NOTICE: When government or other drawings, specifications or other data are used for any purpose other than in connection with a definitely related government procurement operation, the U. S. Government thereby incurs no responsibility, nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use or sell any patented invention that may in any way be related thereto.
Progress Report

RESEARCH AND DEVELOPMENT IN PROGRAMMING STUDY ASSIGNMENTS

Contract No. DA-36-039 SC-75081
File No. 0195-PH-58-91 (4461)
DA Task Number 3B28-04-001-01-04

I. Final Report
II. Flow Charts
III. Programming Manual
22 February 1961

for

UNITED STATES ARMY SIGNAL R&D LABORATORY
FORT MONMOUTH, NEW JERSEY

by

A. J. Perlis

ASTIA AVAILABILITY NOTICE

Qualified requestors may obtain copies of this report from ASTIA

Carnegie Institute of Technology
Schenley Park, Pittsburgh 13, Pennsylvania
Errata and Addenda
to the
Final Report
Contract No. DA-36-039 SC-75061
DA Task Number JB28-C4-001-C1-04.
October 19, 1961
by
A. J. Perlis

for
UNITED STATES ARMY SIGNAL R&D LABORATORY
FORT MONMOUTH, NEW JERSEY

ACTIA AVAILABILITY NOTICE
Qualified requesters may obtain copies of this report from ACTIA.

* Carnegie Institute of Technology
School of Engineering
Schenley Park, Pittsburgh, Pennsylvania
1. p2, L 7  Amplification

The input is in "string" format, i.e., as a concatenation of characters from the 20^4 L alphabet. The only formal structure to such a string is a beginning (1st character) and an end (last character).

The transformation to "tree" form is accomplished by differential behavior of the 20^4 translator on observing certain character combinations in the input string. The "tree" form has punctuation introduced (such as "and") which permits the decomposition of the input string into levels (and thereby sub-levels, etc.)

2. p2, L 12

A terminal such as 1. A terminal such time.

3. p3, L 16  Amplification

"Atomic" machine instructions are those wired in commands, or even those precomposed in fixed order from such commands.

4. p5, L 6  Amplification

:= is a metalinguistic connective meaning "is defined as", i.e., a := b means a is defined as b.

5. p8, L 14

<identifier> :=<letter> ... <letter>|<identifier>|<letter>|<identifier>|<digit>|
<identifier>|<special register identifier>

6. p 11, L 4  2

(7  3

1. a → b means a is replaced by b
2. L 8 means 8th line from the bottom
Element sequencing is explained in detail in (7). Suffice it to say here that its purpose is to examine symbols independently of the structures in which they are imbedded. Word and list sequencing on the other hand are structure dependant.

Amplification: if $k$ is missing, it is assumed to be 1.

Arithmetic Expressions

Amplification

The vertical bar here used is not to be confused with the metalinguistic connective "or". This meaning is given in 3.7.3.

These examples correspond to examples of procedure declarations given in section 5.4.2.
Procedure heading \rightarrow process declaration heading

Procedure heading \rightarrow process declaration heading

(A C t, P ... \rightarrow (A C t, P ...)

 insertion after

\langle storage function \rangle := \langle arithmetic expression \rangle

 insertion after

\langle bound pair set \rangle := [\langle bound pair list \rangle] \langle storage function \rangle

insertion after

\langle bound pair list \rangle \rightarrow [\langle bound pair set \rangle]

 insertion after

real array \mathcal{F}[i:n, l:n] \quad (n-\xi^l+1) \times (\eta^l-1) + \eta^l-1

see 5.2.3.4, for the semantics of \langle storage function \rangle.

 insertion after

5.2.3.4. Storage function:

An arithmetic expression which is a mapping of a two dimensional array of information into a one dimensional ordered set of information.

A systematic method of specifying the row and column indices of the elements in a two dimensional array has to be made. While this can be done by defining the storage function as a procedure with formal parameters this complexity is absurd since very few uses of storage functions will require the procedure mechanism. Thus one should specialize on two characters representing the row, e.g., \xi, and column, e.g., \eta, indices for which substitution is made by evaluation of the expression each time and element of
the array is to be identified. The definition of these special characters can be imbedded in the syntax by adding to the definition of arithmetic expressions the syntactic types \( \langle \text{row primary} \rangle \) and \( \langle \text{column primary} \rangle \) as in section 3.3.1. The characters \( \xi \) and \( \eta \) (or their equivalents) can be added to the \( ^{20} \mathrm{L} \) alphabet.

25. p 50, L 14

\[
; \langle \text{specification part} \rangle \ldots \
\]
\[
\langle \text{value part} \rangle \langle \text{specification part} \rangle
\]

26. p 52, L 14

The use of \( \ldots \)

The use of Procedure Statements (see 4.7) and \( / \) or Function Designators (see 3.2) \( \ldots \).

27. p 55, L 9, L 11

delete \( / \)

Part III.

28. p 99, L 2-

discussed and \( \ldots \) discussed (see Part I) and

29. p 100, L 1

delete

30. L 1-

or combination \( \ldots \) or combination of : 

31. p 102, L 5

\( 101.1 \rightarrow 101 \)

32. p 112, L 5

\( \ldots \leq x < n \) \( \) (divides \( \ldots \))

\( \ldots \leq x < n \) \( \land \) \( \rightarrow \) (divides \( \ldots \))

33. p 112, L 7

begin if (divides \( \ldots \))

begin if \( \rightarrow \) (divides \( \ldots \))

34. p 113, L 1

delete

35. L 5

insert

where \( * \) is used to represent multiplication
Here, the last two digits represent the exponent, \( p \), of a floating point number where the exponent is represented as \( p + 50 \).

The notation \( \ldots \) represents intervening statements immaterial to the points under discussion.

\( R := p \ldots \rightarrow R := p^4 + 3 \times p; \)

\( p = 101 \) (2) and \( p = 5 \) (10) mean \( p = 101 \) in binary and \( p = 5 \) in decimal, respectively.

Part II.

49. p 58, L 1   a 20 L   a 20 L running

50. p 59, bottom append

where \( \sim \) denotes a blank, and \( Bt \) denotes the block tag.

51. p 60 L 8 insert

where CBT is the Code Block Table.

52. L 6- notation.

\[ \sim \] notation and \( \Theta \) is some (at most) binary operator.

53. L 5- missing,

\[ \sim \] missing depending on the operator.

54. p 61 L 1, 2 delete

55. L 5 or all \( \Theta \) of all

56. L 8 insert

- subtraction

57. L 20 insert

J jump

go to transfer

58. p 62, L 9- the right of \( \sim \) the left of

59. p 64 page number should be 63.

60. p 63, L 5 the \( O \) code \( \sim \) the \( S \) code

L 7 " 

L 14 in IS \( \sim \) in the input sequence IS

L 18 in, I C and \( \sim \) in IS and GS.

L 21 will be \( \sim \) will often be

L 2- of the form \( \sim \) of that form
61. p 64, L 8- insert

Note: In the following $O$ and $O_S$, I and IS, refer interchangeably to output and input sequences, respectively. Similarly isrt and insrt are used interchangeably.

62. bottom

Note: 'Blend' is inserted by the main declaration and means block end.

63. p 65, L 6 insert

$MT$ is a transfer to the routine such that when upon completion returns to the statement following the statement $MT$.

replace by

$GBT \{ \{ J \} := Bt, \langle \tau \rangle, (S)$

isrt $(oL, oJ)$, next $(oL, oJ) := 'blend'$

Notes: The vertical $\mid$ is used in the sense of address expressions (see section 3.7).

The routines isrt$(X)$ and next $(Y)$ are defined in reference 3 and essentially cause an empty site to be inserted following the site$X$; and the site one beyond the pointer for $Y$ to be referenced, respectively.

64. p 66, L 2-

Note: the digit 1 is occasionally used in place of the value true; and similarly so for 0 and false.

65. p 68, L 5 next $(oL, oJ) := \rightarrow$ next $(oL, oJ) :=$

66. p 68, L 3 $K := 1 \rightarrow K := K + 1$

$[K] \rightarrow [K']$

67. p 68, L 5 Note: $\sigma$ stands for the operator which is to be compiled in this line of code. It is generated in the expression analyzer.
Comment: declaration handles lists of identifier possessing the same declared attributes and, in particular, handles array declarations. In case of

* column * column, for a rectangular array.

insert:

(2) of dimension, there is computed
space := abs ( n-m+1 )
base := storage base + 1
storage base := storage base + space

and base is stored in the address assigned to A. The mapping function for A[ ] is then base + i.

true → true

String transfer is the table of Macro identifiers → String transfer becoming the number of characters in the Macro.

insert after end:

(this indicates the end of the Macro declaration)

J := line number (library table (A))
9. p 71, L 3- should read

\[ J := \text{field (2, library table } [J]) \]

80. p 72, L 1 should read

\[ \text{next (next (I [J]))} : = \text{procedure} \]

81. p 72, L 3 Note:

Copy serves to copy the list whose starting address is in \( J \)
into \( I [J] \)

82. p 72, L 7- insert after 'there':

is in the output sequence

83. p 72, L 3- a 0 (1) \( \rightarrow \) a = 0 (1), respectively

84. p 72, L 1- b 0 (1) \( \rightarrow \) b = 0 (1), respectively

85. p 73 replaced entirely by: (See page 10, )
The catalogue of actions in the four cases are given in the following table:

<table>
<thead>
<tr>
<th>Case</th>
<th>Declaration</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$a = 0$</td>
<td>$a = 0, b = 0$ generate code for $A + j + A+l \times i$</td>
</tr>
<tr>
<td></td>
<td>for $A$, $i$, $j$</td>
<td>execution of above</td>
</tr>
<tr>
<td></td>
<td>$A := \text{base}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A + 1 := \text{Column}$</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>no action</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Declaration</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$a = 0, b = 1$ generate code for $A + j + A+l \times i$ if local identifier $\gamma_s$ in B.A. routine and compute $\beta_s$ in I. and compute $A + j + A+l \times i + I.$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for $A$, $i$, $j$</td>
<td>execution of above</td>
</tr>
<tr>
<td></td>
<td>$A := \text{base}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A + 1 := \text{column}$</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>no action</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Declaration</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$a = 1, b = 0$ (dynamic declaration, not inside procedure) generate code for $A + j + A+l \times i$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for $A$, $i$, $j$</td>
<td>execution of above</td>
</tr>
<tr>
<td></td>
<td>$A := \text{base}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A + 1 := \text{column}$</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>execution of above</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Declaration</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$a = 1, b = 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for $A[\gamma_s] := \text{base}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A + 1[\beta_s] := \text{column}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A[\gamma_s] + j + A+l[\beta_s] \times i$ if local identifier $\gamma_s$ in B.A. routine compute $\beta_s$ in I. store $\beta_s$ in I.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>execution of above</td>
<td>execution of above</td>
</tr>
</tbody>
</table>
86. p 74, L 3  
Note:

\[ P[i] \] refers to the \( i^{th} \) line of a table containing either an operator or operand of the expression being analyzed. There is assumed to exist an internal list of operators whose order specified the hierarchy of their execution order, e.g.,

\[ \uparrow, \ast, /, +, - \]

means \( \uparrow \) done before \( \ast \) before \( / \), etc.

87. p 74, L 12  
delimiter \( \rightarrow \) 

88. p 74, L 6  
\( := \) digit \( \rightarrow := \) decimal digit

\( := \) \( \rightarrow := \) \( 10 \) (the notation for base 10)

89. p 75, L 5  
Note:

\( m.j \) means code line \( m \) \( j \) is an index

90. p 77 L 4  
\( e(1); \rightarrow e(1)); \)

91. p 77 L 3  
Note:

In each instance the argument \( p \) of code line \( (9) \) should be delimited by \( \rightarrow \) to indicate the nature of the substitutions being employed. 

See p966

92. p 77 L 7  
\( m;i'; \) \( P[i+2] \rightarrow =i;A \) \( P[i+2] \)

93. p 77, L 12  
\( =i \) \( \rightarrow := \)

94. p 77, L 14  
should read

if \( P[i] \) = Expression terminal \( [k] \) then go to \( (\text{expression analyzer}) \)

95. p 77, L 19  
\( I_o \) \( \rightarrow I_\circ \)

96. p 78, L 1  
\( I_o \) \( \rightarrow I_\circ \)

97. p 78, L 2  
(\( R [k] \)) \( \rightarrow (L [k] \))

98. p 78, L 3  
go_to \( (L [k] \) \( \rightarrow \) go_to \( (L [k] \)) ; \)
99. p 79, L 10 should read

else go to (expression analyzer) and

100. p 80, L 13 \[MT_{BA} \rightarrow MT_{(BA)}\]

, L 1

p 81, L 9

p 81, L 1

101. p 81, L 2 \[EX_{16} \sim E \times 20\]

102. p 81, L 3 \[P_{[1-5]} \sim \phi (P[1-5])\]

103. p 82, L 2 \[EX_{16} \sim \sim EX_{20}\]

104. p 83, L 10 should read

\[\text{go to (j, y)}\]

105. p 86, L 4 \[\text{begin : I \sim begin I}\]

106. p 86, L 13 \[\text{for list \sim for list}\]

107. p 86, L 2 \[\text{next (L,j)} \sim \text{next (O,i)}\]

108. p 88, L 14 should read

i = \[\text{generated by comma}\]

109. p 89, L 11 \[x \sim x\]

110. p 90, L 9 \[T[H] \sim \phi (IT[H])\]

111. p 90, L 8 should read

Bl 2: if marker (IT[H]) \(\neq ')\)

112. p 90, L 7 should read

\[\text{then begin} \ H = H + 1; \ \text{go to Bl2 and}\]

113. p 90, L 6 number > number [\(\sim\)]

114. p 91, L 1ff Note:

T and IT are used interchangeably for the identifier table.
norm field $\Rightarrow$ norm (field

is A $\Rightarrow$ is in A

operand $\Rightarrow$ operator

MT $[z] \Rightarrow$ MT $[z]$

seq N (G, $\Rightarrow$ seqw (G,

seqw N (V, $\Rightarrow$ seqw (V,

( -L.e $\Rightarrow$ -L.e

go to $\Rightarrow$ go to

code line (I.O. $\Rightarrow$ code line (I.O.)
INTRODUCTION

A single programming language is described. Though it has, from the programmer's point of view, three forms, there is in reality only one language and only one processor for producing machine code.

In keeping with historical precedence the three forms as mentioned are:

(i) An algebraic language
(ii) A symbolic machine-like coding language
(iii) A symbol manipulation list processing language

The algebraic language is ALGOL with some trivial extensions
The symbolic machine-like language is like TASS
The symbol-manipulation language is like Threaded Lists

Nevertheless the three are described as one language using the same syntax description; and there is only one processor.

The name given to the language is 20AL, that of the processor is 20AP.

The report that follows is divided into sections that:

Define the Language Syntax of 20AL (Part I)
Define the Process Syntax of 20AP (Part II)
Define the Use of 20AL (Part III)

Naturally, a particular computer must be used as a model for (ii) above; and some of the Process Syntax will be similarly machine dependent.

The machine used as a model—where so necessary—is the Bendix G 20. Effort has been made to keep such reference to a minimum.

Associated with the processor are modes of operation which, of course, are machine dependent. One such is described in Part III. The organization
of 20^P is intended to permit flexible operating schemes, consequently the design of the processor is described in terms of actions taken at various phases of the translation process.

The phases are directly related to "passes" through the code sequences admitted to and generated by the system. Three passes are involved in the description of the translation process:

1) Reduction from string representation to tree representation.

2) Reduction from tree representation to machine code sequence representation.

3) Reduction of machine code sequence representation to output data representation.

With respect to each of the syntactic units of 20^P there is associated a "time" of declaration and a "time" of call. With each of these there is a first, an intermediate, and a terminal such

20^P generates tables of relations between properties of syntactic units in the three representations. Operations on these tables do not constitute a pass in the sense above. Instead they are imbedded in transition phases between the passes.

The following diagram will be useful in describing processing tasks on the various syntactic units:

<table>
<thead>
<tr>
<th>Pass</th>
<th>Declaration</th>
<th>Call</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thus within the descriptions that follow, the notation Di3 will refer to an intermediate declaration processing during pass 3.

* Expressions are handled out of deference to storage efficiency—in a 3 address block form, rather than a pure tree form.
Part I. The Language: 20\(^L\)

The form of the description that follows borrows heavily on the report of ALGOL 60. Indeed, since the ALGOL 60 language is so much an integral part of the 20\(^L\), parts of the report are reproduced almost in toto in the sequel. No further mention will be made within those portions of their source.

A word should be said about representation. The characters of the alphabet are assumed to be distinct and individually recognizable. A unique collection of characters underlined is assumed to be a unique character of the alphabet.

What representation one may choose to use on restricted character input devices is not the concern of this report.

The purpose of the algorithmic language, 20\(^L\) is to describe computational processes. The basic concepts used for the description of calculating rules are

(i) "atomic" machine instructions

(ii) list instructions containing as constituents numbers, symbols, variables, functions, and relations.

(iii) the well-known arithmetic expression containing as constituents numbers, variables, functions and relations.

From such basic units are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain control statements and statement clauses are added which may describe, e.g.,

\(+\) The definition of ALGOL 60 was—in part—the responsibility of the senior project member.
alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these control statements that one statement refer to another, statements may be provided with labels. Sequences of statements may be combined into compound statements by insertion of statement brackets.

Statements are supported by declarations which are not themselves computing instructions, but inform of the existence and of certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. Each declaration is attached to and valid for one compound statement. A compound statement which includes declarations is called a block.

A program is a self-contained compound statement, i.e., a compound statement which is not contained within another compound statement and which makes no use of other compound statements not contained within it.

In the sequel the syntax and semantics of the language will be given. (*) Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.
1.1. Formalism for Syntactic Description

The syntax will be described with the aid of metalinguistic formulae. Their interpretation is best explained by an example

\[ <ab> ::= ( | [ <ab> | <ab> <d> ] \]

Sequences of characters enclosed in the brackets \( < > \) represent metalinguistic variables whose values are sequences of symbols. The marks \( ::= \) and \( | \) (the latter with the meaning of \( \text{or} \)) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable \( <ab> \). It indicates that \( <ab> \) may have the value (or \( | \) or that given some legitimate value of \( <ab> \), another may be formed by following it with the character (or by following it with some value of the variable \( <d> \). If the values of \( <d> \) are the decimal digits, some values of \( <ab> \) are:

\[
(137) \\
(234.5) \\
(46) \\
[86]
\]

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e., the sequences of characters appearing within the brackets \( < > \) as \( ab \) in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place. Definition: \( <\text{empty}> ::= \) (i.e., the null string of symbols).
2. Basic Symbols, Identifiers, Numbers, and Strings.

Basic Concepts.

The reference language is built up from the following basic symbols:

<basic symbol> ::= <letter> | <digit> | <logical value> | <delimiter> | <continuation mark>

2.1. Letters

<letter> ::= a | b | c | d | e | f | g | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e., character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2.2. Digits

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<decimal digit> ::= <octal digit> | 8 | 9

Decimal digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical Values

<logical value> ::= true | false

The logical values have a fixed obvious meaning.
2.3. Delimiters

<delimiter> ::= <operator> | <separator> | <bracket> | <declarator>

<specificator>

<operator> ::= <arithmetic operator> | <relational operator> | <logical operator> | <sequential operator> | <list operator>

<arithmetic operator> ::= + | - | x | / | ::-

<relational operator> ::= < | ≤ | = | ≥ | ≠ |

<logical operator> ::= & | | V | ∧ | ⊕ | $ |

<list operator> ::= * | $ | / | \ [ ]

<instruction operator bound> ::= [ ]

<sequential operator> ::= go to | if | then | else | for | do

<separator> ::= , | . | 10 | ; | := | X | step | until | while | comment

<bracket> ::= ( | [ | |[ ] ] )

<declarator> ::= own | Boolean | logical | integer | octal | real | array | switch | procedure | index | macro | parameter | equivalent | library | constant | list

<specificator> ::= string | label | value

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel. Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

<continuation mark> ::= \ψ

The continuation mark is used in the case of symbolic machine code as a punched-card oriented convention specifying that the instruction punched on the card requires (at least) one more card to complete its description.
For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols: \( \text{<comment> <any sequence not containing ;> } \)

\( \text{begin comment <any sequence not containing ;>} \)

\( \text{end <any sequence not containing end or ; or else>} \)

By equivalence is here meant that any of the three symbols shown in the right-hand column may, in any occurrence outside of strings, be replaced by any sequence of symbols of the structure shown in the same line of the left-hand column without any effect on the action of the program.

2.5. Identifiers

2.4.1. Syntax

\(<\text{identifier}> ::= \text{<letter> | <identifier> | <identifier> | <digit>}\)

\(<\text{special register identifier}>\)

2.4.2. Examples

\(q\)

Soup

V17a

a34kTMNs

MARILYN

2.4.3. Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4. Standard Functions).

However, the use of symbolic machine code is more easily mixed with the algebraic and list language if certain machine registers are
recognized by identifiers fixed by convention. Thus, e.g.,

<special register identifier> ::= ACC | MQ | QA | LR

The above is intended as an example and is clearly machine dependent.

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> ::= <decimal digit>

<unsigned integer> ::= <digit>|<unsigned integer> <digit>

<integer> ::= <unsigned integer> | * <unsigned integer> | - <unsigned integer>

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

<exponent part> ::= 10 <integer>

<decimal number> ::= <unsigned integer> | <decimal fraction> | <unsigned integer><decimal fraction>

<unsigned number> ::= <decimal number> | <exponent part> | <decimal number><exponent part>

<number> ::= <unsigned number> | * <unsigned number> | - <unsigned number>

<unsigned octal integer> ::= <octal digit> | <unsigned octal integer> <octal digit>

<octal integer> ::= (8) <unsigned octal integer> | +(8)<unsigned octal integer>

<unsigned octal integer>
2.5.2. Examples

<table>
<thead>
<tr>
<th></th>
<th>-200.084</th>
<th>-_083_10^-02</th>
<th>(8) 1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>177</td>
<td>+07.4310^5</td>
<td>-107</td>
<td>(8) 7777</td>
</tr>
<tr>
<td>+5384</td>
<td>9.3410^10</td>
<td>10^-4</td>
<td></td>
</tr>
<tr>
<td>+0.7300</td>
<td>2_10^-4</td>
<td>+10^5</td>
<td></td>
</tr>
</tbody>
</table>

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10. Octal numbers are used only with machine assembly code.

2.5.4. Types

Integers are of type integer. All other numbers are of type real (cf. section 5.1. Type Declarations).

2.6. Strings

2.6.1. Syntax

<proper string> ::= <any sequence of basic symbols not containing \"or\" | <empty>

<open string> ::= <proper string> | \'<open string>' |

<string> ::=  \'<open string>'

2.6.2. Examples

"5k", "[[[\"f:\^ =/:\"Tt"

".. This \" is \" a \"string"

2.6.3. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes 'and' are introduced. The symbol \" denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2., Function Designators and 4.7. Procedure Statements).
2.7. Quantities, Kinds and Scopes

The following kinds of quantities are distinguished: simple variables, arrays, lists, labels, switches, macros, and procedures.

The scope of a quantity is the set of statements in which the declaration for the identifier associated with that quantity is valid, or, for labels, the set of statements which may have the statement in which the label occurs as their successor.

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), an ordered set of strings (special case: a single string), 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 list identifier is the ordered set of values of the corresponding list. These values are ordered by the element sequencing rule for lists (3.8...)

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

2.9 Threaded Lists

Threaded lists (Tlists) have been reported on elsewhere. The following points are, however, basic to the material of the report. An information site is a place in the machine that can be occupied by 1) a symbol, 2) a Tlist, 3) an address specifying the site of a symbol.
Initially a Tlist named NAME occupies two sites denoted by: NAME: (,).
One information site, denoted by , ) is available in each empty Tlist of the form (,).

New blank sites may be added by an insert operation. Thus:

\[(, ) \xrightarrow{\text{insert}} (, , ) \xrightarrow{\text{insert}} (, , )\,.

In a blank site, an empty Tlist may be inserted. Thus:

\[(, , ) \xrightarrow{\text{list}} (, (, , ), )\,.

In order to access information in a Tlist each list may be sequenced at any time in one of three ways : word, element, and list. Furthermore, if X is the name of a particular Tlist, X# specifies its site currently under scan. X* specifies the "next" site to that currently under scan, and makes it the current one. n(k,X#) specifies the kth next site to that currently under scan but does not change the significance of X#. h(X#) specifies the site of the innermost "(" of the pair "(" and ")" which enclose X#. Similarly the kth head of X# by u(k,X#) = h(h(h(\ldots h(X#))\ldots))\,.

The operators h and u are themselves independent of the sequencing mode employed.

Word sequencing is a left-right sequencing with exactly one stop at each site. List sequencing is a left-right sequencing with stops only at sites which are on the same level. Element sequencing is a left-right sequencing with stops only at sites where symbols or indirect referents may occur. An example will clarify:

Tlist: \((,(,),(,)),(,))\,.

word sequence: 1 2 3 4 5 6 7 8 9 10 11
list sequence: 1 2 3 4
element sequence: 1 2 3 4 5 6 7
Two lists may be combined in several ways to form new lists.

Copy \((x,y)\) copies the list \(x\) into the list site \(y\). Thus:

\[
(x) \quad \text{and} \quad ((, , , ))
\]

\(X\) and \(y\)
gives for copy \((X,y)\): \((,(,\ldots))\).

Join \((x,x,y)\) forms \(z:\ (x,y)\).

Append \((x,y)\) forms, using the above example, \((,(, , , \ldots))\).

The \(\rightarrow\) operation substitutes symbols and control characters into sites from other sites. It is a special copy working on the micro scale of a site.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, 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.

\(<\text{expression}> ::= \text{<arithmetic expression> <Boolean expression> <logical expression> <designational expression> <address expression>}\> 

3.1. Variables

3.1.1. Syntax

\(<\text{variable identifier}> ::= \text{<identifier>}\>

\(<\text{simple variable}> ::= \text{<variable identifier>}\>

\(<\text{subscript expression}> ::= \text{<arithmetic expression>}\>

\(<\text{subscript list}> ::= \text{<subscript expression> <subscript list>}\>

\(<\text{subscript expression}>\>

\(<\text{array identifier}> ::= \text{<identifier>}\>
3.1.2. Examples

\begin{align*}
\text{epsilon} & : R[5, \ne] \\
\text{detA} & : X[p \ne, \star] \\
\text{a17} & \\
\text{Q[7,2]} & \\
\text{x[sin(n x pi/2), Q[3, n, 4]]} & 
\end{align*}

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 the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2. Array Declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables 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 fictitious 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).

3.1.5. List Indices

3.1.5.1. List variables designate components of lists. Each list expression of the subscript list occupies one subscript position of the subscripted list variable, and is called a list subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The list component referred to by a subscripted variable is specified by the action of the list sequencing mode currently operative over the list named. The value of the subscript is defined only if the sequencing action does not exhaust the list elements. Should exhaustion occur before the list component is encountered, control transfer within the program will occur.

3.2. Function Designators

3.2.1. Syntax

<procedure identifier> ::= <identifier>

<actual parameter> ::= <string> | <expression> | <array identifier> |

<switch identifier> | <procedure identifier>

<letter string> ::= <letter> | <letter string> <letter>

<parameter delimiter> ::= , | ) <letter string> :

<actual parameter list> ::= <actual parameter>

<actual parameter list> <parameter delimiter>

<actual parameter>
3.2.2. Examples

\[
\begin{align*}
sin(a-b) \\
J(v,s,n) \\
R \\
S(s-5) Temperature: (T) Pressure: (P) \\
Compile('='; )Stack: (Q)
\end{align*}
\]

3.2.3. Semantics

Function designators define sequencing rules, 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

3.2.4.1. Standard Arithmetic Functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

- \( \text{abs}(E) \) for the modulus (absolute value) of the value of the expression \( E \)
- \( \text{sign}(E) \) for the sign of the value of \( E (+1 \text{ for } E>0, 0 \text{ for } E=0, -1 \text{ for } E<0) \)
- \( \text{sqrt}(E) \) for the square root of the value of \( E \)
- \( \text{sin}(E) \) for the sine of the value of \( E \)
- \( \text{cos}(E) \) for the cosine of the value of \( E \)
- \( \text{arctan}(E) \) for the principal value of the arctangent of the value of \( E \)
\[ \ln(E) \] for the natural logarithm of the value of \( E \)

\[ \exp(E) \] for the exponential function of the value of \( E \) \((e^E)\).

These functions are all understood to operate indifferently on arguments both of type \textit{real} and \textit{integer}. They will all yield values of type \textit{real}, except for \text{sign}(E) which will have values of type \textit{integer}. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.4.2. Standard List Procedures

\text{next}(v) \quad \text{for the extraction of the next list component from } v \text{ without advancing the sequence marker}

\text{list}(v) \quad \text{for the insertion of an empty list into the site } v

\text{insert}(k,v) \quad \text{for the insertion of } k \text{ empty sites immediately following } v

\text{def}(w,v) \quad \text{for the dynamic definition of a list } w \text{ as } v

\text{copy}(v,w) \quad \text{for the creation of a list } w \text{ which is, except for linking addresses, identical to the list } v.

\text{seq}(e,v,w) \quad \text{for the sequencing through list } v \text{ in mode } e \text{ with exit to } w \text{ on completion}

These functions operate on lists according to the formation and sequencing rules regarding lists (cf. section 3.8)

3.2.5. Transfer Functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely

\text{entier}(E)

which "transfers" an expression of \textit{real} type to one of \textit{integer} type, and assigns to it the value which is the largest integer not greater than the value of \( E \)

3.3. Arithmetic Expressions
3.3.1. Syntax.

<adding operator> ::= + | -
<multiplying operator> ::= x | / | \;
<brimary> ::= <unsigned number> | <variable>

<function designator> | (<arithmetic expression>)
<bractor> ::= <primary> | <factor> ↑<primary>
<brerm> ::= <factor> | <term> <multiplying operator> <factor>
<brsimple arithmetic expression> ::= <term>

<bradding operator> <term> | <simple arithmetic expression>
<bradding operator> <term>
<brif clause> ::= if <Boolean expression> then
<brarithmetic expression> ::= <simple arithmetic expression>

<brif clause> <simple arithmetic expression> else
<brarithmetic expression>

3.3.2. Examples

Primaries

7.394 \times 10^{-8}

sum

w[i+2,8]

\cos(r^{x+3})

(a-3/y+v) \uparrow 8

Z[\nu, \phi]

Factors:

\omega

sum \uparrow \cos(y+z \times 3)

7.394 \times 10^{-8} \uparrow w[i+2,8] \uparrow (a-3/y+v) \uparrow 8
Terms:

\[ U \]

\[
\omega x \text{sum} \cos(y+z x^3)/7.394_{10}^{-8} \text{w}[i+2,8] \uparrow
\]

\[(a-3/y^xv-8)\]

Simple arithmetic expression:

\[ U-Y+\omega x \text{sum} \cos(y+z x 3)/7.394_{10}^{-8} \text{w}[i+2,8] \uparrow
\]

\[(a-3/y^xv \uparrow 8)\]

Arithmetic expressions:

\[ w x u Q(S+Cu) \uparrow 2 \]

\[
\text{if } q>0 \text{ then } S+3 x Q/A \text{ else } 2 x S+3 x q
\]

\[
\text{if } a<0 \text{ then } U+V \text{ else if } a x b>17 \text{ then } U/V \text{ else if }
\]

\[
\text{if } y \neq \text{ then } V/U \text{ else } 0
\]

\[ a x \sin(\omega x t) \]

\[
0.57_{10} 12 x a[N x (N-1)/2, 0]
\]

\[
(A x \arctan(y) + Z) \uparrow(7+Q)
\]

\[
\text{if } q \text{ then } n-1 \text{ else } n
\]

\[
\text{if } a<0 \text{ then } A/B \text{ else if } b=0 \text{ then } B/A \text{ else}(z \times R[I, \ast^\ast])
\]

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In 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 arising from the computing rules defining the procedure (cf. section 5.4. Procedure Declarations) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses
the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. 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 (the largest arithmetic expression found in this position is understood). The construction:

```
else <simple arithmetic expression>
```

is equivalent to the construction:

```
else if true then <simple arithmetic expression>
```

3.3.4. Operators and Types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types real or integer (cf., section 5.1. Type Declarations).

List variables occurring in simple arithmetic expressions must of course, refer to that part of their information content which is of type real or integer. 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 x 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, otherwise real.

3.3.4.2. The operations <term>/<factor> and <term> \div <factor> both denote division, to be understood as a multiplication of the term by the reciprocal
of the factor with due regard to the rules of precedence (cf. section 3.3.5).
Thus for example
\[ a/b \times 7/(p-q) \times v/s \]
means
\[ (((a x (b^{-1})) \times 7) \times ((p-q)^{-1})) \times v) \times (s^{-1}) \]
The operator \(/\) is defined for all four combinations of types \texttt{real} and
\texttt{integer} and will yield results of \texttt{real} type in any case. The operator \(\div\)
is defined only for two operands both of type \texttt{integer} and will yield a
result of type \texttt{integer} defined as follows:
\[ a \div b = \text{sign}(a/b) \times \text{entier}(|a/b|) \]
(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation \(<\texttt{factor}>^{<\texttt{primary}>}\) denotes exponentiation,
where the factor is the base and the primary is the exponent. Thus, for
example,
\[ 2^{n^k} \quad \text{means} \quad (2^n)^k \]
while
\[ 2^{(n^m)} \quad \text{means} \quad 2^{(n^m)} \]
Writing \(i\) for a number of \texttt{integer} type, \(r\) for a number of \texttt{real} type,
and \(a\) for a number of either \texttt{integer} or \texttt{real} type, the result is given
by the following rules:
\[ a^{i} \quad \text{if } i > 0, \ a \times a \times \ldots \times a \ (i \text{ times}), \ of \ the \ same \ type \ as \ a. \]
\[ \text{If } i = 0, \text{ if } a \neq 0, 1, \ of \ the \ same \ type \ as \ a. \]
\[ \text{if } a = 0, \text{ undefined.} \]
\[ \text{If } i < 0, \text{ if } a \neq 0, 1/(a \times a \times \ldots \times a) \ (\text{the denominator has} \]
\[ \text{i factors), of type } \texttt{real}. \]
\[ \text{if } a = 0, \text{ undefined.} \]
If \( a > 0 \), \( \exp(r \times \ln(a)) \), of type \text{real}.

If \( a = 0 \), if \( r > 0 \), 0, 0, of type \text{real}.

If \( r < 0 \), undefined.

If \( a < 0 \), always undefined.

3.3.5. Precedence of operators

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

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

- first: \( \uparrow \)
- second: \( x/\cdot \)
- third: \( + - \)

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. Arithmetics of \text{real} quantities

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

3.4.1. Syntax

<relational operator> ::= \leq \mid = \mid \geq \mid \neq

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

<arithmetic expression> | <logical expression> <relational operator>

<logical expression>

<Boolean primary> ::= <logical value> | <variable>

<function designator> | <relation> | (<Boolean expression>)

<Boolean secondary> ::= <Boolean primary> | ¬ <Boolean primary>

<Boolean factor> ::= <Boolean secondary>

<Boolean term> ::= <Boolean factor> | <Boolean term>

<implication> ::= <Boolean term> | <implication> ∨ <Boolean term>

<simple Boolean> ::= <implication>

<Boolean expression> ::= <simple Boolean> | <if clause> <simple Boolean> else <Boolean expression>

3.4.2. Examples

x = -2
Y > v \vee z < q
a \odot b > - 5 \land z - d > q^2
p \land q \lor x \neq y
s \equiv \neg a \land b \land \neg c \lor d \lor e \lor f
if k < l then s > w else h \leq 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 (cf. section 5.1. Type Declarations and sections 5.4.4. Values of Function Designators).

3.4.5. The Operators

Relations 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), and ≡ (equivalent), is given by the following function table.

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>b2</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>¬ b1</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>b1∧b2</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>b1∨b2</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>b1⇒b2</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>b1≡b2</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
3.5. Logical Expressions

3.5.1. Syntax

\[
<\text{simple logical operator}> ::= \lor \mid \land \mid \neg \mid $ \\
<\text{logical primary}> ::= <\text{octal integer}> \mid <\text{integer}> \mid <\text{variable}> \\
<\text{function designator}> \mid <\text{Boolean expression}> \mid ( <\text{logical expression}> ) \\
<\text{shift measure}> ::= <\text{arithmetic expression}> \\
<\text{logical secondary}> ::= <\text{logical primary}> \mid \neg <\text{logical primary}> \\
<\text{logical factor}> ::= <\text{logical secondary}> \mid <\text{logical factor}> \mid $ <\text{shift measure}> \\
<\text{logical term}> ::= <\text{logical factor}> \mid <\text{logical term}> \land <\text{logical factor}> \\
<\text{major}> ::= <\text{logical term}> \mid <\text{major}> \lor <\text{logical term}> \\
<\text{logical expression}> ::= <\text{major}> \mid <\text{if clause}> <\text{major}> \text{ else } <\text{logical expression}> \\
\]

3.5.2. Examples

\[
X \lor Y \\
X \ (I \times 8 - 4) \\
(X \land Y) \ (K) \\
(\text{if } X \ (2 = Y \text{ then } Z \text{ else } K) \ (3) \\
\]

3.5.3. Semantics

A logical expression is a rule for computing the value of a fixed length string of binary digits. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.5.4. Types

Variables and function designators entered as logical primaries must be declared \textit{logical} (cf. section 5.1. Type Declarations and sections 5.4.4. Values of Function Designators).

3.5.5. The Operators

The operator $ refers to a (non-cyclic) shift of the binary pattern. Thus in $l1 \ l2$, the logical variable $l1$ is shifted $|l2| \ (\text{mod } 5)$ places.
left (right) if \( \ell^2 \) has a positive (negative) value. \( \sigma \) is a function of
the register size of a computer and will, of course, vary among computers.

All other operators are as described in sections 3.4.5 and 3.3.4.

3.5.6. Precedence of operators

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

3.5.6.1. According to the syntax given in section 3.4.1, 3.5.1, the
following rules of precedence hold:

- first: arithmetic expressions according to section 3.3.5.
- second: \( \langle \leq \Rightarrow \rangle \neq \)
- third: \( \neg \)
- fourth: \( \$ \)
- fifth: \( \wedge \)
- sixth: \( \vee \)
- seventh: \( \subset \)
- eighth: \( \equiv \)

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

3.6. Designational Expressions

3.6.1. Syntax

\[
\langle \text{label} \rangle ::= \langle \text{identifier} \rangle | \langle \text{unsigned integer} \rangle \\
\langle \text{switch identifier} \rangle ::= \langle \text{identifier} \rangle \\
\langle \text{switch designator} \rangle ::= \langle \text{switch identifier} \rangle [\langle \text{subscript expression} \rangle] \\
\langle \text{simple designational expression} \rangle ::= \langle \text{label} \rangle | \langle \text{switch designator} \rangle | \\
(\langle \text{designational expression} \rangle)
\]

continued --
\[ \text{<designational expression>} := \text{<simple designational expression> |} \\
\text{<if clause>} \text{<simple designational expression>} \text{ else} \text{<designational expression>} \]

3.6.2. Examples

17

\( p^9 \)

\text{Choose}[n-1]

\text{Town} [\text{if } y < 0 \text{ then } N \text{ else } N + 1]

\text{if } A < n \text{ then } 17 \text{ else } q [\text{if } w < 0 \text{ then } 2 \text{ else } n]

3.6.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clause 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 (cf. 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. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.6.4. The subscript expression

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

3.7.1 Syntax

\(<\text{address expression}> ::= | <\text{indirect}> | <\text{direct}> \)

\(<\text{indirect}> ::= <\text{indirect address expression}> | ( <\text{indirect address expression}> <\text{sign}> <\text{indirect address expression}> ) | ( <\text{indirect}> ) \)

\(<\text{direct}> ::= <\text{unsigned address integer}> | <\text{direct}> \)

\(<\text{indirect address expression}> ::= <\text{simple address expression}> | \)

\(<\text{simple address expression}> ::= <\text{elementary address}> | <\text{elementary address}> <\text{sign}> <\text{elementary address}> | <\text{elementary address}> <\text{sign}> <\text{unsigned address integer}> \)

\(<\text{elementary address}> ::= <\text{identifier}> \)

\(<\text{unsigned address integer}> ::= <\text{an unsigned integer} \leq 32,767> \)

\(<\text{sign}> ::= + | - \)

3.7.2 Examples

\[
\begin{align*}
\text{[\text{i.} 26] } & \rightarrow \text{[\text{i.} 26] } \\
\text{[\text{i.} + (8)23] } & \rightarrow \text{[\text{i.} + (8)23] } \\
\text{[\text{c - 46} \times \text{DT4}] } & \rightarrow \text{[\text{c - 46} \times \text{DT4}] } \\
\text{[\text{1 + E4}] } & \rightarrow \text{[\text{1 + E4}] } \\
\text{[\text{2 + ( E9 )}] } & \rightarrow \text{[\text{2 + ( E9 )}] } \\
\text{[\text{( E6 + (Z I 11))}] } & \rightarrow \text{[\text{( E6 + (Z I 11))}] } \\
\text{[\text{(( ALPHA ) + ( MU ) + (TAU))}] } & \rightarrow \text{[\text{(( ALPHA ) + ( MU ) + (TAU))}] }
\end{align*}
\]

3.7.3 Semantics

Address expressions are used to specify the value of operands of the symbolic machine-like code. Their syntax is defined to make maximal use of the operand generating facilities of a particular computer. In particular, in address expressions, the characters "(" and ")" bracketing an identifier
refer to the contents of the storage location which will correspond to that identifier. Nested parentheses provide levels of indirect addressing.

indicates that identifiers not enclosed in parentheses have individually implied parentheses about them. indicates that the value of the address expression is/operand. The value of the address expression will in general be defined modulo (f) where f will depend on the addressable storage capacity of the computer.

3.7.4. Precedence of operators

The sequence of operations within an address expression is generally from left to right. Insofar as sequencing of operations is concerned, the use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.8 List Subscript expressions

3.8.1. Syntax

<address chain> ::= \(||<address chain>| <address chain>| \)

<function designator> | <empty>

<foral list subscript expression> ::= p | <address chain> | p <address chain>

<aflist sequence chain head> ::= * | <function designator>

<aftlist sequence chain> ::= <aftlist sequence chain head> <aftlist sequence chain head> <aftlist sequence chain>

<aftlist subscript expression> ::= s | <aftlist sequence chain>

3.8.2. Examples: (as subscripts)

I [ p H(V[ w ], t) \( w, *; \) ]
3.8.3. Semantics

The list subscript expressions are used to select list components. \( p \) isolates the list component prefix, \( l(4) \) the left (right) portion of the list component. \( \neq \) refers to the current position of the sequence counter on the list in question; * refers to the next as defined by the sequencing rule invoked on the list. \( \# \) alters the sequence counter before extraction of the list component.

3.8.4. Precedence of operators

\( \wedge \) and \( \vee \) are associative to the right, i.e.,

\[
\wedge \vee \wedge \vee x \text{ means } \wedge \text{ of } \vee \text{ of } \wedge \text{ of } x.
\]

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by control statements, i.e., go to statements, which define their successor explicitly; shortened by conditional statements, which may cause certain statements to be skipped; and expanded by for 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 since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.
4.1. Compound Statements and Blocks

4.1.1. Syntax

- `<unlabelled basic statement>` ::=`<assignment statement>`
- `<go to statement>` | `<dummy statement>` | `<procedure statement>` | `<code line>`
- `<basic statement>` ::= `<unlabelled basic statement>` | `<label>`<br>`<basic statement>`
- `<unconditional statement>` ::= `<basic statement>` | `<for statement>`
- `<compound statement>` | `<block>`
- `<statement>` ::= `<unconditional statement>`
- `<conditional statement>`
- `<compound tail>` ::= `<statement> and` | `<statement>` ;
- `<compound tail>`
- `<block head>` ::= `<begin > <declaration>` | `<block head>` ;
- `<declaration>`
- `<unlabelled compound>` ::= `<begin > <compound tail>
- `<unlabelled block>` ::= `<block head>` ; `<compound tail`
- `<compound statement>` ::= `<unlabelled compound>`
- `<label>`<br>`<compound statement>`
- `<block>` ::= `<unlabelled block>` | `<label>`<br>`<block>`

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 and

Block:

L: L; ... begin D ; D ; ... D ; S ; S ; ... S ; S and

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:

\[ a := p \cdot q \]

\textit{go to Naples}

\texttt{START: CONTINUE: W := 7.993}

\texttt{r : b : CIA X + (B) \downarrow}

Compound statement:

\texttt{begin x := 0 ; for y := 1 step 1 until n do x := x + A[y] ;}

\texttt{if x > q then go to STOP else if x > n - 2 then go to S ;}

\texttt{Aw: St: W := x + bob end}

Block:

\texttt{Q: begin integer i, k ; real w ;}

\texttt{for i := 1 step 1 until m do}

\texttt{for k := i + 1 step 1 until m do}

\texttt{begin w := A[i, k] ;}

\texttt{A[i, k] := A[k, i] ;}

\texttt{A[k, i] := w end for i and k end block Q}

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 (cf. 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) occurring 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 in the level immediately outside it. The exception to this rule is presented by labels, which are local to the block in which they occur.

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 one statement.

4.2. Assignment Statements

4.2.1. Syntax

\[
\text{<left part>} ::= \text{<variable>} := \\
\text{<left part list>} ::= \text{<left part> | <left part list> <left part>} \\
\text{<assignment statement>} ::= \text{<left part list> <arithmetic expression> | <left part list> <Boolean expression>} \\
\]

4.2.2. Examples

\[
s := p[0] := n := n + 1 + s \\
n := n + 1 \\
A := B/C - v - q x S \\
y, v, k+2] := 3 - \arctan(s x \text{meta}) \\
V := Q > Y \land Z
\]

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables. The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence 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 any subscript expressions having values as evaluated in step 4.2.3.1.
4.2.4. Types

All variables of a left part list must be of the same declared type.

If the variables are Boolean, the expression must likewise be Boolean. If the variables are of type `real` or `integer`, the expression must be arithmetic.

If the type of the arithmetic expression differs from that of the variables, appropriate transfer functions are understood to be automatically invoked.

For transfer from `real` to `integer` type, the transfer function is understood to yield a result equivalent to

\[
\text{entier}(E \times 0.5)
\]

where \(E\) is the value of the expression.

4.3. GO TO Statements

4.3.1. Syntax

\[
<\text{go to statement}> := \text{go to } <\text{designational expression}>
\]

4.3.2. Examples

\[
\text{go to 8}
\]

\[
\text{go to exit [n+1]}
\]

\[
\text{go to Town [if y < 0 then N else N+1]}
\]

\[
\text{go to if } a < b \text{ then 17 else q [if w < 0 then 2 else n]}
\]

4.3.3. Semantic

A `go to` statement interrupts the normal sequence of operations, defined by the write-up of statements, 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 go to statement can lead from outside into a block.

4.3.5. GO TO an undefined switch designator

A go to statement is equivalent to a dummy statement 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 ... John; end

4.4.3. Semantic

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

4.5. Conditional Statements

4.5.1. Syntax

<if clause> ::= if <Boolean expression> then
<unconditional statement> ::= <basic statement> | <for statement>
<comp unti statement> | <block>
<if statement> ::= <if clause> unconditional statement>
<labels ::= <if statement>
<condition statement> ::= <if statement> | <if statement> else
<statement>
4.5.2. Examples

```plaintext
if x > 0 then n := n-1
if y > a then y := m else go to R
if x > y ∨ y < z then AA: begin if q > y then r := v/s
else y := 2 * a and
else if y > x then r := x else if v > s-1
then go to S
```

4.5.3. Semantics

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

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

```plaintext
if Bl then S1 else if B2 then S2 else S3 ; S4
```

and

```plaintext
if Bl then S1 else if B2 then S2 else S3 ; S4
```

Here Bl to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows:

The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value true is found. Then the unconditional statement following this Boolean is executed.

Unless this statement defines its successor explicitly the next statement
to be executed will be \( S_4 \), i.e., the statement following the complete conditional statement. Thus the effect of the delimiter \textit{else} may be described by saying that it defines the successor of the statement if follows to be the statement following the complete conditional statement.

The construction

\texttt{else <unconditional statement>}

is equivalent to

\texttt{else if true then <unconditional statement>}

If none of the Boolean expressions of the \textit{if} clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:

\[
\begin{array}{c}
\text{if } B_1 \text{ then } S_1 \quad \text{else if } B_2 \text{ then } S_2 \quad \text{else } S_3 \quad ; \\
\text{if } B_1 \text{ false } \quad \text{; } \quad \text{if } B_2 \text{ false }
\end{array}
\]

4.5.4. **GO TO** into a conditional statement

The effect of a \textit{GO TO} statement leading into a conditional statement follows directly from the above explanation of the effect of \textit{else}.

4.6. **For statements**

4.6.1. Syntax

\texttt{<for list element> := <arithmetic expression>} |
\texttt{<arithmetic expression> step <arithmetic expression> until}
\texttt{<arithmetic expression> | <arithmetic expression> while}
\texttt{<Boolean expression>}

\texttt{<for list> := <for list element> | <for list>, <for list element>
\[ \text{<for clause>} ::= \text{for <variable>} ::= \text{<for list> do}
\]
\[ \text{<for statement>} ::= \text{<for clause> <statement> |}
\]
\[ \text{<label>} ::= \text{<for statement>}
\]

4.6.2. Examples

\[
\text{for } q := 1 \text{ step } s \text{ until } n \text{ do } A[q] := B[q]
\]
\[
\text{for } k := 1, v1 \times 2 \text{ while } v1 < N \text{ do }
\]
\[
\text{for } j := 1+G, L, 1 \text{ step } 1 \text{ until } N, C=D \text{ do }
\]
\[
A[k,j] := B[k,j]
\]

4.6.3. Semantics

A \text{ for clause} causes the statement \text{S} which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualised by means of the following picture:

\[
\begin{tabular}{c}
\text{Initialize} \downarrow \text{test} \downarrow \text{statement } S \downarrow \text{advance} \downarrow \text{successor} \\
\text{for list exhausted}
\end{tabular}
\]

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

4.6.4. The \text{for list} elements

The \text{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. This element gives rise to one value,
namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement $S$.

4.6.4.2. Step-until-element. A for element of the form A step B until C,
are arithmetic expressions, gives rise to an execution which may be
described most concisely in terms of additional ACOOL statements as
follows:

$$ V := A ; $$

$$ L1 : \text{if } (V - C) \cdot \text{sign}(E) > 0 \text{ then go to Element exhausted}; $$

Statement: $S$ ;

$$ V := V + E ; $$

$$ \text{go to } L1 ; $$

where $V$ is the controlled variable of the for clause and Element exhausted
points to the evaluation according to the next element in the for list,
or if the step-until-element is the last of the list, to the next statement
in the program.

4.6.4.3. While--element. The execution governed by a for list element of
the form $E$ while $F$, where $E$ is an arithmetic and $F$ a Boolean expression,
is most concisely described in terms of additional ACOOL statements as
follows:
L3 : V := E ;
    if ¬ F then go to Element exhausted ;
    Statement S ;
    go to 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 out of the statement S (supposed to be compound) through
a go to statement, the value of the controlled variable will be the same
as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand,
the value of the controlled variable is undefined after the exit.

4.6.6. go to leading into a for statement

The effect of a go to statement, outside a for statement, is undefined.

4.7. Procedure Statements

4.7.1. Syntax

<actual parameter> ::= <string> | <expression> | <array identifier> |
                    <list identifier> | <switch identifier> | <procedure identifier>
<letter string> ::= <letter> <letter string> <letter>
<parameter delimiter> ::= , | <letter string>
<actual parameter list> ::= <actual parameter> |
                         <actual parameter list> <parameter delimiter>
<actual parameter>
<actual parameter part> ::= <empty> |
                         (<actual parameter list>)
<procedure statement> ::= <procedure identifier>
                         <actual parameter part>
4.7.2. Examples

Spur (A) Orders (7) Result to: (V)
Transpose (W,v+1)
Absmax (A,M,3x,1,e)
Innerproduct(A[t,F,u],B[F],10,P,Y)
Among (V,W,yf )

These examples correspond to examples given in section 4.4.2.

4.7.3. Semantics

A procedure statement 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 WFL, the effect of this execution will be equivalent to the effect of performing the following operations on the program:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading 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. These formal parameters will subsequently be treated as local to the procedure body.

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 conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.
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.

4.7.4. Actual-formal correspondence

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

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct 20^L statement.

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

4.7.5.1. String cannot occur as actual parameters in procedure statements calling procedure declarations having 20^L 60 statements as their bodies (c.f. section 4.7.6).

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which in a variable (special case of expression).
4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

4.8. Code Line

4.8.1. Syntax

\[
\text{<code line>} ::= \text{<line of code>} \downarrow
\]

\[
\text{<line of code>} ::= \text{<instruction designator>} | \text{<macro designator>} | \text{<empty>}
\]

\[
\text{<instruction designator>} ::= \text{<operation designator>} \text{<address expression>}
\]

\[
\text{<operation designator>} ::= \text{<any of the operation mnemonics of the computer machine code>}
\]

\[
\text{<macro designator>} ::= \text{<macro identifier>} \text{<actual parameter part>}
\]

4.8.2. Examples

\[
\text{CLA} \Rightarrow X + 4 \downarrow
\]

\[
\text{STF} \mid (X + (B)) \downarrow
\]

\[
\text{GRP} \mid (Z, (A 9) - (h 13), 7) \downarrow
\]

4.8.3. Semantics

The code line is the unit statement in machine-like symbolic code. In the case of macro designators, the parenthesis conventions of the actual parameters must match the requirements of the formal parameters in the corresponding macro declaration.

4.9. Macro Statements
4.9.9. Syntax

<actual elementary macro parameter> ::= <string not containing parentheses or ,>
<actual macro parameter> ::= <a.e.m.p.> | (<a.m.p.l.>)
<actual macro parameter list> ::= <a.m.p.> | <a.m.p.l.>
<actual macro parameter part> ::= <empty> | (<a.m.p.l.>)
<macro statement> ::= <macro identifier> | <actual macro parameter part>

4.9.2. Examples

Innerproduct ( Act, P, u], B[z], 10, P, Y )
Among ( (R&T) $ 1, W, C )

4.9.3. Semantics

A macro designator specifies that the following sequence of events transpire:

(1) In a copy of the macro declaration corresponding to the
macro designator, the set of characters specifying an actual parameter
are substituted for their corresponding formal parameters in all places
of the latter's occurrence in the macro declaration. Then

(ii) The altered (copy of the) declaration replaces the macro
statement which called it and then

(iii) Processing continues at the code position previously
occupied by the macro statement.
5. **Declarations**

Declarations serve to define certain properties of the identifiers of the program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. 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 significance again.

A declaration may be marked with the additional declarator `g`.

This has the following effect: upon a reentry into the block, the values of `g` quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as `g` are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.
Syntax:

<declaration> ::= <type declaration> | <array declaration> | <switch declaration> | <procedure declaration> | <macro declaration> | 
<parameter declaration> | <equivalence declaration> | <library declaration> | <constant declaration>

5.1. Type Declarations

5.1.1. Syntax

<type list> ::= <simple variable> | 
<simple variable>, <type list>
<type> ::= real | integer | boolean | index | list | 
<local or own type> ::= <type> | own <type>
<type declaration> ::= <local or own type> <type list>

5.1.2. Examples

integer p, q, r

own Boolean Acryl,n

logical Student, 27

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero, and be represented in either decimal or octal form. Boolean declared variables may only assume the values true and false. Logical declared variables are binary strings. List declared variables are empty lists containing one information site. Index declared variables are index registers.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of own, see the fourth paragraph of section 5 above.
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 list>, <bound pair>

<array segment> ::= <array identifier> [ <bound pair list> ]

<array identifier>, <array segment>

<array list> ::= <array segment> | <array list>, <array segment>

<array declaration> ::= array <array list> | <local or own type>

ARRAY <array list>

5.2.2. Examples

array a, b, c[7:2, 2:1], s[-2:10]

case integer array A[if a<0 then 2 else 1:20]

for i array q[-7:1]

5.2.3. Semantics

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

5.2.3.1. Subscript bounds. The subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound 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 type **real** is understood.

5.2.4. **Lower upper bound expressions**

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

5.2.4.2. The expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently in the outermost block of a program only **array** declarations with constant bounds may be declared.

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.

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

5.2.5. **The identity of subscripted variables**

The identity of a subscripted variable is not related to the subscript bounds given in the **array declaration**. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. **Switch Declarations**

5.3.1. **Syntax**

<switch list> ::= <designational expression> |

<switch list>, <designational expression>

<switch declaration> ::= switch <switch identifier> ::= <switch list>
5.3.2 Examples

```c
switch S := s1, s2, Q[n], if v > then s3 else S4
switch Q := p1, w
```

5.3.3 Semantics

A `switch` declaration defines the values corresponding to a switch identifier. 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 (cf. section 3.6. 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.

Any reference to the value of a switch designator from outside the scope of any quantity entering into the designational expression for this particular value is undefined.

5.4 Procedure Declarations
5.4.1. Syntax

<formal parameter> ::= <identifier>

<formal parameter list> ::= <formal parameter> | <formal parameter list> <parameter delimiter>

<formal parameter>

<formal parameter part> ::= <empty> | (<formal parameter list>)

<identifier list> ::= <identifier> | <identifier list>, <identifier>

<value part> ::= value<identifier list> ; | <empty>

<specifier> ::= string | <type> | array | <type> array | label | switch

procedure <type> procedure | list

<specification part> ::= <empty> | <specifier> <identifier list> ;

<procedure heading> ::= <procedure identifier>

<formal parameter part> ; <specification part> <value part>

<procedure body> ::= <statement> | <code>

<procedure declaration> ::= procedure <procedure heading> <procedure body> |

<type> procedure <procedure heading> <procedure body>

5.4.2. Examples (see also the examples at the end of the report).

procedure: Spur(a)Orders(n)Results(s) ; value n ;
array a ; integer n ; real s ;
beg integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s + a[k,k]
end
Examples continued:

```plaintext
procedure Transpose(a)Order: n; value n;
array a; integer n;
begin real w; integer i, k;
for i := 1 step 1 until n do
  for k := 1 to k until n do
    begin w := a[i,k];
      a[i,k] := a[k,i];
      a[k,i] := w
    end
end Transpose

integer procedure Step(u); real u;
Step := if 0 < u < 1 then 1 else 0

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

```plaintext
procedure Innerproduct(a,b)Order(k,p)Result(y);
  value k;
  integer k,p; real y,a,b; begin
  real s; s=0;
  for p=1 step 1 until k do s:=s+a x 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 **procedure** declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (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 non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the **procedure** declaration appears.

5.4.4. Values of function designators

For a **procedure** declaration to define the value of a function designator there must, within the procedure body, occur an assignment of
a value to the procedure identifier, and in addition the type of this
value must be declared through the appearance of a type declarator as
the very first symbol of the procedure declaration.

Any other occurrence of the procedure identifier within the procedure
body denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the
kinds and types of the formal parameters by means of an obvious notation,
may be included. In this part no formal parameter may occur more than
once and formal parameters called by name (cf. section 4.7.3.2) may be
omitted altogether.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-
\LaTeX language. Since it is intended that the use of this feature should
be entirely a question of hardware representation, no further rules concern-
ing this code language can be given within the reference language.

5.5. Macro Declarations

5.5.1. Syntax

\begin{verbatim}
<macro heading> ::= <macro identifier> <formal parameter part> ;
               <specification part>

<macro body> ::= <statement>

<macro declaration> ::= macro <macro heading> <macro body>
\end{verbatim}
5.5.2. Examples

```plaintext
macro Innerproduct (a,b) Orders(k,p)Result(y) ;
integer k,p ; real S ;
S: = 0
for. p: = |step| until k do S: = S + a * b ;
y: = S and Innerproduct
macro Among (x,y) Predicates(B) Error(L)
list y ;
Boolean B ;
logical x ;
comment B is true if and only if the logical variable x is
an (indirect) element of the list y ;
begi. seqe(y,L) ; B: = false ;
S: If y[p,\$] \neq 0 \land y[\$] = x Then begin B: = true ;
else go to S
end Among.
```

5.5.3. Semantics

A macro declaration serves to define a macro associated with a
macro identifier. Macros only exist in the processing interval from their
point of definition in the lexicographic sequencing of code to the end of
the block in which they are defined — and, in time, only during Pass 1.

The principal constituent of a macro declaration is a statement.
Associated with the body is a heading, much as with procedures, except the
concept of name, value have no significance with macros. Whenever a macro
is called, the formal parameters in the macro body will be replaced by the
actual parameters corresponding leading to a new section of code
which will then be immediately subject to processing. Replacement is understood to occur simultaneously on all parameters -- all their occurrences in the macro body.

5.5.4. The replacement process

Any string of characters satisfying the syntactic rules of actual macro parameters, (see section 4.9) may replace an identifier.

5.6. Equivalent declaration

5.6.1. Syntax

<inner element> ::= <simple variable> |
<outer element> ::= <simple variable> | <subscripted variable>
<simple pair> ::= (<inner element>, <outer element>) |
<pair list> ::= <simple pair> | <pair list>, <simple pair>
<equivalent declaration> ::= equivalent (<pair list>)

5.6.2. Examples

equivalent (A,B), (TAU, PHI [1,3:1])

5.6.3. Semantics

An equivalent declaration declares an identifier, (the inner element) within the block containing the equivalence declaration to be—in every respect—identical with an identifier (the outer element) declared in an outer block.
5.7. Library declaration

5.7.1. Syntax

\[
\text{<identifier list> ::= <identifier> | <identifier list>, <identifier>}
\]

\[
\text{<library declaration head> ::= <local or own type> | <empty>}
\]

\[
\text{<library declaration title> ::= library | library procedure}
\]

\[
\text{<library declaration> ::= <library declaration head> <library declaration title> <identifier list>}
\]

5.7.2. Examples

```
library RANDOM, NTHROOT

integer library SORT

library procedure X, MTXINVSE
```

5.7.3. Semantics

The library declaration serves to call a machine coded, non $20^L$, procedure from the library. Connection to the procedure is made by a standard procedure statement. All storage requirements, except actual parameters, are provided within the library.

The library procedure declaration calls a $20^L$ procedure declaration from the library to be substituted in the $20^L$ code for the occurrence of the library procedure declaration. Substitution of these procedure declarations is made in the left to right order of their names in the call.
Part II Flow Charts for 20\(^P\).

1. Philosophy

Flow charting an operation of a processor such as 20\(^P\) must inevitably be complex and—at times—somewhat machine dependent. Every effort has been made to minimize references which are of the latter kind. Nevertheless, where they are required, specific machine instructions will be used and their explanation given at that time. The flow charts will be given at several levels of description and the detail at any given level will be a function of the processes being described. Some descriptions will be in text, others in a semi-formal notation.

Flow charts are extremely difficult to read under the best of circumstances and the use of formal notation exclusively makes them elegant but impossible to comprehend. However, occasionally it serves the admirable purpose of a shorthand notation and it will be used in such places and defined at the point of use.

As has been mentioned the processor operated in three phases: P1, P2, P3, loosely described as passes over the code.

P1 is an assembly phase and a translation phase

P2 is a compiling and loading phase.

P3 is a running or operating phase.

Intervening are two transition phases T1, T2, which work on tables prepared during passes P1, P2 ; and P2 P3, respectively. During these phases for many of the 20\(^L\) elements there is a point of declaration (D) and (often several) subsequent points of call or use (C). The notation P1,D will refer to some action taken at a declaration during pass 1.
2. Pass 3 Disposition

The processing becomes more clear when a storage map of a $20^L$ program is specified. Schematically it is:

<table>
<thead>
<tr>
<th>Fixed Data Storage</th>
<th>Flow Chart designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(own arrays)</td>
<td>FO</td>
</tr>
<tr>
<td>fixed arrays</td>
<td>FA</td>
</tr>
<tr>
<td>constants</td>
<td>FC</td>
</tr>
<tr>
<td>scalar variables</td>
<td>FS</td>
</tr>
<tr>
<td>list table</td>
<td>FL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library and Administration</th>
<th>L</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>List Storage Pool</th>
<th>LSP</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>P</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Dynamic arrays and Scalars</th>
<th>DA</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Parameter stack</th>
<th>PS</th>
</tr>
</thead>
</table>

Distributed through this storage certain tables are present in all or most programs. They are:

1. The list table LT
2. The active block or procedure table ABT
3. The parameter stack PS
4. The exit stack ES

During PL_D as a block is encountered (begin declaration ...) it is assigned a name (Bt), a level (Bd), and a tag (Bt). The names are positive integers satisfying: If the block begin of a block $a$ occurs (lexicographically) before the block begin of a block $b$ then $a < b$. The levels
specify the depth of parenthesis nesting with begin an opening parenthesis
and end a closing one. Each level Bt will correspond to an assigned
Index register. The block tag (Bt) specifies whether the block is internal
to a procedure (Bt = 1) or not (Bt = 0). The term 1-block or O-block will be
used to distinguish blocks by this property. 1-blocks can, of course, be
part of a recursive process and its declared variables must be treated
dynamically.

In the case of 1-blocks, Pl,D must generate code operative during
P3,D which supplies information to the administration routine BA. The
information supplied must carry to BA;

(i) The block name
(ii) The block level

This information is added to ABT in an augmented table line:

<table>
<thead>
<tr>
<th>Block name</th>
<th>Block level</th>
<th>Base address</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>s</td>
<td>Ar</td>
</tr>
</tbody>
</table>

Ar had been generated by ABT and then index register S is loaded with Ar.

In either case, for 1-blocks and O-blocks, a code line is entered
into the Code Block Table (CBT) active during Pl,D:

<table>
<thead>
<tr>
<th>line number</th>
<th>Bt</th>
<th>Block name</th>
<th>Block level</th>
<th>Base address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>g</td>
<td>r</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>
3. Code generation and Table Generation

During P1 and P2 various code and table generation actions take place. If a table has the name $R$ , then $R$ \* indicates the line at which a marker is poised. $R$ * indicates that the marker has been moved forward one line. If $R$ has a field structure, substitution into and extraction from fields is indicated by a variety of obvious notations. Thus, e.g.,

$$\text{CBT} \* \langle\ldots, Bt, Br, Ba, \ldots\rangle$$

specifies that the contents of $Bt$, $Br$, and $Ba$ are put into the 3, 5, and 6th field of the next line of CBT and \langle blank\rangle into the 7th field.

During P1, code generation occurs. This code is either 20^L code (say, as a result of macro calls) or a form of 3 address code, written, e.g.,

$$\gamma_3 \cdot \theta \gamma_2 \cdot \gamma_1$$

meaning $\gamma_3 \leftarrow \gamma_2 \theta \gamma_1$ in conventional notation. In a given line any of $\gamma_1, \gamma_2, \gamma_3$ may be missing. A collection of 3-address code will be recognized as being collected as a unit by the notation

$$(, \xi_1, \xi_2, \ldots, \xi_k)$$

where each $\xi_i$ is such a 3-address code line.
The operators $\Theta$ will be those most immediately derived from the semantics of $20^L$.

The operators $\Theta$ will be those most immediately derived from the semantics of $20^L$ operators. A table containing the notation and meaning of all such follows:

3-address operator table

<table>
<thead>
<tr>
<th>arithmetic</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>\div</td>
<td>integer division</td>
</tr>
<tr>
<td>↑</td>
<td>exponentiation</td>
</tr>
<tr>
<td>\Pi</td>
<td>store in parameter stack</td>
</tr>
<tr>
<td>MT</td>
<td>mark transfer</td>
</tr>
<tr>
<td>σ</td>
<td>store</td>
</tr>
<tr>
<td>Fh</td>
<td>procedure begin</td>
</tr>
<tr>
<td>Pe</td>
<td>procedure end</td>
</tr>
<tr>
<td>Bh</td>
<td>block begin</td>
</tr>
<tr>
<td>Be</td>
<td>block end</td>
</tr>
<tr>
<td>L</td>
<td>label</td>
</tr>
</tbody>
</table>

In $P2$ the three address code is converted into machine code. Here, detailed knowledge of a machine is necessary and only the general method of such translating can be discussed without becoming overinvolved in specific computer details.

In $P1$, where the major assembly and translation functions are
accomplished there are two major code sequences:

(i) \( 20^L \) (input) code

(ii) List organized three-address code.

The organization of these two code sequences is quite different. (i) is organized as a linear chain while (ii) is organized as a tree. The fundamental distinction is the mode of their sequencing. Each has a marker denoted by a sub-line \( \uparrow \); and this marker moves through the sequences in different ways. To be specific:

In the case of

(i) The marker may move relative to its current position designated as \( I \neq \) (Input-current)—any number of character positions (spaces excluded) backwards or forwards.

Whereas

(ii) The marker may only move back by moving forward to a list sentinel and up several lists higher in the tree structure, through the application of the sequence procedure "up". On the other hand the code position \( \theta \neq \) may itself have substituted into it an entire list through which the marker position may pass or by-pass as occasion demands.

In the case of \( P1 \) the scanning of the input sequence (IS) is the clocking mechanism: The number of characters to the right of the marker is non-decreasing.

The scanner is quite elementary. Characters are either components of identifiers, numbers, truth values, or delimiters. In the case of identifiers and numbers, these syntactic units are multi-charactered, and, when encountered, are accumulated in an accumulator until delimiter is reached. In this context this delimiter is known as a terminal element.

The scanner is under control of \( P1 \) translator routines each of which has a set of terminal elements, the occurrence of which create the
conditions under which these routines select their actions.

Of course, these terminal elements may themselves serve to activate
translator routines.

In order to organize the translation scheme, the 8S code itself
is the master control scheme. Sites in the 8 code may contain the names
of translator routines, e.g., \( X \), designated \([X]\), or, of course,
3-address code. Progressing through the 8 code now specifies the control
organization of the translation process.

As a matter of efficiency, certain translator routines have their
own sub-control organization; for example, the "expression" translator, and
that for some of the declarations. Indeed, the "expression" translator
produces 3-address code in a block rather than a list.

In the case of many of these special routines, the sub-routines
function when certain character sequences occur in IS. Their occurrence
causes certain actions, these are combinations of:

(1) Code generation

(ii) Substitution into, and movement of the markers

in, IC and 8C

(iii) Operations on the auxiliary tables generated during

the Pass.

These actions will be noted in the form of productions:

\[ \beta : S_1 \overset{a_1 a_2 \ldots a_k}{\Rightarrow} \alpha \rightarrow \gamma \]

meaning: If the character string \( S_1 \) is of the form \( \alpha \) \( S_2 \) then actions
\( a_1, a_2, \ldots, a_k \) are accomplished; following which, \( \alpha \) is the label of the
next production accomplished. If not of the form, \( \gamma \) is the label of the
next production.
1. The flow charts proper.

The analysis of a 20\(^L\) program is controlled by the block structure of 20\(^P\). Thus the flow charts naturally divide into:

(i) The analysis of declarations since they define that which occurs at the beginning of a block.

(ii) The analysis of statements since they form the content of a block.

(iii) The analysis of expressions since they form the content of most statements.

(iv) The analysis of identifiers since they form the content of most expressions.

(v) The analysis of the block end since administration of storage and identifier scopes is controlled in that way.

The flow charts reduce 20\(^L\) in a 3 address pseudo code using certain tables out of which a machine dependent Pass 2 would produce machine code.

The 3 address code so produced is itself tied together in a list structure. In general, the notation of the charts will be that of 20\(^L\).

5. Flow charts for declarations.

Block: If \(I[i,.\#] = \#\) begin then

\[\text{begin insert}(O[],\#); \text{list}(O[],\#); I[.,\#]\]

\[\text{go to main declaration and}\]

else

If \(I[.,\#] = \text{Blank}\) then

\[\text{begin go to Block and end}\]

else go to statement
comment The next step is to check the occurrence, if any, of a declaration;

Main declaration: macro declare (U, V) label V;

begin if I[.c] = U

then

begin HT Blockhead; go to V end end
declare('own', own)
real:
declare('real', real 1)
declare('integer', integer 1)
declare('logical', logical 1)
declare('index', index 1)
declare('list', list 1)
declare('Boolean', Boolean 1)
array:
declare('array', array 1)
declare('procedure', procedure 1)
declare('library', library 1)
declare('equivalent', equivalent 1)
declare('macro', macro 1)
If I[.c] = 'comment' then go to comment
If I[.c] = 'value' then go to value
go to compound

comment The preceding is the switching table for declarations in 20
Blockhead:

; s := s + 1; r := r + 1; orbik := r;
field(2, IT [k]) := true
DT [*] := Dt, | (r), (s)
Blockhead 1:  If Dt
            then
           begin
            isrt (3, 0[,])
            0[A] := code line (' n. Br: } (r) * (s))
            0[,X] := code line (' n. MT | (DA) ) end
            go_to | (Blockhead)

Comment 1. orbMK holds the index of the current block being processed

2. field gains access to the fields of tables

3. Dt is a block flag indicating whether the block is interior
to a procedure declaration

4. code line is a procedure generating its actual parameter
   as a code line

5. BA is a fixed location whose contents specify the variable
   location of the block administration routine;

own 1: delta [1] := true; I[,X] := 'daler'; I[,9]; go_to real
go_to array

integer 1: delta [3] := true; go_to real 2
logical 1: delta [4] := true; go_to real 2
Decimal 1: delta [5] := true; go_to real 2
go_to declaration;

list 1: delta [7] := true; go_to index 2
array 1: delta [8] := true; go_to index 2
procedure 1: delta [9] := true; Dt := true; I[,X]; procmast := procmast + 1
            isrt (O[,]) ; list (0[,X]) ; isrt (2, 0[,X]) ;
            param := 1; If I[,X] = letter
            then
then
begin A := identifier accumulated; indirect := true
delta [i2] := true; IF identifier declared
0[*,] := code line ( e(A); Pn).--)
next( 0[*] := code line ( e(A); Pn).--); I[*]
if I[*] = ')' then
begin i[*]; if I[*] = letter
then
begin A := identifier accumulated; IF identifier declared
I[*] := I[*]; if I[*] = '!', then go to procedure 3
if I[*] = ')' then go to main declaration
else go to alarm and
else go to alarm and
else go to alarm and

comment: The procedure name is declared and entered as having label character. Each of the actual parameters when declared is declared with indirect and param set to true to indicate their status. pronoun specifies the depth of procedure nesting current.
declaration: begin j := 0
declaration 1: if I[*] ≠ letter then go to alarm 3
else A := identifier accumulated; j := j+1; H[*] := A

IF identifier declared
if I[*] = '!', then go to declaration 2
else if \( I[i, q] = '1' \) then go to declaration 4

else go to clause 4

declaration 2:
\[ \text{Declaration} \]
\[ N_t = 1; \text{Expression terminal,} [k] s = 0 \]  

Expression Analyzer

declaration 3:
\[ 0_s \]  
\[ s = \text{code line (e} [R[j]), e^r = t, l) \]  
\[ 0_s \]  
\[ s = \text{code line (e} [C[j]), e^r = t, b) \]  
\[ \text{if } j \neq 1 \]  
then begin \( j = j - 1; \) go to declaration 3 end

else if \( I[i, q] = '1' \)
then go to clause 1

else if \( I[i, q] = '0' \) then go to declaration 4

else go to clause 4

declaration 4:
\[ \text{for } i = 1 \text{ step } 1 \text{ until } 12 \text{ do delta } [i] = 0 \]  
go to statement and end

Comment: declaration handles lists of identifiers processing the same declaration and, in particular, handles array declarations. In case of arrays:

(1) of 2 dimensions \( A [n; n, r; e] \) there is computed

- \( \text{column} = \text{abs}(s-r+1) \)
- \( \text{space} = \text{column} \times \text{abs}(n-r+1) \)
- \( \text{base} = \text{storage base} + r \times m \times \text{column} \ n \)

storage base = storage base + space

and there is stored in the address assigned to \( A \) and its successor, base and column. The mapping function for \( A [i, j] \) is then base + \( j + i \times \text{column} \).

The expression analyzer provides \( t, 1 \) as base and \( t, 4 \) as column;

equivalent \( I[i, j] = 0; I[i, q] \)

\[ \text{if } I[i, q] = '1' \] then go to clause 5

\[ I[i, q] \]  
\[ \text{if } I[i, q] = '1' \] then go to clause 3
At identifier accumulated; \( j = j + 1 \); \( N[j] = A \)

\[ \text{if} \ I[.?] \neq \% \text{, then go to alarm 6} \]

At identifier accumulated; \( j = j + 1 \); \( N[j] = A \)

\[ \text{if} \ I[.?] = \% \]

then begin

\[ \text{equivalent 2:} \]

\[ K_1 = A; \ ] \text{MT Expression Analyzer} \]

\[ j_1 = j - 1; \ ] \text{O[.?,]: = code line (} e(N[j]), \text{)}} \]

\[ A_1 = N[j]; \text{ indirect: } = \text{ true; MT identifier declared} \]

\[ \text{equivalent 3:} \]

\[ \text{if} \ I[.?] = \% \]

then go to equivalent 1

\[ \text{if} \ I[.?] = \%

then go to declaration 4

go to alarm 4

\]

\[ \text{end} \]

\[ \text{else} \]

\[ \text{if} \ I[.?] = \%

then begin

\[ j_1 = j + 1; \ ] \text{A}_1 = N[j]; \text{ MT identifier declared; } j_1 = j + 1 \]

\[ A_1 = N[j]; \text{ MT chain; go to equivalent 3} \]

\]

\[ \text{end} \]

\[ \text{go to alarm 4} \]

\[ \text{comment} \]

equivalent makes identifiers within blocks identical to those declared outside. In the case of a variable equivalent to an array this equivalence is established dynamically.

\[ \text{value 1:} \]

\[ I[.?] = \text{dollar} \]

\[ \text{value 2:} \]

\[ I[.?] \text{ if } I[.?] \neq \text{ letter} \]
then go_to alarm 3
A:=identifier accumulated; MT set value
if I [,φ] = ', '
then go_to value 2
if I [,φ] = '1'
then go_to declaration k
go_to alarm h

set value: ; delta [10]:= 1; KT identifier declared; go_to set value;
Comment: identifiers declared as values have that property inscribed in the identifier's table.
Comment: I [,φ]; if I [,φ] = '; '
then go_to statement end
and go_to comment
Macro: I [,φ]; delta [11]:= true
if I [,φ] ≠ letter
then go_to alarm 3
A:=identifier accumulated; KT identifier declared;
KT [,φ]:= A;
I [,φ]; I [,φ]
Macro 1:
if I [,φ] = letter
then begin
A:=identifier accumulated; KT [,φ]:= A; KT φ
end
if I [,φ] = '1'
then begin
I [,φ]; go_to macro 1
end
if I $[\phi] \neq '$
then go to alarm 4

MT $\phi$: Macro file; string transfer terminal: = 'end'
string transfer storage: = macro file

if string transfer

Macro file: = Macro file + string transfer norm

go to declaration 4

Comment Macros are stored in some 'external' file whose index is in Macro file.

MT is the table of Macros declared. MT is the table of Macro identifiers. String transfer is the table of Macro identifiers. String transfer is the name of a program which maps I $[\phi]$, up to the first non-matching end, onto the Macro file;

library; delta[13] = 1; I $[*,*]$: if I $[\phi] \neq$ procedure

then

library 1: begin I $[*,*]$: if I $[\phi] =$ letter

then

begin A: = identifier accumulated; delta [12]: = 1;

MT identifier declared; go to library 1 end

if I $[\phi] = '$
then go to library 1

if I $[\phi] = '$
then go to declaration 4

go to alarm 4 end library 1

else

library 2: I $[*,*]$: if I $[\phi] = $ letter

then

begin A: = identifier accumulated;

J: = library table (A)

J1: = field (2, J)

isrt (2, I $[\phi]$

next (I $[\phi]$: = 'library')
next next (I [,r]) : = procedure
    list ( I [,r] )
copy (I[,r],J); go to main declaration end

Comment: library table contains entries by name and peripheral storage location. Procedures named in library declarations are called at the end of Pass 2. Those named in library procedure declarations are inserted into the code at the point of name.

6. The analysis of expressions.

The expression analyzer produces 3 address code from the analysis of expressions. It acts through encountering delimiters. The delimiters upon which it acts are:

<table>
<thead>
<tr>
<th>Delimiter</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>!, *, /, +, -, &lt;, &lt;=, &gt;, $, -, ^, v, $=, $=, $, $=, $</td>
<td>( P [i] ) is the ( i )th position in the as-yet-unfulfilled expression stack.</td>
</tr>
<tr>
<td>( \text{else, then, \text{end}, ;, :=, \leq} )</td>
<td>( O [j] ) is the ( j )th position in the output sequence which is a block. There is an ( O [s, r] ) which points to this block.</td>
</tr>
</tbody>
</table>

6.1. Analysis of arrays.

Arrays may be characterized by two of their properties:

(a) They are of fixed (variable) dimension: a \( 0 (l) \)

(b) They are declared in a block exterior (interior) to a procedure \( b0 (l) \).
The catalogue of actions in the four cases are given in the following table:

<table>
<thead>
<tr>
<th>Case</th>
<th>Declaration</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>for A</strong>[4, 3]**</td>
<td><strong>Declaration</strong></td>
<td><strong>Call</strong></td>
</tr>
<tr>
<td><strong>Pass 1</strong></td>
<td><strong>A = 0</strong></td>
<td><strong>A = 0, b = 0</strong></td>
</tr>
<tr>
<td></td>
<td><strong>A : = base</strong></td>
<td>code for <strong>A + j + A + j = i</strong></td>
</tr>
<tr>
<td></td>
<td><strong>A + i : = column</strong></td>
<td>execution of above</td>
</tr>
<tr>
<td><strong>Pass 2</strong></td>
<td><strong>A = 0, b = 1</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>code for</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A + j + A + j = i + i</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if local or compute <strong>β</strong> in D.A. routine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store <strong>β</strong> in I,</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A + j + A + j = i + i</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Pass 2</strong></td>
<td><strong>A = 1, b = 0</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>code for</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A + j + A + j = i</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>execution of above</td>
<td>execution of above</td>
</tr>
<tr>
<td><strong>Pass 2</strong></td>
<td><strong>a = 1, b = 1</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>code for</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A[β] : = base</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A[β] + j + A + j = i</strong> if local or compute <strong>β</strong> in D.A. routine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>store <strong>β</strong> in I,</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A[β] + j + A + j = i</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>execution of above</td>
<td>execution of above</td>
</tr>
</tbody>
</table>


Expression analyzer:  Ex 7:  If $I_{[.\bar{q}]} = 20^L$ delimiter
then
begin $P[1] = I_{[.\bar{q}]}$

Ex 3:  if operator ($P[i=2]$) and proceeds ($P[i=2], P[i=1]$)
then  go to Ex 1

Ex 7:  if $I_{[.\bar{q}]} = \text{letter}$
then
begin $A_1 = \text{identifier accumulated}$
$M_1 = \text{identifier encountered}$
$P[i] = a_{(A)}$

Ex 2:  if $P[i=1] = 'in' \& P[i=2] = \text{delimiter}$
then
begin $P[1] = a_{(A)}$; $i = i+1$; go to Ex 7  and
if $P[i-1] = 'alga'$
then
begin $P[i-1] = P[1]$; $i = i+1$; go to Ex 7  and
go to alarm 7  and
if $I_{[.\bar{q}]} = \text{digit} \& I_{[.\bar{q}]} = '1' \& I_{[.\bar{q}]} = '0'$
then
begin $A_1 = \text{number accumulated}$
$M_1 = \text{number encountered}$
$P[i] = a_{(A)}$; go to Ex 2  and
go to alarm 7
Ex 1: \( i := i + 1 \); if arithmetic (P[i])

then

begin
if \( P[i-1] = t,1 \land P[i+1] = t,1 \)
then

begin
\( r := \min (1,m) \); \( s := \max (1,m) \); temp [s] = true end

else

begin
if \( P[i-1] = t,1 \)
then \( r := 1 \)
else

if \( P[i+1] = t,1 \)
then \( r := 1 \)
else \( r := \text{first available (temp)} \)

begin
\( r := \text{first available (temp)} \); temp [r] = false and \( \overline{\sigma} [j] = \text{code line (t,r, (P[i], (P[i+1])))} \)

\( j := j + 1 \); \( P[i] := t,1 \); \( t := P[i+1] \);

go to Ex 7 end

if relational (P[i])
then

begin
if \( P[i-1] = t,1 \) then temp [l] = true
if \( P[i+1] = t,1 \) then temp [l] = true
\( \overline{\sigma} [j] = \text{code line (t,l, (P[i], (P[i+1])))} \)

\( j := j + 1 \); \( \delta[j] = \text{code line (n,j, + G, J, + G, J+1)} \), \( P[i] := L, j; P[i+1] := P[i+2] \);

go to Ex 7 end

if logical (P[i]) \( \land \) Boolean (P[i-1]) \( \land \) Boolean (P[i+1])
then

begin
\( n := j + 1 \)
\[ q[i] = \text{field}(2, P[I-1]) \]
\[ r[i] = \text{field}(2, P[I+1]) \]
\[ t[i] = \text{field}(1, \delta[r]) \]
\[ \text{field}(1, \delta[r]): = \text{field}(1, \delta[q]) \]
\[ \text{if } P[i] = 'y' \]
\[ \text{then} \]
\begin{verbatim}
\text{begin} \\
  s[i] = \text{field}(h, \delta[q]) \\
  \text{field}(h, \delta[q]): = t \\
\text{Ex 5: } \text{if } \text{chain}(s) \text{ then} \\
\text{begin} \\
  u[i] = s \\
  s[i] = \text{field}(h, \delta[u]) \\
  \text{field}(h, \delta[u]): = t; \text{ go to Ex 5 and } \\
  \text{field}(3, \delta[r]): = \text{chain tag + q and } \\
  \text{if } P[i] = 'A' \text{ then} \\
\text{begin} \\
  s[i] = \text{field}(3, \delta[q]) \\
  \text{field}(3, \delta[q]): = t \\
\text{Ex 6: } \text{if } \text{chain}(s) \text{ then} \\
\text{begin} \\
  u[i] = s \\
  s[i] = \text{field}(3, \delta[u]) \\
  \text{field}(3, \delta[u]): = t; \text{ go to Ex 6 and } \\
  \text{field}(h, \delta[r]): = \text{chain tag + q and } \\
  P[i-1]: = P[i+1]; P[i]: = P[i+2]; \text{ go to Ex7 end; }
\end{verbatim}
if \( P[i] = 'i' \) \( P[i+2] = 'i' \)
then

begin
\( \delta[j] \) = code line \((t, 2, +, t, 2, t, 1)\); \( j = j+1 \)
\( \delta[j] \) = code line \((t, 2, +, t, 2, e(1))\); \( j = j+1 \)
\( \delta[j] \) = code line \((t, 2, \text{abs}, t, 2, -)\)
P \([i]\) = 'i'; go to Ex 7 end

if \( P[i] = 'i' \) \( P[i+2] = 'i' \)
then

begin
\( \delta[j] \) = code line \((t, 4, +, t, 4, t, 3)\); \( j = j+1 \)
\( \delta[j] \) = code line \((t, 4, +, t, 4, e(1))\); \( j = j+1 \)
\( \delta[j] \) = code line \((t, 4, \text{abs}, t, 4, -)\)
P \([i]\) = 'i'; go to Ex 7 end
else

if \( P[i] = \gamma \) then go to expression.

if \( P[i] = 'i' \)
then

begin
if proaw \([k]\)
then

begin
\( j = j+1 \); \( \delta[j] \) = code line \((t, \text{abs}, (p[i-1])\))
\( j = j+1 \); \( \delta[j] \) = code line \((-\text{go to } \Theta[k])\); \( P[i-1] = \Theta[k] \)
\( j = j+1 \); \( \delta[j] \) = code line \(\text{blank} \)
\( L[k] \) = \( j \) end
\( I[\ast] \); go to Ex 7 end

if \( P[i] = \gamma \)
then

begin
if proaw \([k]\)
then
begin \( j := j+1; \overline{O}[j] := \text{code line} (I_o \cdot (P[i-1])); \)
\( j := j+1; \overline{O}[j] := \text{code line} (\_ \cdot L_o \cdot (R[k])); \)
\( j := j+1; \overline{O}[j] := \text{code line} (\text{go to} \ (L[k])); \)
\( P[i-1] := L[k]; \ k := k-1; \ l := i \)

\textbf{Ex 4:}
\begin{align*}
\text{begin} & \ i := i-1; \ \text{if} \ P[i] = '!' \\
\text{then} & \\
\text{begin} & \ v := i; \ m := l; \ \text{go to} \ \text{Ex 5} \ \text{end} \\
\text{else} & \\
\text{begin} & \ i := i-1; \ \text{go to} \ \text{Ex 4} \ \text{end} \ \text{end} \\
\textbf{Ex 5:} & \ \text{begin} \ j := j+1; \ \overline{O}[j] := \text{code line} (MT \cdot (P[i-1])) \\
\textbf{Ex 6:} & \ \text{if} \ i = 1 \\
\text{then} & \\
\text{begin} & \ i := \nu+1; \ I[.,c]; \ \text{go to} \ \text{Ex 7} \ \text{end} \\
\text{else} & \\
\text{begin} & \ j := j+1; \ \overline{O}[j] := \text{code line} (\_ \cdot \text{st.} \ (P[i-1])); \ \text{TY}(m)) \\
\text{then} & \\
\text{begin} & \ m := m+1; \ i := i+2; \ \text{go to} \ \text{Ex 6} \ \text{end} \\
\text{if} & \ P[i-2] := P[i-1]; \ i := i-2; \ I[.,*]; \ \text{go to} \ \text{Ex 2} \ \text{end} \ \text{end} \\
\text{then} & \\
\text{begin} & \ \text{if} \ P[i-2] = '!', \ \text{P[i-6]} = 'dolar' \\
\text{then} & \\
\text{begin} & \ j := j+1; \ \overline{O}[j] := \text{code line} (t,2*;t,2;t,4) \\
\text{then} & \\
\text{begin} & \ j := j+1; \ \overline{O}[j] := \text{code line} (t,1*;t,1;t,4) \\
\text{then} & \\
\text{begin} & \ j := j+1; \ \overline{O}[j] := \text{code line} (t,1*t;t,1*t,3) \\
\text{then} & \\
\text{begin} & \ j := j+1; \ \overline{O}[j] := \text{code line} (t,1*; \ \text{Storage base} \ t,1) \)
\[ j := j+1; \overline{O}[j] := \text{code line (storage base, storage base, t,2) if proc nest \neq 0 then}
\]
\[ \text{begin } j := j+1, \overline{O}[j] := \text{code line } (e(P[i-5]), I, \text{st. t,1)}
\]
\[ j := j+1, \overline{O}[j] := \text{code line } (e(P[i-5]), I+1, \text{st. t,4) end}
\]
\[ \text{else}
\]
\[ \text{begin } j := j+1, \overline{O}[j] := \text{code line } (e(P[i-5]), \text{st. t,1})
\]
\[ j := i+1; \delta[i] := \text{code line } (e(P[i-1]), I, \text{t,1})
\]
\[ \text{if } P[i-7] \neq \text{array then begin } P[i-5] := P[i-7]; \text{go to Ex 14 end}
\]
\[ \text{else go to expression end}
\]
\[ \text{if } P[i-2] = \text{'}v\text{'} \land P[i-4] = \text{'}dollar\text{'} then
\]
\[ \text{begin } j := j+1; \overline{O}[j] := \text{code line } (t_2, t_2, t_2, t,1)
\]
\[ j := j+1; \overline{O}[j] := \text{code line } (t_2, t_2, t_2, e(1))
\]
\[ j := j+1; \overline{O}[j] := \text{code line } (t_1, t_1, t_1, e(1))
\]
\[ j := j+1; \overline{O}[j] := \text{code line } (t_1, t_1, \text{storage base, t,1})
\]
\[ j := j+1; \overline{O}[j] := \text{code line } (\text{storage base, t,2, storage base})
\]
\[ \text{if proc nest \neq 0 then}
\]
\[ \text{begin } j := j+1; \overline{O}[j] := \text{code line } (e(P[i-3]), I, \text{st. t,1}) \text{ end}
\]
\[ \text{else}
\]
\[ \text{begin } j := j+1; \overline{O}[j] := \text{code line } (e(P[i-3]), \text{st. t,1}) \text{ end end}
\]
\[ \text{if } P[i-2] = \text{'}v\text{'} then
\]
\[ \text{begin if nondynamic } (P[i-5]) \land \text{ proc nest } = 0 \text{ then}
\]
\[ \text{Ex 12: begin } j := j+1; \overline{O}[j] := \text{code line } (t_1, t_1, t_1, e(P[i-5]) + 1) \]
\[ j_s = j+1; \bar{O}[j]; = \text{code line } (I. \ast.t, l_t, t, 2) \]
\[ j_s = j+1; \bar{O}[j]; = \text{code line } (t, l_t. t, e (P[i-5]), I) \] 
\[ \text{end else} \]
\[ \text{if dynamic } (P[i-5]) \wedge \text{procnest} = 0 \]
\[ \text{then go to Ex 12} \]
\[ \text{else} \]
\[ \text{begin if nondynamic } (P[i-5]) \wedge \text{procnest} = 1 \]
\[ \text{then} \]
\[ \text{begin if local } (P[i-5]) \]
\[ \text{then} \]
\[ \text{begin } j_s = j+1; \bar{O}[j]; = \text{code line } (t, 2, t, 2, I) \] 
\[ \text{end else} \]
\[ \begin{align*} \text{begin } j_s &= j+1; \bar{O}[j]; = \text{code line } (\text{MT, BA}) \\
\text{Ex 13: } j_s &= j+1; \bar{O}[j]; = \text{code line } (t, 2, t, 2, I) \] 
\[ \text{end else} \]
\[ \text{if dynamic } (P[i-5]) \wedge \text{procnest} = 1 \]
\[ \text{then} \]
\[ \text{begin if local } (P[i-5]) \]
\[ \text{then} \]
\[ \begin{align*} \text{begin } j_s &= j+1; \bar{O}[j]; = \text{code line } (t, l_t. t, l_t, e (P[i-5]+1), I) \\
\text{Ex 13: } j_s &= j+1; \bar{O}[j]; = \text{code line } (t, l_t. t, l_t, 2) \\
\text{Ex 13: } j_s &= j+1; \bar{O}[j]; = \text{code line } (I. \ast. t, e (P[i-5]), I) \\
\text{Ex 13: } j_s &= j+1; \bar{O}[j]; = \text{code line } (t, l_t. t, l_t, I) \] 
\[ \text{end else} \]
\[ \text{begin } j_s = j+1; \bar{O}[j]; = \text{code line } (\text{MT, BA}); \text{go to Ex 13 and end.} \]
else
if P[i-2] = ']' then
begin if nondynamic (P[i-3]) \& proconest = 0 then
Ex 14: begin j := j+1; δ [j] := code line (I, st. t, 1)
j := j+1; δ [j] := code line (t, 1, st. e(P[i-3]))
Ex 15: P[i-3] := t, 1; i := i+1, go to Ex 7 end
else
begin if dynamic (P[i-3]) \& proconest = 1 then
begin if P[i-3] = local then
Ex 16: begin j := j+1; δ [j] := code line (I, st. t, 1, 1)
j := j+1; δ [j] := code line (t, 1, st. e (P[i-3]))
go to Ex 15; end
else
begin j := j+1; δ [j] := code line (MT, BA); go to Ex 16 end
else
begin if dynamic (P[i-3]) \& proconest = 0 then go to Ex 14
else
begin if dynamic (P[i-3]) \& proconest \neq 0 then begin if local P[i-5] then go to Ex 16
j := j+1; δ [j] := code line (MT, BA); go to Ex 12 end
else go to alarm end; go to Ex 17 end

Ex 16: begin j = j+1; 0[j] = code line (I. st. e P[i-3], I.\ldots)
     j = j+1; 0[j] = code line (I. st. t, 1, I.\ldots)
go to Ex 17 end

if P[i] = 'then'
then

begin q = field (2, P[i-1])
s = field (3, O[q]);
field (3, O[q]) = j+1;

Ex 9: if chain (a)
then

begin u = s; s = field (3, O[u]); field (3, O[u]) = j+1;
go to Ex 9 end; I[*,\ldots] go to Ex 7 end
else if P[i] = else then begin q = field (2, P[i-3])
    s = field (4, O[q]); field (4, O[q]) = j+1;

Ex 10: if chain (a) then begin u = s;
s = field (4, O[u]); field (4, O[u]) = j+1;
go to Ex 10 end;
I[*,\ldots] go to Ex 7 end
else if P[i] = then go to statement
else if P[i] = 'end' then go to statement

Ex 18: if P[i] = '(' then go to Ex 19
if P[i] = '[' then begin i = i+1; go to Ex 7 end
go to alarm

Ex 19: if P[i-1] = identifier
then
begin  
  \$k := k+1; R[k] = \text{label}\\  \$j := j+1; \overline{O}[j] = \text{code line (go to } 1 (R[k]))\\  \$j := j-1; \overline{O}[j] = \text{code line (blank)}\\  L[k] := j; \text{procmw}[k] = \text{true end}\\  I[*] := \text{go to Ex 7}\\

\text{Comment: Actual parameters of procedures are coded as follows:}\\  j := \text{blank}\\  \text{Evaluation of parameter into t,1}\\  \text{address of t,1 into I}\\  \text{go to } j$

These parameters will be entered from procedures via a \text{MT} command. The address $j, v = 1, 2, 3, \ldots, q$ is stored in the currently available parameter position in the procedure parameter stack. Thus the total coding is

\text{go to } K\\  \text{code for parameter } l\\  \text{.....}\\  \text{code for parameter } q\\  \text{code for storing } j_1 \text{ into } \Pi_1\\  \text{code for storing } j_q \text{ into } \Pi_q\\

\text{K: MT Procedure Name}
7. The Analysis of statements

Compound 1: begin

if I [],$] = 'if' then go to if statement
if I [],$] = 'go to' then go to statement
if I [],$] = 'for' then go to for statement

end

Compound 2: begin

If I [],$] = '(' then go to procedure statement
If I [],$] = '=' then go to assignment statement
If I [],$] = '|' then go to assembly code
If I [],$] = '+' then go to assembly code
if I [],$] = ';' then go to statement end
if I [],$] = 'end' then go to compound end
if I [],$] = 'block end' then go to block end

Assignment statement: insert (O[],$]); blocklist (O[],$)); O [],$]

end

go to expression analyser
procedure statement: go to assign statement
if statement: c:=1; isrt (2,O[c]); label (M[c]);
   next next (O[,c]); = code line (_= L.e(M[c]))
   list (O[,*,]); isrt (7,O[,c]); label (M[c+])
   label (M[c+2]); list (O[,*,]) isrt (2,O[,*]);
   o[*,*]: = recognizer (Boolean);
   o[*,*]: = M[c+1]; o [,*,] = e(M[c+2]);
   o [,*,] = code line (_= L.e(M[c+1]))
   o[*,*]: = recognizer (unconditional statement)
   o[*,*]: = code line (_= go to: e(M[c]))
   o [,*,] = recognizer (else)
   o [,*,] = code line (_= L.e(M[c+2]))
   o [,*,] = recognizer (statement)
   o[*,*]: = code line (_= go to: e (M[c]));
   head (O[,c]; go to master control

Comment  The list structure built up for if statements has the form shown in
snapshot form below. Z1, Z2, Z3 are label equivalents generated by label. >B<,
>US<, >else<, and >S< are the inserted recognizers for Boolean, unconditional
statement, else, and statements, respectively.
(, (, >B<, Z2, Z3), , , , , , ), L. Z1,
(, (, >B<, Z2, Z3), L. Z2, >US<, go_to Z1, >else<, L. Z3,
>g<, go_to Z1), L. Z1,

for statement begin: I [],*]; if I[,*] ≠ letter then go to alarm

A: = identifier accumulated
X: = A; insert (2,0[*]);
c: = c+1; d: = c; label (M[d]); c = c+1; e: = c
label (M[e]); next (2) (0[*]): = code line
(= Lo e (M[e]) = _); list (0[*]); insert (3,0[*]) next (1)(0[*])
: = recogniser (forlist); next (2) (0[*]): = code line
(= Lo e (M[d]) = _); next (3) (0[*]): = recogniser
(compound); block list (0[*]); go to Ex 7 end

for list
phrase
if P[1] = 'step' then go to step
if P[1] = 'until' then go to until
if P[1] = 's', A P[1-2] = 'until' then go to until 1
if P[1] = 'while' then go to while
if P[1] = 's', A P[1-2] = 'while' then go to until 1
if P[1] = 'do', A P[1-2] = 's', then go to do 1

phrase: begin insert (3,0 [*]); 0[*]): = code line
(= Lo e (M[d])); block list (0[*]); next ()[c]): =
recogniser(forlist); insert (2, II[*]); next (2) (II[*]): = *a*;
\begin{verbatim}
I[*,*]: = X; go to Ex 7 end

step begin isrt (5, O[*,*]); f := c+1; label (M[f]); g := c+1
label (M[g]); O[*,*]: = code line (._ go_to, e (M[f]))
O[*,*]: = code line (._ L0 e (M[g])); O[*,*]: =
code line (._ M0 e (M[d])); isrt (4, I[*,*])
next (1) (I[*,*]): = X; next (2) (I[*,*]): = X:=
next (3) (I[*,*]): = X; next (4) (I[*,*]): = X
block list (O[*,*]); next (O[*,*]) := recognizer (forlist)
end Ex 7 end

until begin isrt (6, O[*,*]);
O[*,*]: = code line (m. st. t, l.)
O[*,*]: = code line (._. L0 e (M[*3])); variable (Q[p]).
isrt (4, I[*,*]); next (1) (I[*,*]): = Q [p])
next (2) (I[*,*]): = X; next (3) (I[*,*]): = X
next (4) (I[*,*]): = X
next (2) (O[*,*]): = code line (t, l.*, m., Q [p])
next (3) (O[*,*]): = code line (._. l0, m., Q [p])
c := c+1
label (M[c]);
next (4) (O[*,*]): = code line (._. l0 e (M[q]), e (M[c]))
next (5) (O[*,*]): = code line (._. L0 e (M[c]))
next (6) (O[*,*]): = recognizer (forlist)
block list (O[*,*]); go to Ex7 end

until l; begin isrt (2, O[*,*]); next (2) (O[*,*]): = recognizer
(forlist); block list (O[*,*]); isrt (2, I[*,*]);
next (1) (I[*,*]): = X; next (2) (I[*,*]): = X
end go to Ex 7 end

while begin isrt (6, O[*,*]); O[*,*]: = code line (._. L0 e (M[f]))
next (2) (O[*,*]): = code line (._. t, l. true); c := c+1
\end{verbatim}
\text{label (M[c])}; \quad \text{next (3) (O[*,j]): code line (L. o (M[i]). o (M[j]))}
\text{next (4) (O[*,j]): code line (L. o (M[c]))}
\text{isr (I,II,[j]): next (1) (II,[j]): } \text{"if"}
\text{next (5) (O[*,j]): recogniser (forlist); block list (O[*,j])}
\text{go to Ex 7 and}
\text{do 1: begin isrt (2,O[*,j]); O[*,j]: code line (MT, o (M[d]))}
\text{do 3: $0$, [*,j]: code line (L. go to o (M[c])); go to 0 [*,j] and}
\text{do 2: begin isrt (1, O[*,c]); go to do 3}

Comment The for statement is the most complex control statement in $20^\ldots L_0$

Thus the statement for $i\leftarrow E_1, E_2, \text{step } E_3 \text{ until } E_4, E_5, \text{step } E_6 \text{ while } E_7, E_8 \text{ do } S$

would generate code controlled as follows:

\begin{align*}
  i &\leftarrow E_1 & \text{generated by for} \\
  \text{MT } d &\leftarrow i & \text{generated by for} \\
  &\leftarrow E_2 & \text{generated by expression analyser} \\
  \text{go to } f &\leftarrow \text{label } g & \text{generated by step} \\
  \text{MT } d &\leftarrow i+ & \text{generated by } i=+ \\
  \text{E } 3 &\leftarrow & \text{generated by } E_3 \\
  m &\leftarrow t, l & \text{generated by until} \\
  \text{label } f &\leftarrow & \text{generated by until} \\
  p &\leftarrow i = & \text{generated by until} \\
  \text{E } 4 &\leftarrow & \text{generated by until} \\
  \text{if } &\leftarrow p * m \leq 0 & \text{generated by until} \\
  \text{then } &\leftarrow \text{go to } g & \text{generated by until} \\
  i &\leftarrow & \text{generated by until} \end{align*}
go to $f'$
label $g'$ generated by step
MT d
i = i +
E_6
label $f'$
if
B_7 generated by while

= t, i, true
j, $g'$, c'
label c'
i = generated by until l
E_8
MT d
go to e generated by do 1
label $f$
statement
label e continuation

go to statement: P [l] = 'go to'; go to expression analyzer
statement end: go to master control
compound end: go to master control
block end: begin

Comment: The following code unravels all linkages between identifiers established in the current block. The example

\[
\begin{align*}
\text{E} & \quad \text{Y} \quad ( \text{E} \quad \text{X} \quad \text{W} \quad \text{X} ) \quad \text{X} \quad \text{W} \quad \text{Z} \quad ( \text{E} \quad \text{Y} \quad ( \text{E} \quad \text{X} \quad \text{Y} \quad \text{X} ) \quad \text{W} \quad \text{X} ) \quad \text{E} \quad \text{F} \quad \text{W} ) \\
1 & \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 21 \quad 22 \quad 23 \quad 24 \quad 25 \quad 26 \quad 27 \quad 28
\end{align*}
\]

where _ means declaration and () mean block beginning and ending, respectively and the order of occurrence being left to right would cause the following sequence of actions:

1. Assign x
2. Assign y
3. Block begin
4. Assign w
5. Enter x
6. Already assigned w
7. Already entered x
   
8. Block end, so: chain unassigned x to assigned x (1.5)
    remove block begin (3.)

9. Already assigned x through chain
10. Enter w
11. Enter z
12. Block begin
13. Assign z
14. Enter y
12. Block begin
16. Assign x
17. Already assigned x
18. Enter y
19. Already assigned x
20. Block end, so: chain unassigned y to preceding unassigned y
    remove block begin (15.)
21. Enter w
22. Already assigned z
23. Enter x
24. Block end. So chain unassigned x to assigned x through chain (1)
    chain unassigned w to unassigned w (10.)
    chain unassigned y to assigned y (14.)
    remove block begin (12.)
25. Enter s and chain (11.)
26. Assign s through chain (25.)
27. Assign w through chain (21)
28. Block end. So all assigned. Remove Block begin;

Block end: begin H: = K-1; if unassigned (T[M]) then go to Bl 1

Bl 2: if marker 
      ) ≠ ', ' 
      then begin H-1; go to Bl 2 end
      if number > length [s]
      then length [s] = number [r]
      go to Block end 1

Bl 1: Mu = H

Bl 3: 1 if chain (T[M])
      then begin Mu = field (5, T[M]); go to Bl 3 end
BL 4:  \textbf{if} field (3, T[T]) = field (3, T[MU]) \\
\hspace{1em} then go to BL 5 \\
BL 6:  \textbf{if} field (2, T[T]) \neq (*) \\
\hspace{1em} then BL 7: begin. T: = T-1; go to BL 4 \textbf{end} \\
\hspace{1em} Q: = Q-1 \\
\hspace{1em} if Q = 0 then go to BL 2 \\
\hspace{1em} go to BL 7 \\
BL 5:  \textbf{if} unassigned (T[[T]) \\
\hspace{1em} then begin field (5, T[MU]) : = T; go to BL 2 \textbf{end} \\
\hspace{1em} if block (T[T]) \geq \text{crtblk} \\
\hspace{1em} then go to BL 6 \\
BL 8: \text{MU: = H} \\
\hspace{1em} field (4, T[MU]): = field (4, T[T]) \\
\hspace{1em} number [r]: = number [r] + norm field (3, T[MU])) \\
\hspace{1em} if \hspace{1em} field (4, T[MU]) = 0 \\
\hspace{1em} then begin \hspace{1em} MU: = field (4, T[MU]); go to BL 8 \textbf{end} \\
\hspace{1em} field (4, T[MU]): = T; go to BL 2 \textbf{end} \\
Block end 1: \textbf{if} S: = 0 \\
\hspace{1em} then go to Pass 1 \textbf{end} \\
\hspace{1em} S: = S-1; go to master control \\
Master control: \hspace{1em} 0 [,*]; \textbf{if} recognizer (0[,o]) \\
\hspace{1em} then go to recognizer \\
\hspace{1em} go to block \\
Comment: The table T is the identifier table. Its field structure is \\
\hspace{1em} <line number>, <block marker>, <identifier>, <block number>, \\
\hspace{1em} <chain number>, <indirect>, <delta vector>.
identifier declared;

; if field (2, T[K]) = true
then go to ID 2
Hs = K

ID 3;
Hs = H-1; if field (3, T[H]) = A
then go to ID 4

ID 5;
if field (2, T[H]) = true
then go to ID 2
go to ID 3

ID 2;
field (1, T[K]) = K; field (3, T[K]) = A
field (4, T[K]) = crtblk; field (7, T[K]) = delta
Ks = K+1; go to identifier declared

ID 4;
if field (4, T[H]) = 0
then go to ID 7
MUs = H

ID 6;
field (4, T[MU]) = crtblk
if field (5, T[MU]) = 0
then go to identifier declared

MUs = field (5, T[MU])
go to ID 6

ID 7;
if param = 1
then begin field (7, T[H]) = delta; go to identifier declared end

 Identifier encountered but not declared;

Comment This routine is entered automatically whenever any identifier is encountered other than in a declaration. The identifier is H;

Hs = K-1
IE 1:  
if field \((3, T[H]) \neq A\)  
then go to IE 2  
if field \((4, T[H]) = \text{ctrl blk}\)  
then begin correspondent: = H;  
go to identifier encountered but not declared end  
if field \((4, T[H]) = 0\)  
then go to IE 3  
IE 2:  
if field \((2, T[H]) \neq '('\)  
then begin \(H:=H-1;\)  
go to IE 1 end  
field \((3, T[K]) = A;\)  
field \((7, T[K]) = \)  
union (field \((7, T[K])\), delta); \(K:=K+1;\)  
go to IE 4

Chain :  
\(H := K-1\)

Chain 1:  
if field \((3, T[H]) = \text{field (3, T[K])}\)  
then begin  
field \((4, T[K]) = \text{field (4, T[H])}\)  
field \((5, T[K]) = H;\)  
go to chain end  
\(H := H-1\)

if \(H=0\) then go to alarm

go to chain 1

Assembly code:

begin comment This code is machine code and its syntax has been described in Part I. Basically the format is operator | operand or operator -> operand;
if machine operand (A)
then go to AC 1

if macro (A)
then go to AC 2

go_to alarm end

AC 2:
list ([I],[Q])

for Z = 1 step 1 while MT (Z) ≠ terminal signal do

begin
if field (1, MT [Z]) = A

then begin H: = field (2, MI[Z]); go to AC 3 end end

go_to alarm

AC 5:
seqe (V, AC 6);

AC 10:
VI[*]; A: = V [VI,c]; seq1 (G, AC 10);

AC 9:
G[*]; def (W,G[*]); if A=W[*] then go to AC 8;

else go_to AC 9;

AC 9:
copy (W[*], VI[*]); go_to AC 10

AC 6:
copy (V,II[*]); go_to statement

AC 3:
Copy (MT[II], V); list G; seq W (G, AC 5); seqW(V,AC 5)

for Z: = 1 step 1 while MI[Z]≠0 do

begin insert ([G[*]); G[*]: = MI [Z]; I[*]; S: = 1

AC 11:
list (G[*]); if I[*]=(')

then begin S: = S + 1; AC 12: insert (G[*]); G[*]: = I[*] and

if I[*] = (') then begin S: = S - 1; go_to AC 12 end

12

AC 12
if S: = 0 then go_to AC 11

12

AC 11:
end

AC 1: begin comment This code section analyzers machine assembly code. It uses
the address expression analyzer. The constituents of address expressions are
identifiers, integers, the arithmetic characters + and -. The address expressions
are machine dependant and the analysis given is for the Bendix G-20.
AC 25  insert (O[$,f$])

AC 35  if $I[.f] \not\in \text{simple variable} + OA \lor I[.f] \not\in OA + \text{simple variable}$
then go to AC 26

if $I[.f] \not\in \text{simple variable} + OA$
then go to AC 27

if $I[.f] \not\in OA + \text{simple variable}$
then go to AC 27

if $I[.f] \not\in \text{simple variable} + OA$
then go to AC 30

if $I[.f] \not\in OA + \text{simple variable}$
then go to AC 30

if $I[.f] \not\in \text{simple variable} + OA$
then go to AC 32

if $I[.f] \not\in OA + \text{simple variable}$
then to
then go to AC 32

if $I[.f] \not\in \text{simple variable} + \text{simple variable}$
then go to AC 34

Go to alarm

Note: similar code need be written for occurrence of 

AC 26: MT AC 36

O[.]* = code line (_. OCA(O), e (A),_. )

AC 26f: $I[.c] = 'OA';$ go to AC 25

AC 27: MT AC 36
AC 30:  MT AC 36
O[,*]: = code line (_*. OCA(3). G(A). _); go to AC 28
AC 32:  MT AC 36
O[,*]: = code line (_* OCA(1). e(A). _); go to AC 28
AC 34:  MT AC 36
O[,*]: = code line (_* OCA(0). e(A)); go to AC 28
AC 36:  ;A: = simple variable identifier accumulated;
go to AC 36
Appendix 1.

1.1 In declaration s add

Switch 1: If I [i,j] = letter then go to switch 5

    go to alarm

Switch 5: A: = identifier accumulated; delta [12]: = 1

MT identifier declared; SW:A; mu := 0

P [1]: = 'switch', go to expression analyzer

1.2 In the expression analyzer add

1.2.1 code relating to switch

    if P[i] = ';=' P[i-2]: = 'switch' then go to switch 3

    go to Ex 7

Switch 3. P [i] := [i-2]; i := i+1; go to Ex 7

    if P[i] = ';', P[i-2] = 'switch' then go to switch 4

Switch 4 begin if simple (P[i-1])

    then begin code constant ( e(sw) + mu: = code line(_, go to,

    (P[i-1])._.) mu: = mu +1 end

    else begin code constant (e(sw) + mu: = code line(_, go to e(TSW)

    0[j]: = (_., go to 0, I.) end

    go to switch 3 end

    if P[i] = 'switch' then

    begin label (TSW); lsr (2,C[1,c])

    0 [i,*]: = (_., l.s(TSW))

    block list (0[i,*]) end

    go to Ex 7

1.2.2 code relating to go to statements
\[
\text{if} \ (P[i] = T \lor P[i] = 'e'; \ \forall P[i] = 'and') \land P[i-2] = 'go to' \\
\text{then local (P[i-1]) then go to go to 1} \\
\text{else go to go to 2 end}
\]

```
go to Ex 7

go to 1: begin j = j+1; \overline{O}[j]: = \text{code line} (_- \text{go to} \ e (P[i-1])._)
  go to statement end end

go to 2: begin j = j+1; \overline{O}[j]: = \text{code line} (I. \& e (P[i-2])._)
  ETA: = block (P[i-1])
  j = j+1; \overline{O}[j]: = \text{code line} (I. \& e (ETA)._)
  j = j+1; \overline{O}[j]: = \text{code line} (_- \text{go to} \ BA \text{ go to}._)
  go to statement end end
```
PART III

A Programming Manual for 20^L

1. Introduction

20^L is a programming language for -- in the main -- scientific computation. There is, in the language, no extensive input/output facilities. These are provided by procedures but a few sample such will be mentioned in the sequel.

20^L is one language with one processor even though the programer may write in 3 different modes: algebraic language, assembly language, or list language and they may be mixed as desired by the programer. Thus, for programs requiring -- for reasons of speed and efficiency of storage (word packing) -- total control over the machine's abilities the programer may, in continuous transition, skip into symbolic machine language.

On the other hand, if he is doing extensive symbol manipulation he may choose to program more extensively in a list formalism.

Most importantly as a program is debugged it can be altered from the language which it is easiest to test the logic of the program into that in which it is most efficient to operate the program.

2. Programming Principles

The languages used on computers follow very closely two fundamental principles of computer design:

(i) The nature of storage and (ii) The sequencing of control

The computer's storage is divided into units called "words." These are of fixed length -- or sometime small multiples of units of fixed length. Each word can be identified by a natural number called an address which, by
the nature of mechanical devices has a range, e.g., from 0 to $2^{15}$. Words
store numbers and consequently we may say that an address is the name of a
problem variable if the contents of the word having that address change in the
manner specified by operations on that variable.

The storage is further characterized by destructive read-in and
non-destructive read-out, meaning that each time information is stored in
a word the previous contents are replaced by the new information. However,
when information is read from the word the contents are restored on read-out.

Thus:

```
read in
the "new"
number

old number
lost
```

```
read out
the number

also read it
back
```

In some applications the programmer constructs, through programming,
a pseudo-memory called a stack or push down list. In this memory each time a
word is read into the same "cell" the previous contents are not lost but are
pushed down deeper into the stack. Here, reading can be destructive or not as
wished. But this is accomplished by programming and not by hardware. It is
particularly easy to accomplish using Tlists,

The second characteristic is sequencing of control.

Each instruction has an identifier called its label or name (absence
implies a blank label) and an operation part.

In the computer each time an operation is completed a new is chosen
according to the rule:
(i) if the operation does not specifically identify the name of its successor the lexicographic next is next to be executed.

(ii) if it does identify its successor then that one so identified is next executed.

The source of computer flexibility is that the choice of (i) or (ii) in any instance may be made conditional on the consequences of already accomplished computation.

In the case of $20^\wedge$ the sequencing rules are slightly more complicated by the concept of the control statement. The control statement induces a sequencing control over the statements within its scope. The scope is defined as the set of all statements following until a punctuation convention is satisfied. The lexicographic last is called the terminal statement. Then the above sequencing rules hold with the additional rule:

(iii) if the current statement is the terminal of a control statement, the successor is determined by the control statement.

Thus in $20^\wedge L$ programming the programmer must be constantly aware of this interplay between sequencing and assignments of values to variables by which his computational purpose is advanced.

There are two fundamental aspects to programming in $20^\wedge$: one is the programming of cycles or loops and the other is the analysis of arithmetic expressions. The semantics of the language has already been discussed and this manual is thus concerned with principle and example.
1. Programming of cycles or loops.

Almost without exception every algorithm executed on computers contains at least one cycle. For, without a cycle, every step of the algorithm would be executable at most one time. The execution time of the algorithm could then be of the same order as the time required to describe it. Even for algorithms without explicit cycles, its availability in a standard form in a library (from which reference and extraction are possible) imbeds the algorithm in the "library cycle".

```
Extract the
algorithm identified as "..."
from the library and make
a copy of it

Employing the
current copy
execute the algorithm
```

For each use the re-description time is effectively zero. We turn our attention to cycles within algorithms.

A chain (of instructions) is a sequence of instructions $I_1, I_2, \ldots, I_n$ such that for each $k$, $2 \leq k < n$, $I_k$ is the successor of $I_{k-1}$ and the predecessor of $I_{k+1}$.

A chain which starts and ends at the same instruction is called a cycle.

A cycle of instructions clearly permits the same sequence to be carried out several times. In programming we note the obvious constraints on cycles so that they are not carried out an infinite number of times:

1. Every cycle must possess a branch or comparison instruction (else it could never terminate), and

2. At least one storage location must change its contents during the course of a cycle satisfying (1).

The termination condition is of great importance and is one of, or combination
(a) A counter achieves its final value after stepping through a sequence of intermediate values.

(b) A relation is first satisfied (or not satisfied), e.g., \( x < y \), or \((x \neq y) \land (z < y + 3)\), etc.

In the case (a), the flow chart is of the form:

```
set the counter

has the Counter passed its final value?

/\ Y

/\ ~

Execute the instructions of the cycle

step the counter
```

In general the counter, \( i \), steps through an increasing or a decreasing sequence of values with the initialization, the progression, and the termination determined by fixed functions. An obvious notation is: \[ i = E_1 \text{ step } E_2 \text{ until } E_3 \text{ do } < \text{ instructions of the cycle } \] The chart becomes:

```
\[ i = E_1 \]

\[ \text{Sign } (E_2) \land (i - E_3) > 0 \] ? \n
\[ \text{Y} \]

\[ \text{N} \]

Execute the instructions of the cycle

\[ i = E_3 \]
```
Sometimes the cycle continues while an arithmetic relation requires termination.

In these cases we may write:

```
>>> for i = E₁ step E₂ while R
```

Exercises:

1. Draw the cycle chart for case (b) of page 101.
2. Draw the cycle chart for the description `++`.
3. How would the description `++` be modified for the case of cycling until R first becomes satisfied?

In many cases cycles are nested, i.e., within the instructions of the cycle is the complete specification of another cycle. Using outline notation:

```
  o
 / \     
N   Y
   / \   
  I   C
   / \   
  S   N
```

represents two nested blocks. The blocks which must be present are labeled as:

initialize; C: check; S: step. Using parentheses to indicate cycle relationships, we find situations like: ( ) as above, but also ( ( ) ) and combinations like: do not use overlapping cycles such as ( )

Exercise:

1. Suppose we use the recursive cycle beginning at the specified beginning and end of a cycle method. How should we specify which condition must

Best Available Copy
any sequence of such symbols to determine if it is an allowable nested sequence.

2. Modify 1. so that the maximum depth of nesting in such sequences is computed, e.g., (((((())(()))))) has a maximum depth of 5.

Often several progressions, say n, share the same set of instructions over which to cycle. This case may be treated in a simple manner by inducing a master cycle, k, which progresses in steps of 1 from 1 to n selecting and applying each of the progressions in turn to the instructions over which to cycle. The chart involves a selection switch, T, having n steps:

![Diagram showing the selection switch and progression control logic.]

The chart can be treated by this technique.
Examples of cycling:

1. Nested cycling

Order the numbers (in) \( A_1, A_2, \ldots, A_n \) so that \( i > j \) implies that \( A_i \geq A_j \).

Start

\[ i \leftarrow 1 \]

\[ i > n-1 \]

\[ \downarrow \]

\[ N \]

\[ j \leftarrow i+1 \]

\[ j > n \]

\[ \downarrow \]

\[ A_i \geq A_j \]

\[ t \leftarrow A_i \]

\[ A_i \leftarrow A_j \]

\[ A_j \leftarrow t \]

\[ j \leftarrow j+1 \]

\[ i \leftarrow i+1 \]

Stop

2. Shared cycling

Outside of progressions with gaps it is difficult to select a simple example using shared cycling. Nevertheless, consider the problem to compute:

\[ f(\cdot) \text{ for } 0 \leq x \leq 5 \text{ in steps of 0.1, i.e.,} \]

\[ f(x) \text{ for } x = 0 (0.1) 5, \text{ where} \]
Otherwise undefined
and
\[ z = x^2 + \frac{1000}{\pi} \sum_{y=\lfloor \frac{y}{x} \rfloor}^{1000} \frac{1}{y^2}, \] \nu \text{ integral.}

Some of the cycling here can be shared.

The case where the extent of cycling is determined by a relation is treated in a similar way, except that a cycle counter may or may not be present. Thus:

\[ \frac{X-\frac{y+a}{2}}{2} \]

\[ y+\frac{1}{2} \left( x+\frac{a}{y} \right) \]

\[ \frac{y-x}{\sqrt{E}} \geq 0 \quad \text{Stop} \]

\[ X \leftarrow y \]

cycles, not on a counter, but while (meaning as long as) \( \frac{y-x}{\sqrt{E}} \geq 0 \). The above chart represents Newton's method for finding the square of \( x \).

Exercise:

1. Add a counter to the above chart so as to count the number of cycles required before exiting.
2. For all integers \(1, 2, \ldots, N\) generate the sequence of integers \(\sqrt{N}\) which are prime, i.e., possess only 1 and themselves as factors. Hint: If \(N\) is an integer its largest factor different from itself must be \(\leq \sqrt{N}\).

3. Suppose a continuous function \(f(x)\) is specified for \(a \leq x \leq b\) and that \(f(a) \neq f(b) < 0\). Let the \(x\)-intercept of the straight line between \((a, f(a))\) and \((b, f(b))\) be a first approximant of a zero of \(f(x)\). Using a sequence of such straight lines develop a sequence of approximants terminating the process as soon as two successive elements of the sequence differ in magnitude by less than a given \(S > 0\).

2. Propositions or relations.

An influential component of every computation is the set of discriminations which influence the branching and eventually determine terminating conditions. Each such discrimination is a question which is, each time, answered yes or no. But each such question is also a proposition having a value true \((T)\) or false \((F)\). Since propositions take on only two values these may be represented by \(1\) \((T)\) or \(0\) \((F)\). Thus the question: "Is \(x > 7\)?" may be treated as a proposition \(P(x) = (x > 7)\) which in reality is a propositional function.

If \(E_1\) and \(E_2\) are arithmetic expressions, e.g., \(X + 2 \neq 5\) and \(n^{\sqrt{n-s^*}}(9y)\), and \(R\) is an arithmetic relation, e.g., \(>\), \(<\), \(\geq\), \(\leq\), \(=\), \(\neq\), then \((E_1 R E_2)\) is called an elementary (arithmetic) proposition.

Since propositions take on the values 0 and 1 it is convenient to define propositional variables which take on only these values. This permits us to write equations like:

\[b \neq (E_1 R E_2)\]

Propositional variables are sometimes called Boolean variables; these, too, are elementary propositions.

Compound propositions are formed from elementary propositions by combining
107

the latter under basic propositional operations. These basic operations are:

(1.) Complementation (\(\neg\)): If \(P\) is a proposition then so is \(\neg P\) defined as:

\[
P = \neg \neg P = 1, P = 1 - \neg \neg P = 0.
\]

or in tabular form:

<table>
<thead>
<tr>
<th>(P)</th>
<th>(\neg P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\(\neg\) satisfies: \(\neg (\neg P) = P\).

If \(P\) and \(Q\) are propositions then so are:

(2) \(P \lor Q\) defined by the table:

<table>
<thead>
<tr>
<th>(P)</th>
<th>(Q)</th>
<th>(P \lor Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(\lor\) is called the inclusive or, i.e., \(P \lor Q\) is true if either or both of \(P\) and \(Q\) are true.

(3) \(P \land Q\) defined by the table:

<table>
<thead>
<tr>
<th>(P)</th>
<th>(Q)</th>
<th>(P \land Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(\land\) is called and, i.e., \(P \land Q\) is true if and only if \(P\) is true and \(Q\) is true.

(4) \(P \rightarrow Q\) defined by the table

<table>
<thead>
<tr>
<th>(P)</th>
<th>(Q)</th>
<th>(P \rightarrow Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(\rightarrow\) is called implication.
(5) \( P \equiv Q \) defined by the table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( P \land Q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( \equiv \) is called **equivalence** i.e., \( P \equiv Q \) is true if and only if \( P \) and \( Q \) have the same value.

The above tables are called truth tables and each binary propositional operation defines and is defined by such a table.

Unlike binary functions, involving operations on real numbers to form real numbers of which there are non-denumerably many, there are only a finite number of binary operations mapping propositional variables to propositional variables.

**Exercises:**

1. How many such binary propositional operations are there?
2. If \( P, Q, R \) are propositional variables how many propositional functions of 3 variables are there?

Hint: Start the counting analysis from the truth table.

3. If \( \lor \) is used for **exclusive or** i.e., either (but not both of) \( P \) and \( Q \) are true for \( P \lor Q \) to be true, construct the truth table and represent \( \lor \) in terms of (1), (2), (3).

It is often quite convenient to synthesize propositional expressions from the truth table. In order to do this we investigate some properties of these operations, particularly (1), (2), and (3). The following identities are easily proved by truth tables:

\[ (1) \ P \lor \neg P = 1 \]  
\[ (2) \ P \lor 1 = 1 \]  
\[ (3) \ P \lor 0 = P \]  
\[ (4) \ P \land \neg P = 0 \]  
\[ (5) \ P \land 1 = P \]  
\[ (6) \ P \land 0 = 0 \]
Propositions obey the **distributive law** of $\land$ over $\lor$:

$$P \land (Q \lor R) = P \land Q \lor P \land R.$$  

Both $\land$ and $\lor$ obey the associative law:

$$P \land (Q \lor R) = (P \land Q) \lor R = P \land (Q \lor R);$$

and the commutative law

$$P \land Q = Q \land P,$$

where $\circ$ means $\lor, \land$.

Consequently, $t. y$

(7)  $P \lor (P \land Q) = P.$

By constructing the truth tables it is demonstrated that

(8)  $\neg (P \lor Q) = \neg P \land \neg Q$  and

(9)  $\neg (P \land Q) = \neg P \lor \neg Q.$

One defines that $\lor$ is the dual of $\land$ and vice versa; and $\neg \neg$ is the dual of $\neg$ and vice versa.

If $S$ is a propositional expression involving only $\lor, \land, \neg$, and elementary propositions, then the dual of $S$ is obtained by replacing, in turn, from left to right each occurrence of $\lor, \land, \neg$ by its dual.

Example:

$S = (\neg P) \land \{(Q \lor \neg S)\}$  

Note: $Q$ means $\neg \neg Q$.

then dual $(S) = P \lor (Q \lor \neg S)$  

$= \neg S.$

A general theorem is that $\neg S = \text{dual}(S)$ for any propositional function.

In order not to have to write parentheses to excess, as in the case of arithmetic operators, an assumed hierarchy of propositional operations is defined:

$\neg$ before $\land$ before $\lor$ before $\Rightarrow$ before $\equiv$. 

Now consider the case of a propositional function $F$, of, say, 3 variables $P$, $Q$, and $R$, i.e., $F(P, Q, R)$ defined by the table of $2^3 = 8$ entries:

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$R$</th>
<th>$F(P, Q, R)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$f_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$f_1$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$f_2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$f_3$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$f_4$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$f_5$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$f_6$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$f_7$</td>
</tr>
</tbody>
</table>

Where each $f_i$ is 0 or 1. Then $F$ can be "expanded" into a "sum" of eight "products":

$$-P \land \neg Q \land \neg R \land f_0 \lor P \land \neg Q \land \neg R \land f_3 \lor P \land Q \land \neg R \land f_2$$

for any values given to $P$, $Q$, and $R$ cause one and only one of the eight products to differ from zero (i.e., be 1) so that its value is $l f_k$ and $f_k$ is the function value corresponding to the given triplet of values for $P$, $Q$, and $R$.

Example:

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$R$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$f_0$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$f_2$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$f_0$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$f_5$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$f_6$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$f_7$</td>
</tr>
</tbody>
</table>

$$F = \neg P \land \neg Q \land \neg R \lor P \land \neg Q \land \neg R \lor P \land Q \land \neg R$$

$$= (P \land Q) \lor (P \land \neg Q) \lor (Q \land \neg P)$$

$$= P \lor Q$$
In programming, propositional functions are used to determine branching patterns in programming. If P is a proposition we represent it in flow charts as:

\[ P \]

Then, \( PVQ \) is represented as

\[ P \]

or that obtained in permuting \( Q \) and \( P \). \( P \& Q \)

is

\[ P \]

while \( QAP \) is that obtained in permuting \( Q \) and \( P \).

While the propositional affect of \( PVQ \) and \( QVP \) is the same, we will see later there is sometimes reason to choose one over the other.

We will now apply the foregoing to a specific example: Evaluate and print \( F(X, T) \) for \( Y=0(1) \) 20 and for \( X=0(1) \) 20

In English:

\[ F(X, Y) = F_1(X, Y) = X^2 \left[ \frac{X^2}{3} \right] \] if \( 0 \leq X < 3 \) or \( 7 \leq X < 11 \) but not \( X \) divides \( Y \) or \( X \) even; should it not be \( X^2 + \left[ \frac{X^2}{3} \right] \) then

\[ F(X, Y) = F_2(X, Y) = X^2 - X \cdot \left[ \frac{X^2}{2} \right] \cdot \left[ Y + 1 \right] \] unless \( X \) is even in which case

\[ F(X, Y) = F_1(X, Y) - F_2(Y + 1, X). \]

Comments: (1) The above is somewhat ambiguous. It can be clarified by requiring its statement in an unambiguous problem language. (2) What precisely is meant by "for \( Y = 0(1) \) 20 and \( X = 0(1) \) 20"? Does it mean simultaneous or iterated cycling? What happens if \( 11 \leq X \)? Is it the sense of the problem that the scope of the first or is "\( \leq X \) li", or "\( 7 \leq X \) 11 but not \( X \) divides \( Y \)" or the preceding coupled to "or \( X \) even"?
These and similar questions can be answered by the use of operators and parentheses, thusly:

```
for Y = 0 step 1 until 20 do
    begin for X = 0 step 1 until 20 do
        begin if ( 0 ≤ X < 3 V 7 ≤ X < 11 ) (divides (X,X) V divides (2,X) )
            then F : = X↑2 + entier ( X↑2 / 3 )
        else begin if (divides(2,X/2))
            then F : = X↑2 - X x entier (X↑2/3) / (Y+1)
        else F : = Y↑2 + entier (Y↑2/3) - ( (Y+1)↑2 - (Y+1)x entier
            ( (Y+1)↑2/3 ) / (Y+1) )
        print ( F ) end end end

halt;
```
4. Encoding of algebraic formulae.

Everyone is familiar with formulae. Some examples are:

a) \( t = x + 4 \times y \)

b) \( t = (x-z) \times z + y/(2 + r) \)

c) \( s = e \times (r + a/x) \)

The expressions on the right are said to define the variables on the left. In computation the counterpart is that the right hand sides are evaluated, using the current values of the variables appearing there, thus defining a value for the variable on the left. This time sequence is emphasised by using a sign like \( \leftarrow \) or \( := \) in place of \( = \) thus:

a) \( t \leftarrow x + 4 \times y \)

or
b) \( t := (x-z) \times z + y/(2 + r) \).

with this sense of \( := \) (which will be used hereafter), a formula

\[ t := t + x \times t \]

has computational meaning, i.e., it defines a specified set of actions to be carried out. Thus, in Newton's method for computing the square root of \( "a" \), the formula

\[ x := \frac{1}{2} \left( x + \frac{a}{x} \right) \]

yields the iterants successively, with each new one computed from the preceding one.

In computation each variable may be said to be the name of a storage location and its contents the (current) value of the variable. In the above, clearly, the new value of \( x \) reads over the previous value and unless otherwise
where retained it would be lost when read over.

Much of what we have learned about formulae is based on the assumption that formulae are generally simple, i.e., not too many symbols and not too many expressions involving repeated division or exponentiation.

In computation these assumptions are no longer generally true. Thus while \( x^2 \) is quite unambiguous, \( x^2 + 1 \) is less clear unless the printing is very precise, and then what of \( x^{a+1} \) ? Consequently, the formulae dealt with have the property that:

All formulae are represented as finite linear sequences of characters.

Thus such formulae do not contain either superscripts, subscripts, or exponents. Naturally a representation needs to be introduced to take their place. All such representations are based on a partition of the alphabet of recognizable characters into disjoint classes. For example, such a partition might be:

a) The set of operators: \( +, -, \cdot, /, \div, \frac{\text{a}}{\text{b}}, \ldots \).

b) The set of two sided delimiters: "(, )".

c) The set of numbers, e.g., 2, 10.5, 110.

d) The set of variables: \( X, M, N, \ldots \).

These latter two classes may described by example. Now are they to be described explicitly?

In a subsequent set of notes a method for their formal description will be given.

Then a formula
could be represented by the linear character string:

\[ Y := C \cdot (A + D I + E \cdot 2 + 2^2) \]

(1)

where \( \cdot \) denotes exponentiation.

The operators are binary; they have two associated operands, e.g.,

in (1) for the underlined operators the associated operands are:

\[
\begin{align*}
A & \cdot D I \\
R & \cdot 2 \\
2 & \cdot R \cdot 2 \\
E & \cdot (2 + R \cdot 2) \\
A & \cdot D I \cdot E \cdot (2 + R \cdot 2)
\end{align*}
\]

and, even

\[ Y := C \cdot (A + D I \cdot E \cdot (2 + R \cdot 2)) \]

(2.7)

The assignment of operands to operators is determined by:

(i) The direction of scan of the formula, e.g., from left to right;

and

(ii) The hierarchy of the operators, e.g.,

\( \cdot \) before \( \cdot \) before / before \( \cdot \) before + before =.

However, all operations within a matching set of parentheses are accomplished before the operation for which the matching set of parentheses delimit an operand, as (2.3) precedes (2.4).
Exercises:

1. For the following, represent the formulae by linear strings of characters:

   \[ A = \ldots \]
   \[ C = \frac{D}{E \ast F \ast (R - M)} \]
   \[ X := 3 \]
   \[ Z := X \ast C + \frac{E \ast F}{L \ast T \ast y + z^2} \]

2. For the following formulae list the order of operations in their evaluation as in (2.1 thru 2.7):

   \[ Y := Y \ast Z \ast R \ast PHI \ast (A + B) \]
   \[ Z := X^{23/127} \ast Z48/L + 1 \]
   \[ W := PHI \ast 3 + W/L/M \]

3. The following charts each evaluate a single formula. Find them.
In exercise 3, the flow charts can be easily converted to code, with the possible exception of the computation with the operator \( \downarrow \). We defer its analysis. Note that the charts can still not be converted into code until the nature of the operands and operators is clarified as to whether integer or fractional, since conversion may be required.

This clarification proceeds as follows:
The analysis should be based on a cascaded treatment of expressions. The simplest expression is that containing one operand \( E_n \), e.g., \( x, 3.4, 25, M_{ij} \), etc. Then the natural definition is:

The arithmetic of an \( E_n \) is that of its operand.

Now how is the arithmetic of an operand specified? It is natural that its arithmetic be unchanged during a computation, so the rule is:

The arithmetic of an operand named by an identifier is declared by the programmer, not by virtue of its identifier form. Such a declaration will have the form:

\[
\begin{align*}
\text{real} & \quad x, M_{ij}, TIPS, ROO0\% \\
\text{integer} & \quad ZZ, l4, Pl \\
\end{align*}
\]

Operands named by numbers are defined to be those numbers. Any such number consisting of digits alone is understood to be an integer, e.g.,

11, -7, 1230 but not 21.k, 0056, 1.5, 8, 2/7.

These latter are representations of fractional numbers.

Now if both simple operands, \( a_1 \) and \( a_2 \), are of the same arithmetic,
that of the expression formed from them O₁ op O₂ would reasonably have
that arithmetic. However the division of two integers does not necess-
arily produce an integer. Nor does exponentiation, as for example, 3 ↑ ↑ 7.
An acceptable solution is to have two division operators possessing the
sign / and \. The former always yields a fractional number while the
latter always yields an integer. But what integer, e.g., 3 ÷ 7, 25 ÷ 2; 107 ÷ 8?
Convention has established the definition:
\[ \text{a} \; ÷ \; \text{b} = \text{sign} \left( \frac{\text{a}}{\text{b}} \right) \left\lfloor \frac{\text{a}}{\text{b}} \right\rfloor \]
Thus for the above, \( 0, \; 2, \; -5 \). While use of \( / \) would give \( \text{in a typical machine} \)
representation \( 028571b350. \; 1250000052. \; 5875000051. \)

Exponentiation may be treated as follows for \( \text{a} \; \text{b} \).

- if \( b \) is an integer and
  - if \( b > 0 \), then \( a * a * \ldots * a \) (b times) and consequently of
    the same arithmetic type as \( a \).
  - if \( b = 0 \), and if \( a > 0 \), then 1 of the same type as \( a \).
  - else undefined.
  - if \( b < 0 \), and if \( a < 0 \), then \( \frac{1}{a^b} \) (The denominator
    has \( b \) factors!) and is of type real or undefined.

- if \( b \) is real and
  - if \( a > 0 \), then \( \exp (b \ln(a)) \) of type fractional
    - if \( a < 0 \), then if \( b > 0 \), 0 of type fractional; else
  - if \( b \) undefined
    - if \( a > 0 \), always undefined.
The use of conditional expressions is very natural in some instances. Thus:

\[ y := \begin{cases} 
0 & \text{if } x \leq 0 \\
1 & \text{else} \\
\end{cases} \]

defines \( y \) as the discontinuous function:

\[ y = \begin{cases} 
0 & x \leq 0 \\
1 & x > 0 \\
\end{cases} \]

Similarly

\[ b := (x \leq 0 \text{ then } c \land x \geq y \text{ else } g \lor q \lor \neg(x \geq 7) \lor g\]

defines (Boolean) \( b \) in terms of a conditional expression.

It is worthwhile to carry out by hand the following computations using expressions:

```plaintext
real rl, ra, rb;
integer n, i, j;

n := 5;
rl := n/(n + 15);
rb := n + 6/(6 * rl + 0.5);
i := n := n - 2;
j := rb - i;
ra := (j - 1) * rl * (rb - 4);
rl := ra + rb + n + i + j + 8 * rl;
rb := (rl - rb * n + j - ra) \lor (rb - j) + ra;
j := n := 1 + n + (j - 2);
i := n + ra;
```
The following example utilises Boolean expressions

```plaintext
real ra, rb;
integer ia;
Boolean ba, bb;
ra := 7.5;
ia := 5;
rb := 3 * ra - 2 * ia;
ba := rb > ia ia > ra;
ra := 2 * (ra - ia) - 1;
ba := ra > ia ba;
bb := (ba = rb > ia) ra < rb;
ba := (ba bb);
```

The following statements generate a sequence of values for $\text{SUM}$.

Find the first four of these values.

```plaintext
real p, q, SUM;
integer n;

n := 1;
p := 0.5;
SUM := 0;
q := 1;

loop:
  SUM := SUM + q/n;
  q := q * p;
  n := n + 1;
go_to loop;
```
5. Program Units

In a program there will be statements and declarations, Section 4.1.1 of Part I gives the important rules of how to join statements and declarations together to form a program. The main difficulty of this section is that of punctuation, particularly of when to write semi-colon and when not to. The difficulty is directly connected with the use of the delimiter **end**. As a guide the relevant rules may be restated as follows:

**PUNCTUATION RULE 1:** The first symbol following any statement (whether basic or not) must be one of the following three:

```
;
also
end
```

**PUNCTUATION RULE 2:** Any sequence `... end end end ...` must always be terminated by semi-colon or **also**.

Punctuation rule 1 follows directly from the syntactic rules governing statements (Sections 4.1.1, 4.5.1, 4.6.1, and 5.4.1). Punctuation rule 2 follows from observing that an **end**, whenever it occurs, is the last symbol of some statement, and then applying punctuation rule 1.

5.1 The concept of block structure.

Block structure is critical to 20[^4]L for it allows the efficient use of storage through overlay. Critical to its understanding is the concept of global and local relative to a block.
The concept local may be illustrated by an example of a program structure as follows:

1: begin real A, B, C;
3: begin real A, D;
4: Q: A := 2 * B + C;
5: D := 2 + B * A;
6: P: C := 2 * A - D;
7: go to P;
8: go to R;
9: end;
10: R: go to P;
11: end;

Here we have a larger block, from 1 to 11, containing as one statement a smaller block from 3 to 9. In the outer block we work with the identifiers A, B, and C, which are local to this block. In the statement at 2 a value is assigned to this A. The inner block introduces a new, local, A and a D. This A, then, has no relation to the A of the outer block, which is now screened. The variables B and C, on the other hand, are the same in both blocks. At 4 they are used to assign a value to the local A. This value is again used to assign a value to the local D at 5. These operations make no use whatsoever of the A of the outer block. At 6 a value is assigned to the non-local C, using the local A and D. Labels are automatically local. Thus the labels Q and P at 4 and 6 are only accessible from inside the inner block. The go to statement at 7 will therefore lead to the statement at 6. The go to statement at 8, on the other hand, will lead out of the inner block to 10 because the identifier R, being not declared in the inner block, will be non-local. The moment this passage out of the inner block occurs the local variables A and D
are completely lost. The go to statement at 10 will lead to 2 because the label P at 6 is local to the inner block and thus inaccessible from 10.

Using the above example follow the action of the following program and find the values of those variables which are defined at the label STOP.

```plaintext
begin real W, S, B, C;
1:  W := 8;
2:  S := 3;
3:  B := 2 * W - S;
4:  C := B - W;

begin real P, W;
5:  W := B - 2 * C;
6:  P := C + 2 - B;
7:  AA: W := P - 2 * W;
8:  C := C + 1;
9:  if W > 1 then go to AA;
10: S := W - P + S
end
11: W := W - C + S;
STOP:
end
```

The scope of a label comprises, so to speak, all those statements from which the label may be seen.

The concept of scope may be illustrated by the example given. The scopes of the different quantities are as follows:

- **Scope includes statements at**
  - A and P in outer block
  - B, C, and R
  - D, Q, and A and P in inner block

  - 2, 10
  - 2, 4, 5, 6, 7, 8, 10
  - 4, 5, 6, 7, 8
An important step in the planning of a program is the subdivision of
the process into parts which may conveniently be written as blocks or procedures.
In order to be able to do this the programmer must have a clear idea of the
properties of these units.

Blocks are useful for expressing such parts of the program which form a
closed process. A block is indispensable if in a process an array is needed
whose size depends on the results of previous calculations. Such an array must
then be local to a block. In addition any other quantity (simple variable,
label, switch, procedure) which is used only internally during the work of the
block, but which is not useful when the block is completed may be declared to
be local to the block. This is particularly useful when different blocks of a
program are written by different programmers. By using blocks the programmers
will only have to agree on the non-local identifiers of the blocks, while
inside each block the programmer is free to choose the identifiers of working
quantities.

Procedures have two other uses:

Abbreviation of small ad-hoc functions; and a form of communication
of closed processes between programs.

In particular they offer the option of recursive definitions of processes.

Any block may be converted into a procedure by adding a heading to it.
The heading will attach an identifier to the block and usually make some or
all of the non-local identifiers formal parameters. Where the block in
question is written specially for the program this conversion may be efficient
only if the mechanism of the block is used two or more times with different
non-local quantities, corresponding to two or more calls of the procedure,
since a call of a procedure is a more elaborate process than a simple entrance
into the corresponding block.
Frequently the formulae of a program may be shortened through the use of suitable function designators. As above this will be economical only if the corresponding ad-hoc procedure is used more than once during the program.

An example of the use of chained procedure calls is furnished by the following:

Compute \[ y = \int_{a}^{b} H(x) \, dx \]

where \[ H(x) = \int_{T_1(x)}^{T_2(x)} G(x,t) \, dt \]

and \[ G(x,t) = \sqrt{\sin^2(xt) + \cos^2((1-x)t)} \]

If the procedure \texttt{SIMPS} \( F, L, U, \lambda \)

Evaluates the integral of the function named \( F \) from \( L \) to \( U \) to a precision \( \lambda \) then the program would be as follows:

\begin{verbatim}
begin real a, b, y;
    real procedure H(p); real p;
begin real procedure R := p 2 + 3 x p;
    real procedure R := p 3 - 2 x p;
    real procedure G(n); real n;
begin G := sqrt((sin(p x n)) 2 + (cos((1 - p) x n)) 2) and G;
    H := SIMPS(G, R, S, \( \lambda \)) and H;
    real procedure SIMPS(F, L, U, )
    \{20^L code for a Simpson's rule program\}
    y := SIMPS(H, a, b, \( \lambda \)) and y.
\end{verbatim}
6. Declarations

Declarations are really quite straight forward except in the case of arrays, procedures, switches, and macros.

6.1. Arrays

The detailed explanations of Sections 4.2.3.1 - 4.2.3.3 are relevant in a case like:

```plaintext
real n; array A[1:10];
n := 2;
A[n + 1] := n := n + 2;
Section 4.2.3.1 of Part I produces:
Section 4.2.3.2 of Part I gives the value of the expression as 4.
Section 4.2.3.3 of Part I assigns 4 to n and A[3].
Following the code given below will clarify the consequences of non-
dynamic arrays
```

```plaintext
begin integer i, j; integer array A[1:3, 1:2], C[0:2];
j := i := 1;
i := A[3,2];
j := i - j;
A[1, -j-2]; 6 C[1-1] := 7;
STOP: end
```
Dynamic arrays are useful in matrix routines. Thus a declaration `real array X[1: n, 1: n]` allows the block in which the declaration is imbedded to use only storage necessary, say, for a linear equation routine.

6.2. Procedures

In the case of procedure declarations the following should be noted:

(i) The procedure declared is part of the block in which it is declared so its non-declared variables may 'communicate' with those in the block in which it is imbedded.

(ii) In the case of parameters not called by value — they are evaluated every time they are called. Whereas parameters called by value are evaluated once at the very beginning of each execution of the procedure.

(iii) Exits from a procedure involve a return to blocks and consequently may cause re-initialising of storage assignments. Thus in:

```plaintext
begin real x, j
array A[1: 5]
j := j + 1
begin real B
array A[1: j]
procedure M(p) real p
begin integer i
if x < 0 then A[i] := x else go to L

... end and ;
L : A[i] := x . end ;
```

...two different array elements are assigned the value x.
Switches are similarly tricky.

Thus, the kind of situation referred to by the remark of Section 5.3.5 of Part I may be illustrated by the following example:

```plaintext
begin switch W := tt, Q[n + 2] ;
switch Q := Q1, Q2, Q3 ;
... ...
A: begin real n ;
TT: go to W[2] ;
... ...
end block A

and

The go to statement at TT refers to W[2]. The designational expression for W[2] is Q[n+2]. Into this expression the variable n enters. Owing to the declaration real n in the head of block A the statement TT is outside the scope of the n of Q[n+2]. Consequently the go to statement is undefined.

As an exercise follow the action of the following statements and write a list of the labels to which the go to statements successively refer and find the final values of the variables:

```plaintext
begin integer n, s ;
switch S := SB, S2, S3, STOP ;
switch W := TW, S[n - s + y] ;
n := 7 ;
TW: go to S[n - 4] ;
SB: n := n - 1 ;
s := s + n ;
go to W[n - 2] ;
S3: n := n - 2 ;
s := n - 2 ;
go to W[n - s - 1]
STOPs:

end
```
Macros are particularly simple to understand. They are, after all, blocks of code which are substituted into the code wherever they are called.

After substitution their effect is precisely as though they had been put there originally by the programmer, one must be careful about the rules for substitution of parameters. They are:

(i) A formal macro parameter can only be an identifier. For all occurrences of that identifier in a macro definition, an instance of a macro call will cause that identifier to be replaced in a copy of the definition by the actual parameter. Thus the replacement of the formal parameter \( x \) by the string \( xx \) will not create a non-terminating string: \( xx \ldots x \ldots \) since the replacement is into a copy of the definition.

(ii) The actual parameter, \( \# \) delimited by a set of matching parentheses replaces the corresponding formal parameter with the outermost parentheses stripped.

Thus:

```latex
\begin{verbatim}
\textbf{macro} \quad S( A, B, C )
\begin{verbatim}
x := A + t
B
T | Q + ( C )
\end{verbatim}
\textbf{end}
\end{verbatim}
\end{verbatim}
```

A call occurring in the code:

\[
S( \sin( p * x), (S( \sin( p * x), y := \cos( p * x), (R + 2)), (((3)))))
\]

will cause

firstly \[
x := \sin( p * x) + t
\]
\[
S( \sin( p * x), y := \cos( p * x), (R + 2)
\]
\[
T | Q + (((3)))
\]

then:
then:

\[ x := \sin(p \times x) + t \]

finally:

\[ x := \sin(p \times x) + t \]
\[ y := \cos(p \times x) \]
\[ T := Q + (R + 2) \]
\[ T := Q + (((3))) \]

will be produced. Notice that parentheses serve three delimiting functions in this example.

7. **For Statements**

*For* statements are the most complex control statement in the language 20L. Finding the values assigned to the controlled variable in the following for statements and the final value of \( s \):

```plaintext
begin real p, q, r, s; integer k, m;
p := 1; q := 2; r := 3; s := 0;
for k := p + q, q - p, r * p - q do s := s + k;
for m := q step r until 7 * q do s := s - m;
for k := 2, s, 2 step 2 until 6 do s := s + 2 * k;
for m := s + 4, m + 2 while s < 0 do s := s - m;
for k := 1 step 1 until 5 do
for m := 3 step -1 until 0 do s := s + k + m;
```

will clarify the concept of the *for* statement considerably.

For statements are particularly useful for executing operations on vectors and matrices (described in 20L as arrays). A simple example is the addition of two vectors \( VA \) and \( VB \) to give a third \( VC \). This may be expressed as

```plaintext
array VA, VB, BC [1: n]; integer i;
for i := 1 step 1 until n do VC[i] := VA[i] + VB[i];
```

Note that the quantity \( n \) cannot be declared in the same block ahead as the arrays \( VA, VB, VC \) (cf. Section 5.2.4.2 of Part I).
8. Lists

Lists have been reported on elsewhere. Examples of their use are given in the article in "Communications of ACM" (April 1960).

Lists are represented by \( X[a, b] \) where:

- \( X \) is the name of the list;
- \( a \) is \( p, \not\!, \) or \( \forall \) meaning prefix, left, or right-site section, respectively;
- \( b \) is \( \not\! \) (current) or \( * \).

\( p \) has a 3-bit form \( g f e \):

\[
\begin{align*}
    e &= \{ \text{0 list} \} \\
    f &= \{ \text{0 direct} \} \\
    g &= \{ \text{0 non-terminal} \} \\
    \end{align*}
\]

Thus \( p = 101(2) \) or \( p = 5(10) \) means a direct terminal-list entry. The indirect means referral to a word having structure not sequenced by List sequencing modes. In particular, the referencing may be to arrays declared within 20 .

The empty list is represented by (, ) and the list:

\[
(, [2.1], , [2.2[1,2]], [3.2] , [1.2] )
\]

refers to 20 \( L \) objects \( X, \ Y, \) and \( 2[1,2] \) (arrays, variables, lists, procedures, etc.); while a itself is stored in the list. The arrow is omitted in the above representation.

As an example consider the procedure

\[
\text{procedure equal} \ (x, y, E) ; \text{list } x, y ; \text{Boolean } E \begin{align*}
\text{begin} & \text{ seqw}(x, \text{exit}) ; \text{ seqw}\ (y, \text{exit}) ; \ E := \text{true} ; \\
\text{1 : if } & \ x[p,\#] = y[p,\#] \\
\text{then begin} & \text{ seqw}(x, \text{exit}) ; \text{ seqw}(y, \text{exit}) ; \ E := \text{false} ; \\
\text{1 : else if } & \ x[p,\#] = 1 \\
\text{end.} &
\end{align*}
\]
then go to 1 and

E := false

exit; and equal.

which checks if two lists are equal.
Bibliography and Note


4. 20 stands for G=20 Carnegie Tech Algebraic Translator.


6. Several of the examples in this section are due to staff members of the Danish Regecentralen.
ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS
For Algebraic Language see p. 1-2 through 1-3
For Symbolic Coding and Data Processing see p. 1-9

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

def: Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt: Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text: Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words have been collected at the beginning.

The examples have been ignored in compiling the index.

* see: plus
- see: minus
× see multiply
/ see divide
↑ see: exponentiation
<, ≥, =, ≠ see: (relational operator)  
∧, ∨, A, T see: (logical operator)
. see: comma
, see: decimal point
10 see: ten
; see: colon
, see: semicolon
: see: colon equal
/ see: space
( ) see: parentheses
[ ] see: subscript bracket
" " see: string quote

I-1
(actual parameter), def 3.2.1, 4.7.1
(actual parameter list), def 3.2.1, 4.7.1
(actual parameter part), def 3.2.1, 4.7.1
(adding operator), def 3.3.1
alphabet, text 2.1
arithmetic text 3.3.6
(arithmetic expression), def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
(arithmetic operator), def 2, 3 text 3.3.4
array, text 3.1.4.1
array, synt 2.3, 5.2.2, 5.4.1
(array declaration), def 5.2.1 synt 5 text 5.2.3
(array identifier), def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
(array list), def 5.2.1
(array segment), def 5.2.1
(assignment statement), def 4.2.1 synt 4.1.1 text 1, 4.2.3
(basic statement), def 4.1.1 synt 4.5.1
(basic symbol), def 2
begin, synt 2.3, 4.1.1
(block), def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
(block head), def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3
(Boolean expression), def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1, text 3.4.3
(Boolean factor), def 3.4.1
(Boolean primary), def 3.4.1
(Boolean secondary), def 3.4.1

1 - 2
(Boolean term), def 3.0.1
(bound pair), def 5.2.1
(bound pair list), def 5.2.1
(bracket) def 2.3
(code), synt 5.0.1 text 4.7.8 5.0.6
colon, synt 2.3, 3.2.1, 4.0.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
colon equal, synt 2.3, 4.0.2.1, 4.6.1, 5.3.1
comma, synt 2.3, 3.1.1, 3.2.1, 4.0.1, 4.7.1, 5.0.1, 5.2.1, 5.3.1
comment, synt 2.3
comment convention, text 2.3
(compound statement), def 4.1.1 synt 4.5.1 text 1
(compound tail), def 4.1.1
(conditional statement), def 4.5.1 synt 4.1.1 text 4.5.3
(decimal fraction) def 2.5.1
(decimal number), def 2.5.1 text 2.5.3
decimal point, synt 2.3, 2.5.1
(declaration), def 5 synt 4.1.1 text 1.5 (complete section)
declarator), def 2.3
delimiter) def 2.3 synt 2
(designational expression), def 3.6.1 synt 3, 4.3.1, 5.3.1 text 3.6.3
digit), def 2.0.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.3
divide /, synt 2.3, 3.3.1 text 3.3.6.2
do, synt 2.3, 4.0.1
(dummy statement) def 4.1.1 synt 4.1.1 text 4.1.3
else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.0.1 text 4.5.3.2
(empty), def 1.1 synt 2.6.1, 3.2.1, 4.0.1, 4.7.1, 5.0.1

[Incomplete or unclear text, possibly due to image quality or other issues.]
end, synt 2.3, 4.1.1
enter, text 3.2.5

exponentiation, synt 2.3, 3.3.1 text 3.3.43
(exponent part), def 2.5.1 text 2.5.3
(expression), def 3 synt 3.2.1, 4.7.1 text 3 (complete section)

(factor), def 3.3.1
false, synt 2.2.2
for, synt 2.5, 4.6.1
(for clause), def 4.6.1 text 4.6.3
(for list), def 4.6.1 text 4.6.4

(for list element), def 4.6.1 text 4.6.1, 4.6.4.2, 4.6.4.3
(formal parameter), def 5.4.1 text 5.4.3
(formal parameter list), def 5.4.1
(formal parameter part), def 5.4.1
(for statement), def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 complete section

(function designator), def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.1

go to, synt 2.3, 4.1.1

(go to statement), def 4.3.1 synt 4.1.1 text 4.3.3

(identifier) def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
(identifier list), def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
(if clause), def 3.3.1, 5.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.5.2
(if statement), def 4.5.1 text 4.5.3.1

(implementation), def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
(integer), def 2.5.1 text 2.5.4

label, synt 2.3, 5.4.1
(label) def 3.5.1 synt 3.2.1, 4.5.1, 4.6.1 text 1 5.1.3
(left part), def 4.2.1
(left part list), def 4.2.1
(letter) def 2.1 synt 2.2.4.1, 2.5.1, 4.7.1
(letter string), def 3.2.1, 4.7.1
local, text 1 5.3
(local or own type), def 5.1.1 synt 5.2.1
(logical operator), def 2.3 synt 3.4.1 text 3.4.5
(logical value), def 2.2.2 synt 2.3.4.1
(lower bound) def 5.2.1 text 5.2.4
minus - , synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply X, synt 2.3, 3.3.1 text 3.3.4.1
(multiplying operator), def 3.3.1
Nonlocal, text 1 5.1.3
(number), def 2.5.1 text 2.5.3, 2.5.4
(open string), def 2.6.1
(operator) def 2.3
own, synt 2.3, 5.1.1 text 5.2.5
(parameter delimiter) def 3.2.1, 4.7.1 synt 5.4.1 text 1 5.7.7
parentheses ( ) synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2
plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
(primary), def 3.3.1
procedure, synt 2.3, 3.3.1
(procedure body), def 5.4.1
(procedure declaration) def 5.4.1 synt 5 text 5.4.3
(procedure heading), def 5.4.1 text 5.4.3
(procedure identifier) def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4
(procedure statement), def 4.7.1 synt 4.1.1 text 4.7.3
program, text 1
(proper string), def 2.6.1
quantity, text 2.7
real, synt 2.3, 5.1.1 text 5.1.3
(relation) def 3.4.1 text 3.4.5
(relational operator) def 2.3, 3.4.1
scope, text 2.7
semicolon; synt 2.3, 4.1.1, 5.4.1
(separator) def 2.3
(sequential operator) def 2.3
(simple arithmetic expression) def 3.3.1 text 3.3.3
(simple Boolean) def 3.4.1
(simple designational expression) def 3.6.1

I - 5
(simple variable), def 3.1.1 synt 5.1.1 text 2.4.3
space #, synt 2.3 text 2.3, 2.6.3
(specification part) def 5.4.1 text 5.4.5
(specificator) def 2.3
(specifier) def 5.4.1
standard function, text 3.2.4, 3.2.5
(statement) def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
statement bracket, see: begin end
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
(string) def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript bracket synt 2.3, 3.1.1, 3.5.1, 5.2.1
(subscript variable), def 3.1.1 text 3.1.4.1
(subscript expression) def 3.1.1 synt 3.6.1
(subscript list), def 3.1.1
successor, text 4
switch , synt 2.3, 5.3.1, 5.4.1
(switch declaration) def 5.3.1 synt 5 text 5.3.3
(switch designtor) def 3.6.1 text 3.6.3
(switch identifier) def 3.6.1 synt 3.2.1, 4.7.1, 5.3.1
(switch list) def 5.3.1
(term), def 3.3.1
ten synt 2.3, 2.5.1
then, synt 2.3, 3.3.1, 4.5.1
transfer function, text 3.2.5
true, synt 2.2.2
SYMBOLIC CODING AND LIST PROCESSING

address chain, def 3.8.1
address expression, def 3.7
address interpreter, def 4.8.1
code line, def 4.8.1
copy, text 3.2.4.2
elementary address, def 3.7.1
equivalent declaration, def 5.6.1
indirect address expression, synt 3.7.1
insert, text 3.2.4.2
instruction designator, def 4.8.1
library declaration, def 5.7.1
list, text 3.2.4.2
list procedures, text 3.2.4.2
list subscript expression, text 3.8
logical expression, def 3.5
macro declaration, def 5.5.1 text 5.5.3
macro designator, def 4.8.1
macro identifier, synt 4.8.1
macro statement, def 4.9.9
next, text 3.2.4.2
operation designator, def 4.8.1
pair list, def 5.6.1
seq, text 3.2.4.2
shift measure, def 3.5.1

SIGMA/SL 9671

I = 9
A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (1) an algebraic language, via ALGOL; (2) a symbolic machine-like coding language, like BBS; (3) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same system description, in Part II. There is only one processor, 20 P, described in Part III. The design of the processor is described in terms of actions taken at various phases of the translation process.

A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (1) an algebraic language, via ALGOL; (2) a symbolic machine-like coding language, like BBS; (3) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same system description, in Part II. There is only one processor, 20 P, described in Part III. The design of the processor is described in terms of actions taken at various phases of the translation process.

A single programming language is described, through it has from the programmer's point of view three forms. The forms are: (1) an algebraic language, via ALGOL; (2) a symbolic machine-like coding language, like BBS; (3) a symbolic list processing language, like Threaded Lists. The three are described as one language, 20 L, using the same system description, in Part I. There is only one processor, 20 P, described in Part III. The design of the processor is described in terms of actions taken at various phases of the translation process.