A reference type variable is said to have a value which is the one referenced by the contents of the variable.

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

Examples

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

3.1.1 Simple variables

A simple-variable-1 is any variable which is not a subscripted variable. The corresponding values are described in chapter 2.

Note: Certain syntax classes (such as simple-variable-1) are marked with a "–1". The corresponding program terms may contain a remote identifier (see 5.5.6).

Value type variables have values of integer type, real type, **Boolean** or **character**.

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

3.1.2 Text variables

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

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

Let X be a text variable. Then X.OBJ, X.START, X.LENGTH and X.POS denote the components of X, respectively. These four components are not directly accessible to the user. Instead, certain properties of a text variable are represented by procedures accessible through the dot notation. These procedures are described in chapter 8.

The components OBJ, START and LENGTH constitute the text reference part of the variable. They identify the frame referenced (see 2.5.1). POS is used for accessing the individual characters of the frame referenced (see 9.2).

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.**

3.1.3 Array elements

Subscripted variables designate values which are components of multi-dimensional arrays. Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable and is called a subscript. The complete list of subscripts is enclosed by the subscript parentheses ( ) . The array component referred to by a subscripted variable is specified by the actual value of its subscripts. A subscript expression value outside its associated bounds causes a run time error.

Each subscript position acts like a variable of type **integer** and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable.

3.1.4 Function designators

function–designator
= *procedure–identifier–1* [*actual–parameter–part*]

procedure–identifier–1
= *identifier–1*

actual–parameter–part
= (*actual–parameter* { , *actual–parameter* })

actual–parameter
= *expression*
| *array–identifier–1*
| *switch–identifier*
| *procedure–identifier–1*

A function designator defines a value which results through the application of a given set of rules defined by a procedure declaration (see 5.4) to a fixed set of actual parameters. The rules governing specification of actual parameters are given in 4.6.

Note: Not every procedure declaration defines rules for determining the value of a function designator (cf. 5.4.1).

Examples

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

3.1.5 *Remote identifiers*

identifier–1
= *identifier*
| *remote–identifier*

remote–identifier
= *simple–object–expression* . attribute–*identifier*
| *text–primary* . attribute–*identifier*

attribute–*identifier*
= *identifier*

Let X be a simple object expression qualified by the class C, and let A be an appropriate attribute identifier. Then the remote identifier "X.A", if valid, is an attribute identification whose object is the value X and whose qualification is C (cf. 5.5.6).

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

1) The value X is different from none.

2) If the type of A is ref(D), then the qualifying class D must not be declared at any prefix level equal or outer to that of C.

Note: Condition 1 corresponds to a check which causes an error if the value of X is none. Condition 2 is an ad hoc rule intended to simplify the language and its implementations.

A remote identifier of the form

text–primary.attribute–identifier

identifies an attribute of the text variable identified by evaluating the text primary, provided that the attribute identifier is one of the procedure identifiers defined in chapter 8.

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

Example

Let P1 and P2 be variables declared and initialized as in the example in 4.1.4. 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.

3.2 Boolean expressions

Boolean–expression
= *simple–Boolean–expression*
| *if–clause simple–Boolean–expression* **else** *Boolean–expression*

simple–Boolean–expression
= *Boolean–tertiary* { **or else** *Boolean–tertiary* }

Boolean–tertiary
= *equivalence* { **and then** *equivalence* }

equivalence
= *implication* { **eqv** *implication* }

implication
= *Boolean–term* { **imp** *Boolean–term* }

Boolean–term
= *Boolean–factor* { **or** *Boolean–factor* }

Boolean–factor
= *Boolean–secondary* { **and** *Boolean–secondary* }

Boolean–secondary
= [**not**] *Boolean–primary*

Boolean–primary
= *logical–value*
| *variable*
| *function–designator*
| *relation*
| (*Boolean–expression*)

A Boolean expression is of type **Boolean**. It is a rule for computing a logical value. Except for the operators **and then** and **or else** (see 3.4) the semantics are entirely analogous to those given for arithmetic expressions.

Variables and function designators entered as Boolean primaries must be of type **Boolean**.

Examples

```
x = –2
Y>v or z<q
a+b> –5 and z–d>q**2
p and not q or x<>y
t.more and then t.getchar
x == none or else x.a>0
if k<1 then s>w else h<=c
```

3.3 Relations

relation
= *arithmetic–relation*
| *character–relation*
| *text–value–relation*
| *object–relation*
| *object–reference–relation*
| *text–reference–relation*

value–relational–operator
= < | <= | = | >= | > | <>

reference–comparator
= == | =/=

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

3.3.1 *Arithmetic relations*

arithmetic–relation
= *simple–arithmetic–expression* *value–relational–operator* *simple–arithmetic–expression*

The relational operators <, <=, =, >=, > and <> have their conventional meaning (less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to). Arithmetic relations assume the value **true** whenever the corresponding relation is satisfied for the expressions involved, otherwise **false**. If the two constituent expressions are of different arithmetic types conversion to the type with maximum value range is assumed. Consequently overflow cannot occur during the evaluation.

3.3.2 *Character relations*

character–relation
= *simple–character–expression* *value–relational–operator* *simple–character–expression*

Character values may be compared for equality and inequality and ranked with respect to the collating sequence. Let X and Y be simple character expressions, and let rel be any value relational operator. Then the relation "X rel Y" has the same Boolean value as the relation "rank(X) rel rank(Y)".

Note: Because of variations in collating sequences the value of a character relation (and by implication that of a text relation, see below) is implementation-defined. Implementation-independent comparison of character values is obtained by using the procedure "isorank".

3.3.3 Text value relations

text–value–relation
= *simple–text–expression value–relational–operator simple–text–expression*

Two text values are equal if they are both empty, or if they are both instances of the same character sequence, otherwise they are unequal.

A text value T ranks lower than a text value U if and only if they are unequal and one of the following conditions is fulfilled:

1) T is empty.

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

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

3.3.4 Object relations

object–relation
= *simple–object–expression* **is** *class–identifier*
| *simple–object–expression* **in** *class–identifier*

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

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

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

3.3.5 Object reference relations

object–reference–relation
= *simple–object–expression reference–comparator simple–object–expression*

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

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

3.3.6 Text reference relations

text–reference–relation
= *simple–text–expression reference–comparator simple–text–expression*

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

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

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

The following relations are all true (cf. 2.5)

T = **notext** **eqv** T == **notext**
"" == **notext**
"ABC" =/= "ABC" (different occurrences, see 3.7)

Example

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

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

3.4 The logical operators

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

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

The operation "b1 **and then** b2" denotes "conditional and". If the value of b1 is **false** the operation yields the result **false**, otherwise it yields the result of evaluating b2.

The operation "b1 **or else** b2" denotes "conditional or". If the value of b1 is **true** the operator yields the result **true**, otherwise it yields the result of evaluating b2.

Note: The value of "b1 **and then** b2" is given by textual substitution of the Boolean expression "(**if** b1 **then** b2 **else false**)". Similarly, the operation "b1 **or else** b2" is defined by substitution of "(**if** b1 **then true else** b2)". These definitions imply that the evaluation of the second operand is suppressed when the evaluation result is already evident from the value of the first operand alone.

3.4.1 Precedence of Boolean operators

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

According to the syntax given in 3.2 the following rules of precedence hold:

first:	non-Boolean expressions
second:	**< <= = >= > <> == =/= is in**
third:	**not**
fourth:	**and**
fifth:	**or**
sixth:	**imp**
seventh:	**eqv**
eighth:	**and then**
ninth:	**or else**

The use of parentheses is interpreted in the sense given in 3.5.2.

3.5 Arithmetic expressions

arithmetic–expression
= *simple–arithmetic–expression*
| *if–clause simple–arithmetic–expression*
else *arithmetic–expression*

simple–arithmetic–expression
= [+ | −] *term* { ⟨ + | − ⟩ *term* }

term
= *factor* { ⟨ * | / | // ⟩ *factor* }

factor
= *primary* { ** *primary* }

primary
= *unsigned–number*
| *variable*
| *function–designator*
| (*arithmetic–expression*)

An arithmetic expression is a rule for computing a numerical value. In the case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in 3.5.1 below. The value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed by parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if-clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (see 3.2). This selection is made as follows: The Boolean expressions of the if-clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the longest arithmetic expression found in this position is understood). If none of the Boolean expressions has the value **true**, then the value of the arithmetic expression is the value of the expression following the final **else**.

In evaluating an arithmetic expression, all primaries within that expression are evaluated with the following exceptions:

- primaries that occur within any expression governed by an if-clause but not selected by it.
- primaries that occur within a Boolean expression
  1) after the operator **or else** when the evaluation of a preceding Boolean tertiary results in **true**, or
  2) after the operator **and then** when the evaluation of a preceding Boolean equivalence results in **false**.
- primaries that occur after a function designator, and the evaluation of the function terminates with a goto-statement. In this case the evaluation of the arithmetic expression is abandoned.

Primaries are always evaluated in strict lexical order.

Examples

Primaries:
```
7.394_604&-8
sum
w(i+2, 8)
cos( y + z*3.141_592_653_589_793_238&&0 )
( a -3/y + vu**8)
```

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

Terms:
```
u
omega * sum ** cos(y + z*3)/7.394&-8 ** (a -3/y + vu**8)
```

Simple arithmetic expression:
```
u -yu + omega*sum**cos(y+z*3)/7.394&-8**(a-3/y+vu**8)
```

Arithmetic expressions:
```
w*u -q(s+cu)**2

if q>0 then s+3*q//a else 2*s+3*q

if a<0 then u+v else if a*b>17 then u/v

    else if k >= y then v/u else 0

0.57&12 * a( n*(-1)//2, 0 )

( a*arctan(y)+z ) ** (7+Q)

if q then -1 else n
```

3.5.1 Operators and types

Apart from the Boolean expressions of if-clauses, the constituents of arithmetic expressions must be of arithmetic types. The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules, where "i" and "j" are of integer type, "r" of real type, and "x" is any arithmetic type:

– The operators "+", "–", and "*" have their conventional meaning.

– The operator "/" denotes real division. Any operand of integer type is converted before the operation. Division by zero constitutes an error.

– The operator "//" denotes integer division. It is valid only for integer type operands. The meaning of "i//j" is defined by:

```
integer procedure DIV(i, j); integer i, j;
if j=0 then error("..." !div by zero;)
else begin integer m, n;
  m:= 0;  n:= abs(i);
  for n:= n – abs(j) while n>=0 do m:= m + 1;
  DIV:= if i<0 eqv j>0 then –m else m
end DIV;
```

– The operator "**" denotes exponentiation. The value and type of the operation depends upon the types of the operands as follows:

```
x**r:  <type of r> procedure EXPR(x, r); <arithmetic type> x; <real type> r;
       if  x<0 or (x=0 and r<=0.0)
       then error("..." !EXPR undefined;);
       else EXPR := if x>0 then exp(r*ln(x)) else 0.0;

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

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

It is understood that the finite deviations of using the exponentiation operator may be different from those of using the procedures EXPR and EXPN.

If the operands of an arithmetic operator are of different types, or both **short integer**, an appropriate type conversion function is understood to be automatically invoked, except as explicitly noted above, before the operation is evaluated as follows:

> If one operand is of type **long real** the other is converted to **long real**, else if one operand is of type **real** the other is converted to **real**, else **short integer** operands are converted to **integer**.

Note: The result of evaluating an arithmetic expression can never be of type short integer.

Conversion from **short integer** to **integer** is always exact. Conversion from an integer type to a real type is exact within an implementation-defined range which includes zero. Conversion from **real** to **long real** is exact within an implementation-defined range which includes zero.

The type of the operation (and by repeated application also the type of the arithmetic expression) is a consequence of the type conversion rule as follows:

	SI	I	R	LR		
SI	I	I	R	LR	SI:	**short integer**
I	I	I	R	LR	I:	**integer**
R	R	R	R	LR	R:	**real**
LR	LR	LR	LR	LR	LR:	**long real**

The rule also determines the type of a conditional expression, i.e. an arithmetic expression of the form "**if** B **then** SAE **else** AE". The expression is of type **long real** if either SAE or AE is **long real**. Otherwise, if either SAE or AE is of type **real**, the type of the expression is **real**, else the type is **integer**.

Note: The type of a conditional expression is independent of the actual value of the Boolean expression, i.e. it is completely determined by the program text.

3.5.2 Precedence of arithmetic operators

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

In accordance with the syntax given in 3.3.1 the following rules of precedence hold:

first:	**
second:	* / //
third:	+ −

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

Note: The order of evaluation of the primaries is not influenced by the use of parentheses.

3.5.3 *Arithmetics of real quantities*

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

3.6 Character expressions

character–expression
= *simple–character–expression*
| *if–clause simple–character–expression*
else *character–expression*

simple–character–expression
= *character–constant*
| *variable*
| *function–designator*
| (*character–expression*)

A character expression is of type **character**. It is a rule for obtaining a character value (see 2.1.3). Apart from possible if-clauses, all constituents of a character expression must be of type **character**.

3.7 Text expressions

text–expression
= *simple–text–expression*
| *if–clause simple–text–expression* **else** *text–expression*

simple–text–expression
= *text–primary* { & *text–primary* }

text–primary
= **notext**
| *string*
| *variable*
| *function–designator*
| (*text–expression*)

A text expression is of type **text**. It is a rule for obtaining an identification of a text variable. Apart from possible if-clauses, all constituents of a text expression must be of type **text**.

Each textual occurrence of a non-empty string corresponds to a unique constant main text frame. A given occurrence always references that same frame, while different occurrences of the same non-empty string always reference different text frames.

The empty string ("") is textually equivalent to **notext**.

3.7.1 Text concatenation

The operator "&" permits text concatenation. The simple text expression "TP1 & TP2 & ... & TPn", where TPi is a text primary (1<=i<=n), references a new alterable main frame whose contents are formed by concatenating copies of the frames referenced by TP1, TP2, ... , TPn (in that order). The expression is equivalent to CONCATENATE_n(TP1, TP2, ... , TPn) defined by

```
text procedure CONCATENATE_n(T1, T2, ... , Tn); text T1, T2, ... , Tn;
begin text temp;
    CONCATENATE_n:– temp:– blanks(T1.length+T2.length+ ... +Tn.length);
    temp.sub(1, T1.length):= T1;
    temp.sub(1+T1.length, T2.length):= T2;
    ...
    temp.sub(1+T1.length+T2.length+ ... , Tn.length):= Tn;
end CONCATENATE_n;
```

Note: It follows that the text primary constituents of a simple text expression are evaluated in strict lexical order. The evaluation of Ti may influence the result of evaluating Tj, if i<j (due to the specified "by reference" transmission of parameters to the procedure CONCATENATE_n).

Observe further that it follows from the syntax (cf. 3.1.5) that "." is evaluated before "&", thus the two expressions "T1 & T2.sub(1, 2) & T3.main" and "T1 & (T2.sub(1, 2)) & (T3.main)" are equivalent.

3.7.2 Text expression evaluation

The result of evaluating:

- **notext**, or an empty string, identifies an anonymous text variable whose contents are defined by (2) of 3.1.2.
- 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 by parentheses identifies an anonymous text variable which contains a copy of (the contents of) the text variable identified when evaluating the same expression without parentheses.
- a conditional text expression identifies an anonymous text variable which contains a copy of (the contents of) the text variable identified by the branch which was selected for evaluation.

For further information on the text concept, see chapter 8.

3.8 Object expressions

object–expression
= *simple–object–expression*
| *if–clause simple–object–expression* **else** *object–expression*

simple–object–expression
= **none**
| *variable*
| *function–designator*
| *object–generator*
| *local–object*
| *qualified–object*
| (*object–expression*)

object–generator
= **new** *class–identifier* [actual–parameter–part]

local–object
= **this** *class–identifier*

qualified–object
= *simple–object–expression* **qua** *class–identifier*

An object expression is of type **ref**(qualification). It is a rule for obtaining a reference to an object. The value of the expression is the referenced object or **none**. Apart from a possible if-clause all constituents must be of object reference type.

3.8.1 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 5.5.5). 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.

3.8.2 Object generator

The value of an object generator is the object generated as the result of its evaluation. See 4.7.

3.8.3 Local objects

A local object "**this** C" is valid provided that the expression is used 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 4.8).

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 valid. If there is no such block the local object is illegal (in the given context). For an instance of a procedure or a class body, "textually enclosing" means containing its declaration.

3.8.4 *Instantaneous qualification*

Let X represent any simple reference expression, and let C and D be class identifiers such that D is the qualification of X. The qualified object "X **qua** C" is then a legal object expression, provided that C is outer to or equal to D or is a subclass of D. Otherwise, since C and D belong to disjoint prefix sequences, the expression 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. Instantaneous qualification restricts or extends the visibility of attributes of a concatenated class object accessible through inspection or remote accessing (cf. 3.1.5 and 4.8).

3.9 Designational expressions

designational–expression
= *simple–designational–expression*
| *if–clause simple–designational–expression*
else *designational–expression*

simple–designational–expression
= *label*
| *switch–designator*
| (*designational–expression*)

switch–designator
= *switch–identifier* (*subscript–expression*)

switch–identifier
= *identifier*

label
= *identifier*

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

The evaluation of the subscript expression is analogous to that of subscripted variables. The value of a switch designator is defined only if the subscript expression assumes one of the values 1, 2, ... , n, where n is the number of entries in the switch list. A value outside this range causes a run time error.

Note: It is a consequence of the syntax that class attributes which are labels or switches cannot be accessed by the dot notation.

4 STATEMENTS

statement
= { *label* : } *unconditional–statement*
| { *label* : } *conditional–statement*
| { *label* : } *for–statement*

unconditional–statement
= *assignment–statement*
| *while–statement*
| *goto–statement*
| *procedure–statement*
| *object–generator*
| *connection–statement*
| *compound–statement*
| *block*
| *dummy–statement*
| *activation–statement*

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

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

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in chapter 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1 Assignment statements

assignment–statement
= *value–assignment*
| *reference–assignment*

value–assignment
= *value–left–part* := *value–right–part*

value–left–part
= *destination*
| *simple–text–expression*

value–right–part
= *value–expression*
| *text–expression*
| *value–assignment*

destination
= *variable*
| *procedure–identifier*

reference–assignment
= *reference–left–part* :– *reference–right–part*

reference–left–part
= *destination*

reference–right–part
= *reference–expression*
| *reference–assignment*

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

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

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

A procedure identifier in this context designates a memory device local to the procedure instance. This memory device is initialized upon procedure entry following the rules given in 5.7. The type associated with a procedure identifier is given by the type declarator which appears as the first symbol of the corresponding procedure declaration.

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

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

a) Any expression which is, or is part of, the left part of an assignment is evalated prior to the evaluation of the right part. Within a particular left part, constituent expressions such as subscript expressions are evaluated in sequence from left to right.

b) The expression of the ultimate right part is evaluated.

c) The value or reference of this expression is assigned to its immediately preceding left part. If the left part is itself part of an assignment, the resulting value or reference is assigned to its immediately preceding left part. This process is repeated until the left part list is exhausted. All assignments are performed with any left part expression having values as evaluated in step a.

Note: It is not required that, in multiple assignments, all left parts are of the same type, as long as the appropriate type conversion functions are defined.

If the destination is of type **Boolean** or **character**, the value right part must likewise be of type **Boolean** or **character**, respectively.

For the description of the text value assignment, see 4.1.2. There is no value assignment operation for objects.

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

If a destination is a formal parameter called by name, 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.

4.1.1 Arithmetic assignment

If the type of the arithmetic expression differs from that associated with the destinations, an appropriate conversion function is understood to be automatically invoked. For transfer from real type to integer type the conversion function is understood to yield a result which is the largest integral quantity not exceeding E + 0.5 in the mathematical sense (i.e. without rounding error) where E is the value of the expression. Conversion from **integer** to **short integer** is exact within the value range of **short integer**; outside it constitutes a run-time error. Conversion from **long real** to **real** is performed with correct rounding. If the value range of **long real** exceeds that of **real** a run-time error may result. Conversions not mentioned above are performed according to the rules given in 3.5.1.

Note: The value of a real type expression is defined with only finite accuracy.

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

4.1.2 Text value assignment

Let X be the text variable identified as the result of evaluating the left part (see 3.7) of a text value assignment, and let Y denote the text variable identified by evaluating the corresponding right part. If X references a constant text frame, or X.LENGTH < Y.LENGTH, then the assignment constitutes an error. Otherwise, the value of Y is conceptually extended to the right by X.LENGTH - Y.LENGTH

blank characters, and the resulting text value is assigned as the new contents of the text frame referenced by X.

Note: If X==notext, the assignment is legal if and only if Y==notext.

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

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

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

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

4.1.3 Text reference assignment

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

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

4.1.4 Object reference assignment

Let the left part of an object reference assignment be qualified by the class C_l, and let the right part be qualified by C_r. If the right part is itself a reference assignment, C_r 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 C_r, C_l and V.

Case 1. C_l is the class C_r or outer to C_r:

The reference assignment is legal and the assignment operation is carried out.

Case 2. C_l is inner to C_r:

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

Case 3. C_l and C_r satisfy neither of the above relations:

The reference assignment is illegal.

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

Example

Let "point" and "polar" be the classes declared in the example of 5.5.2.

```
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.

4.2 Conditional statement

conditional–statement
= *if–clause* { *label* : } *unconditional–statement* [**else** *statement*]
| *if–clause* { *label* : } *for–statement*

if–clause
= **if** *Boolean–expression* **then**

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

Three forms of the conditional statement exist. Let B be a Boolean expression, Su an unconditional statement, Sfor a for-statement and S a statement. Then, in execution of a statement of the form "**if** B **then** Su", B is evaluated. If the result is true, Su is executed. If the result is false, Su is not executed. If Su is labelled, and a goto-statement leads to the label, then B is not evaluated, and the computation continues with execution of the labelled statement.

The two remaining forms are explained in terms of the above, as follows.

The conditional statement:	is equivalent to the execution of:
if B **then** Sfor	**if** B **then begin** Sfor **end**
if B **then** Su **else** S	**if** B **then begin** Su; **goto** GAMMA **end**; S; GAMMA:

If Su, Sfor or S are labelled they are labelled in the equivalent construct.

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

Examples

```
if x>0 then n:=n+1

if false then abort: terminate_program
else if s<0 or p<q then
else if v>s then  a := v−q
else if v<s−1 then  goto abort
```

4.3 While-statement

while–statement
= **while** *Boolean–expression* **do** *statement*

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

The evaluation of "**while** BE **do** S" is equivalent to

ALPHA: **if** BE **then begin** S; **goto** ALPHA **end**

4.4 For-statement

for–statement
= *for–clause statement*

for–clause
= **for** *simple–variable for–right–part* **do**

simple–variable
= *identifier*

for–right–part
= := *value–for–list–element* { , *value–for–list–element* }
| :– *reference–for–list–element* { , *reference–for–list–element* }

value–for–list–element
= *value–expression* [**while** *Boolean–expression*]
| *text–expression*
| *arithmetic–expression*
step *arithmetic–expression* **until** *arithmetic–expression*

reference–for–list–element
= *reference–expression* [**while** *Boolean–expression*]

The simple variable of the for-clause is called "the controlled variable". The statement following is called "the controlled statement".

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

Assignments may change the value of the controlled variable during execution of the controlled statement. Upon exit from the for-statement, the controlled variable has the value given to it by the last (explicit or implicit) assignment operation.

4.4.1 For list elements

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

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

C: controlled variable
V: value expression
R: reference expression
A: arithmetic expression
B: Boolean expression
S: controlled statement

The effect of the occurrence of expressions as for list elements may be established by textual replacement in the definitions. ALPHA and DELTA are considered local to the fictitious block (4.4.3). DELTA is of the same type as A2.

1. V (value expression)

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

2. A1 **step** A2 **until** A3

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

3. V **while** B

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

4. R (reference expression)

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

5. R **while** B

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

4.4.2 The controlled variable

The controlled variable cannot be 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, see 4.4.1) must be semantically and syntactically valid. In particular each implied reference assignment in cases 4 and 5 of 4.4.1 is subject to the rules of 4.1.4 and 4.1.3, and each text value assignment in cases 1 and 3 of 4.4.1 is subject to the rules of 4.1.2.

4.4.3 The controlled statement

The controlled statement always acts as if it were a block, whether it takes this form or not. Hence, labels on or defined within the controlled statement are invisible outside that statement.

4.5 Goto-statement

goto–statement
= ⟨ goto | go to ⟩ *designational–expression*

A goto-statement interrupts the normal sequence of operations, by defining its successor explicitly by the value of a designational expression (i.e. a program point). Thus the next statement to be executed is the one at this program point.

The program point referenced by the designational expression must be visible at the goto-statement (cf. 5.6.2). See also 7.3.5.

Examples

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

4.6 Procedure statement

procedure–statement
= *procedure–identifier–1* [*actual–parameter–part*]

A procedure statement interrupts the normal sequence of operations by invoking (calling for) the execution of a procedure body. Conceptually this may be described in the following terms.

If the procedure has parameters an additional block embracing the procedure body block (cf. 5.4) is created. All formal parameters correspond to variables declared in this (fictitious) block with types as given in the corresponding specifications. Thus formal parameters are non-local to the procedure body, but local to this block. The evaluation of the procedure statement now proceeds as described in 4.6.5.

4.6.1 Actual-formal parameter correspondence

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

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

– exact type correspondence is required for parameters of type **text**, **character** or **Boolean** as well as for all **array** parameters, irrespective of transmission mode

– an actual parameter corresponding to a formal parameter of arithmetic type which is not an array or procedure can have any arithmetic type. The conversion follows the assignment statement rules for parameters called by value and by reference and 5.4.3 for parameters called by name

– a type procedure can correspond to a formal parameter specified as a proper procedure

– the type of an actual parameter corresponding to a formal parameter of object reference type must coincide with or be subordinate to the formal type (see 2.4.2).

4.6.2 Value parameter replacement (call by value)

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

All formal parameters quoted in the value part of the procedure heading as well as value type parameters not quoted in the name part are assigned the values of the corresponding actual parameters, these assignments being considered as performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications. As a consequence, variables called by value are to be considered as non-local to the body of the procedure, but local to the fictitious block (cf. 4.6.5).

Note: Parameters transmitted by value are evaluated once only, before entry of the procedure body.

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 4.1.3, and "copy" in 8.3).

Value specification is redundant for a parameter of value type.

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

4.6.3 Default parameter replacement (call by reference)

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

A formal parameter called by reference is initially assigned a local copy of the reference obtained by evaluating the corresponding actual parameter. Such assignments are entirely analogous to those described under call by value.

Note: Parameters transmitted by reference are evaluated once only, before entry of the procedure body.

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 4.1.3 and 4.1.4. Since in this case the formal parameter is a reference type variable, its contents may be changed by reference assignments within the procedure body.

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

An array, procedure, label or switch parameter called by reference cannot be changed from within the procedure or class body. It thus references 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 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.

4.6.4 Name parameter replacement (call by name)

Call by name is an optional transmission mode available only for parameters to procedures.

Each occurrence of a formal parameter called by name within the procedure body invokes an evaluation of the actual parameter. This evaluation takes place in the context of the procedure statement, i.e. no identifier conflicts can occur (since the procedure body and its variables are invisible).

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

For an expression within a procedure body which is

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 designator whose procedure identifier is a formal parameter called by name,

the following rules apply:

1) Its type is that prescribed by the corresponding formal specification.

2) If the type of the actual parameter does not coincide with that of the formal specification, then an evaluation of the expression is followed by an assignment of the value or reference obtained to a fictitious variable of the latter type. This assignment is subject to the rules of 4.1. 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 constant (i.e. CONST is **true**), then all occurrences of the formal parameter evaluate to the same text frame (see 3.7).

Section 4.1 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.

Note: 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, since the actual parameter is evaluated at each occurrence.

4.6.5 Body execution

The execution of a procedure call proceeds in the following steps, where 1 and 2 are performed only if the procedure has parameters.

1) The formal parameter block instance is created.

2) Actual parameters corresponding to call by value or call by reference are evaluated as described above and the results are assigned to the corresponding variables of the formal block instance, following the rules in 4.1. Actual parameters corresponding to call by name are treated as described in 4.6.4.

3) The procedure body is instantiated and starts executing.

The execution of the final statement of the procedure body, unless this statement is itself a goto-statement, concludes with the execution of an implicit goto-statement to the program point immediately following the procedure statement.

4.6.6 Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.6.5 lead to a correct statement. This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. The following are some important particular cases of this general rule, and some additional restrictions.

A formal parameter which occurs as a destination within the procedure body and which is called by name can only correspond to an actual parameter which is an identification of a variable (special case of expression).

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

Note: The rules stated above are applicable only where formal arrays or procedure calls are actually evaluated during the execution of the procedure body.

4.7 Object generator statement

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

1) The object is generated and the actual parameters, if any, of the object generator are evaluated. The parameter values and/or references are transmitted.

2) Control enters the object through its initial **begin** whereby it becomes operating in the "attached" state (see chapter 7). The evaluation of the object generator is completed:

   case a: whenever the basic attribute procedure "detach" of the generated object is executed (see 7.1), or

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

The state of the object after the evaluation is either "detached" (case a) or "terminated" (case b). Cf. chapter 7.

4.7.1 Parameter replacement

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

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

For further information on the parameter transmission modes, see 5.5.5.

4.8 Connection statement

connection–statement
= **inspect** *object–expression when–clause* { *when–clause* }
[*otherwise–clause*]
| **inspect** *object–expression* **do** *connection–block–2*
[*otherwise–clause*]

when–clause
= **when** *class–identifier* **do** *connection–block–1*

otherwise–clause
= **otherwise** *statement*

connection–block–1
= *statement*

connection–block–2
= *statement*

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

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

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

The purpose of the connection mechanism is to provide implicit definitions to items 1 and 2 in 5.5.6 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 from left to right. 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, unless 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 inner to 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 by a second fictitious block, called a "connection block". During the execution of a connection block the object X is said to be "connected". A connection block has an associated "block qualification", which is the preceding class identifier for a connection-block-1 and the qualification of the preceding object expression for a connection-block-2.

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

Example

Let "Polar" be the class declared in the example of 5.5.2. Then within the connection-block-2 of the connection statement

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

a procedure "plus" is available for vector addition.

4.9 Compound statement

compound–statement
= **begin** *compound–tail*

compound–tail
= *statement* { ; *statement* } **end**

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

L: L: ... **begin** S; S; ... S; S **end**

Note: Each of the statements S may be a complete compound statement or block.

Example

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

4.10 Blocks

block
= *subblock*
| *prefixed–block*

subblock
= *block–head* ; *compound–tail*

block–head
= **begin** *declaration* { ; *declaration* }

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

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

Note: Each of the statements S may be a complete compound statement or block.

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

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

b) any entity represented by this identifier outside the block is invisible inside the block, unless made visible by connection or remote access.

Identifiers (except those representing labels) occurring within a block and not being declared within this block will be non-local to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement behaves as though declared in the head of the smallest embracing block.

A label is said to be implicitly declared in this block head. In this context, a procedure body, the statement following a for-clause, or a connection block must be considered as if it were enclosed by **begin** and **end** and treated as a block, this block being nested within the fictitious block of 4.6.1 in the case of a procedure with parameters. A label that is not within any block of the program (nor within a procedure body, the statement following a for-clause, or a connection block) is implicitly declared in the implied connection block embracing the program.

Note: System-defined class identifiers used as prefixes within the block as well as identifiers introduced as part of an external head are in this respect treated in the same manner as labels.

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

See also 5.6.

Example

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

4.10.1 Prefixed blocks

prefixed–block
= *block–prefix main–block*

block–prefix
= *class–identifier* [*actual–parameter–part*]

main–block
= *block*
| *compound–statement*

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 5.5.2.

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 (for system-defined class identifiers, see (2) of 5.5.1).

A program is enclosed by a prefixed block (cf. chapter 10).

Example

Let "hashing" be the class declared in the example of 5.5.3. Then within the prefixed block

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

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

4.11 Dummy statement

dummy–statement
= *empty*

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

Example

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

5 DECLARATIONS

declaration
= *simple–variable–declaration*
| *array–declaration*
| *switch–declaration*
| *procedure–declaration*
| *class–declaration*
| *external–declaration*

For external declarations, see chapter 6.

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid within a certain region of the program called its "scope". Outside the scope the particular identifier may be used for other purposes.

Within its scope the identifier declaration may be either "visible" or "invisible". If visible an occurrence of the identifier references this declaration. A declaration is invisible (within its scope) either because the associated identifier has been redefined or because its visibility has been explicitly restricted (see 5.5.4) or because connection or remote access is necessary to make it visible.

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

Apart from labels (see 4.10), formal parameters of procedure and class declarations, possible identifiers introduced implicitly by external declarations, and identifiers declared in the program prefix, each identifier appearing in a program must be explicitly defined within the program.

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

More precise scope and visiblity rules are given in 5.6.

5.1 Simple variable declaration

simple–variable–declaration
= *type type–list*

type–list
= *type–list–element* { , *type–list–element* }

type–list–element
= *identifier*
| *constant–element*

Type declarations serve to declare certain identifiers to represent simple variables of a given type (see 3.1.1 and 3.1.2).

For constant element, see 5.8.

5.1.1 Value type variables

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

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

Boolean variables may assume the values **true** and **false**.

Character variables may assume the implementation-defined character values of the internal character set (cf. 1.2).

5.2 Array declaration

array–declaration
= [*type*] **array** *array–segment* { , *array–segment* }

array–segment
= *array–identifier* { , *array–identifier* }
(*bound–pair–list*)

array–identifier
= *identifier*

bound–pair–list
= *bound–pair* { , *bound–pair* }

bound–pair
= *arithmetic–expression* : *arithmetic–expression*

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

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

Note: An initial "–" in an upper bound may follow ":" directly (cf. 1.3).

The dimension is given as the number of entries in the bound pair list.

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

The expressions are evaluated in the same way as subscript expressions. This evaluation takes place once at each entrance into the block through the block head. The expressions cannot include any identifier that is declared, either explicitly or implicitly, in the same block head as the array in question, unless the identifier is declared by a constant element. An array has elements only when the values of all upper bounds are not smaller than those of the corresponding lower bounds. If any lower bound value is greater than the corresponding upper bound value, the array has no elements. An attempt to access an element of an empty array leads to a run-time error. The array may, however, be created at block entry and it may be passed as a parameter.

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

Examples

```
integer array a(2:20)    ! 19 elements;
real array  q(–7:if c<0 then 2 else 1)    ! 10 or 9 elements;
array  a, b, c(7:n, 2:m), s(–2:10)    ! any value of n or m legal;
```

5.3 Switch declaration

switch–declaration
= **switch** *switch–identifier* := *switch–list*

switch–list
= *designational–expression* { , *designational–expression* }

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

An expression in the switch list is evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved. This evaluation takes place in the contex of the switch declaration.

Examples

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

5.4 Procedure declaration

procedure–declaration
= [*type*] **procedure** *procedure–heading* ; *procedure–body*

procedure–heading
= *procedure–identifier*
[*formal–parameter–part* ; [*mode–part*] *specification–part*]

procedure–body
= *statement*

procedure–identifier
= *identifier*

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement, the procedure body, which through the use of procedure statements and/or function designators may be activated within those parts of the program in which the procedure declaration is visible.

A procedure without type is called a proper procedure.

Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body are, whenever the procedure is activated, assigned values or references from the actual parameters following the rules of 4.6. Identifiers in the procedure body which are not formal are either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears.

The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body.

In addition, a procedure with parameters in certain respects acts as if a fictitious block embraced the procedure body. The formal parameters then correspond to variables declared local to this fictitious block. Thus, if the identifier of a formal parameter is declared anew within the procedure body, it is thereby given a local significance and actual parameters which correspond to it are invisible throughout the scope of this inner local quantity.

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

See also 4.6.

Examples

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

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

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

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

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

5.4.1 Values of function designators

For a procedure declaration to define the value of a function designator, the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. This identifier is implicitly assigned an initial value (see 5.7). The identifier may, in addition, occur (one or more times) as a destination within the procedure body.

The (dynamically) last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any visible occurrence of the procedure identifier within the body of the procedure other than as a destination in an assignment statement denotes (recursive) activation of the procedure.

If a goto-statement within the procedure, or within any other procedure activated by it, leads to an exit from the procedure, other than through its end, then the execution of all statements that have been started but not yet completed and which do not contain the label to which the goto-statement leads, is abandoned. The values of all variables that still have significance remain as they were immediately before execution of the goto-statement (cf. 7.3.5).

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

5.4.2 Parameter specification

formal–parameter–part
= (*formal–parameter* { , *formal–parameter* })

formal–parameter
= *identifier*

specification–part
= *specifier identifier–list* { ; *specifier identifier–list* }

specifier
= *type* [**array** | **procedure**]
| **label**
| **switch**

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

5.4.3 *Parameter transmission modes*

mode–part
= *name–part* [*value–part*]
| *value–part* [*name–part*]

name–part
= **name** *identifier–list* ;

value–part
= **value** *identifier–list* ;

identifier–list
= *identifier* { , *identifier* }

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

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

The available transmission modes are shown in fig. 5.1 for the different kinds of parameters to procedures.

Parameter	Transmission modes		
	by value	by reference	by name
value type	D	I	O
object reference type	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. 5.1 Transmission modes for procedures

5.5 Class declaration

class–declaration
= [*prefix*] *main–part*

prefix
= *class–identifier*

main–part
= **class** *class–identifier*
[*formal–parameter–part* ; [*value–part*] *specification–part*] ;
[*protection–part* ;] [*virtual–part* ;]
class–body

class–identifier
= *identifier*

class–body
= *statement*
| *split–body*

split–body
= *initial–operations inner–part final–operations*

initial–operations
= ⟨ **begin** | *block–head* ; ⟩ { *statement* ; }

inner–part
= { *label* : } **inner**

final–operations
= **end**
| ; *compound–tail*

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

An object is generated as the result of evaluating an object generator, which is analogous to the evaluation (call) of a function designator, see 3.8.2 or 4.7.

A class body acts like a block, whether it takes that form or not. In a split body the symbol inner represents a dummy statement (but see 5.5.2, (5)).

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 5.5.5 below). The following specifiers are accepted:

<type>, array, and <type> array.

Note: Call by name is not available for parameters of class declarations.

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 5.5.3.

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, but subscript bound expressions which refer to attributes other than the formal parameters of the class declaration (or its prefixes, see 5.5.2) are illegal.

5.5.1 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.

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

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

2) A class can be used as prefix only at the block level at which it is declared. A system class used as a prefix is, together with all classes of its prefix chain, considered to be declared in the smallest block enclosing its textual occurrence. Thus redeclarations may occur at inner block levels of a program. An implementation may restrict the number of different block levels at which such prefixes may be used (see chapters 6, 10, 11 and 12).

Let $C_1, C_2, \ldots, C_n$ be classes such that C_1 has no prefix and C_k has the prefix C_{k-1} ($k = 2, 3, \ldots, n$). Then $C_1, C_2, \ldots, C_{k-1}$ is called the "prefix sequence" of C_k ($k = 2, 3, \ldots, n$). The subscript k of C_k ($k = 1, 2, \ldots, n$) is called the "prefix level" of the class. C_i is said to "include" C_j if $i <= j$, and C_i is called a "subclass" of C_j if $i > j$ ($i, j = 1, 2, \ldots, n$). The prefix level of a class D is said to be "inner" to that of a class C if D is a subclass of C, and "outer" to that of C if C is a subclass of D.

Example

Figure 5.2 depicts a class hierarchy consisting of five classes, A, B, C, D and E:

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

A capital letter denotes a class. The corresponding lower case letter represents the attributes of the main part of an object belonging to that class. In an implementation of the language, the object structures shown in fig. 5.3 indicate the allocation in memory of the values of those attributes which are simple variables.

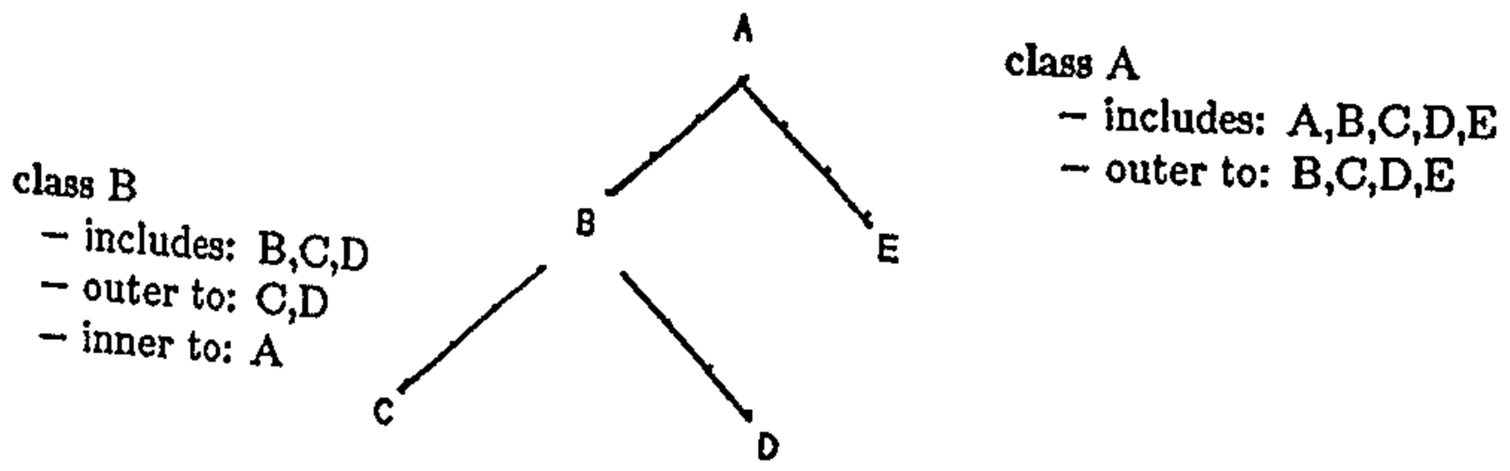


Fig. 5.2

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

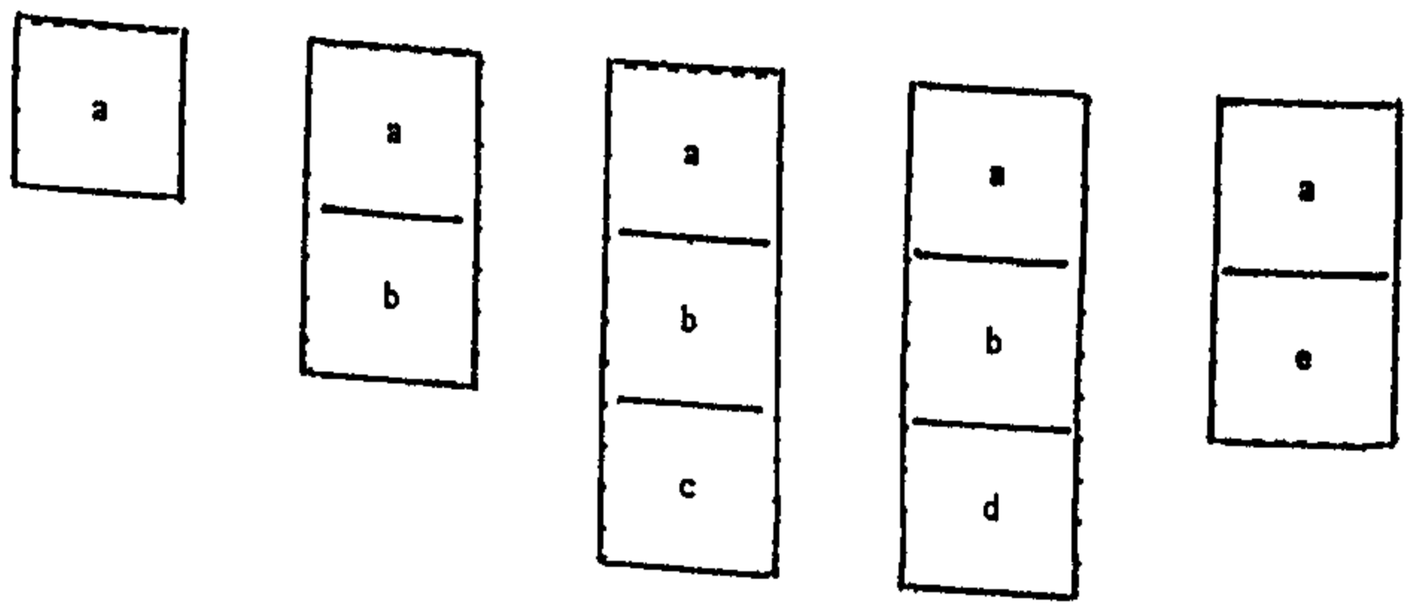


Fig. 5.3

5.5.2 Concatenation

Let C_n be a class with the prefix sequence $C_1, C_2, \ldots, C_{n-1}$, and let X be an object belonging to C_n. Informally, the concatenation mechanism has the following consequences.

1) X has a set of attributes which is the union of those defined in $C_1, C_2, \ldots, C_n$. An attribute defined in C_k ($1 <= k <= n$) 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 C_k 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 made invisible by conflicting definitions at levels < k. (These invisible attributes may nevertheless be accessed, for example 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 5.5.3.)

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 $C_1, \ldots, C_{n-1}$ has a split body the statements in the 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, the following definition is needed.

An occurrence of an identifier which is part of a given block is said to be an "uncommitted occurrence in that block", unless it is the attribute identifier of a remote identifier (see 5.5.6), 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.

Note: 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 of these identifiers within the main part 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 definition 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 5.5.3), the one copied from the prefix class body is deleted.

Example

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

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

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

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

5.5.3 *Virtual quantities*

virtual–part
= **virtual** : *virtual–spec* ; { *virtual–spec* ; }

virtual–spec
= *specifier identifier–list*
| **procedure** *procedure–identifier procedure–specification*

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 procedure may optionally be specified with respect to its type, and the type, kind, and transmission mode of its parameters (if any). If the virtual procedure is so specified, the procedure identifier of the virtual-spec has no significance, while the procedure specification determines the identifier to be used elsewhere in the program, following the rules of 6.3.

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.

A virtual procedure quantity that contains a procedure specification, can only be matched by a procedure of the same type, and with the same procedure heading as that of the procedure specification. Otherwise, the type of the matching quantity (at a given prefix level) must coincide with or be subordinate to (see 2.4.2) 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, and in the absence of a procedure 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 a 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".

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

hashing class ALGOL_hash;
begin integer procedure hash(T); text T;
    begin integer i; character c;
        T.setpos(1);
        while T.more do begin
            c:= T.getchar;
            if c <> ' ' then i:= i + rank(c)
        end;
        hash:= mod(i, n);
    end hash;
end ALGOL_hash;
```

5.5.4 Attribute protection

protection–part
= *protection–specification* { ; *protection–specification* }

protection–specification
= **hidden** *identifier–list*
| **protected** *identifier–list*
| **hidden protected** *identifier–list*
| **protected hidden** *identifier–list*

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

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

1) within the body of C or its subclasses

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

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

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

A class attribute may be specified **protected** only at the prefix level of its definition. Note that a virtual attribute may only be specified **protected** in the class heading in which the virtual specification occurs.

Attributes of the classes Simset and Simulation are protected.

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

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

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

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

Note: Specifying a virtual quantity **hidden** effectively disables further matching at inner levels.

If in the prefix sequence there are several attributes with the same identifier as that of a hidden specification, and these are previously protected, but not hidden, the innermost accessible attribute is hidden.

5.5.5 *Parameter transmission modes*

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

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

The available transmission modes are shown in fig. 5.4 for parameters of class declarations.

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

D: default mode O: optional mode I: illegal

Fig. 5.4. Parameter transmission modes for classes

For further details on parameter transmission modes, see 4.6.

5.5.6 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".

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

5.5.7 Fictitious outermost prefix

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

procedure detach; ... ; (see 7.3.1)

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

5.6 Scope and visibility rules

This section contains the scope and visibility rules governing the identifiers introduced in the program.

An identifier is introduced either implicitly (through a definition in one of the system classes ENVIRONMENT and BASICIO, or as text attribute), or explicitly

1) By its occurrence as a class attribute either in the formal parameter part or in the head of the class body. This is called an "attribute definition".

2) By its occurrence as a class attribute in the virtual part, but not in the head of the class body. This is called an "unmatched virtual definition".

3) By its occurrence in a formal parameter part of a procedure. This is called a "formal parameter definition".

4) By being declared in the head of a subblock or prefixed block. This is called a "local declaration".

5) By its occurrence as a label. In this case an implicit declaration is inserted in the local block. If this block is a class body, the label definition is called a "label attribute definition", otherwise it is considered a special case of one of 3 or 4 above.

6) By its occurrence in an external declaration which is part of an external head of a module. This is called a "head definition".

The occurrence of a system-defined class identifier as a prefix (see 5.5.1, (2)) within some block leads to insertion of implicit declarations both of the referenced class and of all classes in its prefix chain (if any) in the head of the local block. Within any block such declarations are inserted only once. These declarations are then considered special cases of either rule 1 or 4 above.

Use of such an identifier other than as a prefix refers to either such an inserted declaration, or to the definition within ENVIRONMENT or BASICIO.

The "local block" of an identifier definition is the textually closest embracing block (subblock, prefixed block, or procedure or class body), including the fictitious blocks surrounding the controlled statement of a for-statement, a procedure or class declaration, a connection block etc. The identifier (and its definition) is said to be local within this block.

A distinction is made between the "scope" and the "visibility" of an identifier definition and its associated identifier as follows.

The **scope** of an identifier definition is that part of the program text in which it may have an effect.

An identifier definition is said to be **visible** at (or from) a given point in the program text when an occurrence of the identifier at this point can refer to the quantity of the declaration in question.

The same identifier may be defined in several places in the program and may consequently be associated with different quantities. The scopes of such definitions of the same identifier may thus overlap, for instance in the case of an identifier redeclared in an inner block.

At a particular point in the program text where a given identifier is visible there can be at most one definition associated with that identifier, e.g. in the case of redeclaration as mentioned only one of the definitions is visible at any given point within the union of their scopes.

The exact meaning of these terms will be defined in the sequel for each kind of definition.

5.6.1 *Scope of identifier definitions*

1) The scope of an attribute definition is the same as that of the class declaration in which the definition occurs.

2) The scope of an unmatched virtual declaration is the same as that of the class declaration in which the definition occurs.

3) The scope of a formal parameter definition is the procedure body.

4) The scope of a local declaration is the textual extent of its local block, i.e. from the **begin** of the block in question to the matching **end**.

5) The scope of a label or switch attribute definition is the protection part, the virtual part, and the body of the class in which the definition occurs, extended by the protection part and body of all its subclasses.

6) The scope of a head definition is the source module in which it occurs. If the head is followed by a class or procedure declaration the scope is extended by the scope of that declaration.

7) The scope of a separately compiled declaration is the definition itself, extended with the scopes of all external declarations that reference the definition in question.

8) The scopes of attributes of system classes extend over all source modules.

5.6.2 *Visibility of identifiers*

Identifier definitions can only be visible within their scope. The visibility of a particular definition may within its scope be restricted by

a) the occurrence of an identifier definition with the same identifier (a "redefinition" of the identifier) within some construct enclosed by the local block of the former definition. Within their common scope only the innermost of such redefinitions are visible.

b) the occurrence of a redefinition at some inner prefix level.

c) remote access, which may cause some identifier definitions to become invisible within the inspection block or dot notation.

d) use of **this** or **qua** which may cause one or more redefinitions (of type b) to be temporarily suspended.

e) a protection part of the class declaration to which the definition is local (see 5.5.4).

Redefinition of an identifier is not allowed at the head of its local block.

Note: This prohibits the occurrence of two definitions of the same identifier in the same block. A formal procedure parameter identifier may, however, be redefined in the head of the body belonging to its procedure declaration. Such a redeclaration effectively restricts the visibility of the formal parameter to its associated value part, name part and specification part. Thus the only effect of such a parameter is the possible side effects resulting from its evaluation.

In the detailed visibility rules given below the word "visible" means "visible but for possible effects of redefinitions or other restrictions as stated above".

1) An attribute definition is visible within the protection part and the body of its class, and within all classes or blocks prefixed by that class identifier.

   Formal parameters of the class are, in addition, visible within the value part and the specification part of the class declaration. Attributes which are not formal parameters are in addition visible within the virtual parts of its class and of all its subclasses.

Note: The visibility of an attribute definition may be restricted or extended as noted under c)–e) above.

2) An unmatched virtual definition has the same visibility as an attribute definition which is not a formal parameter of the class.

3) A formal parameter definition is visible within the value part, name part, specification part, and within the body of the procedure in which it occurs.

4) A local declaration is visible within its scope.

5) A label or switch attribute definition is visible within its scope.

6) A head definition is visible within its scope.

7) A separately compiled definition is visible, together with all external definitions of its external head (if any), within the scope of an external declaration which refers to the definition.

Note: If an external procedure declaration is given with a procedure binding, the declaration defines two identifiers, occurring textually before and after is. The scope and visibility of the definition given by the procedure binding follows the rules stated above, while the identifier given in the external item is invisible elsewhere.

A possible kind identifier of an external procedure declaration is invisible elsewhere.

5.6.3 *Dynamic aspects of scope and visibility rules*

Apart from the static rules given in the preceding sections, all occurrences of an identifier (except within its definition) have a dynamic aspect. Such an identifier refers not only to its static definition but also to some object generated from the declaration to which the definition is local. Thus, for example, the occurrence of a formal parameter identifier within a procedure or class declaration references some instantiated object of the procedure or class.

See also 2.4.1 and 3.8.1.

5.7 Initialization

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

real-type	0.0
integer-type	0
Boolean	**false**
character	'!0!'
object reference type	**none**
text	**notext**

5.8 Constant declarations

constant–element
= *identifier* = *value–expression*
| *identifier* = *text–expression*

An identifier which is declared by means of a constant element has a fixed value. Any attempt to assign to or otherwise alter the value of such an identifier constitutes an error.

If the identifier is of arithmetic type then the value expression must also be of arithmetic type. Type conversion may be invoked following the rules given for arithmetic assignment (see 4.1.1). Otherwise, strict type correspondence is required between the type of the declaration and the type of the expression.

The constant element is subject to the following exception from the normal rules governing the occurrence of expressions in declarations.

Any occurrence within the expression of an identifier must refer to another constant declaration, which must occur textually before the referencing constant element.

The <constant element>s of a block head are evaluated from left to right.

6 PROGRAM MODULES

SIMULA–source–module
= [*external–head*]
⟨ *program* | *procedure–declaration* | *class–declaration* ⟩ [;]

Program modules constitute compilable programs, procedure declarations, and class declarations.

Example

```
external class b, c;   ! external head of class e;
b class e(f);  ref (c)f;
begin
   external class d;
   external procedure aproc;
   ref (d) dref;
   dref :– new d;
   aproc(dref);
end class e;
```

6.1 External declarations

external–head
= *external–declaration* ; { *external–declaration* ; }

external–declaration
= *external–procedure–declaration*
| *external–class–declaration*

An external declaration is a substitute for a complete introduction of the corresponding source module referred to, including its external head. In the case where multiple but identical external declarations occur as a consequence of this rule, this declaration will be incorporated only once.

Note: An uncommitted occurrence of a standard identifier within a source module refers to the declaration of that identifier within the class ENVIRONMENT or BASICIO, implicitly enclosing the main program (see chapters 9 and 10), with the exception of class identifiers (see 5.5.1).

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

6.2 The main program

program
= *statement*

The statement of the main program is implicitly enclosed in a prefixed block as described in chapter 10.

6.3 External procedure declaration

external–procedure–declaration
= **external** [*kind*] [*type*] **procedure** *external–list*
| **external** *kind* **procedure** *external–item procedure–specification*

kind
= *identifier*

procedure–specification
= **is** *procedure–declaration*

The kind of an external procedure declaration may indicate the source language in which the separatel compiled procedure is written (e.g. assembly, Fortran). The kind must be empty if this language SIMULA. The interpretation of kind (if given) is implementation-dependent.

If an external procedure declaration contains a procedure specification, the procedure body of the pro- cedure declaration must be empty. This specifies a procedure whose actual body, which embodies th algorithm required, is supplied in a separate, non-SIMULA module. The procedure heading (see 5.4) the procedure declaration will determine the procedure identifier (function designator) to be used withi the source module in which the external declaration occurs, as well as the type, order, and transmission mode of the parameters.

A non-SIMULA procedure cannot be used as an actual parameter corresponding to a formal procedure.

6.4 External class declaration

external–class–declaration
= **external class** *external–list*

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

Note: As a consequence of 5.5.1 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 implicitly declares all classes in its prefix chain (since these will be declared in the external head of the class in question).

6.5 Module identification

external–list
= *external–item* { , *external–item* }

external–item
= *identifier* [= *external–identification*]

external–identification
= *string*

The identifier of an external item must be identical to the identifier of the corresponding separately compiled class or SIMULA procedure.

An external item may introduce an external identification to identify the separately compiled module with respect to the environment. The interpretation of the external identification string is implementation-dependent, as is the identification of the module in question in case no external identification is given.

7 SEQUENCING

7.1 Block instances and states of execution

The constituent parts of a program execution are dynamic instances of blocks, i.e. subblocks, prefixed blocks, connection blocks and class bodies.

A block instance is said to be "local to" the one which (directly) contains its describing text. For instance an object of a given class is local to the block instance which contains the class declaration. The instance of the outermost block (see chapter 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, it is said that the PSC is "positioned" at the program point and is "contained" by 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 subblock, a procedure body or a connection block) the PSC returns to the program point immediately following the statement or expression which caused the generation of the block instance. For sequencing of class objects see 7.2 and 7.3.

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 attached state. 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 chapter 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 7.3.1). The object may reenter the attached state through the execution of a call statement (see 7.3.2), 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 7.3.3). 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, since the call by reference parameter transmission mode is applicable to arrays.

7.2 Quasi-parallel systems

A quasi-parallel system is identified by any instance of a subblock 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 chapter 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 7.3. 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.

7.2.1 *Semi-symmetric sequencing: detach - call*

In this section the concept of a quasi-parallel system is irrelevant. Consequently, only object components are considered, 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. The component thereby loses its status as a component.

7.2.2 *Symmetric component sequencing: detach - resume*

In this section only components which belong to a quasi-parallel system are considered.

Initially, i.e. upon the generation of a system head, the main component is the operative and only component of the system.

Non-operative object components of the system are created as described in the previous section, i.e. by detach statements with respect to attached objects local to the system head.

Non-operative object components of the system may be activated by call-statements, whereby they lose their component status, as described in the previous section.

A non-operative object component of the system may also be reactivated through the execution of a resume statement with respect to its detached head, whereby the PSC is moved to its reactivation point. The head of the component enters the resumed state and the component becomes operative.

The previously operative component of the system becomes non-operative and its reactivation point is 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.

7.2.3 *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

$$X = Z_0, Z_1, ..., Z_n = Y \qquad (n>=0)$$

such that for i= 1, 2, ... , n:

- Z_{i-1} is attached to Z_i, or
- Z_{i-1} is a resumed object whose associated system head is attached to Z_i.

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 by 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.

See detailed example in 7.4.

7.3 Quasi-parallel sequencing

A quasi-parallel system is created through the entry into a subblock 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.

7.3.1 *Detach*

Consider a call of the detach attribute of a block instance X.

If X is an instance of a prefixed block the detach statement 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 7.3.2).

   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.

7.3.2 *Call*

"call" is formally a procedure with one object reference type 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.

7.3.3 *Resume*

"resume" is formally a procedure with one object reference type 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.

7.3.4 *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).

7.3.5 *Goto-statement*

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 is moved to P.

3) Otherwise, 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 7.3.4) after which the process is immediately repeated from 1).

See also 4.5.

7.4 Annotated 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                    ! see fig. 7.7;
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 parentheses. 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. 7.1

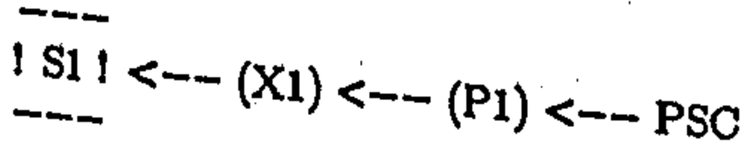
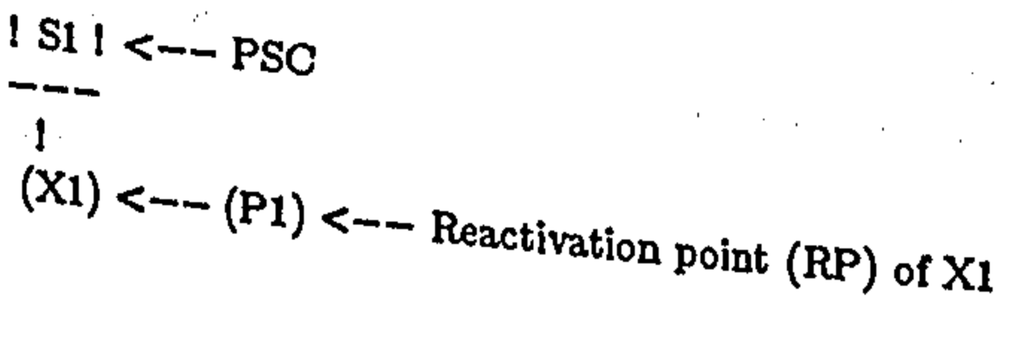


Fig. 7.2



Before and after the detach in line 16:

Fig. 7.3

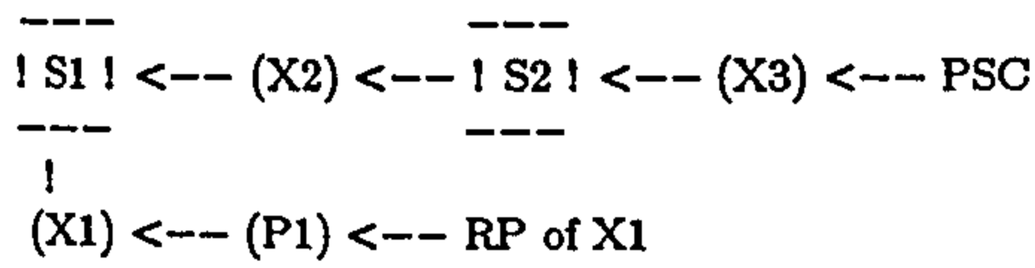


Fig. 7.4

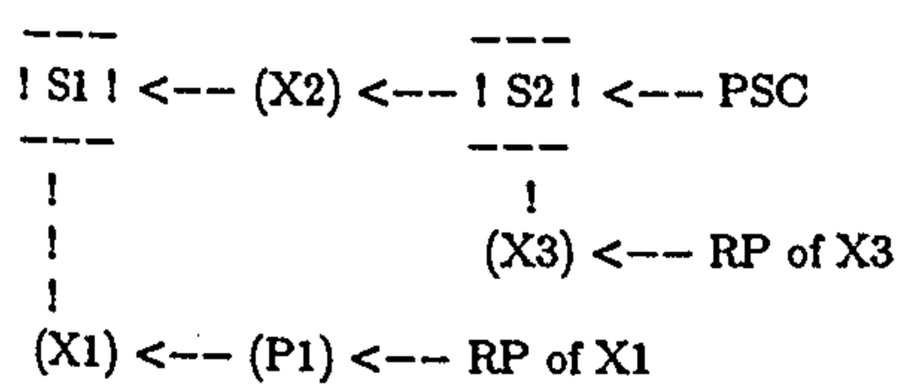
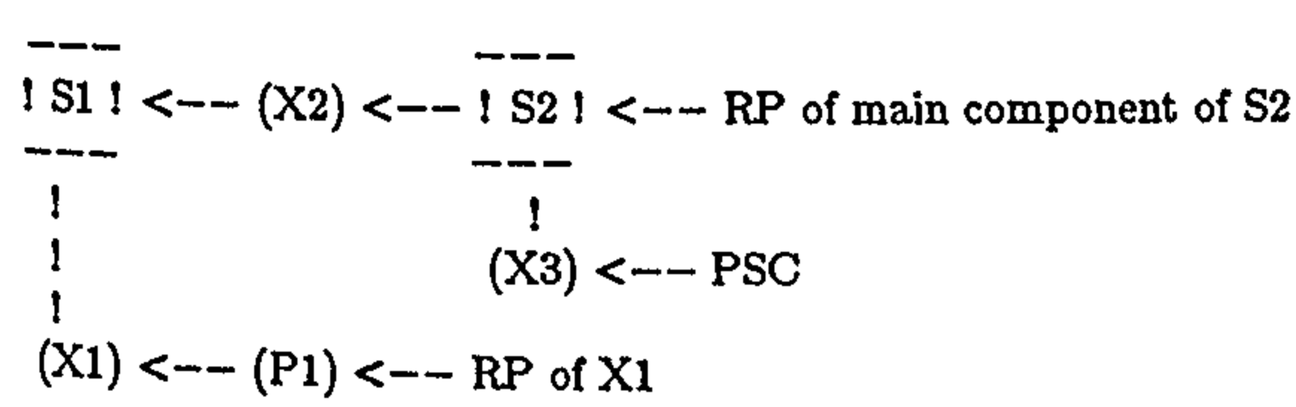


Fig. 7.4 also shows the situation before the resume in line 20. After this resume:

Fig. 7.5



Before and after the detach in line 10:

Fig. 7.6

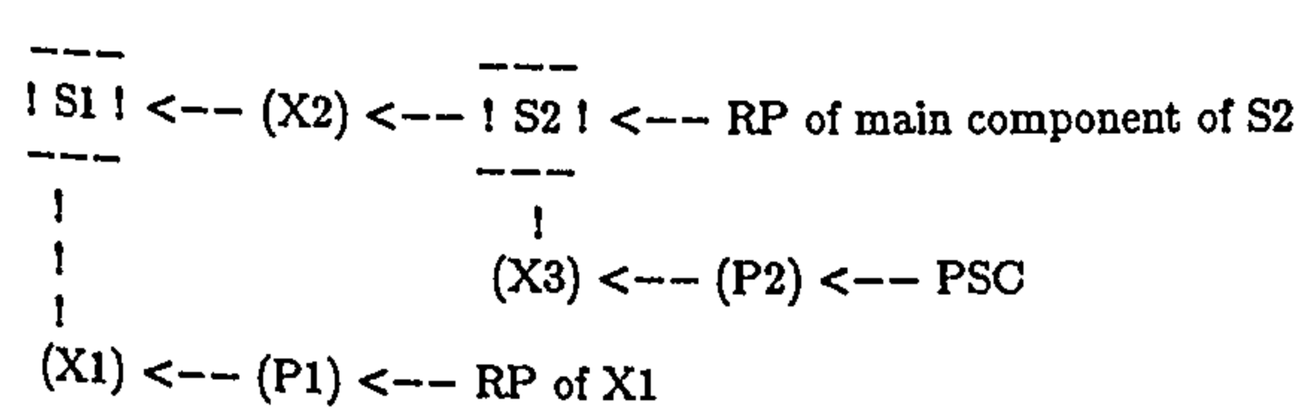
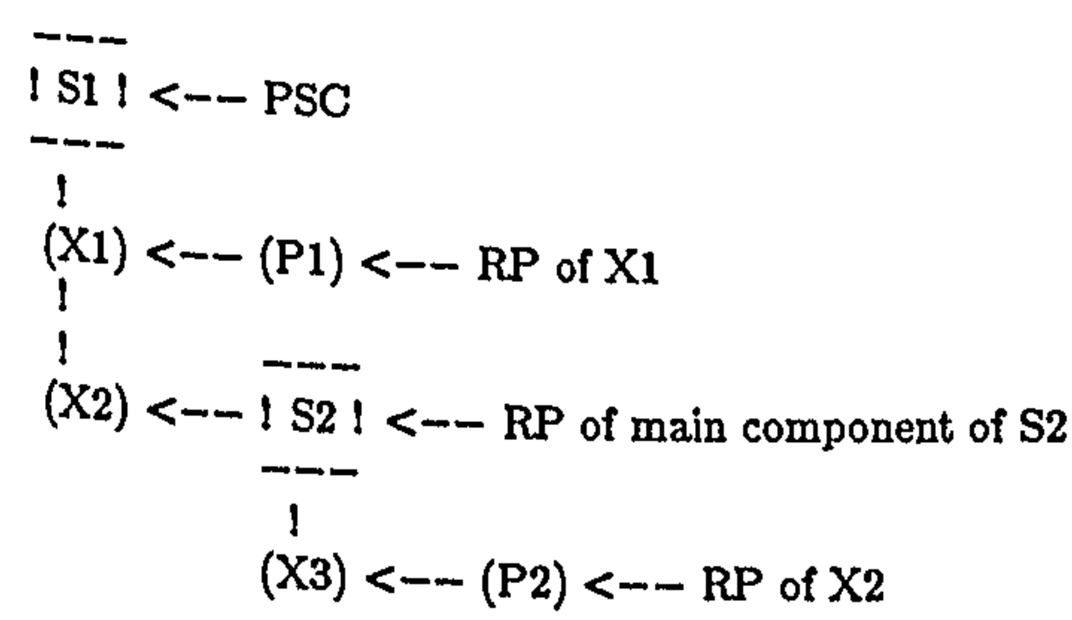
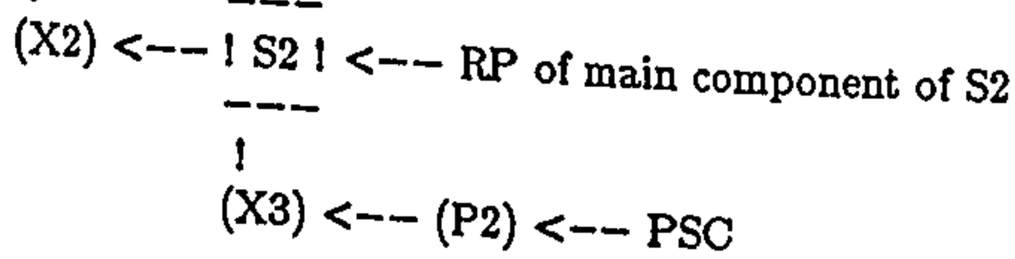


Fig. 7.7



Note that X3 is still the operative component of S2 and does not have a reactivation point of its own. Fig. 7.7 also shows the situation before the call in line 25. After this call, the situation in fig. 7.6 is reestablished. If, however, the call in line 25 is replaced by a "resume(X2)" the following situation arises:

Fig. 7.8



If now a "resume(X1)" is executed at line 11, the PSC is moved to the "RP of X1" in fig. 7.8, leaving an "RP of X2" at the former PSC. If instead a "detach" is executed, fig. 7.8 leads back to fig. 7.7.

8 ATTRIBUTES OF TEXT

A text object is conceptually an instance of

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

Any text value processed by the program is contained within a text frame, i.e. a non-empty segment of the MAIN attribute of some TEXTOBJ instance, or it is empty (i.e. **notext**). See 2.5.

A text variable is conceptually an instance of a composite structure

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

It references (and has as its value the contents of) some text frame defined by the three first components. POS identifies the current character. See 3.1.2.

See also 3.3.3 and 3.3.6 (text relations), 3.7 (text expressions), 4.1.2 and 4.1.3 (text assignments).

This chapter defines all procedure attributes of any text variable. They may be accessed by remote identifiers of the form

```
text–primary . procedure–identifier
```

The attributes are

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

8.1 "constant", "start", "length" and "main"

```
CONSTANT        Boolean procedure constant;
                constant:= OBJ == none or else OBJ.CONST;

START           integer procedure start;   start:= START;

LENGTH          integer procedure length;  length:= LENGTH;

MAIN            text procedure main;
                if OBJ =/= none then
                begin text T;
                  T.OBJ:- OBJ;
                  T.START:= 1;
                  T.LENGTH:= OBJ.SIZE;
                  T.POS:= 1;
                  main:- T;
                end main;
```

"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
```

In addition,

```
notext.main    ==  notext
"ABC".main     =   "ABC"      (but "ABC".main =/= "ABC")
```

Examples

```
Boolean procedure overlapping(X, Y);  text X, Y;
overlapping := X.main == Y.main and then
               (if   X.start <= Y.start
                then X.start + X.length > Y.start
                else Y.start + Y.length > X.start);
```

"overlapping(X, Y)" is true if and only if X and Y reference text frames which overlap each other.

```
Boolean procedure subtext(X, Y);  text X, Y;
subtext := X.main == Y.main
           and then X.start >= Y.start
           and then X.start + X.length <= Y.start + Y.length;
```

"subtext(X, Y)" is true if and only if X references a subframe of Y, or if both reference **notext**.

8.2 Character access

The characters of a text are accessible one at a time. Any text variable contains a "position indicator" POS, which identifies the currently accessible character, if any, of the referenced 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 8.6 and 8.7 may alter the position indicator of the text variable of which the procedure is an attribute.

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.

Note: The implicit modification of POS is lost immediately if "setpos", "getchar" or "putchar" is successfully applied to a text expression which is not a variable (see 3.7.2).

POS

```
integer procedure pos;  pos := POS;
```

SETPOS

```
procedure setpos(i);  integer i;
POS := if i < 1 or i > LENGTH + 1 then LENGTH + 1 else i;
```

MORE

```
Boolean procedure more;  more := POS <= LENGTH;
```

GETCHAR

```
character procedure getchar;
if POS > LENGTH then error("..." ! Pos out of range;)
else begin
  getchar:= OBJ.MAIN(START + POS – 1);  POS:= POS + 1
end getchar;
```

PUTCHAR

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

8.3 Text generation

The following standard procedures are available for text frame generation:

BLANKS

```
text procedure blanks(n); integer n;
if      n < 0 then error("..." ! Parm. to blanks < 0;)
else if  n > 0
then begin text T;
   T.OBJ:– new TEXTOBJ(n, false);
   T.START:= 1;
   T.LENGTH:= n;
   T.POS:= 1;
   T:= notext;     ! blank–fill, see 4.1.2;
   blanks:– T
end blanks;
```

"blanks(n)", with n > 0, references a new alterable main frame of length n, containing only blank characters. "blanks(0)" references **notext**.

COPY

```
text procedure copy(T); text T;
if T =/= notext
then begin text U;
   U.OBJ:– new TEXTOBJ(T.LENGTH, false);
   U.START:= 1;
   U.LENGTH:= T.LENGTH;
   U.POS:= 1;
   U:= T;
   copy:– U
end copy;
```

"copy(T)", with T =/= **notext**, references a new alterable main frame which contains a text value identical to that of T.

Text frame generation is also performed by the text concatenation operator (see 3.7.1) and by the standard procedure "datetime" (see 9.10).

8.4 Subtexts

Two procedures are available for referencing subtexts (subframes).

SUB

```
text procedure sub(i, n); integer i, n;
if i <= 0 or n < 0 or i + n > LENGTH + 1
then error("..." ! Sub out of frame;)
else if n > 0
then 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

STRIP

```
text procedure strip; ... ;
```

The expression "X.strip" is equivalent to "X.sub(1, n)", where n indicates the position of the last non-blank character in X. If X does not contain any non-blank character, **notext** is returned.

Let X and Y be text variables. Then after the value assignment "X:=Y", if legal, the relation "X.strip = Y.strip" has the value **true**, while "X = Y" is **true** only if X.length = Y.length.

8.5 Numeric text values

The names of the syntactic units in this section are in upper case to indicate that these rules concern syntax for data and not for program text.

The syntax applies to sequences of characters, i.e. to text values.

NUMERIC–ITEM
= *REAL–ITEM* | *GROUPED–ITEM*

REAL–ITEM
= *DECIMAL–ITEM* [*EXPONENT*]
| *SIGN–PART EXPONENT*

GROUPED–ITEM
= *SIGN–PART GROUPS* [*DECIMAL–MARK GROUPS*]
| *SIGN–PART DECIMAL–MARK GROUPS*

DECIMAL–ITEM
= *INTEGER–ITEM* [*FRACTION*]
| *SIGN–PART FRACTION*

INTEGER–ITEM
= *SIGN–PART DIGITS*

FRACTION
= *DECIMAL–MARK DIGITS*

SIGN–PART
= *BLANKS* [*SIGN*] *BLANKS*

EXPONENT
= *LOWTEN–CHARACTER INTEGER–ITEM*

GROUPS
= *DIGITS* { *BLANK DIGITS* }

SIGN
= + | –

DIGITS
= *DIGIT* { *DIGIT* }

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

LOWTEN–CHARACTER
= & | ...

DECIMAL–MARK
= . | ,

BLANKS
= { *BLANK* | *TAB* }

BLANK and TAB are the two characters space and horisontal tabulation, respectively.

The default representations of LOWTEN CHARACTER and DECIMAL MARK are "&" and ".", respectively. These values may, however, be changed by appropriate procedure calls, see 9.2.

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.

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

Note: Both the editing and the de-editing procedures are understood to operate on text values represented in the internal character set.

8.6 "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, which is contained by X and w contains the first character of X is located. (Note that leading blanks and tabs are accep as part of any numeric item.)

2) If no such numeric item is found, a run-time error occurs.

3) Otherwise, the numeric item is interpreted as a number.

4) If that number is outside a relevant implementation-defined range, a run-time error occ

5) Otherwise, an arithmetic value is computed, which is equal to or approximates to number.

6) The position indicator of X is made one greater than the position of the last characte the numeric item. Note that this increment is lost immediately if X does not correspond a variable (see 3.7).

The following de-editing procedures are available.

GETINT

integer procedure getint; ... ;

The procedure locates an INTEGER ITEM. The function value is equal to the corresponding integer.

GETREAL

long real procedure getreal; ... ;

The procedure locates a REAL ITEM. The function value is equal to or approximates to the correspon ing number. An INTEGER ITEM exceeding a certain implementation-defined range may lose precisi when converted to **long real**.

Note: No distinction is made between real and **long real** items. In order to preserve precision the procedure assum long real precision.

GETFRAC

integer procedure getfrac; ... ;

The procedure locates a GROUPED ITEM. The function value is equal to the resulting integer. Th digits of a GROUPED ITEM may be interspersed with BLANKS and a single DECIMAL MARK whi are ignored by the procedure.

Note: "getfrac" is thus able to de-edit more general patterns than those generated by "putfrac".

8.7 Editing procedures

Editing procedures of a given text variable X serve to convert arithmetic values to numeric items. After an editing operation, the numeric item obtained, if any, is right-adjusted in the text frame referenced by X and preceded by as many blanks as necessary to fill the text frame. The final value of the position indicator of X is X.length+1. Note that this increment is lost immediately if X does not correspond to a variable, (see 3.7).

A positive number is edited without a sign. A negative number is edited with a minus sign immediately preceding the most significant character. Leading non-significant zeros are suppressed, except possibly in an EXPONENT.

If X references a constant text frame or **notext**, an error results. 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. If the parameters to "putfix" and "putreal" are such that some of the printed digits will be without significance, zeros are substituted for these digits (and no error condition is raised).

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".

PUTINT **procedure** putint(i); **integer** i; ... ;

The value of the parameter is converted to an INTEGER ITEM which designates an integer equal to that value.

PUTFIX **procedure** putfix(r, n); <real–type> 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 run-time error is caused.

PUTREAL **procedure** putreal(r, n); <real–type> 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.

PUTFRAC **procedure** putfrac(i, n); **integer** i, n; ... ;

The resulting numeric item is a GROUPED ITEM with no DECIMAL MARK if n<=0, and with a DECIMAL MARK followed by total of n digits if n>0. Each digit group consists of 3 digits, except possibly the first one, and possibly the last one following a DECIMAL MARK. The numeric item is an exact representation of the number i * 10**(–n).

Examples

```
procedure compact(T); text T;
begin text U; character c;
    T.setpos(1);  U:– T;
    while U.more do begin
      c:=U.getchar; if c <> ' ' then T.putchar(c)
    end;
    while T.more do T.putchar(' ')
end compact;
```

The procedure rearranges 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 blank characters. Since the parameter is called by reference, its position indicator is not altered.

```
begin
    text tr, type, amount, price, payment;
    integer pay, total;
    tr:– blanks(80);
    type:– tr.sub(1, 5);
    amount:– tr.sub(20, 5);
    price:– tr.sub(30, 6);
    payment:– tr.sub(40, 10);
    ... ;
    if type = "order" then begin                    ! ***;
      pay:= amount.getint * price.getfrac;
      total:= total + pay;
      payment.putfrac(pay, 2)
    end
end
```

If tr at *** holds the text

"order 1200 155.75 "

it will after editing contain

"order 1200 155.75 18 690.00 ".

9 THE CLASS "ENVIRONMENT"

The purpose of the environmental class is to encapsulate all constants, procedures and classes which are accessible to all source modules. It contains procedures for mathematical functions, text generation, random drawing, etc.

The general structure of ENVIRONMENT is

```
class ENVIRONMENT;
begin character CURRENTLOWTEN, CURRENTDECIMALMARK;

! Basic operations . . . . . 9.1
      Procedures mod, rem, abs, sign, entier, addepsilon, subepsilon.

  Text utilities . . . . . 9.2
      Procedures copy, blanks, char, isochar, rank, isorank, digit, letter, lowten,
      decimalmark, upcase, lowcase.

  Scheduling . . . . . 9.3
      Procedures call (7.3.2), resume (7.3.3).

  Mathematical functions . . . . . 9.4
      Procedures sqrt, sin, cos, tan, cotan, arcsin, arccos,
      arctan, arctan2, sinh, cosh, tanh, ln, log10, exp.

  Extremum functions . . . . . 9.5
      Procedures max, min.

  Environmental enquiries . . . . . 9.6
      Procedure sourceline.
      Constants maxrank, maxint, minint, maxreal, minreal, maxlongreal, minlongreal, simulaid.

  Error control . . . . . 9.7
      Procedure error.

  Array quantities . . . . . 9.8
      Procedures upperbound, lowerbound.

  Random drawing . . . . . 9.9
      Procedures draw, randint, uniform, normal, negexp, Poisson, Erlang, discrete, linear, histd.

  Calendar and timing utilities . . . . . 9.10
      Procedures datetime, cputime, clocktime.

  Miscellaneous utilities . . . . . 9.11
      Procedure histo.

  Standard system classes . . . . . 9.12
      Classes simset (ch. 11), simulation (ch. 12). ;

    CURRENTDECIMALMARK:= '.';
    CURRENTLOWTEN:= '&'

end ENVIRONMENT;
```

Note: The procedure "terminate_program" is defined in BASICIO, see chapter 10.

9.1 Basic operations

MOD

```
integer procedure mod(i, j);    integer i, j;
begin integer res;
   res := i – (i//j)*j;
   mod := if res = 0 then 0
          else if sign(res) <> sign(j) then res+j
          else res
end mod;
```

The result is the mathematical modulo value of the parameters.

REM

```
integer procedure rem(i, j);    integer i, j;
   rem := i – (i//j)*j;
```

The result is the remainder of an integer division.

ABS

```
<type of e> procedure abs(e);    <arithmetic–type> e;

abs := if e >= 0 then e else –e;
```

The result is the absolute value of the parameter.

SIGN

```
integer procedure sign(e);    <arithmetic–type> e;
   sign := if      e > 0 then  1
           else if e < 0 then –1 else 0;
```

The result is zero if the parameter is zero, one if the parameter is positive, and minus one otherwise.

ENTIER

```
integer procedure entier(r);  <real–type> r;
begin integer j;
   j := r;                    ! implied conversion of "r" to integer ;
   entier:= if j > r          ! implied conversion of "j" to real ;
            then j– 1 else j
end entier;
```

The result is the integer "floor" of a real type item, the value always being less than or equal to the parameter. Thus, entier(1.8) returns the value 1, while entier(–1.8) returns –2.

ADDEPSILON

```
<type of e> procedure addepsilon(e);    <real–type> e;

   addepsilon := e + ... ;  ! see below;
```

SUBEPSILON

```
<type of e> procedure subepsilon(e);    <real–type> e;

   subepsilon := e – ... ;  ! see below;
```

The result type is that of the parameter. The result is the value of the parameter incremented (addepsilon) or decremented (subepsilon) by the smallest positive value, such that the result is not equal to the parameter within the precision of the implementation. Such a value does not exist, however, for the extreme parameter values "maxreal" and "maxlongreal" to "addepsilon", and "minreal" and "minlongreal" to "subepsilon"; in these cases the procedures return the value of the parameter.

9.2 Text utilities

COPY — See 8.3.

BLANKS — See 8.3.

CHAR

```
character procedure char(i);  integer i;
   char := ... ;
```

The result is the character obtained by converting the parameter according to the implementation-defined coding of characters. The parameter must be in the range 0..maxrank.

ISOCHAR

```
character procedure isochar(i);  integer i;
   isochar := ... ;
```

The result is the character obtained by converting the parameter according to an 8-bit character set compatible with the ISO 646 7-bit character set. The parameter must be in the range 0..255.

RANK

```
integer procedure rank(c);  character c;
   rank := ... ;
```

The result is the integer obtained by converting the parameter according to the implementation-defined character code.

ISORANK

```
integer procedure isorank(c);  character c;
   isorank := ... ;
```

The result is the integer obtained by converting the parameter according to an 8-bit character set compatible with the ISO 646 7-bit character set.

DIGIT

```
Boolean procedure digit(c);  character c;
   digit := ... ;
```

The result is **true** if the parameter is a decimal digit.

LETTER

```
Boolean procedure letter(c);  character c;
   letter := ... ;
```

The result is true if the parameter is a letter of the English alphabet ('a' ... 'z', 'A' ... 'Z').

LOWTEN

```
character procedure lowten(c);  character c;
if ... ! c is illegal as lowten;
then  error("..." ! Lowten error ;)
else begin
   lowten:= CURRENTLOWTEN;  CURRENTLOWTEN:= c
end lowten;
```

Changes the value of the current lowten character to that of the parameter. The previous value is returned. Illegal parameters are:

digits, plus ("+"), minus ("–"), period ("."), comma (","), control characters, DEL and all characters not included in the ISO 646 character set (i.e. with an "isorank" greater than 127).

DECIMALMARK

```
character procedure decimalmark(c);    character c;
if c <> '.' and then c <> ','
then error("..." ! Decimalmark error ;)
else begin
  decimalmark:= CURRENTDECIMALMARK;
  CURRENTDECIMALMARK:= c
end decimalmark;
```

Changes the value of the decimal point character used by the text (de)editing procedures (cf. 8.6 and 8.7). The previous value is returned. The only legal parameter values are period and comma.

UPCASE

```
text procedure upcase(t);    text t;
begin t.setpos(1); upcase:– t;   ... end;
```

Converts the letters in the text parameter to their upper case representation. Only letters of the English alphabet are converted. The result is a reference to the parameter.

LOWCASE

```
text procedure lowcase(t); text t;
begin t.setpos(1); lowcase:– t; ... end;
```

Converts the letters in the text parameter to their lower case representation. Only letters of the English alphabet are converted. The result is a reference to the parameter.

9.3 Scheduling

CALL

See 7.3.2.

RESUME

See 7.3.3.

In addition the procedure "detach" (7.3.1) is an attribute of any class.

9.4 Mathematical functions

These procedures return **long real** results whenever a parameter is of this type. Otherwise a **real** type result is returned (cf. 3.3.1).

All procedures return real type approximations to the associated mathematical functions. Their exact definitions (concerning precision, allowed parameter values etc.) are implementation-defined. The procedures return best possible approximations to the exact mathematical results.

The trigonometric functions deal with angles expressed in radians.

SQRT <real-type> **procedure** sqrt(r); <real-type> r;

SIN <real-type> **procedure** sin(r); <real-type> r;

COS <real-type> **procedure** cos(r); <real-type> r;

TAN <real-type> **procedure** tan(r); <real-type> r;

COTAN <real-type> **procedure** cotan(r); <real-type> r;

ARCSIN <real-type> **procedure** arcsin(r); <real-type> r;

ARCCOS <real-type> **procedure** arccos(r); <real-type> r;

ARCTAN <real-type> **procedure** arctan(r); <real-type> r;

The result is in the range (0, pi/2) for non-negative parameters and in the range (-pi/2, 0) for negative parameters.

ARCTAN2 <real–type> **procedure** arctan2(y, x); <real–type> y, x;

The result is in the range(-pi, pi) and a negative value is returned whenever y is negative. Positive y values always result in a positive result, while a zero value returns zero if x is positive and pi if x is negative. If both y and x are zero, a runtime error occurs.

SINH <real-type> **procedure** sinh(r); <real-type> r;

COSH <real-type> **procedure** cosh(r); <real-type> r;

TANH <real-type> **procedure** tanh(r); <real-type> r;

LN <real-type> **procedure** ln(r); <real-type> r;

LOG10 <real-type> **procedure** log10(r); <real-type> r;

EXP <real-type> **procedure** exp(r); <real-type> r;

9.5 Extremum functions

MAX <type> **procedure** max(i1, i2); <type> i1; <type> i2;

MIN <type> **procedure** min(i1, i2); <type> i1; <type> i2;

The value is the greater ("max") or lesser ("min") of the two parameter values. Legal parameter types are **text**, **character**, real type and integer type. The type of the result conforms to the rules of 3.3.1.

9.6 Environmental enquiries

SOURCELINE **integer procedure** sourceline;

The value indicates the line on which the procedure call occurs. The interpretation of this number is implementation-defined.

MAXLONGREAL **long real** maxlongreal = ... ;

MINLONGREAL **long real** minlongreal = ... ;

MAXREAL **real** maxreal = ... ;

MINREAL **real** minreal = ... ;

MAXRANK **integer** maxrank = ... ; ! the largest legal argument to char;

MAXINT **integer** maxint = ...;

MININT **integer** minint:= ...;

These constants define some of the implementation characteristics. The "max..." ("min...") constants have the largest (smallest) values possible for their type.

SIMULAID **text** simulaid = ... ; ! See below ;

The value of "simulaid" is an implementation-defined string of the following general format:
<simid>!!!<siteid>!!!<OS>!!!<CPU>!!!<user>!!!<job>!!!<acc>!!!<prog>

<simid>: Identification of the SIMULA system (name, version etc.)
<siteid>: Identification of the installation (e.g. organization name)
<OS>: Operating system identification (name, version, etc.)
<CPU>: Host system identification (manufacturer, name, number, etc.)
<user>: User identification
<job>: Job identification (session number)
<acc>: Account identification
<prog>: Identification of the executing task or program

9.7 Error control

ERROR

```
procedure error(t);  text t;
begin ... display text "t" and stop program...
end error;
```

"error" stops the execution of the program as if a runtime error has occurred and presents the contents of the text parameter on the diagnostic channel (normally the controlling terminal).

9.8 Array quantities

LOWERBOUND

```
integer procedure lowerbound(a, i);
<type> array a;  integer i;
```

UPPERBOUND

```
integer procedure upperbound(a, i);
<type> array a;  integer i;
```

The procedure "lowerbound" ("upperbound") returns the lower (upper) bound of the dimension of the given array corresponding to the given index. The first dimension has index one, the next two, etc. An index less than one or greater than the number of dimensions of the given array constitutes a run time error.

9.9 Random drawing

All random drawing procedures of SIMULA are based on the technique of obtaining "basic drawings" from the uniform distribution in the interval <0, 1>.

9.9.1 Pseudo-random number streams

A basic drawing replaces the value of a specified **integer** variable, say U, by a new value according to an implementation-defined algorithm.

As an example, the following algorithm may be suitable for binary computers:

U(i+1) = remainder ((U(i) * 5**(2*p+1)) // 2**n)

where U(i) is the i'th value of U, n is an integer related to the size of a computer word and p is a positive integer. It can be proved that, if U(0) is a positive odd integer, the same is true for all U(i) and the sequence U(0), U(1), U(2), ... is cyclic with period 2**n−2. (The last two bits of U remain constant, while the other n−2 take on all possible combinations). Thus there are two sequences - one in the range (1:2**n−3) and the other in (3:2**n−1).

It is a property of this algorithm that any successor to a stream number U(i), e.g. U(i+m), can be computed using modular arithmetic in log2(m) steps.

The real numbers u(i) = U(i) * 2**(−n) are fractions in the range <0, 1>. The sequence u(1), u(2), ... is called a "stream" of pseudo-random numbers, and u(i) (i = 1, 2, ...) is the result of the i'th basic drawing in the stream U. A stream is completely determined by the initial value U(0) of the corresponding integer variable. Nevertheless, it is a "good approximation" to a sequence of truly random drawings.

By reversing the sign of the non−zero initial value U(0) of a stream variable, the antithetic drawings 1−u(1), 1−u(2), ... should be obtained. In certain situations it can be proved that means obtained from samples based on antithetic drawings have a smaller variance than those obtained from uncorrelated streams. This can be used to reduce the sample size required to obtain reliable estimates.

9.9.2 Random drawing procedures

The following procedures all perform a random drawing of some kind. "Normal", "draw", "randint", "uniform", "negexp", "discrete", "linear" and "histd" always perform the operation by means of one single basic drawing, i.e. the procedure has the side effect of advancing the specified stream by one step. The necessary type conversions are effected for the actual parameters, with the exception of the last one. The latter must always be an **integer** variable specifying a pseudo-random number stream. Note, that it must not be a procedure parameter transferred by **name**.

DRAW

```
Boolean procedure draw (a, U);
name U; long 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.

RANDINT

```
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.

UNIFORM

```
long real procedure uniform (a, b, U);
name U; long real a, b; integer U;
```

The value is uniformly distributed in the interval a <= u < b. If b < a, the call constitutes an error.

NORMAL

```
long real procedure normal (a, b, U);
name U; long 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.

NEGEXP

```
real procedure negexp (a, U);
name U; long real a; integer U;
```

The value is a drawing from the negative exponential distribution with mean 1/a, defined by –ln(u)/a, where u is a basic drawing. This is the same as a random "waiting time" in a Poisson distributed arrival pattern with expected number of arrivals per time unit equal to a. If a is non-positive, a runtime error occurs.

POISSON

```
integer procedure Poisson (a, U);
name U; long 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

```
u(0) * u(1) * ... * u(n)  <  e**(–a)
```

The validity of the formula follows from the equivalent condition

```
( –ln(u(0)) – ln(u(1)) – ... – ln(u(n))  ) / 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.

ERLANG

```
long real procedure Erlang (a, b, U);
name U; long 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,

$$-\left(\ln(u(1)) + \ln(u(2)) + \ldots + \ln(u(b))\right) / (a*b)$$

and by c+1 basic drawings u(i) otherwise, where c is equal to entier(b),

$$-\left(\ln(u(1)) + \ldots + \ln(u(c)) + (b-c)*\ln(u(c+1))\right) / (a*b)$$

Both a and b must be greater than zero.

The last formula represents an approximation.

DISCRETE

```
integer procedure discrete (A, U);
name U; <real–type> 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 function value satisfies

lowerbound(A, 1) <= discrete(A, U) <= upperbound(A, 1)+1.

It is defined as the smallest i such that A(i) > u, where u is a basic drawing and A(upperbound(A, 1)+1) = 1.

LINEAR

```
long real procedure linear (A, B, U);
name U; <real–type> 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 type 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

HISTD

```
integer procedure histd (A, U);
name U; <real–type> 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.

9.10 Calendar and timing utilities

DATETIME **text procedure** datetime; datetime:– Copy("...");

The value is a text frame containing current date and time in the form

YYYY–MM–DD HH:MM:SS.sss...

The number of decimals in the field for seconds is implementation-defined.

CPUTIME **long real procedure** cputime;

The value is the number of processor seconds spent by the calling program.

CLOCKTIME **long real procedure** clocktime;

The value is the number of seconds since midnight.

9.11 Miscellaneous utilities

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

The procedure statement "histo(A, B, c, d)" updates a histogram defined by the one-dimensional arrays A and B according to the observation c with the weight d. A(lba+i) is increased by d, where i is the smallest integer such that c <= B(lbb+i) and lba and lbb are the lower bounds of A and B respectively. If the length of A is not one greater than that of B the effect is implementation-defined. The last element of A corresponds to those observations which are greater than all elements of B.

9.12 Standard system classes

The standard system classes are "simset", "simulation" and the I/O classes. They are available at any block level of a program (cfr. 5.5.1).

Class "simset" contains facilities for list (set) manipulation, see chapter 11.

Simset class "simulation" contains facilities for discrete event simulation, see chapter 12.

The I/O classes ("file" and its subclasses) are defined in class BASICIO, see chapter 10.

10 INPUT-OUTPUT

The semantics of SIMULA I/O facilities rely on the intuitive notion of "files" ("data sets"), i.e. collections of data external to the program, organized for sequential or random access. In the language, sequential access files are called "sequential files" and random access files "direct files". When it is necessary to distinguish between the file concept of the language and the underlying files of the environment the latter are called "external files".

Actually a file may in practice be any kind of external device with communication capabilities, such as a terminal, a sensory device, etc.

Examples of sequential files are:

- 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 disk, with each item identified by a unique integer.

The standard I/O facilities are contained by a class called "BASICIO". They are available to the program through block prefixing as described below.

Note: The use of upper case letters indicates that this identifier is inaccessible to the user program (cf. 1.10).

The user's main program acts as if it were embedded as follows:

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

In any program execution the unique instance of this prefixed block constitutes the system head of the outermost quasi-parallel system (see 7.2).

The values of inlength and outlength are implementation-defined; they normally depend upon the actual device(s) associated with SYSIN and SYSOUT (the standard input and output files).

SYSIN and SYSOUT may represent the input and output features of an interactive terminal, in which case inlength and outlength are probably equal. In other cases, for example batch runs, SYSIN may represent record-oriented input and SYSOUT may represent line printer oriented output. Typical values of inlength and outlength would then be 80 and 132, resp.

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

Note: Within this chapter the term "file object" refers to an instance of one of the classes "in(byte)file", "out(byte)file" and "direct(byte)file" or of a subclass of one of these.

The overall organization of "BASICIO" is as follows:

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

          procedure terminate_program;
          begin ... ;  goto STOP  end terminate_program;

          class file . . . . . .
          file class imagefile . . . . . 10.1;
          file class bytefile . . . . . 10.3;
          imagefile class infile . . . . . 10.8;
          imagefile class outfile . . . . . 10.4;
          imagefile class directfile . . . . . 10.5;
          outfile class printfile . . . . . 10.6;
          bytefile class inbytefile . . . . . 10.7;
          bytefile class outbytefile . . . . . 10.9;
          bytefile class directbytefile . . . . . 10.10;
                                                  . . . . . 10.11;

          SYSIN  :- new infile("...");       ! Implementation-defined;
          SYSOUT :- new printfile("...");    ! file names;
          SYSIN.open(blanks(INPUT_LINELENGTH));
          SYSOUT.open(blanks(OUTPUT_LINELENGTH));
          inner;
          STOP:
          SYSIN.close;
          SYSOUT.close
        end BASICIO;
```

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

The class file and its predefined subclasses are available at any block level of a program (but see (2) of 5.5.1). An implementation may restrict, in any way, the use of these classes for class or block prefixing. An implementation may provide additional subclasses of class file.

Within a program, an object of a subclass of file may represent an external file. The effect of several such file objects representing the same external file is implementation-defined.

The procedure "terminate_program" terminates program execution. It closes SYSIN and SYSOUT. It is implementation-dependent with respect to whether or not other open files are also closed.

10.1 The class file

```
class file(FILENAME);  value FILENAME;  text FILENAME;
begin Boolean OPEN;
  text procedure filename;  filename :- copy(FILENAME);
  Boolean procedure isopen;  isopen := OPEN;
                                                                    10.1.1;
  Boolean procedure setaccess(mode);  text mode; . . . . . .
  if FILENAME == notext then error("...");
end file;
```

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

A file is either open or inaccessible as indicated by the variable "OPEN". The procedure "isopen" returns the current value of "OPEN". A file is initially inaccessible (e.g. closed).

Each file object has a text attribute FILENAME. This text value must at "open" identify an external file which, through an implementation-defined mechanism, becomes associated with the file object. If the parameter value is **notext**, a run-time error occurs.

The procedure "filename" retrieves the value of FILENAME.

10.1.1 External file access control

Certain attributes (not specified in the file outline) control the access to the external file. The values of these attributes are set when the file object is opened or closed, from a set of default values possibly modified by successive calls to the procedure "setaccess".

The standard attribute modes are SHARED, APPEND, CREATE, READWRITE, BYTESIZE, REWIND and PURGE.

SHARED: If the value is "shared", the external file may be shared by other programs. The value "noshared" implies that the file must be exclusively assigned to this program.

APPEND: If the value is "append", output to the file is added to the existing contents of the file. The value "noappend" implies for a sequential file that, after "close", the external file will contain only the output produced while the file was open. The mode is not relevant for in(byte)files. For direct files "append" prohibits output before "lastloc". For out(byte)files, the value "append" implies that SHARED has the value "noshared".

CREATE: If the value is "create", the external file associated with FILENAME must not exist at "open" (if it does, "open" returns **false**); a new file is created by the environment. If the value is "nocreate", the associated file must exist at "open". The value "anycreate" implies that if the file does exist at "open" the file is opened, otherwise a new file is created.

READWRITE: If the value is "readonly", output operations cannot be performed. If the value is "writeonly", input operations cannot be performed. The value "readwrite" enables both input and output. This mode is relevant only for direct files.

BYTESIZE: The value of this mode is a positive integer specifying the size of bytes, measured in bits. This mode is relevant only for bytefiles. An implementation may restrict the possible values in any way. If not set explicitly the value is implementation-defined.

REWIND: The value "rewind" indicates that some resetting of the external file occurs at "close" (e.g. rewind of a magnetic tape). The value "norewind" implies no such reset.

PURGE: The value "purge" implies that the external file may be deleted by the environment when it is closed (in the sense that it becomes inaccessible to further program access). The value "nopurge" implies no such deletion.

Additional values and modes may be defined by an implementation.

The parameter "mode" to procedure "setaccess" contains one of the standard values as given above, namely "shared", "noshared", "append", "noappend", "create", "nocreate", "anycreate", "readonly", "writeonly", "readwrite", "bytesize:X" (where X is a positive integer), "rewind", "norewind", "purge" and "nopurge". It is recommended that implementation-defined parameter values have the percent character % as the first character of the text.

The parameter "bytesize:0" (zero) specifies the (implementation-defined) default byte size for bytefiles.

Only one mode may be specified in each "setaccess" call, the case of the individual letters of the parameter being insignificant. Unrecognized modes are ignored and "setaccess" then returns the value **false**. The value is **true** otherwise. A specific mode is interpreted either at next "open" or next "close". A mode which is set after "open" or "close" has no effect until the next "close" or "open" respectively.

The default values of the access modes are given in table 10.1, where "NA" means "not applicable" (i.e. ignored for this file kind) and "*" means that the value is implementation-defined.

Mode:	Files of kind			
	In-	Out-	Direct-	Takes effect at
SHARED	shared	noshared	noshared	open
APPEND	NA	noappend	noappend	open
CREATE	NA	anycreate	nocreate	open
READWRITE	NA	NA	readwrite	open
BYTESIZE:x	*	*	*	open
REWIND	norewind	norewind	NA	open, close
PURGE	nopurge	nopurge	nopurge	close

Table 10.1. Default values of file access modes.

10.1.2 Open and close

Most subclasses of "file" defined in BASICIO contain these procedures.

Procedure "open" establishes the association with an external file (as identified by FILENAME), checks the access modes and causes corresponding opening actions on the external file. If the external file is closed, it is opened.

Procedure "close" causes closing actions on the external file, as specified by the access modes. In addition, the association between the file object and the external file is dissolved. If possible, the external file is closed.

The details of these procedures are subclass- and implementation-defined. All versions conform, however, to the following patterns.

OPEN

```
Boolean procedure open ... ;
if not OPEN
    and ... ! FILENAME association is established;
    and ... ! access modes compatible with external file;
    and ... ! external file is opened;
then begin
  ... ;       ! implementation–defined and access mode specified actions
                on external file, followed by subclass–dependent actions;
  open := OPEN := true;
end open;
```

CLOSE

```
Boolean procedure close;
if OPEN
    and then  ! external file is open and can be closed;
then begin
  ... ;       ! implementation–defined and access mode specified actions
                on external file, followed by subclass–dependent actions;
  OPEN:= false;
  close:= true;
end close;
```

These patterns will not be repeated in detail for each subclass; a short comment will serve to remind the reader of these general actions of the procedures.

10.2 Structure of file subclasses

There are two predefined subclasses of **class** file:

"imagefile" – image (record) oriented files
"bytefile" – character (stream) oriented files

These subclasses each have three subclasses defining the direction of data transfer and the file organization: input-oriented sequential files (i.e. in(byte)files), output-oriented sequential files (i.e. out(byte)files) and bidirectional direct files (i.e. direct(byte)files).

	imagefile subclass	bytefile subclass
sequential input	infile	inbytefile
sequential output	outfile	outbytefile
direct file	directfile	directbytefile

Table 10.2. Subclasses of class "file".

In addition, a standard subclass for line printer oriented output, outfile **class** printfile, is defined.

10.2.1 Procedure "checkpoint"

All files producing output (sequential output or direct files) contain a Boolean procedure "checkpoint". The procedure causes the environment to attempt to secure the output produced so far. Depending on the nature of the associated external device, this causes completion of output transfer (i.e. intermediate buffer contents are transferred). If this is not possible or meaningful, "checkpoint" is a dummy operation in which case the value **false** is returned.

10.2.2 Direct file locking

Direct files contain the following variable and procedures for control of simultaneous access to the external file (cf. access mode SHARED).

```
                    Boolean LOCKED;

LOCKED              Boolean procedure locked;  locked:= LOCKED;

LOCK                integer procedure lock(timelimit, loc1, loc2);
                        real timelimit;  integer loc1, loc2;
                    begin
                      lock := −1;
                      if timelimit>0.0 then begin
                        if LOCKED then unlock;
                        ... ! try to lock indicated part of file, see below;
                        if !success; then begin LOCKED := true;   lock := 0 end
                      end
                    end lock;

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

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

Procedure "lock" enables the program to get exclusive access to all or part of the file. The effect of a "lock" call while the file is locked ("LOCKED" is **true**) is that the previous lock is immediately released (prior to the new locking attempt).

The parameter "timelimit" is the (clock) time in seconds that is the maximum waiting time for the resource. If "timelimit" is less than or equal to zero, the procedure returns immediately without performing any actions upon the file.

The parameters "loc1" and "loc2" identify the part of the file to be locked, by giving the ordinal numbers of two external images (bytes). The program is given exclusive access to a part of the file which includes the requested region. If the two parameters are both zero, this implies locking the whole file. Otherwise, the size of the part of the file that is actually locked, is implementation-dependent; it may even include the entire file.

A return value of zero indicates a successful "lock" operation. The value −1 indicates that "timelimit" was exceeded (or was zero or negative). A negative value less than −1 indicates "lock" failure and its interpretation is implementation-defined.

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

10.3 Imagefiles

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

```
file class imagefile;
    begin text image;
      procedure setpos(i);  integer i;  image.setpos(i);
      integer procedure pos;      pos:= image.pos;
      Boolean procedure more;     more:= image.more;
      integer procedure length;   length:= image.length;
    end imagefile;
```

The individual logical unit of an external file associated with an imagefile is called an "external image". Each external image is an ordered sequence of characters.

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

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

The three standard subclasses of imagefile are "infile" (sequential input file), "outfile" (sequential output file) and "directfile" (bidirectional direct file). In addition, "printfile", a standard subclass of class outfile, is available. It represents a line printer oriented file.

10.4 The class "infile"

```
imagefile class infile;
        begin Boolean ENDFILE;
          Boolean procedure endfile;   endfile:= ENDFILE;

          Boolean procedure open(fileimage); text fileimage; . . . . . 10.4.1;
          Boolean procedure close; . . . . . 10.4.1;
          procedure inimage; . . . . . 10.4.2;
          Boolean procedure inrecord; . . . . . 10.4.2;
          character procedure inchar; . . . . . 10.4.3;
          Boolean procedure lastitem; . . . . . 10.4.4;
          text procedure intext(w); integer w; . . . . . 10.4.5;
          integer procedure inint; . . . . . 10.4.6;
          long real procedure inreal; . . . . . 10.4.6;
          integer procedure infrac; . . . . . 10.4.6;
          ENDFILE:= true;
        end infile;
```

An object of the class "infile" is used to represent an image-oriented sequential input file.

The variable ENDFILE is **true** whenever the file object is closed or the external file is exhausted (i.e. "end of file" has been encountered). The procedure "endfile" gives access to the value of ENDFILE.

10.4.1 Open and close

OPEN

```
Boolean procedure open(fileimage);  text fileimage;
if ... then begin ...  ! see 10.1.2;
  ENDFILE:= false;
  image:- fileimage;
  image:= notext;
  setpos(length+1);
  open:= OPEN:= true
end open;
```

If successful, "open" returns **true** and sets ENDFILE **false**. In addition, "image" references the parameter "fileimage" which is space-filled.

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
  image:- notext;
  OPEN:= false;
  close:= ENDFILE:= true
end close;
```

If successful, "close" returns **true**. In addition, OPEN is **false**, ENDFILE is **true** and "image" references **notext**.

10.4.2 Inimage and inrecord

INIMAGE

```
procedure inimage;
if not OPEN or ENDFILE then error("...")
else begin
  ... ; ! attempt to transfer external image to "image";
  if ... ! "image" too short; then error("...")
  else if ... ! there was no more to read;
  then begin
    ENDFILE:= true;
    image:= "!25!" end
  else ... ; ! pad "image" with space(s);
  setpos(1)
end inimage;
```

The procedure "inimage" performs the transfer of an external file image into "image". A run-time error occurs if "image" is **notext** or too short to contain the external image. If it is longer than the external image, the latter is left-adjusted within "image" and the remainder of the text is filled with spaces. The position indicator is set to one.

INRECORD

```
Boolean procedure inrecord;
if not OPEN or ENDFILE then error("...")
else begin
  ... ; ! transfer external image to "image" (no space–filling);
  if ... ! no more to read;
  then begin
    ENDFILE:= true;
    setpos(1);
    image.putchar('!25!') end  Note – POS = 2 now
  else begin
    setpos(... !number of characters transferred + 1; );
    inrecord:= not ...! whole external image received?;
  end if
end inrecord;
```

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

Note: If an "end of file" is encountered, EM ('!25!') is generated as a single character external image, and the variable ENDFILE is given the value **true**. A call on "inimage" or "inrecord" when ENDFILE already has the value true constitutes a run-time error.

10.4.3 Inchar

INCHAR

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

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

Note: The result may be the "EOF-character" EM (ISO 646 code 1/9).

10.4.4 Lastitem

LASTITEM

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

The purpose of the procedure "lastitem" is to skip past all SP and HT characters (ISO 646 codes 2/0 and 0/9 respectively). The process of scanning may involve the transfer of several successive external images. If the file contains no further non-space, non-tab characters the value **true** is returned.

10.4.5 Intext

INTEXT

```
text procedure intext(w); integer w;
begin text t;
  intext :− t :− blanks(w);
  while t.more do t.putchar(inchar)
end intext;
```

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

Note: The result may be a reference to an "EOF-image" (cf. 10.4.2).

10.4.6 Item-oriented input

ININT

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

INREAL

```
long real procedure inreal;
if lastitem then error("..." ! Inreal: End of file; )
else begin text t;
   t:– image.sub(pos, length–pos+1);
   inreal:= t.getreal;
   setpos(pos+t.pos–1)
end inreal;
```

INFRAC

```
integer procedure infrac;
if lastitem then error("..." ! Infrac: End of file; )
else begin text t;
   t:– image.sub(pos, length–pos+1);
   infrac:= t.getfrac;
   setpos(pos+t.pos–1)
end infrac;
```

The procedures "inint", "inreal" and "infrac" are defined in terms of the corresponding de-editing procedures of "image". These three procedures, starting at the current "pos", skip spaces and tab's, and then scan past and convert a numeric item.

10.5 The class "outfile"

```
file class outfile;
   begin

      Boolean procedure open(fileimage); text fileimage; . . . . . 10.5.1;
      Boolean procedure close; . . . . . 10.5.1;
      procedure outimage; . . . . . 10.5.2;
      procedure outrecord; . . . . . 10.5.3;
      procedure breakoutimage; . . . . . 10.5.4;
      Boolean procedure checkpoint; . . . . . 10.2.1;
      procedure outchar(c); character c; . . . . . 10.5.6;
      procedure outtext(t); text t; . . . . . 10.5.7;
      text procedure FIELD(w); integer w; . . . . . 10.5.8;
      procedure outint(i, w); integer i, w; . . . . . 10.5.8;
      procedure outfix(r, n, w); <real–type> r; integer n, w; . . . . . 10.5.8;
      procedure outreal(r, n, w); <real–type> r; integer n, w; . . . . . 10.5.8;
      procedure outfrac(i, n, w); integer i, n, w; . . . . . 10.5.8;
   end outfile;
```

An object of the class "outfile" is used to represent an image-oriented sequential output file.

Note: See 10.7 for a special property of procedures "open", "close", "outimage" and "outrecord".

10.5.1 Open and close

OPEN

```
Boolean procedure open(fileimage); text fileimage;
if ... then begin ...  ! see 10.1.2;
   image:– fileimage;
   setpos(1);
   open:= OPEN:= true;
end open;
```

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
   if pos <> 1 then outimage;
   image:– notext;
   ... ;  ! perform closing actions on external file;
   OPEN:= false;
   close:= true;
end close;
```

The procedure "close" calls "outimage" if the position indicator is not equal to 1.

10.5.2 Outimage

OUTIMAGE

```
procedure outimage;
if not OPEN then error("..." ! file closed; )
else begin
  ... ; ! transfer "image" to external image;
  image:= notext;
  setpos(1)
end outimage;
```

The transfer of an image from the text "image" to the file is performed by the procedure "outimage". The procedure reacts in an implementation-defined way if the "image" length is not appropriate for the external file. Whether or not trailing blanks from the "image" are actually recorded on the external file, is implementation-dependent. After the transfer, "image" is cleared to blanks and the position indicator is set to 1.

10.5.3 Outrecord

OUTRECORD

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

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

10.5.4 Breakoutimage

BREAKOUTIMAGE

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

The procedure "breakoutimage" outputs the part of "image" that precedes POS. The output is performed as a partial output of an external image, in the sense that implicit line terminators are suppressed. On some external files this operation is not possible, in which case the transfer is identical to that of "outrecord".

After transfer the "image" is blanked and POS is set to one.

One use of "breakoutimage" is to allow input from a terminal display on the same line as one on which output (e.g. a prompt) has already been written.

10.5.5 Checkpoint

The procedure "checkpoint" is described in 10.2.1.

10.5.6 Outchar

OUTCHAR

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

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

10.5.7 Outtext

OUTTEXT

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

Procedure "outtext" always transfers the complete contents of the text parameter to the file.

10.5.8 Item-oriented output

```
text procedure FIELD(w); integer w;
if w>length then error("..." ! Item too long; )
else begin
    if pos+w−1 > length then outimage;
    FIELD:− image.sub(pos, w);
    setpos(pos+w)
end FIELD;
```

OUTINT

```
procedure outint(i, w); integer i, w;
if      w = 0 then FIELD(...).putint(i)   ! see below;
else if w < 0
then begin text f;
  f:− FIELD(−w);
  f:= notext;
  f.sub(1, ...).putint(i) end
else FIELD(w).putint(i);
```

OUTFIX

```
procedure outfix(r, n, w); <real−type> r; integer n, w;
... ; ! as body of outint, with "putfix" substituted for "putint";
```

OUTREAL

```
procedure outreal(r, n, w); <real−type> r; integer n, w;
... ; ! as body of outint, with "putreal" substituted for "putint";
```

OUTFRAC

```
procedure outfrac(i, n, w); integer i, n, w;
... ; ! as body of outint, with "putfrac" substituted for "putint";
```

The procedures "outint", "outfix", "outreal" and "outfrac" are defined in terms of the corresponding editing procedures of "image". They provide facilities for "item-oriented" output. Each item is edited into a "field" (subtext of "image") normally starting at the current accessible character. POS is advanced correspondingly. If the remainder of "image" is too short to contain the item, "outimage" is called implicitly prior to the editing operation. The field is space-filled before the editing operation.

A run-time error occurs if a field cannot be contained within the full length of "image".

Parameter "w" determines both the length of this field and the adjustment of the item within it, as follows.

w > 0	The field length is w, the item is right-adjusted.
w < 0	The field length is abs(w), the item is left-adjusted.
w = 0	The field length is the exact number of characters needed to contain the item (i.e. no leading or trailing spaces).

10.6 The class "directfile"

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

      Boolean procedure open(fileimage); text fileimage;  . . . . . 10.6.1;
      Boolean procedure close;  . . . . . 10.6.2;
      integer procedure lastloc;  . . . . . 10.6.2;
      integer procedure maxloc;  . . . . . 10.6.2;
      procedure locate(i); integer i;  . . . . . 10.6.3;
      procedure inimage;  . . . . . 10.6.4;
      procedure outimage;  . . . . . 10.6.5;
      Boolean procedure deleteimage;  . . . . . 10.6.6;
      character procedure inchar;  . . . . . 10.6.7;
      integer procedure lock(t, i, j); real t; integer i, j;  . . . . . 10.6.7;
      Boolean procedure unlock;  . . . . . 10.2.1;
      Boolean procedure checkpoint;  . . . . . 10.4.4;
      Boolean procedure lastitem;  . . . . . 10.4.5;
      text procedure intext;  . . . . . 10.4.6;
      integer procedure inint;  . . . . . 10.4.6;
      long real procedure inreal;  . . . . . 10.4.6;
      integer procedure infrac;  . . . . . 10.5.6;
      procedure outchar(c); character c;  . . . . . 10.5.7;
      procedure outtext(t); text t;  . . . . . 10.5.8;
      text procedure FIELD(w); integer w;  . . . . . 10.5.8;
      procedure outint(i, w); integer i, w;  . . . . . 10.5.8;
      procedure outfix(r, n, w); <real–type> r; integer n, w;  . . . . . 10.5.8;
      procedure outreal(r, n, w); <real–type> r; integer n, w;  . . . . . 10.5.8;
      procedure outfrac(i, n, w); integer i, n, w;  . . . . . 10.5.8;
      ENDFILE:= true;
    end directfile;
```

An object of the class "directfile" is used to represent an image-oriented direct file in which the individual images are addressable by ordinal numbers.

The variable LOC contains the current ordinal number. When the file is closed, the value of LOC is set to zero. The procedure "location" gives access to the current value of LOC.

The variable ENDFILE is **true** when the file is closed or when an image with location greater than "lastloc" has been input (through "inimage"). It is set after each "inimage" statement. The procedure "endfile" returns the current value.

The variable MAXLOC indicates the highest permitted value of LOC. On some systems this value corresponds to the size of a preallocated file while, on other systems which allow the file to be dynamically extended, this variable is assigned the value "maxint"–1.

10.6.1 Open and close

OPEN

```
Boolean procedure open(fileimage);  text fileimage;
if ... then begin ...  ! see 10.1.2;
   MAXLOC:= ... ;  ! See below;
   image:- fileimage;
   setpos(1);
   locate(1);
   open:= OPEN:= true;
end open;
```

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
   image:- notext;
   if LOCKED then unlock;
   LOC:= MAXLOC:= 0;
   ... ;
   OPEN:= false;
   close:= ENDFILE:= true;
end close;
```

The procedure "open" locates the first image of the file. The length of "image" must, at all "inimage" and "outimage" statements, be identical to the length of "image" at the "open" call. The value assigned to MAXLOC at "open" is either a maximum length determined from the external file, or it is "maxint"–1 if no such length exists.

10.6.2 Locate, lastloc, and maxloc

LOCATE

```
procedure locate(i);  integer i;
if i<1 or i>MAXLOC then error("..." ! Parameter out of range; )
else begin
   LOC:= i;
   ... ;
end locate;
```

LASTLOC

```
integer procedure lastloc;
if not OPEN then error("..." ! file closed; )
else  lastloc:= ... ;
```

MAXLOC

```
integer procedure maxloc;
if not OPEN then error("..." ! file closed; )
else  maxloc:= MAXLOC;
```

Procedure "locate" may be used to assign a given value to the variable LOC. This assignment may be accompanied by implementation-defined checks and (possibly asynchronous) instructions to an external memory device associated with the file; no transfer to/from "image" is, however, performed. A parameter to "locate" less than one or greater than MAXLOC constitutes a run-time error.

The procedure "lastloc" indicates the largest location of any written image. For a new file the value returned is zero.

10.6.3 Inimage

INIMAGE

```
procedure inimage;
if not OPEN then error("..." ! file closed; )
else begin
   locate(LOC);
   setpos(1);
   ENDFILE:= LOC > lastloc;
   if ENDFILE then image:= "!25!" else
   begin
     if ... ! external written image at LOC exists ; then
      ... ! transfer to "image";...
     else begin
         while more do image.putchar('!0!')
         ! Note: now pos = length+1;
     end not written;
     LOC:= LOC + 1;   ! Location for *next* image;
   end;
end inimage;
```

The procedure "inimage" transfers into the text "image" a copy of the external image as currently identified by the variable LOC. If the file does not contain an image with an ordinal number equal to the value of LOC, the effect of the procedure "inimage" is as follows. If the location indicated is greater than "lastloc", then ENDFILE is set to **true** and the end of file text ("!25!") is assigned to "image". Otherwise, if the image is a non-written image but there exists at least one written image whose LOC is greater than current LOC, then the "image" is filled with NUL ('!0!') characters and the position indicator is set to "length"+1 (i.e. "more" becomes **false**). Finally the value of LOC is incremented by one.

10.6.4 Outimage

OUTIMAGE

```
procedure outimage;
if  not OPEN then error("..." ! file closed; )
else if LOC > MAXLOC then error("..." ! file overflow; );
else begin
   locate(LOC);
   ... ; ! output "image" to external image at LOC;
   LOC:= LOC + 1;
   image := notext;
   setpos(1)
end outimage;
```

The procedure "outimage" transfers a copy of the text value "image" to the external image, thereby storing in the file an external image whose ordinal number is equal to the current value of LOC. If the file contains another image with the same ordinal number, that image is overwritten. The value of LOC is then incremented by one.

10.6.5 Deleteimage

DELETEIMAGE

```
Boolean procedure deleteimage;
if OPEN and then ... ! image LOC was written;
then begin
  locate(LOC);
  ... ; ! attempt to delete image;
  if ... ! delete operation successful;
  then begin
    deleteimage := true;
    LOC:= LOC + 1;
  end successful
end deleteimage;
```

The Boolean procedure "deleteimage" makes the image identified by the current value of LOC effectively un-written. Irrespective of any physical differences on the external medium between never-written images and deleted ones, there is no difference from the program's point of view. Note that this means that "deleteimage" may decrement the value returned by "lastloc" (in case LOC was equal to "lastloc").

Note: Outputting a NUL-filled image at location "lastloc" in the file does not necessarily decrement the "lastloc" value; explicit writing (outimage) of such images should be avoided.

10.6.6 Inchar

INCHAR

```
character procedure inchar;
begin
  while not more do inimage;
  inchar:= image.getchar
end inchar;
```

Note: Inchar skips all unwritten images.

10.6.7 Lock and Unlock

The procedures "lock" and "unlock" (see 10.2.2) provide locking mechanisms. The last two parameters of "lock" indicate the minimum range of locations to be locked (inclusive).

10.6.8 Item-oriented input/output

The remaining procedures ("lastitem" to "intext" and "outchar" to "outtext") are defined in accordance with the corresponding procedures of "infile" and "outfile" respectively, i.e. their definitional algorithms are exact copies of those given in these two classes.

10.7 The class "printfile"

The class "printfile" defines a class for line printer oriented output.

```
outfile class printfile;
    begin integer LINE, LINES_PER_PAGE, SPACING, PAGE;
    integer procedure line; line := LINE;
    integer procedure page; page := PAGE;
                                                                    . . . . . 10.7.1;
    Boolean procedure open(fileimage); text fileimage;              . . . . . 10.7.1;
    Boolean procedure close;                                        . . . . . 10.7.2;
    integer procedure linesperpage(n); integer n;                   . . . . . 10.7.3;
    procedure spacing(n); integer n;                                . . . . . 10.7.4;
    procedure eject(n); integer n;                                  . . . . . 10.7.5;
    procedure outimage;                                             . . . . . 10.7.5;
    procedure outrecord;
    SPACING:= 1;
    LINES_PER_PAGE:= ... ;
end printfile;
```

An object of the class "printfile" is used to represent a line printer oriented output file. The class is a subclass of "outfile". A file image normally represents a line on a printed page.

It is a property of this class that "outfile" attributes, which are redeclared at "printfile" level, are not accessible to the user's program through explicit qualification (**qua**). Thus these "outfile" procedures ("open", "close", "outimage", "outrecord") may be envisaged as including the following initial code:

```
procedure X...;
inspect this outfile
   when printfile do X...
   otherwise ...;
```

Note: Consequently, possible implicit calls of outimage from outchar, close and the item-oriented output procedures are understood to invoke "printfile.outimage".

The variable LINE indicates the ordinal number of the line on which the next output will be printed (on the current page), provided that no implicit or explicit "eject" statement occurs. Its value is accessible through the procedure "line". Note that the value of LINE may be greater than LINES_PER_PAGE (see 10.7.5).

The variable PAGE indicates the ordinal number of the current page. Its value may be retrieved by means of procedure "page".

10.7.1 Open and close

OPEN

```
Boolean procedure open(fileimage);  text fileimage;
if ... then begin ...  ! see 10.1.2;
  image:- fileimage;
  PAGE:= 0;
  LINE:= 1;
  setpos(1);
  eject(1);
  open:= OPEN := true;
end open;
```

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
  if pos <> 1 then outimage;
  eject(LINES_PER_PAGE);
  LINE:= 0;
  SPACING:= 1;
  LINES_PER_PAGE:= ... ;
  image:- notext;
  ... ;
  OPEN:= false;
  close:= true;
end close;
```

The procedures "open" and "close" conform to the rules of 10.1.2. In addition, "close" outputs the current value of "image" if POS is not equal to 1 and sets LINE to zero.

10.7.2 Lines per page

LINESPERPAGE

```
integer procedure linesperpage(n);  integer n;
begin  linesperpage:= LINES_PER_PAGE;
  LINES_PER_PAGE:=
    if n > 0 then n
    else if n < 0 then maxint
    else  ... ;  ! default value;
end linesperpage;
```

The variable LINES_PER_PAGE indicates the maximum number of physical lines that may be printed on each page, including intervening blank lines. An implementation-defined value is assigned to the variable at the time of object generation, and when the printfile is closed. The value of the variable may be retrieved by a call on "linesperpage"; in addition the variable is given a new value.

If the parameter to "linesperpage" is zero, LINES_PER_PAGE is reset to the original value (assigned at object generation). A parameter value less than zero may be used to indicate an "infinite" value of LINES_PER_PAGE, thus avoiding any automatic calls on "eject".

10.7.3 Spacing

SPACING

```
procedure spacing(n); integer n;
if 0<=n and n<=LINES_PER_PAGE then SPACING:= n
else error("..." ! Parameter out of range; );
```

The variable SPACING represents the value by which the variable LINE is incremented after the next printing operation. Its value may be changed by the procedure "spacing". A call on the procedure "spacing" with a parameter less than zero or greater than LINES_PER_PAGE constitutes an error. The effect of a a parameter to "spacing" which is equal to zero may be defined as forcing successive printing operations on the same physical line. Note, however, that on some physical media this may not be possible, in which case spacing(0) has the same effect as spacing(1) (i.e. no overprinting).

10.7.4 Eject

EJECT

```
procedure eject(n); integer n;
if not OPEN then error("..." ! file closed; )
else if  n <= 0  then error("..." ! Parameter out of range; )
else begin
  if n > LINES_PER_PAGE then n:= 1;
  if n <= LINE then
  begin
    ... ; ! change to new page on external file;
    PAGE:= PAGE + 1
  end;
  ... ; ! move to line "n" on current (external) page;
  LINE:= n
end eject;
```

The procedure "eject" is used to position to a certain line identified by the parameter, n. The variable "PAGE" is incremented by one each time an explicit or implicit "eject" implies a new page.

The following cases can be distinguished:

n	<=	0	: Error
n	>	LINES_PER_PAGE	: Equivalent to eject (1)
n	<=	LINE	: Position to line number n on the next page
n	>	LINE	: Position to line number n on the current page

The tests above are performed in the given sequence.

10.7.5 Outimage and outrecord

OUTIMAGE

```
procedure outimage;
if not OPEN then  error("..." ! file closed; )
else begin
  if LINE > LINES_PER_PAGE then eject(1);
  ... ;  ! output the image on the line indicated by LINE;
  LINE:= LINE + SPACING;
  image:= notext;
  setpos(1)
end outimage;
```

OUTRECORD

```
procedure outrecord;
if not OPEN then  error("..." ! file closed; )
else begin
  if LINE > LINES_PER_PAGE then eject(1);
  ... ;  ! output image.sub(1, pos−1) on the line indicated by LINE;
  LINE:= LINE + SPACING;
  setpos(1)
end outrecord;
```

The procedures "outimage" and "outrecord" operate according to the rules of 10.5.2 and 10.5.3. In addition, they update the variable LINE (and possibly PAGE).

Note: In addition, the procedure "breakoutimage" is inherited from the class prefix "outfile". This procedure does not update LINE or PAGE.

10.8 Bytefiles

The class bytefile is the common prefix class for all byte-oriented files.

```
file class bytefile;
    begin short integer BYTESIZE;

        short integer procedure bytesize;  bytesize := BYTESIZE;

    end bytefile;
```

Bytefiles read and write files as continuous streams of bytes. The variable BYTESIZE defines the range of the byte values transferred. Byte values are integers in the range (0:2**BYTESIZE−1). The BYTESIZE value of the file object is accessible through the procedure "bytesize".

Note: The procedure "bytesize" returns zero before first "open" of the bytefile.

There are three standard subclasses of "bytefile":

"inbytefile"	representing a sequential file for which input operations are available.
"outbytefile"	representing a sequential file for which output operations are available.
"directbytefile"	representing a direct file with facilities for both input and output.

10.9 The class "inbytefile"

```
bytefile class inbytefile;
    begin Boolean ENDFILE;
        Boolean procedure endfile;   endfile:= ENDFILE;                    . . . . . 10.9.1;

        Boolean procedure open;      . . . . . 10.9.1;
        Boolean procedure close;     . . . . . 10.9.2;
        short integer procedure inbyte; . . . . . 10.9.3;
        text procedure intext(t);    text t;  . . . . . .
        ENDFILE:= true;
    end inbytefile;
```

An object of the class "inbytefile" is used to represent a byte-oriented sequential input file.

The variable "ENDFILE" is **true** if there are no more bytes to read. The procedure "endfile" returns the value of ENDFILE.

10.9.1 Open and close

OPEN

```
Boolean procedure open;
if ... then begin ...  ! see 10.1.2;
   ENDFILE:= false;
   BYTESIZE:= ... ! value of access mode BYTESIZE;
   open:= OPEN:= true;
end open;
```

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
   ... ;
   OPEN:= false;
   close:= ENDFILE := true;
end close;
```

10.9.2 Inbyte

INBYTE

```
short integer procedure inbyte;
if ENDFILE then error("..." ! End of file ;)
else if ... ! no more bytes to read;
then ENDFILE:= true  ! inbyte returns zero;
else inbyte:= ...! next byte of size BYTESIZE;
```

The procedure "inbyte" returns the **short integer** value corresponding to the input byte. If there are no more bytes to read, a zero result is returned. If prior to an "inbyte" call ENDFILE is **true**, a run-time error occurs. The result is always in the range 0 <= inbyte < 2**BYTESIZE.

10.9.3 Intext

INTEXT

```
text procedure intext(t);    text t;
begin
   t.setpos(1);
   while t.more and not ENDFILE do t.putchar(char(inbyte));
   if ENDFILE then t.setpos(t.pos−1);
   intext:− t.sub(1, t.pos−1)
end intext;
```

The procedure "intext" fills the frame of the parameter "t" with successive input bytes.

10.10 The class "outbytefile"

```
bytefile class outbytefile;
   begin
      Boolean procedure open;  . . . . . 10.10.1;
      Boolean procedure close;  . . . . . 10.10.1;
      procedure outbyte(x); short integer x;  . . . . . 10.10.2;
      procedure outtext(t); text t;  . . . . . 10.10.3;
      Boolean procedure checkpoint;  . . . . . 10.2.1;
   end outbytefile;
```

An object of the class "outbytefile" is used to represent a sequential output file of bytes.

10.10.1 Open and close

OPEN

```
Boolean procedure open;
if ... then begin ...  ! see 10.1.2;
   BYTESIZE:= ... ! value of access mode BYTESIZE;
   open:= OPEN:=true;
end open;
```

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
   OPEN:= false;
   close:= true;
end close;
```

10.10.2 Outbyte

OUTBYTE

```
procedure outbyte(x);    short integer x;
if not OPEN then error("..." ! file closed; )
else if x < 0 or else x >= 2**BYTESIZE
then error("..." ! Outbyte, illegal byte value ;)
else ... ;  ! output of x;
```

The procedure "outbyte" outputs a byte corresponding to the parameter value. If the parameter value is less than zero or exceeds the maximum permitted value, as defined by BYTESIZE, a run-time error occurs. If the file is not open, a run-time error occurs.

10.10.3 Outtext

OUTTEXT

```
procedure outtext(t);   text t;
begin
   t.setpos(1);
   while t.more do  outbyte(rank(t.getchar))
end outtext;
```

The procedure "outtext" outputs all characters in the parameter "t" as bytes.

10.11 The class "directbytefile"

The class "directbytefile" defines a byte-oriented direct file.

```
bytefile class directbytefile;
    begin integer LOC, MAXLOC;  Boolean LOCKED;
        Boolean procedure endfile;  endfile := OPEN and then LOC>lastloc;
        integer procedure location;  location := LOC;
        integer procedure maxloc;  maxloc := MAXLOC;
        Boolean procedure locked;  locked := LOCKED;

        Boolean procedure open; . . . . . 10.11.1;
        Boolean procedure close; . . . . . 10.11.1;
        integer procedure lastloc; . . . . . 10.11.2;
        procedure locate(i);  integer i; . . . . . 10.11.2;
        short integer procedure inbyte; . . . . . 10.11.3;
        procedure outbyte(x);  short integer x; . . . . . 10.11.3;
        Boolean procedure checkpoint; . . . . . 10.2.1;
        integer procedure lock(t, i, j);  real t;  integer i, j; . . . . . 10.2.2;
        Boolean procedure unlock; . . . . . 10.2.2;
        procedure intext(t);  text t; . . . . . 10.9.3;
        procedure outtext(t);  text t; . . . . . 10.10.3;
    end directbytefile;
```

An object of the class "directbytefile" is used to represent an external file in which the individual bytes are addressable by ordinal numbers. The variable LOC is defined to represent such ordinal numbers. When the file is closed, the value of LOC is zero.

The variable MAXLOC indicates the maximum possible location on the external file. If this is not meaningful MAXLOC has the value of "maxint"−1. The procedure "maxloc" gives access to the current MAXLOC value.

The procedure "endfile" returns **true** whenever LOC indicates an address greater than "lastloc".

The procedures "intext" and "outtext" conform to the pattern for "inbytefile" and "outbytefile", respectively.

10.11.1 Open and close

OPEN

```
Boolean procedure open;
if ... then begin ...  ! see 10.1.2;
  LOC:= 1;
  MAXLOC:= ...;  ! fixed size, or maxint−1;
  BYTESIZE:= ...;  ! value of access mode BYTESIZE;
  open:= OPEN:= true;
end open;
```

CLOSE

```
Boolean procedure close;
if ... then begin ...  ! see 10.1.2;
  MAXLOC:= 0;
  OPEN:= false;
  close:= true;
end close;
```

10.11.2 Locate and lastloc

LOCATE

```
procedure locate(i);  integer i;
if i < 1 or i > MAXLOC
then error("..." ! Parameter out of range; )
else LOC := i;
```

LASTLOC

```
integer procedure lastloc;
if not OPEN then error("..." ! file closed; )
else lastloc := ...;
```

The current last written location is returned by the procedure "lastloc". The procedure "location" returns the current value of LOC. The procedure "locate" may be used to assign a given value to the variable. A parameter value to "locate" which is less than one or greater than MAXLOC constitutes a run-time error.

10.11.3 Inbyte and outbyte

INBYTE

```
short integer procedure inbyte;
if not OPEN then error("..." ! file closed; )
else if LOC <= lastloc
then begin
  inbyte:= ... ! next byte of size BYTESIZE;...;
  LOC:= LOC+1
end inbyte;
```

OUTBYTE

```
procedure outbyte(x);  short integer x;
if not OPEN then error("..." ! file closed; )
else if x < 0 or else x >= 2**BYTESIZE
then error("..." ! Outbyte, illegal byte value ;);
else if LOC > MAXLOC then error("..." ! file overflow; )
else begin
  ... ! output of x;
  LOC := LOC + 1
end outbyte;
```

The procedure "inbyte" reads one byte, returning its integer value. The result of "inbyte" from an unwritten LOC is zero (cf. 10.9.2).

The procedure "outbyte" outputs a byte according to the given parameter value (cf. 10.10.2).

10.11.4 Lock and unlock

The procedures "lock" and "unlock" (see 10.2.2) provide locking mechanisms. The last two parameters of "lock" indicate the minimum range of (byte) locations to be locked (inclusive).

11 CLASS SIMSET

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

```
class simset;
begin

   class linkage; . . . . . 11.1;
   linkage class link; . . . . . 11.2;
   linkage class head; . . . . . 11.3;
end simset;
```

The reference variables and procedures necessary for set handling are introduced in standard classes declared within the class "simset". Using these classes as prefixes, their relevant data and other properties are made parts of the object themselves.

Both sets and objects which may acquire set membership have references to a successor and a predecessor. Consequently they are made subclasses of the "linkage" class.

The sets are represented by objects belonging to a subclass "head" of "linkage". Objects which may be set members belong to subclasses of "link" which is itself another subclass of "linkage".

11.1 Class "linkage"

```
class linkage;
begin ref (linkage) SUC, PRED;

   ref (link) procedure suc;
         suc:– if SUC in link then SUC else none;

   ref (link) procedure pred;
         pred:– if PRED in link then PRED else none;

   ref (linkage) procedure prev;   prev:– PRED;

end linkage;
```

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

SUC is a reference to the successor of 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 give the value **none** if the designated object is not a set member, i.e. of class "link" or a subclass of "link".

The attributes SUC and PRED may only be modified through the use of procedures defined within "link" and "head". This protects the user against certain kinds of programming errors.

The procedure "prev" enables a user to access a set head from its first member.

11.2 Class "link"

```
linkage class link;
      begin

         procedure out;
         if SUC=/=none then begin
            SUC.PRED:- PRED;
            PRED.SUC:- SUC;
            SUC:- PRED:- none
         end out;

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

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

         procedure into(S); ref (head) S;   precede(S);

      end link;
```

Objects belonging to subclasses of the class "link" may acquire set membership. An object may only be a member of one set at a given instant.

In addition to the procedures "suc" and "pred", there are four procedures associated with each "link" object: "out", "follow", "precede" and "into".

The procedure "out" removes the object from the set (if any) of which it is a member. The procedure call has no effect if the object has no set membership.

The procedures "follow" and "precede" 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 has 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 is inserted immediately after ("follow") or before ("precede") the "linkage" object designated by the parameter.

The procedure "into" removes the object from the set (if any) of which it is a member and inserts it as the last member of the set designated by the parameter. The procedure call has the same effect as "out" if the parameter has the value **none** (except for possible side effects from evaluation of the parameter).

11.3 Class "head"

```
linkage class head;
      begin

            ref (link) procedure first;      first:– suc;

            ref (link) procedure last;       last:– pred;

            Boolean procedure empty;         empty:= SUC == this linkage;

            integer procedure cardinal;
            begin integer i;
               ref (link) ptr;
               ptr:– first;
               while ptr =/= none do begin
                  i:= i+1;
                  ptr:– ptr.suc
               end while;
               cardinal:= i
            end cardinal;

            procedure clear;   while first =/= none do first.out;

               SUC:– PRED:– this linkage
      end head;
```

An object of the class "head", or a subclass of "head" is used to represent a set. "head" objects may not acquire set membership. Thus, a unique "head" is defined for each set.

The procedure "first" may be used to obtain a reference to the first member of the set, while the procedure "last" may be used to obtain a reference to the last member.

The **Boolean** procedure "empty" gives the value **true** only if the set has no members.

The **integer** procedure "cardinal" gives the number of members in a set.

The procedure "clear" removes all members from the set.

The references SUC and PRED initially point to the "head" itself, which thereby represents an empty set.

12 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 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 acts as the head of a quasi-parallel system which may represent a "discrete event" simulation model.

```
simset class simulation;
    begin  ref (head) SQS;

    link class EVENT_NOTICE (EVTIME, PROC);  long real EVTIME;  ref (process) PROC;
      begin
        ref (EVENT_NOTICE) procedure suc;
          suc:– if SUC is EVENT_NOTICE then SUC else none;

        ref (EVENT_NOTICE) procedure pred;    pred:– PRED;

        procedure RANK_IN_SQS (afore);  Boolean afore;
        begin ref (EVENT_NOTICE) evt;
          evt:– SQS.last;
          while evt.EVTIME > EVTIME do     evt :– evt.pred;
          if afore then
            while evt.EVTIME = EVTIME do evt :– evt.pred;
          follow(evt)
        end RANK_IN_SQS;
      end EVENT_NOTICE;

    ref (MAIN_PROGRAM) main;

    ref (EVENT_NOTICE) procedure FIRSTEV;   FIRSTEV:– SQS.first;
    ref (process) procedure current;        current:– FIRSTEV.PROC;
    long real procedure time;               time:= FIRSTEV.EVTIME;

    link class process; . . . . . 12.1;
    procedure ACTIVAT ... . . . . 12.3;
    procedure hold(T);  long real T; . . . . . 12.4;
    procedure passivate; . . . . . 12.4;
    procedure wait(S);  ref (head) S; . . . . . 12.4;
    procedure cancel(X);  ref (process) X; . . . . . 12.5;
    process class MAIN_PROGRAM; . . . . . 12.6;
    procedure accum(a, b, c, d);  name a, b, c;  long real a, b, c, d;

    SQS:– new head;
    main:– new MAIN_PROGRAM;
    main.EVENT:– new EVENT_NOTICE(0, main);
    main.EVENT.into(SQS)
end simulation;
```

When used as a prefix to a block or a class, "simulation" introduces simulation-oriented features through the class "process" and associated procedures.

The variable SQS refers to a set which is called the "sequencing set", and serves to represent the system time axis. The members of the sequencing set are event notices ranked according to increasing value 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".

Note: Since the statements and procedures introduced by "simulation" make implicit use of the sequencing procedures (detach, call and resume) explicit sequencing by these procedures should be done with care.

The sequencing set (SQS) is described here, for purposes of illustration, as a simset list. In actual implementations, however, a more efficient representation is recommended.

12.1 Class "process"

```
link class process;
    begin ref (EVENT_NOTICE) EVENT;
      Boolean TERMINATED;
      Boolean procedure idle;          idle:= EVENT==none;
      Boolean procedure terminated;    terminated:= TERMINATED;

      long real procedure evtime;
        if idle then  error("..." ! No Evtime for idle process)
        else evtime:= EVENT.EVTIME;

      ref (process) procedure nextev;
        nextev:- if  idle or else EVENT.suc == none then none
                 else EVENT.suc.PROC;

      detach;
      inner;
      TERMINATED:= true;
      passivate;
      error("..." ! Terminated process;)
    end process;
```

An object of a class prefixed by "process" is 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 and its reactivation point positioned in front of the first statement of its user-defined operation rule. The process object remains detached throughout its dynamic scope.

The procedure "idle" has the value **true** if the process object is not currently represented in the sequencing set. It is said to be in the passive or terminated state depending on the value of the procedure "terminated". An idle process object is passive if its reactivation point is at a user-defined prefix level. If and when the PSC passes through the final **end** of the user-defined part of the body, it proceeds to the final operations at the prefix level of the class "process", and the value of the procedure "terminated" becomes **true**. (Although the process state "terminated" is not strictly equivalent to the corresponding basic concept defined in chapter 7, 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", unless 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" references the process object, if any, represented by the next event notice in the sequencing set.

12.2 Activation statement

activation–statement
= *activation–clause* [*scheduling–clause*]

activation–clause
= *activator object–expression*

activator
= **activate** | **reactivate**

scheduling–clause
= *timing–clause*
| ⟨ **before** | **after** ⟩ *object–expression*

timing–clause
= *simple–timing–clause* [**prior**]

simple–timing–clause
= ⟨ **at** | **delay** ⟩ *arithmetic–expression*

An activation statement is only valid within an object of a class included in "simulation", or within a prefixed block whose prefix part is such an object.

The effect of an activation statement is defined as being that of a call on the sequencing procedure ACTIVAT local to "simulation".

```
procedure ACTIVAT (REAC, X, CODE, T, Y, PRIO);
ref (process) X, Y; Boolean REAC, PRIO; text CODE; long real T;
```

The actual parameter list is determined from the form of the activation statement, by the following rules.

1) The actual parameter corresponding to REAC is **true** if the activator is **reactivate**, **false** otherwise.

2) The actual parameter corresponding to X is the object expression of the activation clause.

3) The actual parameter corresponding to T is the arithmetic expression of the simple timing clause if present, otherwise it is zero.

4) The actual parameter corresponding to PRIO is **true** if **prior** is in the timing clause, **false** if it is not used or 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
- absent -	"direct"
at arithmetic expression	"at"
delay arithmetic expression	"delay"
before object expression	"before"
after object expression	"after"

12.3 Procedure ACTIVAT

```
procedure ACTIVAT(REAC, X, CODE, T, Y, PRIO);
   ref (process) X, Y;
   Boolean REAC, PRIO;  text CODE;  long real T;
inspect X do if not TERMINATED then
begin  ref (process) z;  ref (EVENT_NOTICE) EV;
    if REAC then EV:- EVENT
    else if EVENT =/= none then goto exit;
    z:- current;
    if      CODE = "direct" then
direct:     begin EVENT:- new EVENT_NOTICE(time,X);
               EVENT.precede(FIRSTEV)
            end direct
    else if CODE = "delay" then
            begin T:= T + time;
                  goto at_
            end delay
    else if CODE = "at" then
at_:        begin if T < time then T:= time;
              if T = time and PRIO then goto direct;
                 EVENT:- new EVENT_NOTICE(T, X);
                 EVENT.RANK_IN_SQS(PRIO)
            end at
    else if Y == none or else Y.EVENT == none
    then EVENT:- none
    else begin
      if X == Y then goto exit;
          comment reactivate X before/after X;
      EVENT:- new EVENT_NOTICE(Y.EVENT.EVTIME, X);
      if CODE = "before" then EVENT.precede(Y.EVENT)
                         else EVENT.follow(Y.EVENT)
    end before or after;
    if EV =/= none
    then begin EV.out;  if SQS.empty then error("...") end;
    if z =/= current then resume(current);
exit:
  end ACTIVAT;
```

The procedure ACTIVAT represents an activation statement, as described in 12.2. 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 has 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 into 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 the 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 takes 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)".

12.4 Sequencing procedures

HOLD

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

PASSIVATE

```
procedure passivate;
begin
  inspect current do begin EVENT.out; EVENT:- none end;
  if SQS.empty then error("...") else resume(current)
end passivate;
```

WAIT

```
procedure wait(S); ref (head) S;
begin current.into(S); passivate end wait;
```

CANCEL

```
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;
```

The sequencing procedures serve to organize the quasi-parallel operation of process objects in a simulation model. Explicit use of the basic sequencing facilities (call, detach, resume) should be made only after thorough consideration of its effects.

The statement "hold(T)", where T is a long real number greater than or equal to zero, halts the active phase of the currently active process object, and schedules 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" stops the active phase of the currently active process object and deletes 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" includes the currently active process object in a referenced set, and then calls the procedure "passivate".

The statement "cancel(X)", where X is a reference to a process object, deletes the corresponding event notice, if any. If the process object is currently active or suspended, it becomes passive. Otherwise, the statement has no effect. The statement "cancel(current)" is equivalent to "passivate".

12.5 The main (simulation) program

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

It is desirable that the main component of a simulation model, i.e. the "simulation" block instance, should respond to the sequencing procedures of 12.4 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 represents the main component with respect to the sequencing procedures. Whenever it becomes operative, the PSC immediately enters the main component as a result of the "detach" statement (cf. 7.3.1). The procedure "current" references this process object whenever the main component is active.

A simulation model is initialized by generating the MAIN_PROGRAM object and scheduling an active phase for it at system time zero. Then the PSC proceeds to the first user-defined statement of the "simulation" block.

12.6 The procedure "accum"

ACCUM

```
procedure accum (a, b, c, d);  name a, b, c;  long real a, b, c, d;
begin
  a:= a + c * (time−b);  b:= time;  c:= c + d
end accum;
```

A statement of the form "accum (A, B, C, D)" may be used to accumulate the "system time integral" of the variable C, interpreted as a step function of system time. The integral is accumulated in the variable A. The variable B contains the system time at which the variables were last updated. The value of D is the current increment of the step function.

Annex A SIMULA Syntax

Chapter 1: Lexical tokens

letter
= A | B | C | D | E | F | G | H | I
| J | K | L | M | N | O | P | Q | R
| S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i
| j | k | l | m | n | o | p | q | r
| s | t | u | v | w | x | y | z

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

space
= SP

identifier
= *letter* { *letter* | *digit* | _ }

unsigned–number
= *decimal–number* [*exponent–part*] | *exponent–part*

decimal–number
= *unsigned–integer* [*decimal–fraction*] | *decimal–fraction*

decimal–fraction
= . *unsigned–integer*

exponent–part
= ⟨ & | && ⟩ [+ | –] *unsigned–integer*

unsigned–integer
= *digit* { *digit* | _ *digit* } | *radix* R *radix–digit* { *radix–digit* | _ *radix–digit* }

radix
= 2 | 4 | 8 | 16

radix–digit
= *digit* | A | B | C | D | E | F

string
= *simple–string* { *string–separator simple–string* }

string–separator
= *token–separator* { *token–separator* }

simple–string
= " { *isocode* | *non–quote–character* | "" } "

isocode
= ! *digit* [*digit*] [*digit*] !

non–quote–character
= ... *any printing character (incl. space) except the string quote* ".

character–constant
= ' *character–designator* '

character–designator
= *isocode* | *non–quote–character* | "

format–effector
= BS | HT | LF | VT | FF | CR

token–separator
= ... *a direct comment* | ... *a space (except in simple strings and character constants)*
| ... *a format effector (except as noted for spaces)* | ... *the separation of consecutive lines*

Chapter 2: Types and values

type
= *value–type* | *reference–type*

value–type
= *arithmetic–type* | **Boolean** | **character**

arithmetic–type
= *integer–type* | *real–type*

integer–type
= [**short**] **integer**

real–type
= [**long**] **real**

reference–type
= *object–reference–type* | **text**

object–reference–type
= **ref** (*qualification*)

qualification
= *class–identifier*

Chapter 3: Expressions

expression
= *value–expression* | *reference–expression* | *designational–expression*

value–expression
= *arithmetic–expression* | *Boolean–expression* | *character–expression*

reference–expression
= *object–expression* | *text–expression*

variable
= *simple–variable–1* | *subscripted–variable*

simple–variable–1
= *identifier–1*

subscripted–variable
= *array–identifier–1* (*subscript–list*)

array–identifier–1
= *identifier–1*

subscript–list
= *subscript–expression* [, *subscript–expression*]

subscript–expression
= *arithmetic–expression*

function–designator
= *procedure–identifier–1* [*actual–parameter–part*]

procedure–identifier–1
= *identifier–1*

actual–parameter–part
= (*actual–parameter* { , *actual–parameter* })

actual–parameter
= *expression* | *array–identifier–1* | *switch–identifier* | *procedure–identifier–1*

identifier–1
= *identifier* | *remote–identifier*

remote–identifier
= *simple–object–expression* . *attribute–identifier* | *text–primary* . *attribute–identifier*

attribute–identifier
= *identifier*

Boolean–expression
= *simple–Boolean–expression*
| *if–clause simple–Boolean–expression* **else** *Boolean–expression*

simple–Boolean–expression
= *Boolean–tertiary* { **or else** *Boolean–tertiary* }

Boolean–tertiary
= *equivalence* { **and then** *equivalence* }

equivalence
= *implication* { **eqv** *implication* }

implication
= *Boolean–term* { **imp** *Boolean–term* }

Boolean–term
= *Boolean–factor* { **or** *Boolean–factor* }

Boolean–factor
= *Boolean–secondary* { **and** *Boolean–secondary* }

Boolean–secondary
= [**not**] *Boolean–primary*

Boolean–primary
= *logical–value* | *variable* | *function–designator* | *relation* | (*Boolean–expression*)

relation
= *arithmetic–relation* | *character–relation* | *text–value–relation*
| *object–relation* | *object–reference–relation* | *text–reference–relation*

value–relational–operator
= < | <= | = | >= | > | <>

reference–comparator
= == | =/=

arithmetic–relation
= *simple–arithmetic–expression* *value–relational–operator* *simple–arithmetic–expression*

character–relation
= *simple–character–expression* *value–relational–operator* *simple–character–expression*

text–value–relation
= *simple–text–expression* *value–relational–operator* *simple–text–expression*

object–relation
= *simple–object–expression* **is** *class–identifier* | *simple–object–expression* **in** *class–identifier*

object–reference–relation
= *simple–object–expression* *reference–comparator* *simple–object–expression*

text–reference–relation
= *simple–text–expression* *reference–comparator* *simple–text–expression*

arithmetic–expression
= *simple–arithmetic–expression*
| *if–clause simple–arithmetic–expression* **else** *arithmetic–expression*

simple–arithmetic–expression
= [+ | –] *term* { ⟨ + | – ⟩ *term* }

term
= *factor* { ⟨ * | / | // ⟩ *factor* }

factor
= *primary* { ** *primary* }

primary
= *unsigned–number* | *variable* | *function–designator* | (*arithmetic–expression*)

character–expression
= *simple–character–expression*
| *if–clause simple–character–expression* **else** *character–expression*

simple–character–expression
= *character–constant* | *variable* | *function–designator* | (*character–expression*)

text–expression
= *simple–text–expression* | *if–clause simple–text–expression* **else** *text–expression*

simple–text–expression
= *text–primary* { *text–primary* }

text–primary
= **notext** | *string* | *variable* | *function–designator* | (*text–expression*)

object–expression
= *simple–object–expression* | *if–clause simple–object–expression* **else** *object–expression*

simple–object–expression
= **none** | *variable* | *function–designator* | *object–generator*
| *local–object* | *qualified–object* | (*object–expression*)

object–generator
= **new** *class–identifier* [*actual–parameter–part*]

local–object
= **this** *class–identifier*

qualified–object
= *simple–object–expression* **qua** *class–identifier*

designational–expression
= *simple–designational–expression*
| *if–clause simple–designational–expression* **else** *designational–expression*

simple–designational–expression
= *label* | *switch–designator* | (*designational–expression*)

switch–designator
= *switch–identifier* (*subscript–expression*)

switch–identifier
= *identifier*

label
= *identifier*

Chapter 4: Statements

statement
= { *label* : } *unconditional–statement*
| { *label* : } *conditional–statement*
| { *label* : } *for–statement*

unconditional–statement
= *assignment–statement* | *while–statement* | *goto–statement*
| *procedure–statement* | *object–generator* | *connection–statement*
| *compound–statement* | *block* | *dummy–statement* | *activation–statement*

assignment–statement
= *value–assignment* | *reference–assignment*

value–assignment
= *value–left–part* := *value–right–part*

value–left–part
= *destination* | *simple–text–expression*

value–right–part
= *value–expression* | *text–expression* | *value–assignment*

destination
= *variable* | *procedure–identifier*

reference–assignment
= *reference–left–part* :– *reference–right–part*

reference–left–part
= *destination*

reference–right–part
= *reference–expression* | *reference–assignment*

conditional–statement
= *if–clause* { *label* : } *unconditional–statement* [**else** *statement*]
| *if–clause* { *label* : } *for–statement*

if–clause
= **if** *Boolean–expression* **then**

while–statement
= **while** *Boolean–expression* **do** *statement*

for–statement
= *for–clause statement*

for–clause
= **for** *simple–variable for–right–part* **do**

simple–variable
= *identifier*

for–right–part
= := *value–for–list–element* { , *value–for–list–element* }
= :– *reference–for–list–element* { , *reference–for–list–element* }

value–for–list–element
= *value–expression* [**while** *Boolean–expression*]
| *text–expression*
| *arithmetic–expression* **step** *arithmetic–expression* **until** *arithmetic–expression*

reference–for–list–element
= *reference–expression* [**while** *Boolean–expression*]

goto–statement
= ⟨ **goto** | **go to** ⟩ *designational–expression*

procedure–statement
= *procedure–identifier–1* [*actual–parameter–part*]

connection–statement
= **inspect** *object–expression when–clause* { *when–clause* } [*otherwise–clause*]
| **inspect** *object–expression* **do** *connection–block–2* [*otherwise–clause*]

when–clause
= **when** *class–identifier* **do** *connection–block–1*

otherwise–clause
= **otherwise** *statement*

connection–block–1
= *statement*

connection–block–2
= *statement*

compound–statement
= **begin** *compound–tail*

compound–tail
= *statement* { ; *statement* } **end**

block
= *subblock* | *prefixed–block*

subblock
= *block–head* ; *compound–tail*

block–head
= **begin** *declaration* { ; *declaration* }

prefixed–block
= *block–prefix* *main–block*

block–prefix
= *class–identifier* [*actual–parameter–part*]

main–block
= *block* | *compound–statement*

dummy–statement
= *empty*

Chapter 5: Declarations

declaration
= *simple–variable–declaration* | *array–declaration* | *switch–declaration*
| *procedure–declaration* | *class–declaration* | *external–declaration*

simple–variable–declaration
= *type* *type–list*

type–list
= *type–list–element* { , *type–list–element* }

type–list–element
= *identifier* | *constant–element*

array–declaration
= [*type*] **array** *array–segment* { , *array–segment* }

array–segment
= *array–identifier* { , *array–identifier* } (*bound–pair–list*)

array–identifier
= *identifier*

bound–pair–list
= *bound–pair* { , *bound–pair* }

bound–pair
= *arithmetic–expression* : *arithmetic–expression*

switch–declaration
= **switch** *switch–identifier* := *switch–list*

switch–list
= *designational–expression* { , *designational–expression* }

procedure–declaration
= [*type*] **procedure** *procedure–heading* ; *procedure–body*

procedure–heading
= *procedure–identifier* [*formal–parameter–part* ; [*mode–part*] *specification–part*]

procedure–body
= *statement*

procedure–identifier
= *identifier*

formal–parameter–part
= (*formal–parameter* { , *formal–parameter* })

formal–parameter
= *identifier*

specification–part
= *specifier identifier–list* ; { *specifier identifier–list* ; }

specifier
= *type* [**array** | **procedure**] | **label** | **switch**

mode–part
= *name–part* [*value–part*] | *value–part* [*name–part*]

name–part
= **name** *identifier–list* ;

value–part
= **value** *identifier–list* ;

identifier–list
= *identifier* { , *identifier* }

class–declaration
= [*prefix*] *main–part*

prefix
= *class–identifier*

main–part
= **class** *class–identifier* [*formal–parameter–part* ; [*value–part*] *specification–part*] ;
[*protection–part* ;] [*virtual–part* ;] *class–body*

class–identifier
= *identifier*

class–body
= *statement* | *split–body*

split–body
= *initial–operations inner–part final–operations*

initial–operations
= ⟨ **begin** | *block–head* ; ⟩ { *statement* ; }

inner–part
= { *label* : } **inner**

final–operations
= **end** | ; *compound–tail*

virtual–part
= **virtual** : *virtual–spec* ; { *virtual–spec* ; }

virtual–spec
= *specifier identifier–list* | **procedure** *procedure–identifier procedure–specification*

protection–part
= *protection–specification* { ; *protection–specification* }

protection–specification
= **hidden** *identifier–list* | **protected** *identifier–list*
| **hidden protected** *identifier–list* | **protected hidden** *identifier–list*

constant–element
= *identifier* = *value–expression* | *identifier* = *text–expression*

Chapter 6: Program modules

SIMULA–source–module
= [*external–head*] ⟨ *program* | *procedure–declaration* | *class–declaration* ⟩ [;]

external–head
= *external–declaration* ; { *external–declaration* ; }

external–declaration
= *external–procedure–declaration* | *external–class–declaration*

program
= *statement*

external–procedure–declaration
= **external** [*kind*] [*type*] **procedure** *external–list*
| **external** *kind* **procedure** *external–item procedure–specification*

kind
= *identifier*

procedure–specification
= **is** *procedure–declaration*

external–class–declaration
= **external class** *external–list*

external–list
= *external–item* { , *external–item* }

external–item
= *identifier* [= *external–identification*]

external–identification
= *string*

Chapter 12: Class Simulation

activation–statement
= *activation–clause* [*scheduling–clause*]

activation–clause
= *activator object–expression*

activator
= **activate** | **reactivate**

scheduling–clause
= *timing–clause* | ⟨ **before** | **after** ⟩ *object–expression*

timing–clause
= *simple–timing–clause* [**prior**]

simple–timing–clause
= ⟨ **at** | **delay** ⟩ *arithmetic–expression*

Annex B Implementation Aspects

According to section 0.5.1.3 of the Standard, an implementation of the language must document all implementation-dependent and implementation-defined issues. This annex summarizes all characteristics of the language which in some manner depend upon the particular implementation. It may thus be used as a checklist to be filled out for any given implementation.

This annex should also be useful when preparing programs intended to be independent of any particular SIMULA implementation.

B.1 Language extensions

Extensions are allowed only if the following conditions are fulfilled:

a) The implementor provides a translator program, which takes any program accepted by that implementation and translates it into a valid program. The resulting 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 the language.

b) Each implementation has a switch which must be set to make the compiler accept programs with extensions.

   An implementation which allows extensions, shall give warning messages for the use of such extensions.

c) All such 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.

File access modes and access mode values additional to those defined in 10.1.1 may be defined by an implementation.

An implementation may provide additional subclasses of class file.

B.2 Allowed implementation restrictions

It is recognized that all language processing systems have some restrictions due to, e.g., limited capacity of the underlying hardware. Such restrictions are not mentioned here (but cf. 0.5.2).

1. An implementation may restrict the value range of the isocode construct, and the character set defined in table 1.1, as long as the "basic" characters of the table are included.
2. An implementation may restrict the number of different block levels at which a system class may be used as a prefix.
3. An implementation may restrict, in any way, the use of "file" and its subclasses for prefixing or block prefixing.
4. The type **short integer** is allowed to be unsupported by an implementation in the sense that it must then be mapped onto **integer** (i.e. the keyword **short** is ignored).
5. The type **long real** is allowed to be unsupported by an implementation in the sense that it must then be mapped onto **real** (i.e. the keyword **long** is ignored, and the special symbol "&&" is treated as "&").
6. An implementation may restrict the use of the standard access modes and their possible standard values. In that case, the procedure "setaccess" must treat such values as unrecognized, and return **false**.
7. An implementation may restrict the number of block levels at which an external class declaration may occur.
8. An implementation may restrict the possible values of the access mode BYTESIZE in any way.

B.3 Implementation-dependent characteristics

1. Whether or not the procedure "terminate_program" will close open external files (except those associated with sysin and sysout), is implementation-dependent.
2. The effect of a parameter to printfile.spacing with value zero, may be device and implementation-dependent, if the standard effect of "overprint" cannot be achieved.
3. The interpretation of directive lines (apart from the "%-space"-convention) is implementation-dependent.
4. The interpretation of "kind" and of the external identification string in an external procedure declaration is implementation-dependent, as is the identification of a separately compiled module if no external identification is given.
5. It is implementation-dependent whether trailing blanks of image are actually transferred to the external file on outfile.outimage.
6. The size of the part of the file that is actually locked after a call to procedure "lock" is implementation-dependent.

B.4 Implementation-defined characteristics

1. The internal character set is implementation-defined. An implementation is required to document the translation between the internal character set and the standard character set (as defined by ISO 646).

Note: The collating sequence of character (and text) values is decided by the internal character set. Thus the values of character and text relations are implementation-defined.

2. The values of "inlength" and "outlength" (see ch. 10) are implementation-defined.
3. The actual external files connected to SYSIN and SYSOUT are implementation-defined.
4. The relative value ranges of **real** and **long real** are implementation-defined.
5. The ranges in which conversions of an integer type to a real type, or from **real** to **long real**, are exact, are implementation-defined.
6. The range of a numeric item in a de-editing procedure is implementation-defined.
7. If the REAL ITEM located by text procedure "getreal" is an integer within an implementation-defined range, the conversion is exact.
8. The EXPONENT of the numeric item resulting from "putreal" has a fixed, implementation-defined number of characters.
9. The maximum length of a text frame is implementation-defined.
10. The function values of "char" and "rank" are implementation-defined.
11. The exact definitions of the standard mathematical functions are implementation-defined.
12. The association between a file object and an external file is implementation-defined.
13. The effect of several file objects representing the same (external) file is implementation-defined.
14. The details of procedures "open" and "close" are implementation-defined.
15. The interpretation of a function value of "lock" less than −1 is implementation-defined.
16. Outimage of outfile reacts in an implementation-defined way if the length of the internal image is incompatible with the format of the external file.
17. Locate of directfile may invoke implementation-defined checks and possibly instructions to an external memory device.
18. LINES_PER_PAGE of printfile: the value at object generation and after close is implementation-defined.

19. The "basic random drawing" algorithm is implementation-defined.

20. The effect of the conditions demanded for the parameters to "linear" not being fulfilled, is implementation-defined.

21. The number of decimals in the field for seconds of the function "datetime" is implementation-defined.

22. The effect of the parameters to "histo" not fulfilling the condition: length of A = length of B + 1, is implementation-defined.

23. The default BYTESIZE for bytefiles is implementation-defined.

24. Evaluation of arithmetic expressions may give different results for different implementations.

Annex C Index of Syntactic Meta-symbols

Symbol	Defined in:	Referenced in:
activation-clause	12.2	12.2
activation-statement	12.2	4.
actual-parameter	3.1.4	3.1.4
actual-parameter-part	3.1.4	3.1.4, 3.8, 4.6, 4.10.1
arithmetic-expression	3.5	3., 3.1, 3.5, 4.4, 5.2, 12.2
arithmetic-relation	3.3.1	3.3
arithmetic-type	2.	2.
array-declaration	5.2	5.
array-identifier	5.2	5.2
array-identifier-1	3.1	3.1, 3.1.4
array-segment	5.2	5.2
assignment-statement	4.1	4.
attribute-identifier	3.1.5	3.1.5
block	4.10	4., 4.10.1
block-head	4.10	4.10, 5.5
block-prefix	4.10.1	4.10.1
Boolean-expression	3.2	3., 3.2, 4.2, 4.3, 4.4
Boolean-factor	3.2	3.2
Boolean-primary	3.2	3.2
Boolean-secondary	3.2	3.2
Boolean-term	3.2	3.2
Boolean-tertiary	3.2	3.2
bound-pair	5.2	5.2
bound-pair-list	5.2	5.2
character-constant	1.7	3.6
character-designator	1.7	1.7
character-expression	3.6	3., 3.6
character-relation	3.3.2	3.3
class-body	5.5	5.5
class-declaration	5.5	5., 6.
class-identifier	5.5	2., 3.3.4, 3.8, 4.8, 4.10.1, 5.5
compound-statement	4.9	4.
compound-tail	4.9	4.9, 4.10, 5.5
conditional-statement	4.2	4.
connection-block-1	4.8	4.8
connection-block-2	4.8	4.8
connection-statement	4.8	4., 4.2
constant-element	5.8	5.1
decimal-fraction	1.5	1.5
decimal-number	1.5	1.5
declaration	5.	4.10
designational-expression	3.9	3., 3.9, 4.5, 5.3
destination	4.1	4.1
digit	1.	1.4, 1.5, 1.6
dummy-statement	4.11	4.
equivalence	3.2	3.2
exponent-part	1.5	1.5
expression	3.	3.1.4
external-class-declaration	6.4	6.1
external-declaration	6.1	5., 6.1

Symbol	Defined in:	Referenced in:
external-head	6.1	6.
external-identification	6.5	6.5
external-item	6.5	6.3, 6.5
external-list	6.5	6.3, 6.4
external-procedure-declaration	6.3	6.1
factor	3.5	3.5
final-operations	5.5	5.5
for-clause	4.4	4.4
formal-parameter	5.4.2	5.4.2
formal-parameter-part	5.4.2	5.4, 5.5
format-effektor	1.9	1.9
for-right-part	4.4	4.4
for-statement	4.4	4., 4.2
function-designator	3.1.4	3.2, 3.5, 3.6, 3.7, 3.8
goto-statement	4.5	4.
identifier	1.4	3.1.5, 3.9, 4.4, 5.1, 5.2, 5.4, 5.4.2, 5.4.3, 5.5, 5.8, 6.3, 6.5
identifier-1	3.1.5	3.1, 3.1.4
identifier-list	5.4.3	5.4.2, 5.4.3, 5.5.3, 5.5.4
if-clause	4.2	3.2, 3.5, 3.6, 3.7, 3.8, 3.9, 4.2
implication	3.2	3.2
initial-operations	5.5	5.5
inner-part	5.5	5.5
integer-type	2.	2.
isocode	1.6	1.6, 1.7
kind	6.3	6.3
label	3.9	3.9, 4., 4.2, 4.9, 4.10, 4.10.1, 5.5
letter	1.	1.4
local-object	3.8	3.8
main-block	4.10.1	4.10.1
main-part	5.5	5.5
mode-part	5.4.3	5.4
name-part	5.4.3	5.4.3
non-quote-character	1.6	1.6, 1.7
object-expression	3.8	3., 3.8, 4.8, 12.2
object-generator	3.8	3.8, 4.
object-reference-type	2.	2.
object-reference-relation	3.3.5	3.3
object-relation	3.3.4	3.3
otherwise-clause	4.8	4.8
prefix	5.5	5.5
prefixed-block	4.10.1	4.10
primary	3.5	3.5
procedure-body	5.4	5.4
procedure-declaration	5.4	5., 6., 6.3

Symbol	Defined in:	Referenced in:
procedure-heading	5.4	5.4, 6.3
procedure-identifier	5.4	4.1, 5.4, 5.5.3
procedure-identifier-1	3.1.4	3.1.4, 4.6
procedure-specification	6.3	5.5.3, 6.3
procedure-statement	4.6	4.
program	6.2	6.
protection-part	5.5.4	5.5
protection-specification	5.5.4	5.5.4
qualification	2.	2.
qualified-object	3.8	3.8
radix	1.5	1.5
radix-digit	1.5	1.5
real-type	2.	2.
reference-assignment	4.1	4.1
reference-comparator	3.3	3.3.5, 3.3.6
reference-expression	3.	3., 4.1, 4.4
reference-for-list-element	4.4	4.4
reference-left-part	4.1	4.1
reference-relation	3.2.1	3.2.1
reference-right-part	4.1	4.1
reference-type	2.	2.
relation	3.3	3.2
remote-identifier	3.1.5	3.1.5
scheduling-clause	12.2	12.2
simple-arithmetic-expression	3.5	3.3.1, 3.5
simple-Boolean-expression	3.2	3.2
simple-character-expression	3.6	3.3.21, 3.6
simple-designational-expression	3.9	3.9
simple-object-expression	3.8	3.1.5, 3.3.4, 3.3.5, 3.8
simple-string	1.6	1.6
simple-text-expression	3.7	3.3.3, 3.3.6, 3.7, 4.1
simple-timing-clause	12.2	12.2
simple-variable	4.4	4.4
simple-variable-1	3.1	3.1
simple-variable-declaration	5.1	5.
SIMULA-source-module	6.	
space	1.	1.9
specification-part	5.4.2	5.4, 5.5, 5.5.3
specifier	5.4.2	5.4.2
split-body	5.5	5.5
statement	4.	4.2, 4.3, 4.4, 4.8, 4.9, 5.4, 5.5, 6.2
string	1.6	3.7, 5.8, 6.5
string-separator	1.6	1.6
subblock	4.10	4.10
subscripted-variable	3.1	3.1
subscript-expression	3.1	3.1, 3.9
subscript-list	3.1	3.1
switch-declaration	5.3	5.
switch-designator	3.9	3.9
switch-identifier	3.9	3.1.4, 3.9, 5.3
switch-list	5.3	5.3

Symbol	Defined in:	Referenced in:
term	3.5	3.5
text-expression	3.7	3., 3.7, 4.1, 4.4
text-primary	3.7	3.1.5, 3.7
text-reference-relation	3.3.6	3.3
text-value-relation	3.3.3	3.3
timing-clause	12.2	12.2
token-separator	1.9	1.6
type	2.	5.1, 5.2, 5.4, 5.4.2, 6.3
type-list	5.1	5.1
type-list-element	5.1	5.1
unconditional-statement	4.	4., 4.2
unsigned-integer	1.5	1.5
unsigned-number	1.5	3.5
value-assignment	4.1	4.1
value-expression	3.	3., 4.1, 4.4, 5.8
value-for-list-element	4.4	4.4
value-left-part	4.1	4.1
value-part	5.4.3	5.4.3, 5.5
value-relational-operator	3.3	3.3.1, 3.3.2, 3.3.3
value-right-part	4.1	4.1
value-type	2.	2.
variable	3.1	3.2, 3.5, 3.6, 3.7, 3.8, 4.1
virtual-part	5.5.3	5.5
virtual-spec	5.5.3	5.5.3
when-clause	4.8	4.8
while-statement	4.3	4.

Annex D STATUTES of the SIMULA STANDARDS GROUP (SSG)

Article 1. Objectives

The SIMULA Standards Group (SSG) is an organization which at all times shall:

1.1 be the final arbiter in the interpretation of the SIMULA language definition and be a centre for custody of this formal definition.

1.2 standardize the SIMULA language by minimizing its changes and preventing occurrence of SIMULA translators violating the language definition.

1.3 provide a forum for discussion and exchange of information relating to the use of the SIMULA language and its support.

Article 2. Membership

2.1 The SSG offers three categories of membership. These are called full, associate and extraordinary membership. Full and associate members have voting rights. In addition, full members have veto. Regardless of its number of memberships, no institution can have more than two votes.

2.2 Full membership is open to organizations and firms responsible for the maintenance and support of the whole of a SIMULA system, i.e. an independent implementation.

Associate membership is open to organizations and firms who are responsible for maintenance and support of parts of a SIMULA system adapted from another system, i.e. a sub-implementation.

Any organization or firm meeting one or other set of criteria may apply for membership and be voted a member of the SSG in the appropriate category. The decision as to whether full or associate membership is appropriate is left to a majority of existing members.

2.3 The Norwegian Computing Center (NCC), Oslo, Norway is ex officio a full member of the SSG. In addition, the SIMULA Development Group (SDG) Chairman and the Association of SIMULA Users (ASU) Chairman are also full members of the SSG.

2.4 The SSG can offer extraordinary membership to individuals in recognition of their contribution to the SSG work.

2.5 Once granted an SSG membership lasts until:

- it is resigned by a particular member

- it is revoked by the SSG because the conditions under which it was granted cease to exist or the member acts against the objectives of the SSG.

- it is revoked because, after 12 months notice from the Chairman of the SSG, the implementation does not meet the current SIMULA Standard.

2.6 There is no fee for SSG membership.

Article 3. **Representation, Voting and Meetings**

3.1 Each SSG member shall appoint one person to be its representative in the SSG. The duration of this person's appointment is determined by the member.

3.2 The SSG shall meet once every calendar year for an Annual Meeting. This meeting will, in addition to eventual matters of an administrative character, handle proposals related to the SIMULA language definition as described in the Formal Rules of the SSG operation. The Annual Meeting shall also elect one of the full member's representatives as SSG Chairman.

In addition to the Annual Meeting the SSG may have extraordinary meetings when this is approved by a majority of the members.

Decisions can only be taken regarding matters on the agenda presented to the members at least 3 weeks before the meeting, unless all members agree otherwise.

3.3 To constitute a quorum, all members of the SSG shall be notified of the Meeting and at least one more than a half of the member representatives shall be present or give their votes by mail.

3.4 Decisions by SSG are made by a majority vote among the representatives taking part in the vote. Changes to the SSG Statutes or a decision to dissolve the SSG, require 4/5 majority, as well as the consent of the NCC. Full members have the right to veto changes to the SIMULA language definition, but not to the Statutes.

3.5 The SSG meetings are open to non-members who wish to attend. Such non-members will be granted observer status by the SSG. Observers have no voting rights and must apply to the SSG for each meeting they wish to attend. Observers may not normally speak, unless they have submitted a proposal or have received permission from the meeting.

Formal rules of the SSG operation

1. The main task of the SSG is the maintenance of the SIMULA language definition (SIMULA Standard) which consists of:

   a) a clarification of obscure parts of the definition.

   b) removal of possible conflicts in the definition.

   c) alteration of the definition in line with the approved changes in the language.

2. The following types of language changes can be directly considered by the SSG:

   a) obvious oversights that have occurred in the text of the language definition during editing/printing.

   b) removal of language restrictions that are proved obsolete for language consistency and implementation.

   c) trivial extensions to the existing concepts that are felt relevant for continued use of the language in a changing environment.

   All other changes must first be handled by the Simula Development Group and passed over in the form of a formal recommendation.

3. All proposals for language changes conforming to the above rules must be formulated in writing in a concise manner and submitted to the SSG secretariat. Any SIMULA user may submit such a proposal.

4. It is the responsibility of the SSG Chairman to notify receipt of each proposal, to register it and schedule its processing at one of the SSG meetings. Alternatively the SSG Chairman may point out any inadequacies in a proposal to its submitter.

   In either case the proposal will be announced at the next SSG meeting which may approve or revoke the Chairman's decision related to this proposal.

   Complete material related to a particular proposal will only be submitted to SSG members when the proposal is to be processed at the next meeting or on specific request.

5. In its final form every proposal will be an updating text to the SIMULA Standard. It will further indicate the original submitter, date of submission and its motivation. Alternative forms of the proposal and reasons for their rejection are valuable parts of the document. An example of a suitable form is attached to these rules.

   The logical consistency of the text, its clarity and conciseness are of utmost importance. To this aim the SSG or its Chairman may allocate one particular SSG member to bring the proposal into the required shape if this is deemed necessary.

6. It is the responsibility of the SSG Chairman to notify the submitter of a proposal about the result of its processing by the SSG if this is not otherwise obvious. It is also his/her responsibility to minimize the time spent on each proposal.

7. The final text of the proposals will be available from the SSG secretariat as a supplement to the SIMULA Standard until they are incorporated into this at its next revision. To simplify this process, at its approval a proposal will be assigned a number reflecting both the SIMULA Standard version it applies to and its chronological order.

Reg. No.: _____

Accepted as Standard SIMULA Change No.:

Title :
Submitter :
Date :
Affected section(s) :

_____

Proposal:

Motivation:

SSG decision on the above proposal: