RC9000-10/RC8000

SW8585 Compiler Collection

ALGOL8 Reference Manual

PN 99200389

RC Computer

Keywords:
RC9000-10, RC8000, Compiler, ALGOL, ALGOL8, Reference Manual

Abstract:
This manual is the reference manual for the ALGOL compoler for RC9000-10 and RC8000 systems.

Date:
March 1989.

PN: 991 11278

Table Of Contents

Introduction

This book contains the formal description of the ALGOL language and the corresponding ALGOL8 compiler used on RC8000/RC9000-10.

The description of the reference language follows the structure of Revised Report of ALGOL 60 (cf. ref. 11 and 12). The elements which are in ALGOL7 but not in ALGOL 60 are merged into this description.

The book gives a syntax and a semantic description for all language elements. Furthermore short examples are given to illustrate the syntax, further examples are found in [14] ALGOL8 User's Guide, Part 2.

1. Notation

The syntax of the ALGOL8 elements are described with aid of an improved BNF (BackusNaur FORM) where repetition suffixes are used. (cf. ref. 15)

The following example and short explanation may help to understand at least the basic idea of the language.

```
                          {            }1 {          }*
<transaction> ::=    tr {<transdate>}0 {<trfield>}1

<transdate> ::=       <day>.<month>.19<year>

                     {<payment>}
<trfield> ::=        {<invoice>}
```

or more compact:

```
<transaction> ::=

                                    {          }*
                                    {<payment>}
tr {<day>.<month>.19<year>}0        {<invoice>}
                                    {          }1
```

A BNF description is composed of a set of statements, each one naming and defining a syntactial unit or string by means of a definition symbol ::= to the right of which the possible components of the string are stated. (What is in fact defined is not a single string but a whole class of strings having the properties specified by the right side). The components may either be other strings or data constants i.e. sequences of socalled terminal symbols which, in fact, are nothing but typographical representations of elements of the data spectrum. (In the example tr,. and 19).

A string is denoted by a name enclosed in brackets < >. The name, which may be a whole sentence, will normally be chosen to give the reader some associations with the information represented by the string. The order of the statements in a description is immaterial, but all referred strings must be defined somewhere else in the description. On the other hand, the order of the components to the right of the ::= symbol prescribe exactly the order in which the data elements must

appear. In this way concatenation of the real data elements is implied in the description.

Alternative data structures are described as separate lines enclosed in { }. The same bracket is used around repetitive data structures. The repetive factor is specified by the lower and upper bound numbers connected to the bracket. * denotes any number.

2. Basic Symbols, Identifiers, Numbers and Strings

The Algol 8 language is built up from the following basic symbols:

```
                    {<letter>       }
<basic symbol>::=   {<digit>        }
                    {<logical value>}
                    {<delimiter>    }
```

2.0.1 Character Set and Coding

The Algol 8 character set is a subset of the ISO 7bit character set extended with the Danish letters: {, [, |, \, },].

(See ref. 13). At run time, the program may choose any alphabet, but the ISO 7bit code is offered as standard. It is possible in a simple way to use paper tapes in flexowriter code as source and data, because the i/o system may convert the code to ISO 7-bit code (see ref. 3 and 14).

The table 2.1 shows for each character of the ISO 7-bit alphabet:

V: The internal value.

G: The graphic representation or the name of the character.

S: The character class as source to the translator.

D: The character class as data read with the standard alphabet.

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

Table 2.1 Character Set and Input Class

D,Data classes

`0,blind:`	The character is skipped by all read procedures.
`1,shift character:`	Not used in the standard alphabet.
`2,digits:`	May be used as digits in a number or in a text string.
`3,signs:`	May be used as a sign of number or in a textstring.
`4,decimal point:`	May be used as the decimal point of a number, in a text string or in a field reference.
`5,exponent mark:`	May be used as the exponent mark of a number or in a text string.
`6,letters:`	May be used as part of a text string. Will terminate a number.
`7,delimiters:`	Will terminate a number or a text string.
`8,terminator:`	Works as class 7, but terminates a call of readall. EM (25) will immediately terminate a call of read or readstring.

S,Source text classes

`Basic:`	Significant in all contexts.
`Blind:`	Skipped in all contexts.
`Graphic:`	Allowed inside text strings and comments, causes a warning outside.
`Illegal:`	Produces a warning during the translation, but does not harm.
`In text:`	Works as a space inside text strings, blind outside.

Control characters

The control characters which are used in algol are the following:

`10,NL:`	New Line. The changetonewline character.
`12,FF:`	Form Feed. Causes a change of page on the printer, but works syntactically as New Line outside text strings.
`25,EM:`	End Medium. See 2.0.3.
`32,SP:`	Space.
`127,DEL:`	Delete. Used for overpunching of wrong characters.

2.0.2 Source Text

The program consists either of one block, of one compound statement, or of one procedure declaration surrounded by "external" and "end".

All characters up to the first "begin" or "external" are skipped, but appear in a possible listing.

After the last "end", the compiler reads as many characters as are necessary to distinguish the "end" (usually a space or a new line).

2.0.3 Source files

The source text to the compiler consists of one or more files of text as specified in the File Processor command that started the translation. The compiler may read source files from standard input devices such as paper tape, cards, typewriter, magnetic tape, and backing storage.

A file terminates either when an *EM*character is read from the file or when the file physically is exhausted. A file on a roll of paper tape is exhausted when the tape end is met. A file on the backing storage is exhausted when the end of the backing storage area is met. A file on magnetic tape is exhausted when tape mark is met.

When the compiler meets the file termination before the source text is complete, it looks for the next file specified in the File Processor command and continues reading from that file. If the list of files is exhausted, the compiler prints an error message, generates the necessary number of string terminations and "end"s, and compiles the program completed in this way.

The compiler handles the peripheral devices in accordance with the rules of the File Processor (ref. 6 or 7).

2.0.4 Space and New Line

Space and New Line may be used freely in numbers and between identifiers, compound symbols, and other delimiters. They are not, however, allowed inside identifiers, compound symbols, or delimiters.

Space and New Line are significant characters in a text string and will be printed out at run time when the string is printed.

The character " " represents a space inside strings, but is completely blind outside. The latter property may be used to divide identifiers and compound symbols (cf. 2.3).

2.1 Letters

```
                    { a }                          { A }
                    { b }                          { B }
                    { c }                          { C }
                    { d }                          { D }
                    { e }                          { E }
                    { f }                          { F }
<small letter>::=   { g }   <capital letter>::=    { G }
                    { h }                          { H }
                    { i }                          { I }
                    { j }                          { J }
                    { k }                          { K }
                    { l }                          { L }
                    { m }                          { M }
                    { n }                          { N }
                    { o }                          { O }
                    { p }                          { P }
                    { q }                          { Q }
                    { r }                          { R }
                    { s }                          { S }
                    { t }                          { T }
                    { u }                          { U }
                    { v }                          { V }
                    { x }                          { X }
                    { y }                          { Y }
                    { z }                          { Z }
                    { æ }                          { Æ }
                    { ø }                          { Ø }
                    { å }                          { Å }

                 { <small letter>   }
<letter> ::=     { <capital letter> }
```

Letters do not have individual meaning. They are used for forming identifiers and strings (cf. sections 2.10 Identifiers, 2.12 Strings).

2.2 Digits and Logical Values

2.2.1 Digits

```
                (0)
                (1)
                (2)
                (3)
<digit> ::=     (4)
                (5)
                (6)
                (7)
                (8)
                (9)
```

Digits are used for forming numbers, identifiers, and strings.

2.2.2 Logical Values

2.2.2.1 Syntax

```
<logical value> ::=     ( true                )
                        ( false               )
                        ( "<graphic or name>" )
```

<graphic or name> is one of the mnemonics for the ISO 7/bit characters shown in column G in table 2.1. The mnemonics for the ISO values 0...32 and 127 must be spelled with small letters.

2.2.2.2 Semantics

The logical values true or false have a fixed obvious meaning.

The logical value "<graphic or name>" has the value false add <ISO value>, where <ISO value> is the corresponding value in the column V in table 2.1.

2.3 Delimiters

The underlined delimiters (compound symbols) of the reference language are written without underlining. A Space or a New Line is required to separate a compound symbol from a preceding identifier or a succeeding letter or digit. Thus the delimiter space is forbidden inside a delimiter, but the symbol " " may be used instead. The delimiter "goto" and "boolean" may not be written as "go to" and "Boolean".

```
                    { <operator>           }
                    { <separator>          }
<delimiter> ::=     { <bracket>            }
                    { <declarator>         }
                    { <specificator>       }
                    { <compiler directive> }

                    { <arithmetic operator> }
                    { <relational operator> }
<operator> ::=      { <logical operator>    }
                    { <sequential operator> }
                    { <pattern operator>    }
                    { <context operator>    }

                              { +      }
                              { -      }
                              { *      }
                              { /      }
                              { /      }
<arithmetic operator> ::=     { mod    }
                              { *      }
                              { round  }
                              { abs    }
                              { entier }
                              { extend }

                              { <  }
                              { <= }
relational operator> ::=      {  = }
                              { >= }
                              { >  }
                              { <> }

                              { ==  }
                              { =>  }
                              { or  }
<logical operator> ::=        { !   }
                              { and }
                              { &   }
                              { not }
                              { ,   }

                              { goto   }
                              { if     }
                              { then   }
                              { else   }
<sequential operator> ::=     { for    }
                              { do     }
```

```
                                  { case   }
                                  { of     }
                                  { repeat }
                                  { while  }

                                  { real   }
<transfer operator> ::=           { long   }
                                  { string }

<context operator> ::=            { exit     }
                                  { continue }

                                  { add     }
<pattern operator> ::=            { extract }
                                  { extend  }
                                  { shift   }

                                  { ;       }
                                  { ,       }
                                  { .       }
                                  { ,       }
                                  { :       }
<separator> ::=                   { :=      }
                                  { _       }
                                  { step    }
                                  { until   }
                                  { while   }
                                  { comment }

                                  { (     }
                                  { )     }
                                  { <:    }
<bracket> ::=                     { :>    }
                                  { <     }
                                  { >     }
                                  { <*    }
                                  { *>    }
                                  { begin }
                                  { end   }

                                  { own       }
                                  { boolean   }
                                  { integer   }
                                  { long      }
                                  { real      }
<declarator> ::=                  { array     }
                                  { switch    }
                                  { zone      }
                                  { procedure }
                                  { context   }
                                  { field     }

                                  { string   }
<specificator> ::=                { label    }
                                  { value    }
```

```
                                  { algol    }
<compiler directive> ::=          { external }
                                  { message  }
```

Delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent with
;comment<any sequence not containing";">;	;
begin comment<any sequence not containing";">;	begin
end <any sequence not containing "end",";", "else" or "until"	end
;message <any sequence not containing";">;	;
begin message <any sequence not containing";">;	begin
<*< any sequence not containing "<*" or "*>" > *>	space

By equivalence is here meant that any of the structures shown in the left hand column may, in any occurrence outside of strings, be replaced by the symbol shown on the same line in the right hand column without any effect on the action of the program. The comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4 Identifiers

2.4.1 Syntax

```
                                  {          }*
<identifier> ::= <letter>         { <letter> }
                                  { <digit>  }
                                  {          }0
```

The words for compound symbols (see 2.6.1 and 2.7) can never be used as identifiers.

2.4.2 Examples:

```
q
Soup
V17a
a34kTMNs
MARILYN
goto  go_to          Both are interpreted as the
                     delimiter "goto".
go to                An erroneous construction
                     consisting of two identifiers.
13 do a7:=           The number 13, the delimiter "do",
                     the identifier a7, and the
                     delimiter :=
begin of_line : =    An erroneous construction
                     consisting of the delimiter
                     "begin", the identifier "of_line",
                     the delimiter :
                     and the delimiter =
```

2.4.3 Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, procedures, field variables, zones, and zone arrays. They may be chosen freely, except for the limitation mentioned above.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7 Quantities, Kinds and Scopes and section 5. Declarations).

2.5 Numbers

2.5.1 Syntax

```
                      { <digit>              }*
<digit sequence>::= { '<graphic or name>' }1

                        {                    }*
<unsigned integer>::= { <digit sequence> }
                        {                    }1

              { + }1
<integer>::= {   }0  <unsigned integer>

<decimal fraction>::= .<unsigned integer>

<exponent part>::= '<integer>

<decimal number>::=
{                    {                    }1}
{<unsigned integer> {<decimal fraction>}0}
{<decimal fraction>                       }

<unsigned number>::=
{                  {                 }1 }
{<decimal number> {<exponent part>}0 }
{<exponent part>                     }

             { + }1
<number>::= {   }0 <unsigned number>
```

The numbers of digits are restricted, (cf. sections 2.5.5 and 2.5.6).

<graphic or name>, cf 2.2.2.1

2.5.2 Examples

```
        0          200.084       .083'02        'sp'
      177         +07.43'8           '7         'p'
    .5384         9.34'+10           '4       'del'
  +0.7300              2'4         +'+5       1'ne'
```

2.5.3 Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10. The digit sequence '<graphic or name>' consists of the digits in the ISO value in column V in table 2.1.

2.5.4 Types

Integers are either of type integer or of type long, depending on the value. All other representable numbers are of type real.

2.5.5 Integer and Long Literals

Integers and longs may not exceed the interval

```
-140 737 488 355 327 <= integer <= 140 737 488 355 327.
```

If the literal is within the interval

```
-8 388 607 <= integer <= 8 388 607
```

it is classified as being of type integer. Outside this interval it is classified as being of type long (cf. section 3.3.4).

2.5.6 Real Literals

The real may not have more than 14 significant digits or 14 decimals. The exponent part may not exceed the interval 1000 < exponent < 1000. The total number is confined to the range -1.6'616 <= number <= 1.6'616.

The number is converted to internal binary form using the same methods as the procedures read and readall. The relative error of the result is about 3'11.

2.6 Strings

2.6.1 Syntax

```
<string literal>::= (<text string> )
                    (<layout string>)

<text string>::=
<:<any sequence of text symbols not containing ":>"or
"<:">:>

<layout string>::= << <layout> >

<layout>::=

(         )1(       )1
(<spaces>)0(<sign>)0<layout number part>
                              (                         )1
                              (<layout exponent part>)0

             (       )*
<spaces>::= (<space>)1

            (   )
<space>::=  ( _ )

            ( + )
<sign>::=   ( - )

<layout number part>::=
(<first letter>d's>.<d's><zeroes>  )
(<first letter>d's><zeroes>.zeroes>)

                   ( z )
                   ( d )
<first letter>::=  ( f )
                   ( b )

              ( (          )1    )*
<d's>::=      ( ( <space> )0 d )0

              ( (          )1     )*
<zeroes>::=   ( ( <space> )0  0 )0

<layout exponent part>::= '<sign><first letter><d's>
```

A text symbol is a character belonging to one of the classes basic, graphic, or in text (see 2.0.1) or it is a positive integer of at most 3 digits enclosed in >. The latter construction has precedence over the char tion, and represents the character with the integer as internal value. The value must obey 0 value 128. The general string concept is described in 3 6.

2.6.2 Examples

```
<:a<b'c'>>d<'ne'>        will be printed by a
                         running program as

a<b c>d
<<_-d.ddd'+d>            is a layout string.
```

2.7 Quantities, Kinds and Scopes

The following **kinds** of quantities are distinguished: simple variables, arrays, labels, switches, procedures, field variables, zones and zone arrays.

The **scope** of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8 Values and Types

A **value** is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The value of a zone is a set of values called the zone descriptor, plus a set of values in the zone buffer area, plus a set of values called the share descriptors (see 5.5).

The value of a zone array is the set of values of the corresponding subscripted zones.

The various **types** (integer, real, long, boolean) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

A field variable possesses an integer value, but has an associated type denoting the type of a field. A field is either a simple field or an array field. Fields are subsets of arrays or zones. Variable fields posses a single value. Array fields are one dimensional arrays.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, boolean, designational, zone, and string expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, pattern, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

```
                    { <arithmetic expression>     }
                    { <boolean expression>        }
<expression>::=     { <designational expression>  }
                    { <string expression>         }
                    { <zone expression>           }
```

3.1 Variables and Fields

3.1.1 Syntax

```
<variable identifier> ::= <identifier>

<simple variable> ::= <variable identifier>

<simple field variable> ::= <identifier>

<array field variable> ::= <identifier>

<field variable> ::= <simple field variable>
                     <array field variable>

<subscript expression> ::= <arithmetic expression>

<subscript list> ::=
<subscript expression>  {                          }*
                        { ,<subscript expression>  }0

<array identifier> ::= <identifier>

<zone identifier> ::= <identifier>
```

```
<zone array identifier> ::= <identifier>

<zone expression> ::=
(<zone identifier>
(<zone array identifier> (<subscript expr.>))

<field base> ::=
(<array identifier>)
(<zone expression> )
(<array field>     )

<array field> ::= <field base>.<array field variable>

<simple field> ::= <field base>.<simple field variable>

<record variable> ::=
(<zone identifier>(<subscript expression>)           )
(<zone array ident.>(<subscr. expr.>,<subscr. expr.>))

<subscripted variable> ::=
(<array identifier>(<subscript list>) )
(<array field>(<subscript expression>))


<variable> ::=
(<simple variable>     )
(<simple field>        )
(<subscripted variable>)
(<record variable>     )
(<field variable>      )

<field reference> ::=
(<array field>    )
(<simple field>   )

<field> ::= (<field reference>)
```

3.1.2 Examples

```
epsilon
detA
al7
Q(7, 2)
x(sin(n*pi/2), Q(3, n, 4))
P.type
term (termno).entry
```

3.1.3 Semantics

A **variable** is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2).

The **type of a simple variable** is defined in the declaration of the variable itself (cf. section 5.1 Type declaration).

The **type of a variable field** is defined in the declaration of the simple field variable (cf. section 5.7 Field declaration).

The **type of a subscripted variable** is defined in the declaration of the array identifier (cf. section 5.2 Array declaration) or in the declaration of the ultimate array field variable defining the array field (cf. section 5.7 Field declaration).

The **type of a record variable** is real, and the type of a field variable is integer.

3.1.3.1 Zones and Record Variables.

Record variables designate values which are components of zone buffer areas. The subscript expressions are evaluated like subscripts of ordinary subscripted variables.

In case of a zone array with subscripts, the first subscript expression selects a zone from the zone array. This subscript must obey

```
1 <= subscript <= number of zones declared in zone array.
```

The last subscript selects a variable within the zone record, which in turn is a set of consecutive variables of the selected zone buffer area. This subscript must obey

```
1 <= subscript <= number of variables currently in the
record.
```

When an expression is assigned to a record variable, the location (see 4.2.3) of the selected buffer element is not influenced by possible changes of the record caused by procedure calls in the right hand expression.

3.1.3.2 Fields

Fields are subsets of arrays and zone records. A field consists of a number of halfwords located within an array or a zone. This array or zone is the field base for the field. A field variable is a pointer indicating a field within an array, a zone record, or an array field. The type of the field depends only upo a type declared together with the field variable (cf. section 5.7 Field declaration). All the halfwords of a simple field must be located within the field base.

3.1.4 Subscripts

3.1.4.1 Subscripted variables may designate values which are components of multidimensional arrays (cf. section 5.2 Array declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript

brackets (). The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3 Arithmetic expressions).

3.1.4.2 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 fictive variable (cf. section 4.2.4).

The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2 Array declarations).

3.1.4.3 An array field is an array always considered one dimentional. The ordering of the halfwords in the field base and in the array field follows the lexicographical ordering (cf. 5.2.6 Lexicographical ordering) The subscript bounds are defined by means of the halfword bounds (cf. 5.2.7). The halfword bounds for the array field are obtained by subtracting the value of the array field variable from the halfword bounds of the field base (possibly an array field). An element must be located within the field base.

3.1.5 Initial Values of Variables

The value of a variable, not declared own or context, is undefined from entry into the block in which it is declared until an assignment is made to it.

The value of a variable declared own or context is binary zero (if arithmetic) or false (if boolean) on first entry to the block in which it is declared. On subsequent entries it has the same value as at the preceding exit from the block.

3.1.6 Ranges of Values. Type Length. Binary Patterns

Depending on the type, each variable is represented by an integral number of halfwords. Each halfword is of 12 bits. The number of halfwords representing a variable is called the type length. The type length may some times be expressed in bits.

3.1.6.1 Booleans are represented as 12 bits quantities. The type length of a boolean variable is 1 halfword. The binary pattern of a boolean is extended with zeroes to the left whenever needed. The last of the 12 bits is 0 when the boolean is false, 1 when it is true.

The logical constants "true" and "false" and the result of applying the relational operators will always be 12 zeroes for false, 12 ones for true. Other binary patterns may be obtained by applying the operators add and shift, and/or using the constants expressed by "<graphic or name>". The 5 logical operators work on all 12 bits in parallel.

3.1.6.2 Integers are represented in 24bits, 2"s complement, binary form. This gives the range:

```
-8 388 608 <= integer <= 8 388 607.
```

The type length of an integer variable is 2 halfwords, and the binary pattern of an integer is the 24 bites of its representation extended with zeroes to the left whenever needed. The binary patterns are used in connection with the operators add, extract, and shift.

3.1.6.3 Longs are represented in 48 bits, 2's complement, binary form. The range of longs should be confined to:

```
-140 737 488 355 327 <= long <= 140 737 488 355 327.
```

The type length of a long variable is 4 halfwords, and the binary pattern of a long is the 48 bits of its representation. The binary patterns are used in connection with the operators add, extract, and shift.

3.1.6.4 Reals are represented as 48bits builtin floating point numbers. This gives the following range of nonzero real values:

```
1.6'-617 < abs(real) < 1.6'616
```

The precision of real values correspond to 35 significant bits. Thus one unit added to the last binary place will correspond to a relative change of the number of between 6'-11 and 3'-11.

The type length of a real variable is 4 halfwords. The 3 first halfwords are used for the number part and the last halfword for exponent part of the real.

The binary pattern of a real consists of a 36-bits, 2's complement, number part followed by a 12-bits, 2's complement, exponent part so that the real value is:

```
number*2**exponent.
```

The number is either 0 or in the range -1 < = number < -0.5, 0.5 < = number < 1. The exponent is in the range -2048 < = exponent < = 2047. The exponent of 0.0 is -2048, but other exponents might be obtained by the operator "add".

If r is a floating point zero with an exponent < >-2048, the relation r = 0 will be false because the operands are compared bit by bit. The relations r < = 0 or r > = 0 will both be true, however.
Operations like r + b cannot be expected to give b (see ref.4.).

3.1.7 Reals Used as SemiLong Integers

As there is neither builtin long multiplication nor builtin long division, programs using many of these operations on large integers may be speeded up some what by representing them as real variables.

This can be done with full accuracy as long as all results are kept in the range

```
-2**35 - 34 359 738 368 <= real <= 34 359 738 367 - 2**35
-1
```

If the results exceeds this range, the last bits of the semilong integer are lost.

A kind of integer division may be obtained by a real division followed by a cutoff of decimals caused by the addition of a large constant. For results in the range 0 < = result < = 2**34, this is done as follows:

```
roundconstant: - 2**34,
result: - r1/r2 + roundconstant roundconstant,
```

Safety against loss of accuracy may be obtained by scaling the semilong integers so that loss of accuracy will cause a floating point overflow. The scale factor f is chosen so that f*2**35 = 2**2048 and f*(2**35) = 2**2048. This is fulfilled by f = 2**2013. Addition (i1 + i2) and multiplication (i1 * i2) with check for loss of accuracy may be performed like this:

```
r1: - i1*f,          r2: - i2*f,
r1 + r2              r1/f*r2
```

3.2 Function Designators

3.2.1 Syntax

```
<procedure identifier> ::= <identifier>

<actual parameter> ::=
{ <string literal>        }
{ <expression>            }
{ <array identifier>      }
{ <array field>           }
{ <switch identifier>     }
{ <procedure identifier>  }
{ <zone identifier>       }
{ <zone array identifier> }

<letter string> ::=
{          }*
{ <letter> }0

<parameter delimiter> ::=
{     ,              }
{ )<letter string>:( }

<actual parameter list> ::=
<actual parameter>{<parameter delimiter>
          *
<actual parameter>}
          0

<actual parameter part> ::=
                              1
{(<actual parameter list>)}
                              0

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

The <parameter delimiter> known as "fat comma" defined by)<letter string>:(may not contain compound symbols, to prevent code errors. Comment strings may not be used either.

3.2.2 Example

```
sin (a b)
J(v + s, n)
R
S(s 5)Temperature:(T)Pressure:(P)
write(out, <:good morning:>, <<d.ddddd>, folks)
```

3.2.3 Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4 Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7 Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4 Standard Functions

The numerical standard functions of ALGOL8 are listed below. They are described in detail in User's Manual, ref. 14.

```
arcsin    cos       random  sin
arctan    exp       sgn     sinh
arg       ln        sign    sqrt
```

3.2.5 Transfer Functions

The operators add, extend, entier, extract, long, real, round and string take care of type transfers.

3.3 Arithmetic Expressions

3.3.1 Syntax

```
<adding operator> ::=          { + }
                               { - }

                               { *    }
<multiplying operator> ::=     { /    }
                               { //   }
                               { mod  }

                               { shift   }
<pattern operator> ::=         { add     }
                               { extract }

                               { abs     }
                               { entier  }
<monadic operator> ::=         { round   }
                               { extend  }
                               { real    }
                               { string  }
                               { long    }

<primary> ::=
                           {<unsigned number>          }
                           {<variable>                 }
{                   }1     {<function designator>      }
{<monadic operator>}       {(<arithmetic expression>)}
{                   }0     {real <string primary>      }
                           {long <string primary>      }
<factor> ::=
{<factor>**                 }
{<factor><pattern operator>} <primary>
{<boolean basic> extract    }

<term> ::=
{                            }1
{<term><multiplying operator>} <factor>
{                            }0

<simple arithmetic expression> ::=
{{                             }1                  }
{{<simple arithmetic expression>} <adding operator>}
<term>
{{                             }0                  }

<if clause> ::= if <boolean expression> then

<case clause> ::= case <arithmetic expression> of

<arithmetic expression list> ::=
                                                    *
<arithmetic expression> {,<arithmetic expression>}
                                                    0
```

```
<arithmetic expression> ::=
{<simple arithmetic expression>                }
{<if clause><simple arithmetic                 }
  expression>else<arithmetic   expression>     }
{<case clause> (arithmetic expression list>)}

<string primary>    cf. 3.6.1

<boolean basic>     cf. 3.4.1

<boolean expression> cf. 3.4.1
```

3.3.2 Examples

Primaries:

```
7.394'-8
sum
w(i+2,8)
cos(y+z*3)
(a3/y+vu**8)
long(if b then <:abc:> else <<dd.d0>)
abs round ra(i)
entier cos(y+z)
```

Factors:

```
omega
round r shift (-6) add j
(a < b) extract 1
sum ** cos (y+z*3)
7394'8**w (i+2,8) ** (a-3/y+vu**8)
```

Terms:

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

Simple arithmetic expression:

```
U-Yu+omega*sum**cos(y+z*3)/7.394'8**w(i+2,8)**(a3/y+vu**8)
```

Arithmetic expressions:

```
w * u - Q(S+Cu)**2
if q>0 then S+3 * Q/A else 2 * S+3 * Q
if a<0 then U+V else if a * b=17 then U/V
                     else if k<>y then V/U else 0
a * sin (omega * t)
0.57'12 * a(N * (N  1)/2, 0)
(A * arctan(y) + Z) ** (7 + Q)
if q then n-1 else n
```

```
if a<0 then A/B else if b=0 then B/A else z
case 1 + f of (i mod j, if b then r**j else i, case i of (j))
if b then (case i of (j,r)) else case i of (1,5)
```

3.3.3 Semantics

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 Section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value rising from the computing rules defining the procedure (see Section 5.4.4 Values of function designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

If
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 Section 3.4 Boolean expressions). 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. 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.

Case
The expressions of an expression list in a case expression are separated by commas and numbered 1, 2, 3,...

A case expression is evaluated as follows: First, evaluate the arithmetic expression and if necessary round it to an integer. Next, select the list element corresponding to the result. If no such list element exists, the run is terminated. Evaluate the selected expression and take the result as the value of the caseexpression.

The order of evaluation of primaries within an expression is not defined. If different orders of evaluation would produce different results, due to the action of side effects of function designators, then the program is undefined.

In evaluating an arithmetic expression, it is understood that all the primaries within that expression are evaluated, except those within any arithmetic expression that is governed by an if clause but not selected by it. In the special case where an exit is made from a function designator by means of a goto statement (see section 5.4.4), the evaluation of the expression is abandoned, when the goto statement is executed.

3.3.4 Operators and Types

Apart from the boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of real, long or integer types (see Section 5.1 Type declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1 The operators +, , and * have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both of the operands are of integer type. The type is real if at least one of the operands is of real type. Otherwise the type is long.

3.3.4.2 The operations term / factor and term // factor both denote division. The operations are undefined if the factor has the factor value zero, but are otherwise to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (see Section 3.3.5). Thus for example

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

means

$$((((a * (b^{-1})) * 7) * ((p - q)^{-1})) * v) * (s^{-1}).$$

The operator / is defined for all nine combinations of real, long, and integer and will yield results of real type in any case.

The operator // is defined only for operands of integer or long type. The result is integer if both operands are integer and long otherwise.

The result can be defined by the function:

```
long procedure div (a,b);
value                a,b;
long                 a,b;
if b = 0 then
   begin
   if  spilltest then div:=a
                 else alarm (:integer:>)
   end
else
   begin long q,r;
        q:=0, r:=abs a;
        for r:=r - abs b while r>=0 do q:=q+1;
        div:= if a<0 == b>0 then -q else q
   end;
```

If both operands are integer, an equivalent algorithm where all long specifications and declarations are replaced by integer specifications and declarations, must be used as definition.

3.3.4.2.1 The operation <term> mod factor> is defined where // is defined, and yields values of the same types as //. For long resulting values, the result may be defined by the function:

```
long procedure modulus (a,b);
value                    a,b;
long                     a,b;
modulus:=     a - div(a,b)*b;
```

3.3.4.3 The operation <factor>**<primary> denotes exponentation where the factor is the base and the primary is the exponent.

Thus for example

2**n**k means $(2^{n})^{k}$

while

2**(n**m) means $2^{(n^{m})}$??

The result of an exponentation is always real. The precision of the exponentation is explained in ref. 14.

3.3.4.4 Type of a Conditional Expression.

The result of

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

is of type integer if both expressions are of type integer, of type real if at least one expression is of type real, and of type long otherwise.

The result of

```
<case clause>(<arithmetic expression list>)
```

is of type integer if all expressions in the list are of type integer, of type real if at least one expression is of type real, and of type long otherwise.

3.3.4.5 Types of the monadic operators and the pattern operators

Pattern operators: can be used in arithmetic and boolean expressions.

add This dyadic pattern operator will perform a binary addition. The type of the left hand operand will yield the type of the result.

extract This pattern operator extracts a number of the rightmost bits. The result is of type integer.

shift This pattern operator will perform a logical shift of the left hand operand.

Monadic operators:

abs This operator yields the absolute value of integer, long or real expressions.

The following monadic operators are the transfer functions:

extend This operator converts an integer expression into a type long.

entier This operator transfers a real expression to the largest integer not greater than the expression.

long This operator changes the type of a string or real expression into type long.

real This operator changes the type of a string or long expression into type real.

round This operator rounds the value of a real or long expression to the nearest integer.

string This operator changes type of real or long expression into type string.

Further description see Algol 8 User's Manual ref. 14.

3.3.5 Precedence of Operators

Function calls in an expression may cause "sideeffects", but the result will correspond to a left to right evaluation of the expression, so that sideeffects may only influence variables to the right of the function call.

3.3.5.1 According to the syntax given in section 3.3.1 the following rules of precedence hold:

```
first: abs entier real round long extend string
second: ** add extract shift
third: * / // mod
fourth: + -
```

3.3.5.2 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.

3.3.6 Arithmetic of Real, Long, and Integer Quantities

The operations + - * / ** (for integer, long or real exponents) are performed by the built in floating point operations whenever the result is of type real, and by the fixed point operations whenever the result is of type integer. Whenever the result is of type long, + and are performed by the built in double length operations, whereas the operation * is performed by a subroutine.

The operations // and mod are performed by the built in fixed point division whenever the result is of type integer, and by a subroutine whenever the result is of type long.

When necessary integer operands are floated by means of the built in float operation or converted to a long by extension of the sign. Conversion of operands of type long to type real is performed by a subroutine.

The range of values of type real and integer is given in 3.1.6. The action when the range of reals is exceeded, is controlled at run time by means of the two standard integer variables "overflows" and "underflows" (cf. ref. 14). The action when the range of integers or longs is exceeded, is determined at translation time by means of the translation parameter "spill" (cf.ref. 14).

3.4 Boolean Expressions

3.4.1 Syntax

```
<relational operator> ::=
( < )
( <= )
( = )
( >= )
( > )
( <> )

<and> ::=
( and )
( & )

<or> ::=
( or )
( ! )

<not> ::=
( not )
( -, )

<relation> ::=
<simple arithmetic expression><relational operator>
<simple arithmetic expression>

<boolean pattern operator> ::=
( add )
( shift )

<boolean basic> ::=
(<logical value>                                          )
(<variable>                                               )
(<function designator>                                    )
(<boolean basic><boolean pattern operator><primary>)
((<boolean expression>)                                   )

<boolean primary> ::=
( <boolean basic> )
( <relation>      )

<boolean secondary> ::=
(     )1
(<not>)0 <boolean primary>

<boolean factor> ::=
(                       )1
(<boolean factor><and>)0 <boolean secondary>

<boolean term> ::=
(                    )1
(<boolean term><or>)0 <boolean factor>
```

```
<implication> ::=
(                )1
(<implication> =>)0 <boolean term>

<simple boolean> ::=
(                  )1
(<simple boolean>==)0 <implication>

<boolean expression list> ::=
                      (                              )*
<boolean expression> (,<boolean expression list>)0

<boolean expression> ::=
(<simple boolean>                                              )
(<if clause><simple boolean> else <boolean expression>)
(<case clause>(<boolean expression list>)                      )
```

3.4.2 Examples

```
if b add 1 shift 3 then (case i of (true, b or c)) else
case j of ((u=v) shift 1, false)

x = -2
Y > V or z < q
a + b > -5 and z - d > q **2
p or q and x <> y
g == not a and b and not c or d or e => not f
if k < 1 then s > w else h<= c
if if if a then b else c then d else f then g else h < k
```

3.4.3 Semantics

A boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in Section 3.3.3.

3.4.4 Types

Variables and function designators entered as boolean primaries must be declared boolean (see Section 5.1 Type declarations and Section 5.4.4 Values of function designators).

3.4.5 Operators

The relational operators <,<=,=,>=,>, <> have the meaning: less than, less than or equal to, equal to, greater than or equal to, greater than, not equal to, respectively. A relation take on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The meaning of the logical operators <not>, <and>, <or>, => (implies), == (equivalent), is given by the following function table:

b1 b2	false false	false true	true false	true true
not b1 b1 and b2 b1 or b2 b1 -> b2 b1 == b2	true false false true true	true false true true false	false false true false false	false true true true true

3.4.6 Precedence of Operators

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

3.4.6.1 According to the syntax given in Section 3.4.1 the following rules of precedence hold:

```
first:     arithmetic expressions and pattern
           operators according to Section 3.3.5
second:    <,<=,=,>=,>
third:     -, not (both notations equivalent)
fourth:    &, and (both notations equivalent)
fifth:     !, or (both notations equivalent)
sixth:     ->
seventh:   ==
```

3.4.6.2 The use of parentheses will be interpreted in the sense given in Section 3.3.5.2.

3.4.7 Arithmetic of Boolean Quantities

The representation of booleans and some rules for boolean arithmetic is given in 3.1.6. Here, we add the rules for relational operators:

<,<=,>=,>
are in most cases executed as a subtraction (floating point or fixed point) of the two operands. Thus, you must be prepared for overflow, underflow, or spill.

=,<>
are always performed as a bit by bit comparison of the two operands. This may for instance be utilised to compare two text strings packed into real variables without risk of overflow.

3.5 Designational Expressions

3.5.1 Syntax

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

<switch identifier> ::= <identifier>

<switch designator> ::= <switch
identifier>(<subscript expression>)

<simple designational expression> ::=
{<label>                          }
{<switch designator>              }
{(<designational expression>)}

<designational expression list> ::=
                              {                              }*
<designational expression> {,<designational expression>}0

<designational expression> ::=
{<simple designational expression>
{<if clause>simple designational expression>                    }
                                  else<designational expression>}
{<case clause>(<designational expression list>)                 }
```

3.5.2 Examples

```
L17
p9
Choose(n-1)
Town(if y < 0 then N else N+1)
if Ab < c then L17
          else q (if w <=0 then 2 else n)
case p+q of (L17, P13, Choose(n-1))
```

3.5.3 Semantics

A designational expression is a rule for obtaining a label of a statement (see Section 4 Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (see Section 3.3.3). In the general case the boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (see Section 5.3 Switch declarations) 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 from 1 and up. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4 The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (see Section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1,2,3...,n, where n is number of entries in the switch list.

3.5.5 Switch versus case statement

It is recommended to use the case statements instead of "goto sw(i)". Case statements are much faster and may give a clearer program.

3.6 String Expressions

3.6.1 Syntax

```
<formal string> ::= <identifier>

<string primary> ::=
(<formal string>                )
(<string literal>               )
(string <arithmetic expression>)
((<string expression>)          )
(string primary> add <primary> )

<string expression list> ::=
(                    )*
(<string expression>,)0 <string expression>


<string expression> ::=
(<string primary>                                              )
(<if clause><string primary> else <string expression>)
(<case clause>(<string expression list>)                       )
```

3.6.2 Examples

```
if b then <:ok:> else <:error:>
case i of (<:first:>,<:second>,string ra(increase(j)))
if b then (case i of (string r,fs))
          else case i of(<:ab:>,<<d.dd>)
```

3.6.3 Semantics

A string expression is a rule for computing a string value. The principles of evaluation are analogous to the evaluation of an arithmetic expression. The monadic operator "string" changes the type of a real or long expression into type string. The value of string <real> or string <long> has the same bit pattern as the value of the operand (see below).

String expressions are used as actual parameters and as arguments of the operators "real" and "long".

The value of a string expression is

a short text string	(a literal text string of at most 5 characters, for example <:abcde:>),
a long text string	(a literal text string of more than 5 characters, for example <:result:>),
a layout strin ;	(for example <"dd.dd"+d>), or
a text portion	(6 characters none of which are Nulls. This cannot occur as a literal text, but may

be obtained by the operators "`string`" or "`add`", for example `<:abcde:>add 92`).

3.6.4 Types

The argument of the operator "string" must be of type real or long. A formal string must be a formal parameter specified as string.

3.6.5 Binary Pattern

The binary pattern of a string value is 48 bits with the values given below.

3.6.5.1 Text portion and short text string. The characters of the t string (omitting the string quotes) are represented as their internal value (see 2.0.1) and packed as 8bit bytes from left to right. The 48 bits are filled up to the right with zeroes.

3.6.5.2 Long text string. The text strings are stored in portions of 48 bits (two words). These words may contain

```
long text string reference:
word 1 (24 bits): segm shift 12 add rel
word 2 (24 bits): 1 followed by some undefined bits

character portion:
word 1:    char N+3 shift 16 add char N+4 shift 8 add char N+5
word 2:    char N shift 16 add char N+1 shift 8 add char N+2
or
short text string.
```

The long text string is referenced with a long text string reference specifying on which segment (segm) and at which relator (rel) the text starts.

The characters of the text string are stored as text portions on the backing storage area which is occupied by the algol program.

The first text portion representing the first 6 characters is found on segment "segm" word rel//21 and rel//2 (the 256 words of a segment are numbered 0, 1, 2, ...). The next text portions are found in word rel//23 and rel//22 and so on, until 48 bits representing a new long text string are found (which specifies the continuation of the string on a new segment) or until 48 bits representing a short text string are found (signaling the string end).

3.6.5.3 Layout string. The first 24 bits represent the spaces of the lay out a · follows: First, a 1 followed by a 1 for each leading space of the layout. Second, one 0.

The following bits correspond to the digit positions of the number part (z, f, d, and 0). A bit is 1 if the corresponding digit position is followed by a space, otherwise 0. T The last 24 bits contain:

```
bit  0   0
bit  15  b  -  number of significant digits (z, b, f,
               and d).
bit  69  h  -  number of digit positions before the
               point.
bit 1013 d  -  number of digit positions after the
               point.
bit 1415 pn -  first letter of number part (z=10, f=01,
               d=00, b=11).
bit 1617 fn -  sign of number part (+ =10, =01, no sign
               =00).
bit 1819 s  -  number of digits in exponent.
bit 2021 pe -  first letter of exponent part (z=10,
               f=01, d=00).
bit 2223 fe -  sign of exponent part coded as fn.
```

3.6.5.4 Reference of Strings. When a standard procedure references a string parameter and obtains a text portion as the result, it will accept these 6 characters as the first part of the string and reference the parameter again and again to obtain the next text portions. When a short or a long text string is obtained, the string end is met (null character). This rule implies that the string parameter must have sideeffects to supply new text portions when it is referenced repeatedly. The standard procedure "increase" assists you with this tasks as explained in the following example.

Example:
Let the real array ra(1:n) hold a sequence of text portions terminated by a null character.

This variable text may be used as a string parameter in this way, for instance:

```
i:= 1; write(out,string ra(increase(i)));
```

Write will reference the second parameter, which in turn calls increase(i) and yields the value of ra(1). At the same time i becomes 2. Write will print the text portion held in ra(2) and if it does not contain a null character, write will reference the second parameter again, and so on until the null character signals the end of the text.

3.7 Zone Expressions

3.7.1 Syntax

```
<zone expression> cf. 3.1.1
```

3.7.2 Examples

```
in polyfase (p) polyfase (input(i))
```

3.7.3 Semantics

The value of a zone expression is a zone. Zone expressions are used as actual parameters.

The arithmetic expression is evaluated as a subscript expression. It selects a zone from the zone array. The subscript must obey

```
1 <= subscript <= number of zones declared in the array.
```

4. Statements

The units of operation within the language are called statements. They will normally by executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, shortened by conditional statements, which may cause certain statements to be skipped, and lengthened by repetitive statements which cause certain statements to be repeated.

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

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also declarations are supposed to be already defined as they are used in syntactic definition of statements.

4.1 Compound Statements and Blocks

4.1.1 Syntax

```
<basic statement> ::=
            (<assignment statement>)
            (<goto statement>      )
(         )1                       )
(<label>:)  (<dummy statement>     )
(         )0 (<procedure statement> )
            (<context statement> *))


<unconditional statement> ::=
( <basic statement>    )
( <compound statement> )
( <block>              )

<statement> ::=
( <unconditional statement> )
( <conditional statement>   )
( <repetitive statement>    )

<compound tail> ::=
                ( end             )
<statement>     ( ;<compound tail> )

<simple block head> ::=
( begin             ) (                      )*
(<simple block head>) (;<simple declaration>)1

<context block head> ::=
                              (                                    )
begin <context declaration>( ;<context variable declaration>)0

<context block> ::=
<context block head>;<compound tail>

<block head> ::=
( <simple block head>  )
( <context block head> )

<unlabelled compound> ::= begin <compound tail>

<compound statement> ::=
(          )1 (<unlabelled compound>)
( <label>:)0 (<compound statement> )

<block> ::=
(          )1 ( <block head>;<compound tail> )
( <label>:)0 ( <block>                       )
```

*) Note: a <context statement> is only allowed within a context block

```
<program> ::=
{<block>                                    }
{<compound statement>                       }
{external <procedure declaration>;end}
```

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

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

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

4.1.2 Examples

Basic statements:

```
a:= p+q
goto Naples
START:CONTINUE: W:= 7.993
```

Compound statement:

```
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
```

Block:

```
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
```

Context block:

```
R: begin context (i,r,mode);
integer q1,q2;real w;
          if i = type 1 then
             w:= B(i,1)
```

```
                else
                w:= B(i,2)
end block R
```

Program:

```
external procedure X(p);
integer p;
p:= p + 1;
end external
```

4.1.3 Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (see Section 5 Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

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

A label is said to be **implicitly declared** in this block head, as distinct from the explicit declaration of all other local identifiers. In this context a procedure body, or the statement following a for clause, must be considered as if it were enclosed by begin and end and treated as a block, this block being nested within the fictive block of Section 4.7.3.1 in the case of a procedure with parameters by value.

A label that is not within any block of the program (nor within a procedure body, or the statement following a for clause) is implicitly declared in the head of the environmental block.

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

4.2 Assignment Statements

4.2.1 Syntax

```
<destination> ::=
{ <variable>             }
{ <procedure identifier> }

<left part> ::= <destination> :=
                                    }*
<left part list> ::= { <left part> }1

<assignment statement> ::=
{<left part list><arithmetic expression>}
{<left part list><boolean expression>   }
```

4.2.2 Examples

```
s:= p(0):= n:= n + 1 + s
n:= n + 1
A:= B/C - v - q * S
S(v,k + 2):= 3 - arctan(s * zeta)
V:= Q > Y and Z
ia ('x'):= 6 shift 12 + 'x'
```

4.2.3 Semantics

Assignment statements serve for assigning the value of an expression to one or several destinations.

Assignment to a procedure identifier may only occur within the body of a procedure defining the value of the function designator denoted by that identifier (see Section 5.4.4). If assignment is made to a subscripted variable, the values of all the subscripts must lie within the appropriate subscript bounds.

The location of a zone buffer element designated by a record variable is not influenced by expressions to the right of the record variable, even if these change the position of the record within the zone buffer.

The location of a variable is an absolute address in the RC8000/RC9000-10. The assignment process takes place in three steps as follows.

4.2.3.1 The location of all variables, including subscripted variables, record variables, and simple fields, occuring in the left part are evaluated from left to right.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3 The value of the expression is assigned to all the left part variables with locations as evaluated in step 4.2.3.1 in sequence from right to left.

4.2.4 Types

The type associated with all destinations of a left part list must be the same.

If this type is boolean, the expression must likewise be boolean. If the type is real, long, or integer, the expression must be arithmetic.

If the type of the arithmetic expression differs from that associated with the destinations, an appropriate transfer function is automatically performed. For transfer from real to long or integer type the transfer function yields a result which is the largest integral quantity not exceeding E + 0.5 in the mathematical sence (i.e. without rounding error) where E is the value of the expression. It should be noted that E, being of real type, is defined with only finite accuracy (see Section 3.3.6).

The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (see Section 5.4.4).

Field variables may be used as variables of type integer. The conversion of a real value to an integer or long and the conversion from a long value to an integer are performed so that spill alarm (see User's Manual ref. 14) may occur.

4.3 Goto Statements

4.3.1 Syntax

```
<goto statement> ::= goto <designational expression>
```

4.3.2 Examples

```
goto L8
goto exit(n + 1)
goto Town(if y < 0 then N else N + 1)
goto if Ab < c then L17
       else q(if w < 0 then 2 else n)
goto case p + q of (L17, p13, choose(n - 1))
```

4.3.3 Semantics

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

4.3.4 Restriction

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

4.3.5 Goto an Undefined Switch Designator

A goto statement will terminate the program with an alarm if the designational expression is a switch designator whose value is undefined.

4.4 Dummy Statements

4.4.1 Syntax

```
<dummy statement> ::= <empty>
```

4.4.2 Examples

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

4.4.3 Semantics

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

4.5 Conditional Statements

4.5.1 Syntax

```
<case statement> ::=
<case clause> begin <statement set> end

<if statement> ::=
<if clause><unconditional statement>

<conditional if statement> ::=
{         }*      {<if statement>                }
{<label>:}0       {<if clause> else <statement>}
                  {<if clause><rep. statement> }

<conditional statement> ::=
{<conditional if statement>   }
{ {         }*                }
{ {<label>:}0 <case statement>}

<statement set> ::=
             {            }
<statement> {;<statement>}0

<case clause> cf. 3.3.1

<if clause> cf. 3.3.1

<unconditional statement> cf. 4.1.1

<repetitive statement> cf. 4.6.0.1
```

4.5.2 Examples

```
if x > 0 then n:= n + 1
if v > u then V: q:= n + m else goto R
if s < 0 or P <= Q then
          AA: begin if q < v then a:= v/s
                             else y:= 2 * a
              end
else if v > s then a:= v - q
else if v > s - 1 then goto S
```

4.5.3 Semantics

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

4.5.3.1 If statement. An if statement is of the form

```
if B then Su
```

where B is a boolean expression and Su is an unconditional statement. In execution, B is evaluated; if the result is true, Su is executed; if the result is false, Su is not executed.

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

4.5.3.2 Conditional if Statement. Three forms of unlabelled conditional statement exist, namely:

```
if B then Su
if B then Sfor
if B then Su else S
```

where Su is an unconditional statement, Sfor is a for statement and S is a statement.

The meaning of the first form is given in Section 4.5.3.1. The second form is equivalent to

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

The third form is equivalent to

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

Where localL is an anonymous local label.

4.5.4 Goto into a conditional if statement

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

4.5.5 Case statement

4.5.5.1 Semantics

The case statement

```
case i of
begin S1; S2;...; SN end
```

where S1;.. are statements is equivalent to

```
switch SWIT := SWIT1, SWIT2,...SWITN
goto SWIT(i);
SWIT1:    S1; goto STOP;
.
.
.
SWIT2:    S2; goto STOP;
SWITN:    SN;
STOP:
```

The statements of the statement set are separated by semicolons and numbered 1, 2, 3....

A case construction is executed as follows: First, evaluate the arithmetic expression and if necessary round it to an integer. Next, select the set element corresponding to the result. If no such set element exists, the run is terminated. Execute the selected statement and continue the execution after the complete casestatement (provided that a goto was not executed).

4.5.5.2 Example

```
case i of
  begin
       a:= p+1;
       begin
          q:=p;
          r:=q+1
       end;
       p:=q
  end
```

4.6 Repetitive Statements

4.6.0.1 Syntax

```
<repetitive statement> ::=
(<for statement>   )
(<repeat statement>)
(<while statement> )
```

4.6.0.2 For Statements

4.6.1 Syntax

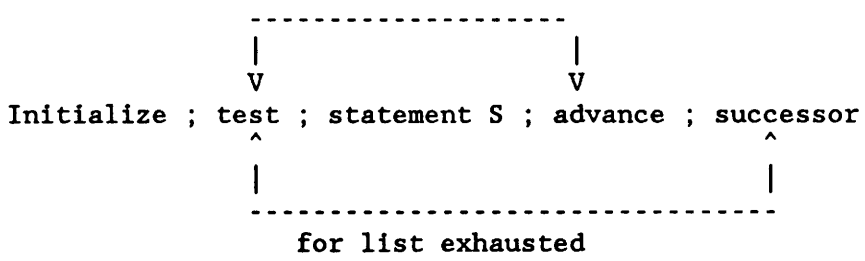
```
<for list element> ::=
(<arithmetic expression>                              )
(<arithm. express.>step<arithm. express.>
(                    until<arithm. express>           )
(<arithmetic expression>while<boolean expression> )

<for list> ::=
                  (                    )*
<for list element>(,<for list element>)0

<for clause> ::= for <variable identifier> := <for list>
do

<for statement> ::=
(        )1
(<label>:)0 <for clause><statement>
```

4.6.2 Examples

```
for q:= 1 step s until n do A(q):= B(q)
for k:= 1, V1 * 2 while V1 < N do
for j:= I + G, L, 1 step 1 until N, C + D do
        A(k,j):= B(k,j)
```

4.6.3 Semantics

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable, which must be a simple variable of real, long, or integer type or a field variable. The process may be visualized by means of the following picture:



In this figure the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so the execution continues with the successor of the for statement. If not the statement following the for clause is executed.

4.6.4 The For List Elements

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written.

The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1 Arithmetic Expression Element

If X is an arithmetic expression then

```
for V := X do S
```

is equivalent to

```
begin
V := X;S
end
```

where S is treated as if it were a block (see Section 4.3.1)

4.6.4.2 Step-Until-Element

If A, B and C are arithmetic expressions and V the controlled variable then

```
A step B until C do S
```

is equivalent to

```
  V:= A;   localB:= B;
L1:  if (V-C)*(localB) > 0 then goto Element_exhausted;  S;
  localB:= B; V:= V + localB;
  goto L1;
```

Where localB is an anonymous variable and Element_exhausted is the end of this step-until-element.

4.6.4.3 While Element

If E is an arithmetic expression, F a boolean expression and V the controlled variable then

```
E while F do S
```

is equivalent to

```
L3:V:=E;
   if-, F then goto Element_exhaused;
S;
goto L3;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5 The value of the controlled variable upon exit.

Upon exit from a for statement, the value of the controlled variable is defined by the algorithms in 4.6.4.2, 4.6.4.1, and 4.6.4.3 above.

4.6.6 Goto Leading into a For Statement.

Any occurrence outside a for statement of a label which labels a statement inside the for statement is forbidden.

4.6.7 Repeat statement

4.6.8 Syntax

```
<repeat statement> ::= repeat <statement set>until
<boolean expression>
```

4.6.9 Examples

```
repeat
   a:= a+1;
   sum:= sum+a
until a > 99

repeat AP(3,4,i) until i=0
```

4.6.10 Semantic

The statement

```
repeat S1; S2;...;SN until boo
is equivalent to the ALGOL construction:
for i:= i, i while - ,boodo
begin
    S1; S2;...;SN;
end
```

Nesting of repeat statements is allowed.

4.6.11 While statement

4.6.12 Syntax

```
<while statement> ::= while <boolean expression> do
<statement>
```

4.6.13 Example

```
while a>b do a:=b
while A(i) do
      begin
        a(i,j):=5;
        a(j,i):=7
      end
```

4.6.14 Semantics

The statement

```
while boo do S
```

is equivalent to the ALGOL construction:

```
for i:= i while boo do
S
```

Nesting of while statements is allowed.

4.7 Procedure Statements

4.7.1 Syntax

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

<actual parameter part> cf. 3.2.1
```

4.7.2 Examples

```
Transpose (W, v+1)
Spur (A)Order:(7)Result to:(V)
is equivalent to Spur (A,7,V)
Absmax (A)beginof search:(N)endof search:(M,Yy,I)
is equivalent to Absmax (A,N,M,Yy,I)
Innerproduct(A(t,P,u),B(P),10,P,Y)
```

These examples correspond to examples given in section 5.4.2.

4.7.3 Semantics

A procedure statem ent serves to invoke (call for) the execution of a procedure body (cf. section 5.4 Procedure declarations). Where the procedure body is a statement written in ALGOL. The effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

The zone of a zone expression is always evaluated before the procudure is entered.

An array field is evaluated before the procedure is entered. The evaluation is made like this:

a) The bound halfwords are computed as shown in section 5.7.5.

b) The lower bound halfword is adjusted relative to the value found above. The adjustment will assure that all halfwords in the lowest array element are available. The adjustment is made as follows:

```
lower_bound_halfword:=
lower_bound_halfword     +     (llower_bound_halfword)
extract log2(type_length)
```

where

```
log2(x) = ln(x)/ln(2)
```

c) A description of a onedimensional array of the resulting type and with these bound bytes is set up local to the procedure. If the

procedure uses this array as an actual array field parameter in subsequent procedure calls, this cutting may be performedagain. Thus, from a certain step, the bytes of an array may be unaccessible from the procedures, if the values of the array field variables are not chosen appropriately.

A parameter specified as a field variable may correspond to an actual parameter of type integer. A field variable as an actual parameter behaves as a variable of type integer.

A parameter specified as a field variable cannot be called by value.

4.7.3.1 Value Assignment (call by value). All formal parameters quoted the value part of the procedure declaration heading (see Sections 5.4 and 4.7.4) are assigned the values (cf. section 2.8 Values and types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictive block with types as given in the corresponding specifications (cf. section 5.4.5). As a consequence, variables called by value may be considered as non local to the body of the procedure, but local to the fictive block (cf. section 5.4.3).

4.7.3.2 Name Replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflict between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

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

4.7.4 Actual-formal Correspondance.

The correspondance 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.

4.7.5 Restrictions

For a procedure statement to be defined it is usually necessary that the operations on the procedure body defined in section 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL8 statement.

This poses the general 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. For reasons of effectivity of the compiled code certain deviations from this general rule are imposed. Some particular cases of this rule together with the deviations are listed in the following:

4.7.5.1 If a string is supplied as an actual parameter in a procedure statement or function designator, where the procedure body is an ALGOL8 statement (as opposed to nonALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in nonALGOL code. (In very specific cases this rule may be circumvented, if the actual string parameters are short strings (cf. Section 3.6.3.)).

4.7.5.2 A formal name parameter which occurs as a left part variable in an assignment statement within the procedure, may actually be an expression which is not a variable (a constant for instance). In this case, the assignment takes place to a fictive variable.

If the actual parameter is a constant, the future value will be taken from this fictive variable, and if it is an expression, the assignment disappears as the fictive variable is a normal used work variable (UV cf. ref. 10).

4.7.5.3 An actual parameter which is an array identifier can only correspond to a formal array parameter with the same number of subscripts or with one subscript. In the latter case, the lexicographical ordering of the array elements is used as explained in 5.2.6. An array field is considered as a onedimensional array (see 4.7.3).

The number, kind and type of any parameters of a formal procedure parameter must be compatible with those of the actual parameter. If a formal not checked whether the actual parameter is a field variable. A formal parameter specified as real array may actually be parameter is specified as a field variable, it is a zone expression. In this case, the array elements are that part of the zone buffer which is selected as the zone record at the moment of the call.

4.7.5.4 (This section has been deleted)

4.7.5.5 Restrictions imposed by specifications of formal parameters must be observed. The correspondence between actual and formal parameters should be in accordance with the following table.

If the actual parameter is itself a formal parameter the correspondence (as in the table on the previous page) must be with the specification of the immediate actual parameter rather than with the declaration of the ultimate actual parameter.

Formal parameter	Mode	Actual parameter
integer	value	arithmetic expression yielding a real, long or integer value.
	name	arithmetic expression yielding an integer value
long	value	arithmetic expression yielding a real, long or integer value
	name	arithmetic expression yielding a long value
real	value	arithmetic expression yielding a real, long or integer value
	name	arithmetic expression yielding a real value
boolean	value	boolean expression
	name name	boolean expression
label		designational expression
integer array long array real array boolean array	name name	integer array long array real array or zone boolean array
integer field	name	arithmetic expression yielding an integer value
long field	name	arithmetic expression yielding an integer value
real field	name	arithmetic expression yielding an integer value
boolean field	name	arithmetic expression yielding an integer value
integer array field	name	arithmetic expression yielding an integer value
long array field	name	arithmetic expression yielding an integer value
real array field	name	arithmetic expression yielding an integer value
boolean array field	name	arithmetic expression yielding an integer value
typeless procedure	name	arithmetic procedure, or typeless procedure, or boolean procedure
integer procedure long procedure real procedure boolean procedure	name name	integer procedure long procedure real procedure boolean procedure
switch string zone zone array	name name	switch string expression zone expression zone array

4.7.6 (This section has been deleted)

4.7.7 Parameter Delimiters.

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones (known as "fat commas") is entirely optional.

4.7.8 Procedure body expressed in slang code.

The restrictions imposed on a procedure statement calling a procedure having its body expressed in nonALGOL code can only be derived from the characteristics of the code used and the intent of the user. Therefore it is outside the scope of this part of the manual.

4.7.9 Standard Procedures

The standard procedures belonging to the ALGOL8 system are described in ref. 14.

4.7.10 Recursive Procedures

Recursive procedures are handled fully in ALGOL8, note however the possible "cutting" of array parameters which are actually arrays fields described in 4.7.3.

If a variable is declared "own" in a procedure body and the procedure is called recursively, the same own variable is used in all the dynamic incarnations of the procedure.

4.8 Context Statements

4.8.1 Syntax

```
<context statement> ::=
(<exit statement>     )
(<continue statement>)

<exit statement> ::=
exit (<designational expression>)

<continue statement> ::= continue
```

4.8.2 Examples

```
exit (L)
exit (Q(1))
continue
```

4.8.3 Semantics

The context statement serves to control special jumps out of a context block and within a context block. The statements should be found in a context block only.

Exit
The exit statement is a goto statement, which leaves a context block in such a manner that the same incarnation (cf. 5.8.1) can start the execution next time with the ALGOL statement immediately following the exit call.

The block level at which an exit statement is found shall be identical with the level of the context block.

The statement shall be found outside forstatements embedded in context blocks.

The exit statement has the following effect:

The return point (also known as the **continue point**), i.e. the logical address for the ALGOL statement following immediately after the exit statement, is stored in an anonymous context variable (cf. 5.8.3). This variable is known as the **context label** belonging to the incarnation.

Jumping to the value of <designational expression>, is done exactly in the same manner as via a goto statement (cf. 3.3).

Several exit statements are permittet in the same context block, and each incarnation has its specific **context label.**

The value of a context label is either a continue point or 0 (zero). A continue point can be defined solely by exit. The zero value can be obtained in the following manner:

First time this incarnation is executed.
When read bit has not been set.
When "new block bit" or "new incarnation bit" is set. (cf. 5.8.3).

Continue
The continue statement is a goto statement that jumps to a context label. The block label at which it is found may be different from the context block level (i.e. inner block). The statement has the following effect:

If the context label belonging to the incarnation is zero, the statement is blind. This means that the block goes on with the next ALGOL statement.

If the context label belonging to the incarnation has the value of a continue point, the block jumps to the point concerned.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (see Section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the begin since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations 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.

At the time of an exit from a block (through end, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration of a simple variable may be marked with the additional declarator own. This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Variables declared in a context block will behave as if they were marked with the declarator own.

No identifier may be declared either explicitly or implicitly (see Section 4.1.3) more than once in any one block head.

All programs may be thought of as surrounded by one common block (the standard identifier block). The declarations of this block are given in the backing storage catalog of the RC 8000. New procedure declarations are inserted in this block when external procedures are translated (see 5.4.7). Procedures expressed in machine language, simple variables, and zones may be inserted in the standard identifier block as described in ref. 10.

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

Syntax

```
<declaration> ::=
(<simple declaration> )
(<context declaration>)

<simple declaration> ::=
(<context variable declaration>)
(<zone declaration>            )
(<zone array declaration>      )

<context variable declaration> ::=
(<type declaration>     )
(<array declaration>    )
(<switch declaration>   )
(<procedure declaration>)
(<field declaration>    )
```

5.1 Type Declarations

5.1.1 Syntax

```
<type list> ::=
                        {                    }*
<simple variable> {,<simple variable>}0

<type> ::=
{real    }
{long    }
{integer}
{boolean}

<type declaration> ::=
{    }1
{own}0 <type><type list>
```

5.1.2 Examples

```
integer p,q,s
own boolean Acryl,n
long lg
```

5.1.3 Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. The range and representation of variables are given in 3.1.

5.2 Array Declarations

5.2.1 Syntax

```
<lower bound> ::= <arithmetic expression>
<upper bound> ::= <arithmetic expression>
<bound pair> ::= <lower bound>:<upper bound>

<bound pair list> ::=
              (              )*
<bound pair> (,<bound pair>)0

<array segment> ::=
<array identifier> ((<bound pair list>))
                   (,<array segment>   )

<array list> ::=
                (                 )*
<array segment> (,<array segment>)0

<array declarer> ::=
(      )1
(<type>)0 array

<array declaration> ::=
(      )1
(<type>)0 array <array list>
```

5.2.2 Examples

```
array a, b, c(7:n, 2:m), s(-2:10, -5:-1, 7:10, 13:i)
real array q(-7:if c < 0 then 2 else 1)
long array zjjj(1:case i of (7, i, 3))
integer array ('nul':'del'
```

5.2.3 Semantics

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

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

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

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

5.2.4 Lower Upper Bound Expressions.

At least one element must be declared.

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (see Section 3.1.4.2).

5.2.4.2 The expressions cannot include any identifier that is declared either explicitly or implicitly (see Section 4.1.3), in the same block head as the array in question. Own variables, having an initial value may however be used.

5.2.4.3 An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds. The array should be declared so that:

```
The product of the lower subscript bounds and the
type_length is >-2**22.

The product of the upper subscript bounds and the
type_length is <2**22.

The product of the number of subscribt elements and the
type_length is <2**22.
```

5.2.4.4 The expressions will be evaluated once at each entrance into the block.

5.2.5 (This section has been deleted)

5.2.6 Lexicographical Ordering

The elements of an array are stored in a sequence, and a multidimensional array declared

```
Am(low1:up1,low2:up2,...lown:upn)
```

may in certain connections (specified in 5.2.6.1 and 5.2.6.2) be considered as a onedimensional array

```
Ao(low:up).
```

Whenever the mapping of Am and Ao makes sense, the element

```
Am(i1,i2,...,in)
```

may be found as

```
Ao(...((i1*c2+i2)*c3+i3)*...+in)
```

where

```
c2 = up2 - low2 + 1, c3 = up3 - low3 + 1, and so on.
```

This mapping of elements is called the lexiographical ordering because it is a linear ordering of the elements obtained by varying the first subscripts at the lowest rate.

The values of low and up may be seen to be:

```
low = ...((low1*c2 + low2)*c3 + low3)* ... + lown
up = ...((up1 *c2 + up2 )*c3 + up3 )* ... + upn
```

It may also be seen that the (possibly fictive) element

```
Am(0,0,...,0) is the same as Ao(0).
```

5.2.6.1 Multidimensional array as actual parameter.

A multidimentional array may occur as an actual parameter where the corresponding formal is a one dimensional array. The mapping above is used in that case.

5.2.6.2 Multidimensional array as field base.

Whenever a multidimentional array is used in a field reference as the (ultimate) field base, the halfword numbering and addressing described in 5.2.7 and 5.2.8 is found by mapping the multidimensional field base on a onedimensional field base according to the rules above.

5.2.7 Bound Halfwords and Halfword Numbering.

Each element of an array is represented by a number of halfwords. This number is the type length explained in section 3.1.6.

The first halfword in an array is called the lower bound halfword and the last one the upper bound halfword. Let an array be declared

```
A(low:up)
```

then

```
lower_bound_halfword = (low 1)*type_length + 1
upper_bound_halfword = up*type_length.
```

The halfwords of an array are numbered relative to the rightmost halfword in the (possibly fictive) element A(O). The element A(i) contains the halfwords

```
(i 1)*type_length + 1 = halfword_number <= i* type
length.
```

5.2.8 Word Boundaries and Addresses.

When an array is declared, it is created so that the word boundaries are between an even numbered halfword and its odd numbered successor.

An array element, A(i), is addressed within the array be the halfword with the number i*type_length.

5.3 Switch Declaration

5.3.1 Syntax

```
<switch list> ::=
                                {                                }*
<designational expression>{,<designational expression>}0

<switch declaration> ::= switch<switch identifier>:=
<switch list>
```

5.3.2 Examples

```
switch S:= S1,S2,Q(m), if v > 5 then S3 else S4
switch Q:= p1,w
```

5.3.3 Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2,..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (see Section 3.5 Designational expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4 Evaluation of Expressions in the Switch List.

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

5.3.5 Influence of Scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4 Procedure Declaration

5.4.1 Syntax

```
<formal parameter> ::= <identifier>

<formal parameter list> ::=

(                                          )*
<formal parameter> (<parameter delimiter><formal
parameter>)0

<formal parameter part> ::=
(                            )1
((<formal parameter list>))0

<identifier list> ::=
            (              )1
<identifier>(,<identifier>)0

<value part> ::=
(                            )1
(<value <identifier list>;)0

<specifier> ::=
( string              )
( <type>              )
( array               )
( <type> array        )
( label               )
( switch              )
( procedure           )
( <type> procedure    )
( <type> field        )
( array field         )
( <type> array field  )
( zone                )
( zone array          )

<specification part> ::=
(                               )1
(<specifier>identifier list>;)0

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

<procedure body> ::= <statement>

<procedure declaration> ::=
(      )1
(<type>)0 procedure<procedure heading><procedure body>
```

5.4.2 Examples

```
procedure Spur(a,n,s);
value n; real array a; integer n; real s;
<*sum all elements in the diagonal up to element a(n,n)*>
                    begin integer k;
                              s:=0;
                              for k:= 1 step 1 until n do s := s + a(k,k)
                    end
procedure Transpose(a,n);
value 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 stepfct(u);
value u; real u;
stepfct:= if 0<=u and u<=1 then 1 else 0

procedure Absmax(a,n,m,y,i);
value n,m;real array a;integer n,m,i;real y;
<* The absolute greatest element in the vector a, and with a
subscript interval n<= index <=m is transferred to y, and the
subscript of this element to i *>
                begin integer p;
                   y:= 0; i:= n;
                   for p:= n step 1 until m do
                   if abs a(p) > y then
                   begin y:= abs a(p);
                      i:=p
                   end
                end Absmax

procedure Innerproduct(a,b) Order:(k,p) Result:(y);
<* here the fat comma )<letter string>:( is used as parameter
delimiter, the above is equivalent to procedure Innerproduct
(a,b,k,p,y);*> integer k,p; real y,a,b;
                begin real s;
                   s:=0;
                   for p:= 1 step 1 until k do s:= s + a * b;
                   y:=s
                end Innerproduct
```

5.4.3 Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators

may be activated from other parts of the block in the head of which the pr block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certainidentifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2 Function designators and section 4.7 Procedure statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal 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 labeling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity. If the procedure body is a block, an identifier of a formal parameter must not be declared anew in this outermost block.

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

5.4.4 Values of Function Designators.

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

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

If a function designator is used as a procedure statement, the resulting value is discarded, but such a statement may be used, if desired, for the purpose of invoking sideeffects (cf.3.3.5).

5.4.5 Specifications

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

Restrictions on the actual/formal correspondence are listed in section 4.7.5.5. Note that arrays cannot be value specified.

5.4.6 Code as Procedure Body

Procedures may be expressed in machine language and introduced into the standard identifier block (see the introduction to chapter 5) as it is explained in ref. 10.

5.4.7 Procedures Translated Alone

Procedures expressed in Algol may be translated alone, provided that they have at most seven parameters. A procedure translated alone will be introduced into the standard identifier block mentioned in the introduction to chapter 5. In order to translate the procedure alone, the procedure declaration must be surrounded by the delimiters external and end, and a semicolon (;) must delimit the procedure declaration from the end.

A procedure translated in this way becomes a standard procedure included in the standard identifier block. The name of the procedure is the name of the backing storage area into which the procedure is translated.

All identifiers used as standard identifiers by the procedure must be present at the time of compilation. The name of an external procedure must not contain capital letters as these are forbidden in names of backing storage areas.

5.4.7.1 Example

```
external real procedure greatest (x,y,i);
value x,y;real x,y;integer i;
begin
    real work;
    i:= 0;
    work:= if x > y then x else y;
    if work = y then i:= 1;
    greatest:= work;
end;
end
```

5.5 Zone Declarations

5.5.1 Syntax

```
<length> ::= <arithmetic expression>

<shares> ::= <arithmetic expression>

<block proc> ::= <procedure identifier>

<zone segment> ::=
<zone identifier> [(<length>,<shares>,<block proc>)]
                  [,<zone segment>                  ]

<zone list> ::=
                [                 ]*
<zone segment> [,<zone segment>]0

<zone declaration> ::= zone <zone list>
```

5.5.2 Examples

```
zone master(2*bl,2,stderror)
zone m1,m2(a,b,c),m3(900,3,pr)
```

5.5.3 Semantics

A zone declaration declares one or several identifiers to represent zones. The arithmetic expressions in the declaration are evaluated once for each entrance into the block. Each zone consists of:

a buffer area
a zone descriptor
one or more share descriptors (often just called shares)

Inside the block, a zone identifier may occur as an actual parameter, as a constituent of a record variable, or as a field base (cf. 3.1).

Buffer area
The length in reals of the buffer area for any zone is given by <length> in the first parenthesis following the zone identifier

Each element of the buffer area may be used as a real variable as explained for zone record below. The elements are in some connections identified by a halfword number in the range

```
1 <= halfword number <= 4*length.
```

Zone descriptor

A zone descriptor consists of the following set of quantities, which specify a process or a document (see ref. 1) connected to the zone and the state of this process:

process name
A text string specifying the name of a process or a document connected to the zone.

mode and kind
An integer specifying mode and kind for a document +see ref. 14, open).

logical position
A set of integers specifying the current position of a document.

give up mask
An integer specifying the conditions under which block proc is to be called.

state
An integer specifying the latest operation on the zone.

record
Two integers specifying the part of the buffer area nominated as the zone record.

used share
An integer specifying the share descriptor within the zone, which is used for the moment.

last halfword
An integer specifying the end of a physical block on a document.

block procedure
The procedure block proc in the first parenthesis following the zone identifier. The give up mask and status of the connected document specifies when this procedure is called.

Share descriptor

Each zone contains the number of share descriptors given by shares in the first parenthesis following the zone identifiers. The share descriptors are numbered 1, 2,...,shares.

A share descriptor consists of a set of quantities which describe an external activity sharing a part of the buffer area with the running program.

An activity may be a parallel process transferring data between a document and the buffer area, or it may be a child process executed in the buffer area under supervisiory control of the algol program. See ref. 14.

The set of quantities forming one share descriptor is:

share state
An integer describing the kind of activity going on in the shared area.

shared area
Two integers specifying the part of the buffer area shared with another process by means of the share descriptor.

operation
Specifies the latest operation performed by means of the share descriptor.

Zone record

A number of consecutive halfwords of the buffer area may at run time be nominated as the zone record. The halfwords of the zone record may be available as record variables, which may be thought of as a kind of real subscripted variables. The record variables are numbered 1, 2..., record length (the max addressable word in the buffer), and referenced as described in 3.1.

All halfwords of the record may be referenced by means of field references, as the zone may be used as a field base.

Zone after declaration

The following parts of the zone is defined just after declaration.

used share is set to first share record is set to the entire buffer area block proc

The entire buffer area can be accessed as one zone record.

5.5.4 Types

The two expressions <length> and shares> must be of type integer. The procedure <block proc> must be declared like this:

```
procedure block proc> (z,s,b); zone z; integer s,b;
```

5.5.5 Scope

All identifiers occuring in <length> and <shares> must be nonlocal to the block. However, <block proc> may also be local.

At the time of exit from the block (through end, or by a go to statement), the activities described by the share descriptors are terminated as follows: A communication with a parallel process is completed by means of the monitor function wait answer (see ref. 1). A running child process is stopped (but not removed, see ref.1).

5.5.6 Standard Zones

Two zones, "in" and "out", are available without declarations. The declaration is similar to zone in,out (128,1,stderror). (See ref. 14).

5.5.7 Standard Block Procedure

A procedure "stderror" exists which can be used as standard block procedure without declaration (cf. ref.14).

5.6 Zone Array Declarations

5.6.1 Syntax

```
<zones> ::= <arithmetic expression>

<length> ::= <arithmetic expression>

<shares> ::= <arithmetic expression>

<block proc> ::= <procedure identifier>

<zone array list> ::=
{<zone array list>,<zone array list>                                    }
{<zone array identifier>(<zones>,length>,<shares>,<block proc>)}

<zone array declaration> ::= zone array <zone array
list>
```

5.6.2 Examples

```
zone array inmerge(3,2*600,2,stderror),
           outmerge(3,2*600,2,stderror)
```

5.6.3 Semantics

A zone array declaration declares one or more identifiers to represent onedimensional arrays of zones. The arithmetic expressions in the declaration are evaluated once for each entrance into the block. Each zone array consists of as many zones as specified by zones . All these <zones> are declared with <length>, <shares>, and <block proc> as specified (cf. section 5.5). The zones of a zone array are numbered 1, 2, . . . , <zones>. Inside a block, a zone array identifier may occur as an actual parameter, as a constituent of a subscripted zone occurring as a parameter (cf. 3.7), or as a constituent of a record variable (cf. 3.1).

5.6.4 Types

<zones> must be of type integer. See section 5.5.4 for <length>, <shares>, and <block proc>.

5.6.5 Scope

All identifiers occurring in zones must be nonlocal to the block. See section 5.5.5 for <length>, <shares>, <block proc>, and the exit from the block.

5.7 Field Declarations

5.7.1 Syntax

```
<field list> ::=
                    {                    } *
<field variable> {,<field variable>} 0

<simple field declaration> ::= <type> field <field list>

<array field declaration> ::=
{       }1
{<type>}0 array field <field list>

<field declaration> ::=
{<simple field declaration>}
{<array field declaration> }
```

5.7.2 Examples

```
integer field           if1, if2
long field              lf1
boolean array field     bfa1, bfa2
array field             rfa1, rfa2, rfa3
```

5.7.3 Semantics

A field declaration serves to declare one or several identifiers as field variables.

Field variables are integers and may be used whereever an integer variable may be used.

A variable field declaration declares simple field variables and an array field declaration declares array field variables.

The type declared together with the field variables, the associated type, has no meaning outside field references.

All field variables declared in one declaration have the same associated type. If no type declarator is given in an array field declaration the type real is understood.

5.7.4 Location of a Variable Field

A variable field is located within a field base which may be an array, a zone record, or an array field.

The denotation of a variable field is shown in section 3.1.

The variable field consists of as many halfwords as the type length of the associated type shows.

A variable field cannot occupy halfwords outside the bound halfwords (cf. section 5.2.7 and section 3.1.4.3).

The integer value of the field variable specifies a halfword number.This halfword number is used as an address in the corresponding field base.

Boolean fields are addressed by their halfword number. Integer, long, and real fields are synchronized with the word boundaries (cf. section 5.2.8) of the RC8000/RC9000-10. Integer fields are addressed by one of the 2 bytes forming the integer word. Long and real fields are addressed by one of the 2 halfwords in the right hand word. The address must be

```
>= lower bound halfword + type length - 1
```

and it must be <= upper bound halfword, of the corresponding field base.

5.7.5 Location and Bounds of an Array Field

An array field is located within the field base. For an array field variable the halfword number should be lower bound halfword 1, where lower bound halfword is the lower bound of the wanted array addressed within the field base. The halfword number referring to a certain piece of data in the array field is found by subtracting the value of the array field variable from the corresponding halfword number in the field base.

If the field base is an array field, this rule may be used recursively.

The bound halfword numbers are given by the formula:

```
bound halfword of array field =
bound halfword of field base - value of array field
variable.
```

A subscripted element in an array field is addressed according to the rule in section 5.2.8. The address of a subscripted element must be >= lower bound halfword + type length - 1 and it must be <= upper bound halfword.

5.8 Context Declarations

5.8.1 Syntax

```
<context declaration> ::=
context(<incarnation>,<no of incarnations>,<context
mode>)

<incarnation> ::= <arithmetic expression>

<no of incarnations> ::= <arithmetic expression>

<context mode> ::= <arithmetic expression>
```

5.8.2 Examples

```
context (i, M+N, 3)
context (if p then 1 else 4, Q, 1 shift 1)
```

5.8.3 Semantics

A context declaration serves to indicate that the actual block is a context block (cf. 4.1.1). The variables declared in the block are denoted context variables.

A context declaration defines a number of incarnations of the declared block; these incarnations are numbered: 1, 2, 3 ..., <no of incarnations>; the declarator parameter <incarnation> represents such an incarnation number. To each execution of a context block is related an incarnation number. If two different executions of a context block have the same incarnation number, they define the same incarnation of the context block. The number of different incarnations is thus equal to the value of the declarator parameter: <no of incarnations>.

The effect of a context declaration is now that the context variables declared in the context block are initialized to values depending on the incarnation before the first statement in the block is executed. When the block is left (via the last end in the block or by means of a goto statement) the values of its context variables are stored, and these variables will be initialized to the stored values in the following execution of the same incarnation of the block.

The parameter <context mode> affects this initialization and storing of context variables. See 5.8.3.4. In more detail, the context declarator functions as follows:

5.8.3.1 Incarnation Interval.

When a context block is executed first time, the actual value of <no of incarnations> defines the number of different incarnations of the block. This value will then remain unchanged for the rest of the program run, although the value may be changed by the program during the run. The value can be changed using the context mode "new block bit" (cf. 5.8.3.4). This value defines the incarnation interval: 1 <= incarnation <= no of incarnations.

5.8.3.2 Initialization of Context Variables.

When a context block is entered, and before the first statement is executed, the following is done:

The value of <incarnation> is evaluated, and it is verified that it is within the incarnation interval. This incarnation is used throughout the execution of the context block, although the value of <incarnation> may be changed in the block. This concept is similar to value parameters in procedures. To an incarnation is related just one record, the fields of which are identical with the declared context variables. To each context block are thus connected <no of incarnations> records. Such records are known as context records. The context variables of the block are initialized if the incarnation concerned has been executed before, the contents of the corresponding context record will be transferred to the context variables of the block. If it is a firsttime execution of the incarnation concerned, all the context variables of the block are set to 0. This zeroset of all variables is performed when "new incarnation bit" is specified, too (cf. 5.8.3.4).

The actual array lengths defines at the same time the maximum array lengths applicable to this incarnation. This means that an array length, in all subsequent runs of this incarnation, shall be less than or equal to the maximum length.Transfer of values between context records and context variables is done in accordance with the common lexicographic procedure.

5.8.3.3 Storage of Context Variables.

When a context block is left (via the last end in the block, an exit statement, or a goto statement) the context variables of the block are stored in the context record belonging to the incarnation, as follows:

If it was the first time the incarnation was executed the corresponding context record is established in the virtual memory connected with the program. The values of the context variables of the block are transferred to the context record belonging to the block and its incarnation.

If several context blocks are nested into one another, and if jumpouts occur from several block levels, the process described above will develop for each of the context blocks thus being left.

5.8.3.4 Context Mode.

The actual value of <context mode> affects the process described in 5.8.3.3. The value is regarded as a bit pattern:

1 shift 0 (read bit):
The updating of the context record described in 5.8.3.3 is not executed unless read bit is set; if read bit is not set, solely the zerosetting of context variables is performed.

1 shift 1 (write bit):
The updating of the context record in virtual memory described in 5.8.3.3 is not executed unless write bit is set. Write bit = 0 is therefore usable for references to and searching in context records.

1 shift 2 (save bit):
Every time a context block is left the context variables of the actual incarnation are saved on backing storage and in virtual memory.

1 shift 3 (new block bit):
Same function as if this context block was executed first time. If context records has been established, such records are ignored.

1 shift 4 (new incarnation bit):
Same function as if this incarnation was executed first time.

5.8.4 Types

<incarnation>, <no of incarnations> and <context mode> must all be of type integer.

Appendix A. References

Part numbers in references are subject to change as new editions are issued and are listed as an identification aid only. To order, use package number.

1 PN: 991 11255
RC9000-10 System Software
delivered as part of SW9890I-D, Monitor Manual Set

2 PN: 991 11259
Monitor, Reference Manual
delivered as part of SW9890I-D, Monitor Manual Set

3 PN: 991 03435
Monitor, Part 3, Definition of External Processes
(for model -10, equivalent information is found in the *LAN Device Processes* and *Channel Device Processes* manuals, parts of SW9890I-D.

4 PN: 991 04162
RC8000, Computer Family, Reference Manual

6 PN: 991 11263 (Part 1), 991 11264 (Part 2)
System Utility Programs, User's Guide
delivered as part of SW8010I-D, System Utility Manual Set.

PN: 991 11294 (Part 3)
System Utility Programs, Part Three
delivered as part of SW8585-D, Compiler Collection Manual Set.

7 PN: 991 11274
BOSS User's Guide
delivered as part of SW8101I-D, BOSS Manual Set.

8 PN: 991 11260
Operating System s, Reference Manual
delivered as part of SW9890I-D, Monitor Manual Set

9 PN: 991 11292
RC Fortran, User's Manual
delivered as part of SW8585-D, Compiler Collection Manual Set.

10 PN: 991 11296
Code procedures and the run time organisation of ALGOL programs
available on request

11 R.M. De Morgan et al.: *Modified Report on the Algorithmic Language Algol 60.* The computer Journal, Vol 19, no. 4, pp 364-379.

12 J.W. Backus et al: *Revised Report on the Algorithmic Language Algol 60* (ed. Peter Naur), Comm. ACM 6 no.1 (1963), pp 1-17.

13 ISO: R646 - 1967 (E) *6 and 7 bit coded character set for information processing interchange.*

14 PN: 991 11280
ALGOL8, User's Guide, Part 2 part of SW8585-D, Compiler Collection Manual Set.

Appendix B. Index

All references are given through section numbers.
The references are given in three groups:

Definition:	Following the word "definition", reference to the syntactic definition (if any) is given.
Use:	Following the word "use", references to the occurrences in metalinguistic formulae are given. References already quoted in the defgroup are not repeated.
Text:	Following the word "text", the references to definitions given in text are given.