Peter Naur: NOTES ON ALGOL TRANSLATOR AND RUNNING SYSTEM

CHAPEL HILL July - December 1961

Contents

[redacted]

ALGOL TRANSLATOR

LOADING SYSTEM

ALGOL RUNNING SYSTEM

etc.

Computation Center
Univ. of North Carolina
Chapel Hill, N.C.

MAIN PRINCIPLES OF THE UNIVERSITY OF NORTH CAROLINA ALGOL 60 PROCESSOR.

Introduction.

The following notes provide the background and an explanation of the main solutions of the design of the ALGOL 60 translator for the UNIVAC 1105 at the University of North Carolina at Chapel Hill, North Carolina. These notes are written in December 1961 when the translator is still far from completed. Their main purpose is to serve as a general explanation of the preliminary notes on the "Algol running system" and the "Algol translator" which are also being written at this time.

Basic approach.

The starting point of the work is the decision to implement the complete ALGOL 60 language, without exceptions. Owing to the generality of the language this has not yet been done anywhere and has been approached in only a few places. However, the experience gained in those projects where such an approach has been made indicates that if the problem is attacked in the proper manner a complete ALGOL 60 processor is entirely feasible. Under these circumstances this approach would seem to be the obvious one to choose in a university institution where programming languages already are at the center of the interest.

The second major consideration is that of limiting the sheer bulk of the work of writing the compiler. This has dictated the following design decisions: (1) No attempt is made to provide facilities for the user to run ALGOL programs which cannot be held completely in the core memory of the machine. In other words programs which require more than 8192 words of store for instructions and variables cannot be handled by the basic ALGOL system. Work with such programs will require the use of prodedures written in machine code. (2) Optimisation of the efficiency [illegible] running program produced by the translator in the case of special simple constructions in the source program will only be attempted in those cases where the optimisation can be achieved with virtually no extra effort as far as the design of the translator is concerned. This means that the complete design will start with a consideration of the most general and complicated situations which are possible within the language. The principal effort will go into the design of solutions for these situations which are as efficient as possible. These solutions will to a considerable extent be chosen with the available machine characteristics in mind. These general solutions will be used throughout, even in cases where an analysis of the source program might reveal that they are unnecessarily general.

The third major consideration is the speed of compilation. Since it is anticipated that a major share of the programs to be compiled by the system will be short (student work) and will be used comparatively little for running it is considered basic that the translator will work very fast, particularly on short programs. This consideration is entirely compatible with the above mentioned decision to make use of the general solutions even when they are not strictly necessary.

The fourth consideration is checking. It has been considered essential that virtually all errors of syntax and consistency would be detected by the system and that extensive error print-outs would be produced automatically. This again has been found to be compatible with generality. Indeed, the uniform, general treatment of all occurrences of each feature of the language has greatly facilitated the design of the run-time error signaling.

Major divisions of work.

Previous experience has indicated that the above principles of design dictate the division of the project into two distinct parts:

1. The running system.
2. The translator.

Further that the logical order of dealing with these parts is the one indicated. In other words, the focus of the attention is the running system.

The reason for this insistence on the run-time events is that owing to the complexity of ALGOL 60 it is not at all clear how the control of the running program will be achieved in present-day computers. It is obvious, however, that the running program will make use of a number of permanent, internal, administrative, programs (or subroutines) for performing such tasks as procedure calls, storage allocation, etc. The generality of the final system will be critically dependent on the logic embedded in these administrative programs. Again the structure of the running program itself will of course reflect the conventions of the available administrative programs.

Now the proper work of the translator is to produce a running program as its output. This means that it cannot be designed completely before the exact form of the running program has been established. Since this again depends on the design of the running system it is clear that the design must start with this latter.

Main features of the running system.

The running system will be described under 5 subheadings as follows:

1. Description and notation.
2. Storage allocation.
3. Addressing.
4. Procedure entry.
5. Own variables.

Description and notation. Although the design of the running system in its basic features has been directly influenced by the characteristics of the UNIVAC 1105 the primary development and description of it has been made in a slightly adapted ALGOL notation. Some features of this notation are the following: The core store of the machine has been described in several ways, essentially reflecting the fact that the distinction in ALGOL between the program and the operands on which it works does not exist in present-day general purpose machines. Thus the instructions of the running program itself are represented as being the components of an array

array store [some lower bound : some upper bound]

This representation is used when an instruction or a parameter within the running program itself is used as an operand or changed. At run time the array store will only occupy a part of the core store of the machine, other parts being occupied by the programs of the administrative routines and the stack (see storage allocation below).

However, the instructions of the running program will alternatively be represented as labelled basic ALGOL statements, the absolute addresses being pictured as a set of unique labels. Control is transferred to an instruction of the running program by means of a go to statement to an element of a switch:

switch instruction := instruction 1, instruction 2, instruction 3, ... ;
Basically the task of the translator is to initialize the components of "store" and a few additional universal variables (such as "first free", see below) and to transfer control to the corresponding program through the statement: go to instruction [some lower bound] .

All variables of a program, including also some variable program parts, will be stored as the components of another array:

array stack [stack lower bound : stack upper bound]
This will occupy a part of the core store of the machine which is entirely separate from that occupied by "store". The components of the stack are initially undefined.

Storage allocation. The recursive procedures of ALGOL 60 dictate a completely dynamic storage allocation for all variables. It is well known that owing to the bracketing character of the ALGOL 60 block delimiters the logical way of arranging the variable storage is in the form of a stack (see Dijkstra, Numerische Mathematik 2 (1960) 312-318). The essential features of the stack, as this concept is used here, are the following:

1. The stack is a linearly arranged section of the store in which at any one time one end up to a certain dividing point has been reserved for specific variables, while the other end is free storage, ready to be used for any purpose.

2. The amount of storage reserved in the stack will in general vary during the run of the program. Additional reservations are always made from the current dividing point, using the first free locations. Likewise cancellations of reservations will only take place at the top of the reserved section. In other words, reservations and cancellations will treat the

stack like a push-down list.

3. References to the items held in the reserved part of the stack are not confined to the top element, but may be made to any element. The same holds for changes of the values of items.

Reservations will be made at the time of block entries, procedure calls, and references to formal parameters called by name. The amount of storage reserved at a specific action will be determined partly by the translator, ~~except~~ partly by the run-time administrative programs. A complete list of the reservations made at a procedure call is given in "Algol running system" page 2. Here the items FIXED FORMAT FIXED ORDER and VARIABLE FORMAT FIXED ORDER are reserved according to information collected by the translator. The remaining items are reserved according to information developed during the procedure call, at run time.

The parameters needed at block or procedure entry and the administrative programs performing the appropriate reservations are shown on pages 1 and 3 - 4 in "Algol running system". The most important universal parameter in these programs is the "first free". This defines the current top of the stack. In fact, the locations stack[first free], stack[first free + 1], stack[first free + 2], ... are the first free locations in the stack area, while the locations stack[first free - 1], stack[first free - 2], etc. are the last reserved locations.

Note that the reserved section includes temporaries. This corresponds to the fact that the translator has replaced all anonymous intermediate quantities by local internal ones. Note also that reservations are made for certain internal, administrative, quantities. These are the following:

stack reference. This indicates where in the stack the entries for the previous block entferdd into the stack are located.

current address modifier. See section on addressing below.

return address. This indicates the place in "store" to which control should be transferred when an exit from the present block is made.

REFERENCE. This indicates the place in "store" where the block parameters of the present block will be found (cf."reference" on page 1).

The exact form of most of the other items in the stack will be described in various pàaces of "Algol running system".

Addressing. Since no variables are allocated absolutely at translate time all references to variables of the program must be completed at run-time. Since the UNIVAC 1105 has no index registers, and since the use of subroutines would be intolerable because of the fast built-in floating point operations, the final addressing is established by a direct address modification technique. This works briefly as follows: Since all variables declared in the same block head will share fate as far as their existence is concerned the translator will be in a position to place all of them relatively to each other. In fact, the reservations VARIABLE FORMAT FIXED ORDER shown on page 2 of"Algol running system" show exactly the order in which the translator will place the variables belonging to one block. This means that in the running code all varibbles belonging to the same block head can be addressed completely, except for one common additive constant. This again means that the only addressing work left to the running system is the addition of the appropriate constant to all occurrences of addresses referring to variables of each particular block head at each entry into this block. This scheme requires the following information:

1. Associated with each block a variable indicating the current absolute addressing of the variables belonging to the block must be kept. This is the "current address modifier" placed at reference+7 (page 1).

2. Information about which addresses in the program belong to each block. This is supplied in the form of a series of bit words attached to each block (address modification code, see page 1 at reference+11+p). These bit words will have one bit for each address of the running program within the range of the block. Clearly this method assumes that the running program is stored in the same order as the original ALGOL program. Note also that where blocks are nested all addresses inside the inner blocks will appear in several address modification codes.

As to the efficiency of this method note first that in simple programs consisting only of one block with no procedures there is no loss of run time whatever since all addresses will be modified once at the start of the program, and never again. Also, since the administrative codes have been written so that unneccessary address modifications are omitted, programs which have no recursive procedure calls and no arrays with variable bounds and in which each procedure is only called in one procedure statement will settle down in a state where no more modifications are necessary as soon as all program parts have been entered once. Thus in these cases very little time will be wasted on address modifications at run time. The worst cases will be programs with recursive procedures and/or frequently varying array bounds in outer blocks and little or no looping in inner blocks. In these cases there can be no question of talking about efficiency, however, since there exist no alternative methods for handling these programs. It may be of interest to note, however, that since the modification of one single address may be expected to be accomplished by the running administration in less than the time of a floating point operation, the time needed for address modifications should never exceed that needed for arithmetic operations as long as real arithmetics is used. If the innermost block includes loops with operations on real variables the situation will be more favorable since one modification will give rise to many arithmetic operations.

Procedure entry. The implementation of procedure statements is based on well-established principles and techniques. The mathhing of a procedure statement with the corresponding procedure declarations takes place entirely at run time. References from inside the procedure body to the information supplied in the call will make use of linking information stored in a set of formal locations. These are initialized at each call of the procedure. Thus, essentially the task of the procedure entry administration is to take the information given in the actual parameters and the procedure heading and form the proper contents in the formal locations. The logic of this transformation process is described in the table of actions,"Algol running system" page 12, and the associated programs, pages 13 - 15.

Own variables. Own variables fall outside the range of the principles of storage allocation described above. Their behaviour when occurring within recursive procedures is still not finally settled within the language. Here they are treated as being similar to variables declared in the outermost block of the program. However, a special area of the store must be set aside for them.

The translator.

In accordance with the basic approach the methods used for translation have been chosen with a view to the speed of translation, and not with any consideration of the generality of the method used. For this reason all methods based on general symbol manipulation maneuvres, as well as those based on a mechanical use of the metasyntactic description of the language,have been rejected.

Like the running system the translator is described mostly in Algol, although with frequent use of tables describing the logic. In spite of this it is not intended to make use of any kind of bootstrapping techniques for transforming the translator code into machine code. Indeed, it is felt that by far the larger amount of work in writing a translator is the development of the logical principles and the statement of these principles in a complete manner. Once this has been done the transformation into any specific language for a machine will be a very minor matter. Bootstrapping only affects the transformation part of the job. Since bootstrapping implies a non-negligible amount of extra work in setting up intermediate languages and translators for them it is felt that the use of this technique might easily waste more effort than it saves.

For a discussion of the actual principles used, see "Algol translator", notes beginning 31. October 1961. Note that since these notes were written while the development work was actually proceding there are frequent corrections or modifications of statements made earlier in the later parts of the text.

Dependence on other work. New solutions.

Since the main stress in the project has been on arriving at a completed workable system no particular stress has been placed on obtaining original solutions. In fact, the solutions have been shosen from whatever suggestions were judged to be the best within the framework of the basic approach. The primary sources are the following:

1. The work of Dijkstra and Zonneveld of the Mathematical Center, Amsterdam, The Netherlands. We owe to this group the conviction that a complete system for ALGOL 60 is a practical proposition and the basic scanning method of pass 2 of the translator. References: E.W. Dijkstra, "Ein ALGOL-60-Ubersetzer fur die X1," Mathematik Technik Wirtschaft, Vol. 8, Vienna, Austria (1961), pp 54-56 and 115-119. E.W. Dijkstra, "Making a Translator for ALGOL 60," Automatic Programming Information Bulletin No. 7, APIC, College of Technology, Brighton, England (1961), pp 3-11. Also personal communications to Peter Naur in March 1960 and April 1961.

2. The work of the group at Regnecentralen, Copenhagen, Denmark: J. Jensen, P. Mondrup, and P. Naur. Also some work of B. Mayoh. The work in this group has influenced the implementation of the progedure call. Also the practical experience of this group in using a stack at run-time has been decisive. References: J. Jensen and P. Naur: "An Implementation of ALGOL 60 Procedures", BIT 1 (1961), 38-47. J. Jensen, P. Mondrup, and P. Naur, "A Storage Allocation Shheme for ALGOL 60," BIT 1 (1961), 89-102; Comm. ACM 4, 10 (October 1961) 441-445.

3. The work of the "Rump Group". The treatment of own arrays is essentially that of Ingerman. Ref: P. Z. Ingerman, "Dynamic Declarations", Comm. ACM 4,1 (January 1961) 59-60.

However, during the work some solutions were adopted which as far as we know have not been described elsewhere. The more interesting ones of these are the following:

1. The addressing scheme (page 7 of the Main Principles). The use of a direct address modification technique was suggested by John W. Carr, III.

2. The scanning logic of pass 2 ("Algol translator"), particularly the treatment of multiple delimiter meanings, as specified in the table of delimiter meanings (page 25) and the associated algorithm (page 36-37).

3. The mechanisms for collecting declarations ("Algol translator", pages 9 - 12, with additions page47-48).

History of project and members of the group.

The project was initiated by John W. Carr, III, Director of the Computation Center. The work described in these notes was accomplished during July to December 1961 during the stay of Peter Naur at Chapel Hill. In December the active members of the group were:

Peter Brown

Robert B. DesJardins

Peter Naur

Miriam Shoffner.

The running system was largely developed during a series of lectures held from July to August by P. Naur. Subsequently the remaining members of the group checked the system out manually by means of specific examples (programs including Ackermann's function and the General Problem Solver by Knuth and Merner and others). The programs for array declarations and the run-time alarm output were written by Miriam Shoffner. The part of the translator developed thus far was written as lecture notes by P. Naur from Oct. to Dec.

MAIN FEATURES OF THE TRANSLATION PASSES.

Tentatively it is assumed that the translation will include 4 separate scans of the source program, i.e. 4 passes. The main functions of each of these and some of the reasons for this division of work will first be described.

Pass 1: Reduction to the standard Algol form. This is a fairly simple process. It will convert the hardware form of the program to a uniform internal representation in which each Algol basic symbol has its unique character. This internal representation has 116 different characters: 52 letters, 10 digits, 2 logical values, 52 delimiters. In this process typographical features (space, change to new line, etc.) are removed. Algol comments are kept, however. (?) No checking is attempted. However, in order to determine when the end of the program has been reached a count of begins and ends must be included. This must take special account of strings enclosed in string quotes and comments.

Pass 2: Identifier matching, declaration collecting, build-up of constant table, delimiter checking. In this pass an identifier table is compiled. This will have one item for each distinct identifier in the program, with no regard to scopes. In the output from the pass every identifier will have been replaced by the number of the identifier in this table.

When scanning block heads the identifiers declared are compiled in a declaration stack. At the corresponding block end the declarations for this block are removed from the declaration stack into the output.

Literal constants (i.e. unsigned numbers, and strings) are compiled in a list of constants.

Pass 2, cont'd.

With the exception of arithmetic, relational, and logical operators, the consistency of the program with respect to the occurrence of all delimiters is checked. In addition, a number of delimiters, which do not appear in the Algol text, are added (so-called pseudobrackets are converted into proper brackets).

Pass 3: Analysis of simple expressions. This is a backward scan. Using the declarations assempled in pass 2 the meaning of any identifier at any place is now known. The analysis will include a complete check of the expressions and the conversion to machine instruction form.

Pass 4: Loading, internal references. In this pass the final absolute addressing will be made. All implicit references (for-statements, then, else, etc.) are worked out by the loader from the context. Explicit references (labels, procedure identifiers) are based on a simple symbolic address system.

Discussion. It has been considered basic that only simple scans would be made, i.e. that in each scan the text of the program would be taken in order from one end to the other. Secondly no restrictions on the order in which the program is written, other than those of Algol 60, have been imposed. Thirdly, a fairly complete checking has been aimed at.

These considerations force the use of a two-scan process. Indeed, no complete processing of expressions is possible in a one-scan process since the declarations will not in general be known. Pass 1 and pass 2 might very well be merged. It seems desirable to separate the machine dependent process of pass 1

Discussion of passes, cont'd.

and the machine independent pass 2. Again the division of work among passes 3 and 4 is not necessary. The advantage of the division is that no absolute addressing of the program, or even calculation of lengths of code becomes necessary until the loading stage.

The following is a more detailed discussion of various problems, beginning with pass 2.

IDENTIFIER HANDLING (pass 2).

The main advantages of the present method for handling identifiers are:

1. Identifiers are at once replaced by an internal representation.

2. The tables used are few and short.

3. The tables are relocatable.

4. No sorting is used.

5. It imposes no restrictions on the language: arbitrarily long identifiers can be handled.

The IDENTIFIER TABLE. This table is generated during pass 2. It will have one entry for each distinct identifier. Even if the same identifier is used with different meaning in different blocks the IDENTIFIER TABLE will have only one entry for it. Thus each identifier may be completely characterized by its number in the IDENTIFIER TABLE.

Before the start of translation of a program the identifiers

Identifier handling, cont'd.

of standard procedures are placed as the first items of the IDENTIFIER TABLE.

The IDENTIFIER TABLE has two parts: 1) the primary words, and 2) the secondary words.

Short identifiers, i.e. those having 5 characters or less, only use the primary words. The corresponding secondary word may be used for holding a part of another long identifier, as explained below.

Long identifiers use one primary word for the first 5 characters, and any number of secondary words, holding 4 characters each.

Assuming an alphabet of 52 letters and 10 digits each character occupies 6 bits. When dealing with groups of 4 or 5 characters no gain can be achieved by packing these characters as tightly as theoretically possible.

Structure of primary words: 3 parts:

1) 1 bit: 0 for short, 1 for long identifier.

2) 30 bits: For short identifiers: all characters.

" long " : first 3 and last 2 characters.

3) 5 bits: The number of characters modulo 32.

This structure has the following advantages: 1) It will make spurious coincidences of the primary words of long identifiers exceedingly rare. 2) It retains the first few characters, which is useful for error print-out during translation and the like.

Secondary words. If primary word no. n refers to a long identifier the first secondary word belonging to this identifier will also be no. n. Further secondary words of this identifier

Identifier handling, cont'd.

will have numbers less than n, making use of such positions in the secondary word table which correspond to short identifiers. The secondary words of the same identifier, as well as the free locations in the secondary word table, are linked together.

Structure of a <u>secondary word</u>:

1) 24 bits: 4 characters of the identifier.

2) 12 bits: Link to next secondary word of the identifier, if there are more. For the secondary word at position q the link is always less than q (might be negative).

Initially the link part of all secondary words with index ≦ 0 is set to indicate the immediately preceding word. As long identifiers are added all free words will remain linked together.

<u>Example</u> of identifier table: For simplicity assume that each word will only hold 2 characters (not 5 or 4). Further assume that the sequence of identifiers shown in the left column have been entered in the table, in the order shown. Then the situation will be as shown in the right hand columns:

Identifier:	Primary Mark	Primary Char.	Primary No.	Index	Secondary Char.	Secondary Link
a						
b1b2				-2		
c				-1		-2
d1d2d3d4				0	g5	-1
e	0	a	1	1	d4	0
f	1	b1	4	2	b2	1
g1g2g3g4g5	0	c	1	3	d3	1
h	1	d1	8	4	d2	3
i	0	e	1	5	g4	0
j1j2	0	f	1	6	g3	5
k	1	g1	10	7	g2	6
m1m2	0	h	1	8		-1
	0	i	1	9		8
	1	j1	4	10	j2	9
	0	k	1	11		9
	1	m1	4	12	m2	11
				13		11

PROGRAM FOR IDENTIFIER HANDLING.

The program will:

1. Read from input the letters and digits up to the next delimiter and form the proper internal representation.

2. Check whether the identifier is already in the identifier table, and if it is not insert it in the table.

3. ~~A~~ In any case exit with a value of the proper identifier number placed in i.

The exact structure of the primary word is takes as follows: (bit 35 is the most significant):

```
Bit  35:       more mark
Bits 34 to 30: number of characters modulo 32.
 -   29 -  24: 1st character
 -   23 -  18: 2nd      -
 -   17 - 12 : 3rd      -
 -   11 -   6: 4th      -
 -    5 -   0: 5th      -
```

Structure of secondary word:

```
Bits 35 to 30: 1st character
 -   29 -  24: 2nd      -
 -   23 -  18: 3rd      -
 -   17 -  12: 4th      -
 -   11 -   0: Link
```

```
array word list [1:  ]; identifier table [0:  ]; secondary [-q:  ];
comment Enter here with symbol = letter, showing that an identi-
fier is coming;
take identifier: n := 0; word counter := 0; short := true; word := 0;
                 for k := 1, 2, 3 do
                     begin
                       word := word + 64↑(5-k) × symbol;
                       n := n + 1; input(symbol);
```

Program for identifier handling, cont'd.

```
                    if class(symbol) = delimiter then
                        go to assemble 3
                  end reading of first 3 characters;
             lastbutone := symbol; last :=  dummy; input(symbol);
             if class(symbol)=delimiter then
                  begin
                    word := word + 64 × lastbutone;
                    go to assemble2
                  end;
             last := symbol; n := n + 1;
new word:    word counter := word counter + 1;
             word list[word counter] := 0;
             for k := 1, 2, 3, 4 do
                  begin
                    input(symbol);
                    if class(symbol)=delimiter then
                        go to assemble 1
                    word list[word counter] :=
                        word list[word counter]  +
                        64↑(6-k) × lastbutone;
                    lastbutone := last; n := n + 1;
                    last := symbol
                  end;
             go to new word;
assemble 1:  if k=1 then word counter := word counter - 1;
             word := word + 64 × last;
assemble 2:  word := word + 64↑2 × lastbutone;
```

Program for identifier handling, cont'd.

```
assemble 3:   word := word + (n ÷ n + 32 × 32) × 2↑30 +
                   (if n ≤ 5 then 0 else moremark);
              i := highest number;
search:       for I := identifiertable[i] while
                   I ≠ word ∧ i > 0 do i := i - 1;
              if i = 0 then
                   begin
                     m := i := highest number := highest number+1;
                     identifier table[i] := word;
                     for k := 1 step 1 until word counter do
                         begin
                           secondary[m]:= secondary[m] +
                              word list[k];
                           m=linkpart(secondary[m])
                         end;
                     secondary[highest number+1] := m
                   end i = 0
              else begin
                     m := i;
                     for k := 1 step 1 until word counter do
                         begin
                           if wordlist[k]≠
                              identifierpart(secondary[m])
                           then begin i:=i-1; go to search end;
                           m := linkpart(secondary[m])
                         end for k
                   end;
```

COLLECTING DECLARATIONS AND SPECIFICATIONS (pass 2).

The functions of this mechanism are:

1. To collect the declarations and specifications of the program in a form suitable
   - a. to be used during the analysis and checking during pass 3,
   - b. to form the information to be inserted at the end of blocks and procedures (appetite, etc.),
   - c. to form the full specifications of formal parameters, and
   - d. to construct the relative addresses of all variables within each block.
2. To check that no two identifiers are declared twice in the same block head.
3. To check that full specifications are available for formals.

<u>Structure of the DECLARATION STACK.</u> The above functions are executed with the aid of a declaration stack. This is a table operated in a stack like manner, holding the information supplied in declarations and specifications. Within the declaration stack all items of identical nature are linked together, forming a chain. Altogether 23 independent chains are maintained, one for each of the combinations marked by an x in the following table:

	No type	<u>real</u>	<u>integer</u>	<u>Boolean</u>
Simple variable, local		x	x	x
" " , own		x	x	x
Array, local		x	x	x
" , own		x	x	x
Switch	x			
Procedure	x			
<type> procedure, call only		x	x	x
" " , call and assign		x	x	x
Label	x			
Formal	x			
Stop	x			

Collecting declarations and specifications (pass 2). cont'd.

The following table shows the information held in the various kinds of items and a suggested bit assignment within a 36 bit word:

	Identifier Bits 35-26	Link 25-16	Other	
type	x	x		
array identifier	x			
array bounds		x	35-26:	number of identifiers
			15-0 :	- - subscripts
switch	x	x	15-0 :	- - expressions
procedure (no type)	x	x	15-0:	symbolic address
type procedure	x	x	15-0:	- -
label	x	x	15-0:	- -
formal	x	x	15-0:	specification and value
stop		x	15-0:	kind of stop: 1) Block 2) procedure (no type) 3) type procedure

Notes on the table:
Symbolic addresses are integers associated with procedure identifiers and labels, identifying each of these uniquely throughout the program. Each array segment will give rise to an entry having one word for each identifier plus one common word describing the bounds. Block begin will cause entry of a stop. Procedure identifier without type enters two words, one describing the identifier, followed by a stop. Type procedure identifiers cause entry of 3 words: 1. procedure identifier linked as call only, 2. stop, and 3. procedure identifier linked as call and assign.

Dynamics of the DECLARATION STACK. Each new declaration will cause the appropriate word to be entered and the corresponding link to be up-dated. Also a check that the identifier has not already been declared in the same block is carried out.

Formal parameters are entered in a similar manner. Specifications cause the appropriate information to be inserted in the

Collecting declarations and specifications (pass 2). cont'd.

- - - -
word already reserved for this formal parameter. This word must be available (check).

At block end all entries corresponding to the latest block are removed from the table. Since this must be done separately for each chain the declarations will be sorted according to their nature just by following each chain down to the latest stop. The information removed from the declaration stack may be transmitted to the output string of pass 2, as in the present description. This will assume that pass 3 is a backward scan. Alternatively it may be transferred to a special table on the drum. If this is done special account must be taken of the location of the declarations for each block in this table in such a manner that in the forward scan of pass 3 the proper declarations may be referenced at each block begin.

Example of the use of the DECLARATION STACK. Consider the contents of the declaration stack during the pass 2 of the following program:

```
begin real A, B;
      real procedure P(A, B); value A; real A; procedure B;
           begin real C, D;
                 E:
                 F:
           end of P;
      integer C, D;
      array E, G[1:2, 1:3];
      F:
end of program;
```

The following tables show the values of all relevant variables, including the identifier table and the declaration stack, both just before the scanning of"end of P" and before the scanning

Collecting declarations and specifications (pass 2) cont'd.

of "end of program".

	Initial	Just before end of P	Just before end of prog.
General variables:			
current top	1	12	10
next symbolic	1	4	5
End of chain variables:			
last real	-1	9	2
last integer	-1	-1	5
last real array	-1	-1	8
last real procedure to call	-1	3	3
last real procedure to call and assign	-1	5	-1
last label	-1	11	9
last formal	-1	7	-1
last stop	0	4	0

Identifier table just before end of program:

identifier number	1	2	3	4	5	6	7	8
identifier	A	B	P	C	D	E	F	G

Declaration stack:
Items 1 to 3 do not change between "end of P" and "end of program".

Item no.	Identifier number	link	Other	Identifier number	Link	Other
1	1 (=A)	-1				
2	2 (=B)	1				
3	3 (=P)	-1	symbolic 1			
	Just before end of P			Just before end of program		
4	(stop)	0	type proc.	4 (=C)	-1	
5	3 (=P)	-1	symbolic 1	5 (=D)	4	
6	1 (=A)	-1	real value	6 (=E)		
7	2 (=B)	6	procedure	8 (=G)		
8	4 (=C)	2			-1	2ident:2subsc.
9	5 (=D)	3		7 (=F)	-1	symbolic 4
10	6 (=E)	-1	symbolic 2			
11	7 (=F)	10	symbolic 3			

The algorithms for handling the declaration stack might be included at this stage. However, since they are intermixed with the scanning procedure of pass 2 this latter procedure will first be discussed.

THE SCANNING METHOD OF PASS 2.

The scanning method bescribed below is essentially based on the method used by E. W. Dijkstra (private communication to P. Naur, April 1961). The basic algorithm of this method is as follows:

1. Read the source program up to and including the next delimiter.
2. Perform the program for the interpretation of the new delimiter.
3. Go to point 1.

In this process it is convenient to exclude the ALGOL delimiters entering into literals (i.e. unsigned numbers and strings) from the class of delimiters. If this is done point 1 may cause reading of one out of 3 combinations: 1) Delimiter only, 2) Identifier and delimiter, and 3) Literal and delimiter. As an example of this method the following string

a[p + 5.83] := w;

would require 5 of the above cycles, the parts read in these cycles being:

a[p+   5.83]   :=   w;

Before developing the programs for the interpretation of each of the delimiters the question of syntactic checks during pass 2 will be discussed. Two aspects of this will be distinguished: microchecking and macrochecking.

The scanning method of pass 2, cont'd.

Microchecking. By microchekking will be understood the checking of the compatibility of adjecent symbols. When deriving the appropriate rules for this it is useful to introduce the class of operands, meaning the conjunction of the classes of(1) Identifiers not followed by (or [(2) Subscripted variables (3) Function designators (4) Expressions enclosed in parentheses and (5) Literals (i.e. unsigned numbers, strings, logical values). For each basic step of the scan it is now passible to define the value of the operand situation as being an integer describing what has been found immediately preceding the new delimiter, according to the following table:

Operand situation	Construction preceding delimiter
0	Not operand
1	Identifier
2	Subscripted variable
3	Function designator or procedure statement
4	Expression enclosed in parentheses
5	Literal

Basic principle for microchecking: Derive from the ALGOL syntax information on whether in the given situation the new delimiter is compatible with the operator situation.

The usefulness of this approach is due to the fact that for many combinations the situation is irrelevant in determining compatibility. For example the following are universally inadmissible combinations:

A <u>go to</u>   P[s] <u>for</u>   =/

The scanning method of pass 2, cont'd.

In fact, the following general rules hold:

The following 16 delimiters can never follow an operand:

Group A.

¬ go to if for comment begin own Boolean integer

real array switch procedure string label value

The following 25 (14) delimiters must always follow an operand:

Group B.

× / ÷ ↑ < ≤ = ≥ > ≠ ≡ ⊃ ∨ ∧ then do

⊗ : := step until while) [ ]

Of these [will only accept identifier and := will only accept identifier or subscripted variable.

The following 6 (7) delimiters may or may not follow an operand:

Group C. (because of commas following array segments)

+ − ; end else (,

The remaining 5 ALGOL 60 delimiters all belong to literals:

. 10 ␣ ‘ ’

These rules can be derived rigorously from the syntax of ALGOL 60. The ones of group A will be more or less obvious to anybody familiar with the language. Many of those of group B follow from the fact that any expression must end with an operand. The proof of this can be derived directly from the ALGOL 60 syntax. We must consider the 3 possible expressions separately.

The scanning method of pass 2, cont'd.

First arithmetic expressions. According to the section 3.3.1 of the ALGOL 60 report the last part of any arithmetic expression must be a simple arithmetic expression. The last part of this must be a term. The last part of this must be a factor. The last part of this must be a primary. But since a primary is an operand in the sense used here it follows that any arithmetic expression ends with an operand. The demonstration for the two other cases follows in a similar manner. Consequently any expression ends with an operand. In addition the proof shows that the same holds for <term>, <factor>, <implication>, <Boolean term>, <Boolean factor>, and <Boolean secondary>.

Now it is easy to verify from the ALGOL 60 syntax that each of the following delimiters, in any occurrence, will be preceded by one or other of the above mentioned constructions:

× / ÷ ↑ < ≤ = ≥ > ≠ ≡ ⊃ ∨ ∧ <u>then</u>

<u>do</u> <u>step</u> <u>until</u> <u>while</u>]

This proves the membership of group B for each of these delimiters. For the remaining members of group B quoted above:

, : :=) [

an individual investigation of the various uses of each of these symbols is necessary to prove the membership of group B. This may, however, be carried through in a straightforward manner.

The above rules are situation independent. They will serve to catch a number of errors by testing whether the class of the new delimiter is compatible with the operand situation. The further microchecking will make use of situation dependent parameter

The scanning method of pass 2, cont'd.

having the form of a one-dimensional Boolean array (a bit word) accomodating one truth value for each combination of operand and delimiter which has not already been checked for. Thus according to this scheme the action of each delimiter program (i.e. the program associated with each delimiter) will do 3 things: (1) Check that the delimiter is compatible with the current situation parameter. (2) Do whatever action is necessary for this delimiter. (3) Assign a new value to the situation parameter. As a simple illustration of this approach consider the scanning of the following piece of program:

begin integer a, b;

Scanning begin will set the situation parameter to admit a great variety of delimiters, in fact all those which may appear at the beginning of a declaration or a statement: go to if for comment begin own Boolean integer real array switch procedure ; end (: [:=

The appearance of integer immediately restricts the set of admissible successors to the following: , ; array procedure

The appearance of , restricts the successors even further: , ;

Finally the ; again opens up all the same possibilities as existed after begin.

It should be noted that this does not yet exhaust the possibilities of microchecking. Obviously this scheme would let such errors which arise from incorrectly writing one kind of operand at a place where only another is correct pass by. Example: begin integer 7, b; However, detection of such errors depends on the meaning of the delimiter, which again depends on the context. For this reason it is convenient to merge the microchecking and the mechanism for handling the multiple uses of delimiters into a single unified scheme. This will be described next.

Multiple meaning of delimiters. Practically all delimiters are used for more than one purpose and the particular meaning of a delimiter must be derived from the context. This will be handled by means of an extension of the basic scanning method in combination with the scheme for microchecking as follows:

The program associated with each delimiter will be split up into as many programs as there are meanings for this delimiter. Which particular program is to be used will be given in the current situation parameter. This then will now be an integer array with one element for each delimiter. The value given for a particular delimiter will at any time tell whether this delimiter is admissible, and if so, what meaning of it is pertinent.

The above scheme is sufficient for the complete scanning of ALGOL 60 declarations except where these contain expressions or statements. It is therefore possible to give complete information on the necessary delimiter programs. This is included below, in the following form: For each subprogram for a delimiter the particular meaning of this delimiter handled by the subprogram is briefly described. Then follows, for those delimiters which admit operands, the admissible operand situation (see table page 14). Finally the list of admissible successors.

own

own1 First symbol of declaration.
Successors: type2.

integer real Boolean

(19) type1 First symbol of declaration.
comma1 semicolon1 array1 procedure1

type2 Following **own**
Successors: comma1 semicolon1 array1

type3 In specification
Successors: comma 5 semicolon 3 array2 procedure2

array

array1 In declaration
Successors: comma2 leftbrace1

array2 In specification
Successors: comma5 semicolon3

Multiple meaning of delimiters (pass 2), cont'd.

state number

switch

(23) switch1 First symbol in declaration
Successors: colonequal1

(14) switch2 In specification.
Successors: comma5 semicolon3

procedure

(16) procedure1 In declaration
Successors: leftparenthesis1 semicolon2

(14) procedure2 In specification
Successors: comma5 semicolon3

value

value1 Following formal parameter part.
Successors: comma4 semicolon 6

string

string1 Specification
Successors: comma5 semicolon3

label

label1 Specification
Successors: comma5 semicolon3

;

(28) semicolon1 Following type declaration
Operand situation: 1
Successors: goto1 if1 for1 comment1 begin1 own1 integer1 real1 Boolean1 array1 switch1 procedure1 semicolon7 end1 leftparenthesis2 colon1 colonequal2 leftbracket2

(26) semicolon2 Following procedure <identifier>
Operand situation: 1
Sucessor: goto2 if2 for2 comment1 begin2 semicolon ~~8~~ 7 leftparenthesis3 colon2 leftbracket3 colonequal3 code1

(31) semicolon3 Following specification.
Operand situation: 1
Successors: comment1 integer3 real3 Boolean3 array2 switch2 procedure2 string1 label1 goto ~~3~~ 2 if ~~3~~ 2 begin ~~3~~ 2 semicolon8 leftparenthesis3 colon2 leftbracket3 colonequal3 for 2 code1

(30) semicolon4 Following formal parameter part
Operand situation: 0
Successors: comment1 integer3 real3 Boolean3 array2 switch2 procedure2 string1 label1 value1

(28) semicolon5 Following array segment
Operand situation: 0
Successors: Same as for semicolon1

(29) semicolon6 Following value part
Operand situation: 1
Successors: comment1 integer3 real3 Boolean3 array2 switch2 procedure2 string1 label1

(27) semicolon7 After ~~dummy~~ statement ~~or procedure call without parameters~~
Operand situation: 0 or 1
Successors: ~~goto1 if1 for1 comment1 begin1 semicolon7 end1 leftparenthesis2 colon1 leftbracket2 colonequal2~~
Depend on matching symbol in stack (see page 33-34)

Multiple meaning of delimiters (pass 2), cont'd.

- - - - - - - - -

semicolon8 Following procedure ~~identifier~~ heading %
Operand situation: 0 or 1
Successors: Same as for semicolon 1

semicolon9 In expression (finishing assignment or goto statement)
Operand situation: 1 to 5
Successors: Depends on the matching symbol in stack as follows:

goto1	Like semicolon	7
goto2 or goto3	- -	1
colonequal2	- -	7
colonequal3	- -	1

} for successors see page 33-34

(27) semicolon10 Following normal procedure call with parameters
Operand situation: 0
Sucessors: Same as for semicolon7

(28) semicolon11 Following end of procedure body
Operand situation: 0
Successors: Like semicolon1

begin

begin1 Statement
Successors: goto1 if1 for1 comment1 begin1 own1 integer1 real1 Boolean1 array1 switch1 procedure1 semicolon7 end1 leftparanthesis2 colon1 leftbracket2 colonequal2

begin2 Procedure body
Successors: Same as for begin 1

(17) comma1 Type declaration list
Operand situation: 1
Successors: comma1 semicolon1

comma2 Array declaration identifier list
Operand situation: 1
Successors: unchanged

comma3 Formal parameter list
Operand situation: 1
Successors: unchanged

comma4 Value list
Operand situation: 1
Successors: unchanged

comma5 Specification list
Operand situation: 1
Successors: comma5 semicolon3

comma6 Array segment
Operand situation: 0
Successor: comma2 leftbracket1

comma7 In expression
Operand situation: 1 to 5
Successors: not1 if1 plus1 minus1 semicolon9 end2 else2 leftparanthesis4 bioperator1 do1 colon3 step1 until1 while1 leftbracket4 rightbracket21 comma7 rightparenthesis2 then1
Note: This set of successors will be referred to as the begin of expression successors.

Multiple meaning of delimiters (pass 2), cont'd.

¬

notl — Anywhere
Successors: plusl minusl semicolon9 end2 else2 leftpar4 binaryoperatorl thenl dol leftbracket4 comma7 rightparenthesis2 (no. 7)

go to

gotol — Normal statement
Successors: Begin of expression (no. 2)

goto2 — Following procedure heading
Successors: Begin of expression (no. 2)

if

ifl — Normal statement
Successors: Begin of expression (no. 2)

if2 — Following procedure heading
Successors: Begin of expression (no. 2)

if3 — Begin of expression
Successors: Begin of expression (no. 2)

if4 — Following else
Successors: Begin of expression (no. 2)

for

forl — Normal statement
Successors: colonequal4 leftbracket5 (no. 22)

for2 — Following procedure heading
Successors: colonequal4 leftbracket5 (no. 22)

comment

commentl — Anywhere
Successors: Unchanged

\+ -

plusl minusl — Begin of arithmetic expression
Operand situation: 0 - 5
Successors: notl plus2 minus2 semicolon9 end2 else2 leftparenthesis4 binaryoperatorl thenl dol colon3 stepl untill whilel leftbracket4 rightbracketl comma7 rightparenthesis2 (no. 1)

plus2 minus2 — In expression.
Operand situation: 1 - 5
Successors: No. 1

end

endl — Following statement
Operand situation: 0 or 1

end2 — In expression
Operand situation: 1 - 5
Successors for endl or end 2 depend on matching symbol in stack as follows:

beginclear, beginblock: <any string..> endl semicolon7 elsel (no. 10).

beginbody: <anystring ..> semicolonll (special treatment)

Multiple meaning of delimiters (pass 2), cont'd.

---

else

else1 — In statement
Operand situation: 0 or 1

else2 — In expression
Operand situation: 1 - 5
Successors for else 1 and else2 depend on matching ~~if~~ then in stack as follows:

- ~~if~~ then statement: goto1 if4 for1 begin1 semicolon7 end1 leftparenthesis2 colon1 leftbracket2 colonequal2 (no. 9)
- ~~if~~ then expression: not1 if4 plus1 minus1 semicolon9 end2 else2 leftparenthesis4 binaryoperator1 do1 colon3 step1 until1 while1 leftbracket4 rightbracket1 comma7 rightparenthesis2 (no. 3)

(

leftparentheis1 — Procedure heading
Operand situation: 1
Successors: comma3 rightparenthesis1 (no. 21)

leftparenthesis2 — Procedure statement, normal
Operand situation: 1
Successors: Begin of expression (no. 2)

leftparenthesis3 — Procedure statement as body
Operand situation: 1
Successors: Begin of expresson (no. 2)

leftparenthesis4 — Subexpression or function designator
Operand situation: 0 or 1
Successors: Begin of expression (no. 2)

x / ÷ ↑

In expression (these form part of binaryoperator)
Operand situation: 1 - 5
Successors: not1 plus2 minus2 semicolon9 end2 else2 leftparenthesis4 binaryoperator1 then1 do1 colon3 step1 until1 while1 leftbracket4 rightbracket1 comma7 rightparenthesis2 (no. 1)

< ≤ = ≥ > ≠ ∧ ∨ ⊃ ≡

In expression (these are the remaining binary operators)
Operand situation: 1 - 5
Successors: plus1 minus1 not1 semicolon9 end2 else2 leftparenthesis4 binaryoperator1 then1 do1 leftbracket4 comma7 rightparenthesis2 (no. 7)

then

then1 — In expression
Operand situation: 1 - 5
Successors depend on matching if:

- ifstatement: goto1 for1 begin1 semicolon7 end1 else1 leftparenthesis2 colon4 leftbracket2 colonequal2 (no. 8)
- ifexpression: not1 plus1 minus1 else2 leftparenthesis4 binaryoperator1 leftbracket4 (no. 5)

Multiple meaning of delimiters (pass2), cont'd.

- - - - - - - - - - - -

do

dol — In expression
Operand situation: 1 - 5
Successors: gotol ifl forl beginl semicolon7 endl elsel leftparenthesis2 colonl leftbracket2 colonequal2 (no. 11)

:

colonl — Label of statement
Operand situation: 1
Successors: gotol ifl forl beginl semicolon7 endl elsel leftparenthesis2 colonl leftbracket2 colonequal2 (no. 11)

colon2 — Following procedure heading
Operand situation: 1
Successors: No. 11

colon3 — In expression
Operand situation: 1 - 5
Successors: Begin of expression (no. 2)

colon4 — Label of unconditional
Operand situation: 1
Successors: No. 8 (see thenl)

step until while

stepl
untill
whilel — In expression
Operand situation: 1 - 5
Successors: Begin of expression (no. 2)

]

rightbracketl — In expression
Operand situation: 1 - 5
Successors depend on matching [as follows:
- [array: : comma6 semicolon5 (no. 13)
- [left part : colonequal2 (direct check)
- [subscr.var. : No. 1 (see x / + ↑) with operand sit.=2
- [for-variable : colonequal4 (direct check)
- [left part or assignment expression: plus2 minus2 semicolon9 end2 else2 binaryoperatorl colonequal5 (no. 6) with operand sit.=2

[

leftbracketl	Array declaration	Operand situation: 1
leftbracket2	Assignm. statement	Successors: Begin of expression (no. 2)
leftbracket3	Following proc.head.	
leftbracket4	Subscr. var.	
leftbracket5	For-controlled var.	
leftbracket6	Continued assignment	

:=

colonequall — Switch declaration
Operand situation: 1
Successors: Begin of expression (no. 2)

colonequal2 — Normal assignment
Operand situation: 1 or 2
Successors: notl if3 plusl minusl semicolon9 end2 else2 leftparenthesis4 binaryoperatorl leftbracket6 colonequal5 (no. 4)

Multiple meaning of delimiters (pass 2), cont'd.

colonequal3 Following procedure heading
Operand situation: 1
Successors: No. 4

colonequal4 For clause
Operand situation: 1 or 2
Successors: Begin of expression (no. 2).

colonequal5 Continued assignment
Operand situation: 1 or 2. Successors: No. 4.

)
rightparenthesis1 Formal parameter part.
Operand situation: 1
Successors: <letter string>:(semicolon4 (special treatment)

rightparenthesis2 In expression
Operand situation: 1 - 5
Successors depend on matching (:
(proc. statement : <letter string>:(semicolon10 7 endl elsel (no. 20) with operand sit. = 0
(subexpression : No. 1 with operand sit. = 3
(func.desig. : <letter string>:(No. 1 with operand sit. = 4

code
codel Following procedure heading.
Operand situation: 0
Successors: Depends on code language.

The information on successors given above may be condensed into the following brief table, which lists the permissible successors in each of 31 different states. The numbers of these states have also been given above. In this table those delimiters which behave in an identical manner as far as their occurrence is concerned have been combined into a single entry. The groups which have been formed in this way are:

goto, covering go to, begin, and for

type, - integer, real, Boolean

string, - string and label

bi.op. - $\times$ / $\div$ $\uparrow$ $<$ $\leq$ $=$ $\geq$ $>$ $\neq$ $\wedge$ $\vee$ $\supset$ $\equiv$

step - step, until, while, and]

Multiple meaning of delimiters (pass 2), cont'd.

TABLE OF DELIMITER MEANINGS.

State number

Delimiter										10+										20+										30+	
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
¬ (not)	? (1)	1	1	1	1		?																								
go to begin for								1	1		1															2	1	1			2
if		3	4	3					4		1															2	1	1			2
comment																										1	1	1	1	1	1
own																												1			
integer real Boolean																									2			1	3	3	3
array															2			1	1									1	2	2	2
switch																												1	2	2	2
procedure															2				1									1	2	2	2
string label																													1	1	1
value																														1	
+ -	2	1	1	1	1	2	1				2															9					9
;	9	9	9	9		9	9	7	7	7	7	6	5	3	3	2	1	1	1	~~10~~ 7						7	7	7			7
end	2	2	2	2		2	2	1	1	1	1									1							1	1			
else	2	2	2	2	2	2	2	1		1	1									1											
(	4	4	4	4	4		4	2	2		2					1										3	2	2			3
binary operator	1	1	1	1	1	1	1																								
then	1	1					1																								
do	1	1	1				1																								
:	3	3	3					4	1		1															2	1	1			2
step until while]	1	1	1																												
[	4	4	4	6	4		4	2	2		2											5		1		3	2	2			3
,	7	7	7				7					4	6	5	5		1	1	1		3			2							
:=				5		5		2	2		2											4	1			3	2	2			3
)	2	2	2				2														1										
code																										1					1

Note that in this table two states have been omitted since they admit only one delimiter. These are: (1) Expecting semicolon4, resulting from rightparenthesis1, and (2) Expecting semicolon11, resulting from end2matching beginbody. In both cases the elimination of possible comments in the text will require a special treatment anyway.

The above 31 states correspond to well defined situations in the input string. The following is an approximate description of these situations and a list of the delimiters which may precede each of them:

Multiple meaning of delimiters (pass 2), cont'd.

MEANING OF STATES AND PRECEDING SYMBOLS.

1. In expression. + - × / ÷ ↑)]

2. Expecting expression. , go to if step until while (:= :

3. Expecting expression after else. else

4. Expecting left part or expression. :=

5. Expecting unconditional expression. then

6. Following subscripted variable which follows :=.]

7. In Boolean expression. ¬ < ≤ = ≥ > ≠ ∧ ∨ ⊃ ≡

8. Expecting unconditional statement. then :

9. Expecting statement after else. else

10. Following end of block or compound statement. end

11. Expecting statement, not comment. do :

12. In value part. value ,

13. Following array segment.]

14. In specification. array switch procedure string label

15. Following <type> as specifier. integer real Boolean

16. In procedure declaration heading. procedure

17. In type list. ,

18. Following own <type>. integer real Boolean

19. Following non-own type declarator. integer real Boolean

20. Following procedure statement.)

21. In formal parameter list. (,

22. Following for. for

23. Following switch as declarator. switch

24. Expecting array segment. array ,

25. Following own. own

Multiple meaning of delimiters (pass 2), cont'd.

26. Expecting procedure body. ;
27. Expecting statement or comment. ;
28. Expecting declaration or statement. ; begin
29. Expecting specification. ;
30. Expecting value part or specification. ;
31. Expecting procedure body or specification. ;

The information given in the table of delimiter meanings (page 25) may of course be handled in many different ways. The whole table may be stored in the machine. If it is packed as closely as possible in a binary machine it will need 31 items of 50 bits. Several cases lend themselves to a special treatment, however. Thus value is only possible in state 30, while the delimiters ¬ , comment, string, label, binary operator, step, until, while, and] may be checked more simply by testing the magnitude of the state number when these are chosen as above. If this is done the table only needs 31 items of 45 bits. It is thus clear that the storage requirements of the present mechanism are very modest.

It should be noticed at this stage that the above mechanism is designed to ignore any possible checking of types. The reason for this is that it is impossible to do a complete type checking because declarations for identifiers are generally not available at this stage. The complete type checking will be performed during pass 3. However, the above mechanism also does not check that delimiters on each side of expressions match properly. This is the task of the macrochecking which will be described next. This also will provide the mechanism for determining the kind of left parenthesis, bracket, end, etc. which matches a given right one. This has already been used in some of the above discussions on the successors of delimiters.

MACROCHECKING AND THE DELIMITER STACK.

For the purpose of checking and matching delimiters which permit arbitrary expressions to occur in between them a stack (push-down list) of delimiters will be used during the scanning of pass 2. This stack will at any time during the scan contain one entry for each delimiter having a left parenthesis character, which has not yet been matched by a corresponding right symbol, and which will admit arbitrary nesting of other brackets to appear before this matching will take place.

Each symbol in the delimiter stack will be one out of 28 different possibilities. In order to describe the meaning and dynamics of these symbols the life history of each of them will now be given, in terms of the following four kinds of events: (1) Creation. An item is said to be created when it is entered at the top of the stack, the other items being pushed down. (2) Changes. These convert the symbol in question to some other symbol. This happens only at the top of the stack, and all other items remain unchanged. (3) Recreation. This denotes that the symbol in question is formed from some other symbol. Only at top of stack. (4) Annihilation. This indicates that the symbol in question is removed from the top of the stack, the other items being popped up. Where in the following descriptions one or more of these events are omitted it means that no event of this kind will ever take place for that particular symbol.

1. beginclean.
Creation: beginl
Changes: To beginblock by ownl, typel, arrayl, switchl.
        To beginprocedure by procedurel.
Annihilation: endl or 2.

2. beginblock
Changes: To beginprocedure by prodedurel
Recreation: From beginclean by ownl, typel, arrayl, switchl.
        - beginprocedure by semicolon7, 9, 10, 11.
Annihilation: endl or 2.

Macrochecking and the delimiter stack, cont'd.

---

3. beginprocedure
Changes: To beginblock by semicolon7, 9, ~~10~~, 11.
Recreation: From beginclean or beginblock by procedure1

4. beginbody
Creation: begin2
Annihiliation: end1,2.

5. (call
Creation: leftparenthesis2,3
Annihilation: rightparenthesis2

6. (subexpression
Creation: leftparenthesis4 with operand situation = 0
Annihil.: rightparenthesis2

7. (function desig.
Creation: leftparenthesis4 with operand situation = 1
Annihil.: rightparenthesis2

8. [array,
Creation: leftbracket1
Changes: To [array: by colon3
Recreation: From [array: by comma7

9. [array:
Changes: To [array, by comma7
Recreation: From [array, by colon3
Annihil.: rightbracket1

10. [leftpart
Creation: leftbracket2, 3
Changes: To :=assign by rightbracket1

11. [left or assign
Creation: leftbracket6
Annihil.: rightbracket1

12. [subscr.var.
Creation: leftbracket4
Annihil.: rightbracket1

13. [for-var.
Creation: leftbracket5
Changes: To :=for by rightbracket1

14. :=switch
Creation: colonequal1
Annihil.: semicolon9

21. Nov. 1961.

Macrochecking and the delimiter stack, cont'd.
- - - - - - - - - - - - -

15. :=assign
Creation: colonequal2,3
Recreation: From [leftpart by rightbracket1
Annihil.: semicolon9, end2, else2

16. :=for
Creation: colonequal4
Changes: To do by do1, to step by step1, to while by while1
Recreation: From until and while by comma7

17. goto
Creation: goto1, 2
Annihil.: semicolon9, end2, else2

18. ifstatement
Creation: if1,2
Changes: To thenstatement by then1
Recreation: From elsestatement by if4

19. ifexpression
Creation: if3
Changes: To thenexpression by then1
Recreation: From elseexpression by if4

20. thenstatement
Creation: None
Changes: To elsestatement by else1, 2
Recreation: From ifstatement by then1
Annihil.: semicolon7, 9, ~~10~~, 11, end1,2

21. thenexpression
Changes: To elseexpression by else2
Recreation: From ifexpression by then1

22. elsestatement
Changes: To ifstatement by if4
Recreation: From thenstatement by else1,2
Annihil.: semicolon7, 9, ~~10~~, 11, end1, 2

23. elseexpression
Changes: To ifexpression by if4
Recreation: From thenexpression by else2
Annihil.: semicolon9, end2, do1, colon3, step1, until1, while1, rightbracket1, comma7, rightparenthesis2 then1

24. step
Changes: To until by until1
Recreation: From :=for by step1

Macrochecking and delimiter stack, cont'd.

25. until
Changes: To :=for by comma7, to do by dol
Recreation: From step by untill

26. while
Changes: To :=for by comma7, to do by dol
Recreation: From :=for by whilel

27. do
Recreation: From :=for, untill, and while by do
Annihil.: semicolon7, 9, 10, 11, endl, 2, elsel, 2

28. program
Creation: By initialization of translator
Annihiliation: semicolon 7,9

In describing the actions performed on the stack by the various delimiter programs it is convenient to divide the relevant delimiter programs into four groups, as follows:

Group 1: Programs entering a new item into the stack. These programs correspond to symbols having the character of left brackets or pseudobrackets. The groups has the following 20 members: beginl, 2, leftparenthesis2, 3,4, leftbracketl, 2, 3, 4, 5, 6, colonequall, 2, 3, 4, gotol, 2, ifl, 2, 3.

Group 2: Programs changing the top element of the stack, without any need for search or check. There are 8 members: if4, ownl, integerl, reall, Booleanl, arrayl, switchl, procedurel.

Group 3: Programs performing simple search and check. These programs represent delimiters which all terminate an expression, but not a statement. They will all perform an action having two steps: (1) Test whether the top of the stack is "elseexpression". If so annihilate this item. (2) Test the (possibly new) top of the stack and perform an appropriate action, according to the indications in the following table. In this table each delimiter

Macrochecking and delimiter stack, cont'd.

- - - - - - - - - - -

is represented by a column and the elements in the top of the stack which are of interest in this connection each have a line. A symbol at the crossing between the line for an element and the column for a program indicates that this element is acceptable for the program and will induce an action according to the following code:

L means: leave the element unchanged in the stack

A - : annihilate this element

Ch - : change the element.

TABLE OF SIMPLE SEARCH AND CHECK LOGIC.

In stack	comma7 ↓	rightbracket1 ↓	do ↓	step	while ↓	rightparenthesis2 ↓	then1 ↓	colon3	until
9. [array:	Ch	A							
10. [leftpart	L	(A)							
12. [subscr.var.	L	A							
13. [for var.	L	A							
11. [left or assign	L	A							
14. :=switch	L								
16. :=for	L		Ch	Ch	Ch				
25. until	Ch		Ch						
26. while	Ch		Ch						
5. (call	L					A			
7. (function desig.	L					A			
6. (subexpression						A			
18. ifstatement							Ch		
19. ifexpression							Ch		
8. [array,								Ch	
24. step									Ch

<u>Group 4: Programs performing a general search and check</u>. The programs in this group represent delimiters which terminate expressions and/or statements. Owing to the fact that arbitrarily deep nesting of for and if clauses is possible in ALGOL the search performed by the delimiter programs of this

Macrochecking and delimiter stack, cont'd.

- - - - - - - - - - - - - -

group may remove an arbitrary number of elements from the delimiter stack. The logic of this search is described in the following table. The meaning of this is as follows: At each stage of the search there is defined the value of an integer called the Search State. Using the current Search State and the symbol in the top of the stack as arguments, the table gives the action to be performed (L, A, And Ch having the same meaning as above) and the new value of the Search State. The letter e in the position of the new Search State indicated that the search has been completed. The integer following this e gives the new state number associated with this completion. Prior to the search the delimiter program will initialize the Search State as follows:

Delimiter program	Initial Search State
semicolon7	2
9	1
end1	5
2	4
else1	8
2	7

TABLE OF ACTIONS AND NEW SEARCH STATES.

Search State	semicolon			end			else		else (alt.)	
	1	2	3	4	5	6	7	8	8a	9a
In stack										
beginclean		L,e27	L,e27		A,e10	A,e10				L,e
beginblock		L,e27	L,e27		A,e10	A,e10				L,e
beginprocedure		Ch,e28	Ch,e28							L,e
beginbody		L,e27	L,e27		A,e(special)	A,e(special)				L,e
:=switch	A,e28									
:=assign	A,2			A,5			A,8			
goto	A,2			A,5			A,8			
thenstatement		A,3			A,6			Ch,e9	A,9a	
thenexpression							Ch,e3			
elsestatement		A,3			A,6					
elseexpression	A,1			A,4			A,7			
do		A,2	A,2		A,5	A,5		A,8	A,8a	A,8a
program		L,e11	L,e11							

Macrochecking and delimiter stack, cont'd.

In considering this table it should be noted that a certain simplification has already been made use of in Search States 1 and 4 and 7. In fact, these ~~two~~ three columns form the combination each of two columns, one of which admits elseexpression while the other does not. This combination of two columns into one clearly would be inadmissible if nothing were known about the items in the stack. However, the very detailed microchecking reflected in the table on page 25 will already have avoided that any illegal sequence of entries into the delimiter stack will ever have hade the chance of building up. For this reason, although the above table certainly reflects the way in which the actual searching will take place it is unnecessarily complex. As a matter of fact only three columns, one for each of the three delimiters, is necessary:

TABLE OF REDUCED SEARCH LOGIC.

In stack	Delimiter: semicolon	end	else
beginclean	L,e27	A,e10	
beginblock	L,e27	A,e10	
beginprocedure	Ch,e28		
begin body	L,e27	A,e(special)	
:=switch	A,e28		
:=assign	A,repeat	A,repeat	A,repeat
goto	A,repeat	A,repeat	A,repeat
thenstatement	A,repeat	A,repeat	Ch,e9
thenexpression			Ch,e3
elsestatement	A, repeat	A,repeat	
elseexpression	A,repeat	A,repeat	A,repeat
do	A,repeat	A,repeat	A,repeat
program	L,e11		

Here the word repeat means that the search should be continued, using the rules in the same column.

It should further be noted that this searching logic is based on a definite rule for the interpretation of the correspondence between <u>thens</u>

Macrochecking and delimiter stack, cont'd.

and following elses. This rule is that else will search back to the first then in the stack, but no further. Thus the association of then and else in the following example would be as indicated in the lines:

begin if .. then for ... do if ... then .. := ... else .. := .. ;

An alternative rule would be to have any else which does not find an expression then search back to the previous begin as indicated here:

begin if .. then for ... do if ... then .. := ... else .. := .. ;

The searching logic appropriate to this rule is given as else(alt.) in the table on page 33. It is obvious that the present treatment will take care of either rule with very little change.

The items in the stack will of course be represented by suitably chosen integers. The following assignment will make the integers relevant to each delimiter form an unbroken sequence:

1. thenexpression
2. thenstatement
3. goto
4. :=assign
5. do
6. beginclean
7. beginblock
8. beginbody
9. elsestatement
10. beginprocedure
11. program
12. :=switch
13. [array:
14. [leftpart
15. [subscr var.
16. [for
17. [left or assign
18. :=for
19. until
20. while
21. (call
22. (function desig.
23. (subexpression
24. ifexpression
25. ifstatement
26. [array,
27. step
28. elseexpression

The only exception is "elseexpression" which will be treated in a special way because of its unique character (in fact, it will be treated alike by all delimiters).

```
Initialize:  do:= blockno := next symbolic := 1;
             DELIMITER STACK [do] := "program";
             for j:=1 step 1 until 23 do  last item [j] := -1;
             last localized old := -1;
             state := H;
             highest number := current top := last top := 0;
clear type and next:  decl := re;
             type has appeared := false;
normal next:  ~~~~~~
```

THE CENTRAL READING PROGRAM FOR PASS 2.

If the logic developed in the preceding sections is included, the basic scanning process of page 13 will be given approximately by the following algorithm:

```
Initialize:     ~~k := ds := 1;~~ ds := blockno := next symbolic := 1;
                DELIMITER STACK[ds] := "program";
                state := 11;
                decl      :=  re   ;
                type has appeared := false;
normal next:    operand situation := 0;
normal next2:   input(symbol);
                comment The label take identifier is on page 6;
                if class(symbol) = letter then go to take identifier;
                if class(symbol) = numeric then go to take number;
                if symbol = left string quote then go to take string;
                if class(symbol) = logical value then
                    begin operand situation := 5;
                          i := if symbol=true then 1 else 2 ;
                          go to next after operand
                    end;
                if class(symbol)=B thengo to alarm;
                go to check occurrence;
                comment The following entry is used by rightbracket1 and
                rightparenthesis2 and after input of logical value;
next after operand: input(symbol);
```

The central reading program for pass 2, cont'd.

check delimiter following operand:

```
                if class(symbol) ≠ delimiter of class B or C then
                    go to alarm;
check occurrence:   case := DELIMITER MEANING [state, symbol] ;
                if case = 0 then go to alarm;
                go to pass2 program [symbol] ;
```

The classes of symbols used in this program are slight modifications of the classes of page 15:

Class name	Symbols belonging to class
numeric	<digit> . $_{10}$
B	× / ÷ ↑ < ≤ = ≥ > ≠ ≡ ⊃ ∨ ∧ **then** **do** : := **step** **until** **while**) [ ] ␣ '
B or C	× / ÷ ↑ < ≤ = ≥ > ≠ ≡ ⊃ ∨ ∧ **then** **do** : := **step** **until** **while**) [ ] + - ; **end** **else** (,

The array DELIMITER MEANING is given in the table on page 25. The switch pass2program has one element for each delimiter, i.e. 48 elements. The labels "take number" and "take string" lead to programs which perform actions similar to those of the identifier handling program on pages 6 to 8, i.e. as many input symbols as are necessary to complete the construction in question are processed. The output will be an item number in a constant table, assigned to i.

The following is a first sketch of the delimiter programs which will be entered through the "pass2program" switch, and which will handle declarations.

The central reading program for pass 2, cont'd.

First note that in consequence of the above logic the operand situation at the time of entry into the delimiter programs is known as follows:

Class characteristic In "B"	In "B or C"	Members	Known operand situation
Yes	Yes	× / + ↑ < ≤ = ≥ > ≠ ≡ ⊃ ∨ ∧ then do : := step until while) [ ]	≠ 0
Yes	No	␣ ›	Alarm
No	Yes	+ - ; end else (,	0 to 5
No	No	¬ go to if for comment begin own Boolean integer real array switch procedure string label value code	= 0

In some of the delimiter programs additional checking of the operand situation must be carried out. The required operand situation for each delimiter sub-program is given on pages 18 - 24.

In addition most of the delimiter programs must assign a new value to the state according to the information on pages 18 - 24. In the brief descriptions below the appropriate information on the new state and the operand situation has been stated in an abbreviated form, thus:

own1 (25) means that the successor state should be 25 while no operand checking is necessary,

comma2 (-,1) means that the state should remain unchanged, while the operand situation must be 1,

comma7 (2,1-5) means that the new state should be 2, while the operand situation must be 1, 2, 3, 4, or 5.

Unless otherwise stated all delimiter programs will return to "normal next" or "normal next2".

DELIMITER PROGRAMS FOR PASS 2.

own1 (25):SET BLOCK;

Boolean: decl := bool; go to type[case];

integer: decl:= int; go to type[case];

real: go to type[case];

type1 (19): SET BLOCK; type has appeared := true;

type2 (18): decl := decl + ownmark;

type3 (15): type has appeared := true;

array1 (24): SET BLOCK; decl := decl + arraymark; counter := 0;

array2 (14): decl := decl + arraymark;

switch1 (23): SET BLOCK; decl := switchmark;

switch2 (14): decl := switchmark;

procedure1 (16): SET BLOCK; goto procedure 2; comment but still state := 16;

procedure2 (14): decl := if type has appeared then decl + type procmark else procmark;

value1 (12): ;

string1 (14): decl := stringmark;

label1 (14): decl := labelmark;

semicolon1 (28,1): DECLARE TYPE; decl:= re; type has appeared := false;

semicolon2 (26,1): DECLARE PROCEDURE; decl:=re; type has appeared := false;

semicolon3 (31,1): SPECIFY; decl:= re; type has appeared := false;

semicolon4 (30,0): ; is executed by right parenthesis 1, page 42;

semicolon5 (28,0): ~~COMPLETE ARRAY SEGMENT;~~ decl:=re; type has appeared := false;

semicolon6 (29,1): SET VALUE;

semicolon7: Depends on search in stack (page 33 - 34).

semicolon8: (28, 0-1): FINISH HEADING; COMPLETE PROCEDURE DECLARATION;

Delimiter programs for pass 2, cont'd.

semicolon9: Depends on search in stack (pag. 33 - 34).

semicolon11 (28,0): COMPLETE PROCEDURE DECLARATION; is done by end, see page 46, line 12 from below

begin1 (28): Ent(beginclean); decl:= re;

begin2 (28): FINISH HEADING; Ent(beginbody);

comma1 (17,1): DECLARE TYPE;

comma2 (-,1): DECLARE ARRAY; counter:=counter+1;

comma3 (-,1): DECLARE FORMAL;

comma4 (-,1): SET VALUE;

comma5 (14,1): SPECIFY;

comma6 (24,0): ~~COMPLETE ARRAY SEGMENT;~~ counter:=0;

comma7 (2,1-5): Depends on simple search in stack (pag. 32)

not1 (7): Produces output

code1 (state suitable for scanning of machine language,0): FINISH HEADING;

goto1 (2): Ent(goto);

goto2 (2): FINISH HEADING; Ent(goto);

if1 (2): Ent(ifstatement);

if2 (2): FINISH HEADING; Ent(ifstatement);

if3 (2): Ent(ifexpression);

if4 (2): Ch(<u>if</u> delimiter stack[ds] = elsestatement <u>then</u> ifstatement <u>else</u> ifexpres-

for1 (22): ;

for2 (22): FINISH HEADING;

comment1 (-): ;

plus1 (1): Produces output
plus2 (1,1-5): Produces output
minus1 (1): Produces output
minus2 (1,1-5): Produces output

Delimiter programs for pass 2, cont'd.

- - - - - - - - - - - - - - - -

end1 : Depends on search in stack (pag. 33 - 34).

end2: Depends on search in stack (pag. 33 - 34).

else1: Depends on search in stack - - .

else2: - - - - - - - .

leftparenthesis1 (21,1): DECLARE PROCEDURE; decl:=~~pe~~ formal; type has appeared:= <u>false</u>;

leftparenthesis2 (2,1): Ent("(call");

leftparenthesis3 (2,1): FINISH HEADING; Ent("(call");

leftparenthesis4 (2,0-1): Ent(<u>if</u> operand situtation=0 <u>then</u> "(subexpr"<u>else</u>"(functF")

× / + ↑ (1): Produce output

< ≤ = ≥ > ≠ ≡ ⊃ ∧ ∨ (7): Produce output

then1: Depends on simple search in stack (pag. 32) { Ch(<u>if</u>DELIMITER STACK [ds] = ifst <u>then</u> thenst <u>else</u> thenex)

do1 (11): Performs simple search in stack (pag. 32); Ch(do);

colon1 (11,1): DECLARE LABEL;

colon2 (11,1): FINISH HEADING; DECLARE LABEL;

colon3 (2): Performs simple search in stack (pag.32); { subsc counter:=1+subscrcount; Ch("[array:") }

colon4 (8,1): DECLARE LABEL;

step1 (2): Simple search in stack (pag.32); Ch(step);

until1 (2): - - - - - - ; Ch(until);

while1 (2): - - - - - - ; Ch(while);

rightbracket1: Depends on simple search - ;

leftbracket1 (2,1): subsc counter:=0; Ent("[array,"); DECLARE ARRAY;

leftbracket2 (2,1): Ent("[leftpart");

leftbracket3 (2,1): FINISH HEADING; Ent("[leftpart");

leftbracket4 (2,1): Ent("[subscr");

leftbracket5 (2,1): Ent("[for");

Delimiter programs for pass2, cont'd.

- - - - - - - - - - - - - -

```
leftbracket6 (2,1): Ent("[leftor assign")

colonequal1 (2,1): DECLARE SWITCH; Ent(":=switch"); counter := 0;

colonequal2 (4,1-2): Ent(" :=assign");

colonequal3 (4,1): FINISH HEADING; Ent(":=assign");

colonequal4 (2,1-2): Ent(":=for")

colonequal5 (4,1-2): ;
                                              DECLARE FORMAL;
~~rightparenthesis1 (unique successor,1): Search for letter string or semicolon4~~

rightparenthesis2: Depends on simple search.

rightparenthesis1 (unchanged or 30.1): DECLARE FORMAL; if ¬
     LETTER STRING FOLLOWS then
     begin if symbol = semicolon then ALARM("semicolon missing");
       decl := re;
       type has appeared := false;
       state := 30;
     end;
```

Delimiter programs for pass 2, cont'd.

The programs which perform a simple search in the stack (see page 31 - 32) will now be described in detail. They all make use of procedures which will be described later. However the following procedure is used so frequently that a description is in place already here:

```
procedure TEST FOR ELSE EXPRESSION;
    begin top of stack := DELIMITER STACK[ds] ;
          if top of stack = else expression then
             begin Produce output; comment Output will be discussed later;
                   ds := ds - 1;
                   top of stack := DELIMITER STACK [ds]
             end
    end;
```

The following programs will also make use of the numerical equivalents of the elements in the stack given on page 35.

```
comma7:   TEST FOR ELSE EXPRESSION;  state := 2;
          begin switch comma7match := switchelement, arraybound, leftpart,
                       subscript, forvariable, left or assign, for element,
                       until, while, procedure call, function designator;
               go to comma7match[top of stack - 11] ;
               ALARM("impossible comma");
          switchelement: COMPLETE SWITCH ELEMENT; go to normal next; counter:=counter+1;
                                                               goto procedure call;
          arraybound:  DELIMITER STACK [ ds ]  :=  "[array,";
          procedure call:
          function designator: COMPLETE ACTUAL PARAMETER; go to normal next;
          leftpart:
          subscript:
          forvariable:
          left or assign;:  COMPLETE SUBSCRIPT; go to normal next;
          for element:  COMPLETE FOR ELEMENT; go to normal next;
          until:   COMPLETE UNTIL;
          reset for list: DELIMITER STACK [ds] := ":=for"; go to normal next;
          while:  COMPLETE WHILE; go to reset for list;
          end comma 7 switching;
rightbracket1:  TEST FOR ELSE EXPRESSION; ds := ds - 1; operand situation := 2;
          begin switch rightbracketmatch := arraybound, leftpart, subscript,
                        forvariable, left or assign;
               go to rightbracketmatch [top of stack - 12] ;
               ALARM("impossibe rightbracket");
          arraybound: COMPLETE ACTUAL PARAMETER;
                      COMPLETE ARRAY SEGMENT; state := 13; go to normal next;
```

Delimiter programs for pass 2, cont'd.

```
          leftpart: COMPLETE LEFT, SUBSCRIPT LIST;
                    input(symbol);
                    if symbol = colonequal then go to colonequal2;
                    ALARM("colonequal missing");
          subscript: COMPLETE SUBSCRIPT LIST; state := 1; go to normal next2; next after operand;
          forvariable: COMPLETE FOR SUBSCRIPT LIST;
                    input(symbol);
                    if symbol = colonequal then go to colonequal4;
                    ALARM("colonequal missing");
          left or assign: COMPLETE SUBSCRIPT LIST; state := 6; go to next after operand;
          end rightbracketl switching;
dol:      TEST FOR ELSE EXPRESSION;
          state := 11; DELIMITER STACK [ds] := "do"
          begin switch domatching := for variable, until, while;
              go to domatching[top of stack - 17];
              ALARM("impossible do");
          for variable: COMPLETE FOR ELEMENT; go to for clause finished;
          until: COMPLETE UNTIL; go to for clause finished;
          while: COMPLETE WHILE;
          for clause finished: COMPLETE FOR CLAUSE; go to normal next
          end do switching;
stepl:    TEST FOR ELSE EXPRESSION;
          if top of stack ≠ ":=for" then ALARM("impossible step");
          DELIMITER STACK [ds] := step; state := 2; go to normal next;
whilel:   TEST FOR ELSE EXPRESSION;
          if top of stack ≠ ":=for" then ALARM("impossible while");
          DELIMITER STACK [ds] := while; state := 2; go to normal next;
rightparenthesis2: TEST FOR ELSEEXPRESSION;
          begin switch rightparenthesismatching := call, function designator,
                        subexpression;
              go to rightparenthesismatching[top of stack - 20];
              ALARM("impossible right parenthesis");
          call: COMPLETE ACTUAL PARAMETER;
              if LETTER STRING FOLLOWS then begin state := 2; go to normal next end;
              COMPLETE PROCEDURE CALL; state := 20; operand situation := 0;
              go to check delimiter following operand;
          function designator: COMPLETE ACTUAL PARAMETER;
              if LETTER STRING FOLLOWS then begin state := 2; go to normal next end;
              COMPLETE FUNCTION DESIGNATOR; state := 1; operand situation := 4;
              go to check delimiter following operand;
          subexpression: COMPLETE SUBEXPRESSION; state := 1; operand situation := 3;
              go to next after operand;
          end rightparenthesis2 switching;
thenl:    TEST FOR ELSE EXPRESSION;
          COMPLETE IF CLAUSE;
          if top of stack = ifstatement then
              begin DELIMITER STACK [ds] := thenstatement;
                    state := 8
              end
```

Delimiter programs for pass 2, cont'd.

```
         else if top of stack = ifexpression then
            begin DELIMITER STACK [ds] := thenexpression; state := 5 end
         else ALARM("impossible then");
         go to normal next;
colon3:  TEST FOR ELSE EXPRESSION;
         COMPLETE ACTUAL PARAMETER; subsc counter := subsc counter +1;
         if top of stack = "[array, " then
            begin DELIMITER STACK [ds] :=  "[array:" ;
                  state := 2; go to normal next
            end
         else ALARM("impossible colon");
untill:  TEST FOR ELSE EXPRESSION;
         COMPLETE UNTIL;
         if top of stack = step then
            begin DELIMITER STACK[ds] := until;
                  state := 2; go to normal next
            end
         else ALARM("impossible until");
```

Next the programs performing a general search in the stack will be described. These are based on the logic described on page 34. They all make use of procedures which will be defined later. The following one should, however, be stated already here:

```
procedure TEST FOR PROCEDURE CALL;
      if operand situation = 1 then COMPLETE CALL WITHOUT PARAMETERS
      else if operand situation ≠ 0 then ALARM("impossible operand");

semicolon9: if operand situation = 0 then ALARM("impossible semicolon");
            TEST FOR ELSE EXPRESSION;
            go to semicolon search 3;
semicolon7: TEST FOR PROCEDURE CALL;
            go to semicolon search 2;
semicolon search 1: ds := ds - 1;
semicolon search 2: top of stack := DELIMITER STACK[ds];
semicolon search 3: begin switch semicolonmatch := thenstatement, goto, assign,
                          do, beginclean, beginblock, beginbody, elsestatement,
                          beginprocedure, program, switchdeclaration;
                          go to semicolonmatch[top of stack - 1];
                          ALARM("impossible semicolon");

thenstatement:
elsestatement:            COMPLETE CONDITIONAL STATEMENT;
                          go to semicolon search 1;
goto:                     COMPLETE GO TO; go to semicolon search 1;
assign:                   COMPLETE ASSIGN; go to semicolon search 1;
do:                       COMPLETE FOR; go to semicolon searchl;
```

Delimiter programs for pass 2, cont'd.

```
beginclean:
beginblock:
beginbody:                  state := 27; go to normal next;
beginprocedure:             COMPLETE PROCEDURE DECLARATION;
                            DELIMITER STACK[ds] := "begin block";
                            state := 28; go to normal next;
program:                    COMPLETE PROGRAM;
switch declaration:         COMPLETE ACTUAL PARAMETER;
                            COMPLETE PROCEDURE CALL;
                            ds := ds - 1; state := 28; go to normal next;
                       end semicolon switching;

end2:      if operand situation = 0 then ALARM("impossible end");
           TEST FOR ELSE EXPRESSION;
           go to eliminate comment;
end1:      TEST FOR PROCEDURE CALL;
           top of stack := DELIMITER STACK[ds];
eliminate comment: input(symbol);
           if symbol = begin then ALARM("impossible end comment"");
           if symbol ≠ end ∧ symbol ≠ semicolon∧symbol ≠ else then
                                         go to eliminate comment;

           go to end search 2;
end search 1:top of stack := DELIMITER STACK[ds];
end search 2: ds := ds - 1;
           begin switch endmatch := thenstatement, goto, assign, do, beginclean,
                          beginblock, beginbody, elsestatement;
                           go to endmatch[top of stack - 1];
                           ALARM("impossible end");
thenstatement:
elsestatement:             COMPLETE CONDITIONAL STATEMENT; go to end search 1;
goto:                      COMPLETE GO TO; go to end search 1;
assign:                    COMPLETE ASSIGN; go to end search 1;
do:                        COMPLETE FOR; go to end search 1;
beginblock:                COMPLETE BLOCK;
beginclean:                operand situation := 0;
                           state := 10; go to check occurrence;
beginbody:                if symbol ≠ semicolon then ALARM("semicolon missing");
                           COMPLETE PROCEDURE DECLARATION;
                           state := 28; go to normal next;
          end end switching;

else2:     if operand situation = 0 then ALARM("impossible else");
           TEST FOR ELSE EXPRESSION; go to else search 3;
else1:     TEST FOR PROCEDURE CALL; go to else search 2;
else search1: ds := ds - 1;
else search2: top of stack := DELIMITER STACK[ds];
else search3: begin switch elsematch := thenexpression, thenstatement, goto,
                            assign, do;
                           go to elsematch[top of stack];
                           ALARM("impossible else");
```

Delimiter programs for pass 2, cont'd.

```
thenexpression:       COMPLETE THEN EXPRESSION;
                      DELIMITER STACK[ds] := "elseexpression";
                      state := 3; go to normal next;
thenstatement:        COMPLETE THEN STATEMENT;
                      DELIMITER STACK[ds] := "else statement";
                      state := 9; go to normal next;
go to:                COMPLETE GO TO; go to else search 1;
assign:               COMPLETE ASSIGN; go to else search 1;
do:                   COMPLETE FOR; go to else search1;
          end else switching;
```

This essentially finishes the description of the scanning process for pass 2. It is now possible to return to the description of the algorithms for handling the declaration stack (see page 12). Before this is done it is however necessary to make an addition to the description of the declaration stack. This follows next.

THE CHECK LIST.

In addition to the identifier table (pag. 3 ff) and the declaration stack (pp. 9) a check list will be used. This will have one item for each item on the identifier table. Purposes:

1) To check against double declarations.
2) Facilitate specifications.

Each item in the check list has two parts:

1) The block number ~~where~~ belonging to the ~~current~~ quantity currently associated with the identifier described in the corresponding item of the identifier table.
2) The item number of the DECLARATION STACK where the declaration (if any) for the corresponding identifier is found.

If the identifier has not yet been declared the check list entry will be =.0.

When an identifier is redeclared in another block the entry in the check list

The check list, cont'd.

- - - - - - - - - - -

is put into the DECLARATION STACK. All such entries will form a new chain in the DECLARATION STACK, being connected with links. The structure of the corresponding machine words will be assumed to be as follows:

Entry in check list: Bits 35-26: DECLARATION STACK index
15- 0: block no.

When the entry is transferred to the DECLARATION STACK the link is added:

Link: Bits 25-16.

The chain of such entries starts at the point in the DECLARATION STACK indicated by the index;

last localized old.

CHAIN TERMINATIONS FOR THE DECLARATION STACK.

In the following programs the following values of constants are assumed:

arraymark = 3	int = 3	re = 2
blockcontant=24	labelmark= 1	stringmark = 12
Bool = 4	ownmark = 12	switchmark = 11
formal = 23	procmark = 13	typeprocmark = 6

The chain terminations for the chains in the DECLARATION STACK will be placed in a vector

<u>integer</u> <u>array</u> last item [1:23]

The subscripts of this vector corresponding to the different chains is given in the following table. The extra note in this table indicates whether in a procedure heading the kind of quantity indicated in the declaration is possible as a specification (S) and whether it is compatible with a value part quetation (V).

1 label	SV	8 real proc., call only	S(V)	14 own real
2 real	SV	9 int. - - -	-	15 own integer
3 integer	SV	10 Bool. - - -	-	16 own Boolean
4 Boolean	SV	11 switch	S	17 own real array
5 real array	SV	12 string	S	18 own integer array
6 int.array	SV	13 procedure	S	19 own Boolean array
7 Bool.array	SV			20-22 <type>proc.,call and ass.
				23 formal

Chain terminations for the declaration stack, cont'd.

Note that 12 string is never used as a chain termination. The numerical assignment is convenient because of checking. In the case of <type> procedures a value quotation is only possible if the corresponding actual parameter is a procedure without parameters. Under these circumstances the specification <type> procedure is unnecessarily restrictive and it is in fact converted to <type> in the programs below.

DECLARATION PROGRAMS.

Now many of the programs called on pages 39 to 47 can be defined:

```
procedure SET BLOCK;
    Comment This will be called at the beginning of each declaration. It will
    do the block entry work if this has not already been done;
    if DELIMITER STACK[ds] = beginclean then
        begin DELIMITER STACK[ds] := beginblock;
            DECLARATION STACK[current top] := last stop×2↑16+blockconstant×2↑9;
            last stop := current top;
            current top := current top + 1;
            block no := block no + 1
        end SET BLOCK;

procedure DECLARE(mark);
    comment This takes care of several different mechanisms which have had
    individual  identifiers in the programs above, as follows:
    Previous identifier:            Use:
    DECLARE TYPE                    DECLARE(0)
    DECLARE ARRAY                   DECLARE(0)
    DECLARE SWITCH                  DECLARE(no of elements)
    DECLARE FORMAL                  DECLARE(0)
    In addition the procedure is called by DECLARE PROCEDURE and DECLARE
    LABEL;
    begin if identifier is old then
        begin if blocknumberpart(check list[i]) = block no then
                                        ALARM("double declaration");
            DECLARATION STACK[current top]:= check list[i]+last localized old×2↑16
            last localized old := current top;
            current top := current top + 1;
        end stacking of previous meaning;
        check list[i] := block no + current top × 2↑26;
        DECLARATION STACK[current top]:=
            i×2↑26 + last item[decl]×2↑16 + mark;
        last item[decl] := current top;
        current top := current top + 1
    end DECLARE;
```

Declaration programs, cont'd.

```
procedure SET VALUE;
     begin integer k, item;
          k := declaration stack part(check list[i]);
          item := DECLARATION STACK[k];
          if k≤last stop∨other part(item) ≠ 0 then ALARM("impossible value quote");
          DECLARATION STACK[k] := item + value mark
     end SET VALUE;

procedure SPECIFY;
     begin integer k, item, note, specifier;
          k := declaration stack part(check list[i]);
          item := DECLARATION STACK[k];
          if k≤last stop then ALARM("impossible specification")
          else begin note := otherpart(item);
                    if note = valuemark then
                         begin if decl > 10 then
                                   ALARM("impossible combination of value and spec")
                              else specifier := if decl>7 then decl-6 else decl;
                              comment The previous statement converts type pro-
                              cedure into  type ;
                         end check of consistency of value
                    else if note = 0 then begin specifier := decl;


                    else ALARM("impossible or double specification");
                    DECLARATION STACK[k] := item + specifier × 2↑9
               end doing the specification
     end SPECIFY;

procedure DECLARE PROCEDURE;
     begin DECLARE(next symbolic);
          DECLARATION STACK[current top] := last stop × 2↑16 + decl;
          last stop := current top;
          current top := current top + 1;
          block no := block no + 1;
          if type has appeared then
               begin DECLARATION STACK[current top] :=
                         i×2↑26 + last item[decl+12]×2↑16 + next symbolic;
                    last item[decl + 12] := current top;
                    current top := current top + 1;
                    check list[i] := check list[i] + 2 × 2↑26  + 1
               end entering second entry;
          output(                          );
          last symbolic := last symbolic + 1;
          print(first 3 characters(primary word[i])
     end DECLARE PROCEDURE;
```

Declaration programs, cont'd.

```
procedure DECLARE LABEL;
    begin decl := 1;
          DECLARE(next symbolic);
          Output(                        );
          next symbolic := next symbolic + 1;
          print(first 3 characters(primary word[i]);
    end DECLARE LABEL;

Boolean procedure LETTER STRING FOLLOWS;
    begin Boolean read on;
          input(symbol);
          LETTER STRING FOLLOWS := read on := class(symbol) = letter;
          if ¬ read on then go to finished;
    repeat: input(symbol);
          if class(symbol) = letter then go to repeat;
          if symbol ≠ colon then ALARM("impossible parameter delimiter");
          input(symbol);
          if symbol ≠ leftparenthesis then ALARM("impossible parameter delimiter");
    finished:
    end LETTER STRING FOLLOWS;

procedure FINISH HEADING;
    begin integer k, specifier, item; integer index;
          k := 0;
          index := last item[23]; comment This is position of last formal;
          specifier := "no more parameters";
    repeat: if index > last stop then
              begin item := DECLARATION STACK[index];
                    DECLARATION STACK[current top + k] :=
                        first 3 characters(primary word[identifier part(item)])
                        + specifier;
                    specifier := otherpart(item);
                    if specifier = 0 then ALARM("specification missing");
                    index := linkpart(item);
                    k := k + 1;
                    go to repeat
              end;
    output(first 3 characters(primary word[identifier part(
                                   DECLARATION STACK[last stop - 1])])
          + specifier);
    for j := k-1 step -1 until 0 do output(DECLARATION STACK[j + current top])
    end FINISH HEADING;

procedure Ent(s); integer s; begin ds := ds + 1; DELIMITER STACK[ds] := s end;

procedure Ch(s); integer s; DELIMITER STACK[ds] := s;
```

Algol translator.
21. Dec. 1961.

CORRECTIONS AND ADDITIONS.

Page 1.
In pass 1 the "change to new line" should be kept as an extra character in the output, in order to facilitate ALARM output (see page 8b).
In ALGOL comment only the symbol **comment** itself need to be kept, unless it is desired to output comments during ALARM output. I.e. all symbols following **comment** up to and including ; may be deleted. This is assumed in the program for comment on page 40. Comments following **end** need not be removed by pass 1, however (see the program at "eliminate comment" on page 46).
Page 6.
In line 10 read:

3. In any case exit with a value of the proper identifier number placed in i and the Boolean variable "identifier is old" set to **true** if the identifier did not have to be added to the identifier table, otherwide to **false**.

In 5th line from bottom read:
take identifier: n := 1; . . .
In the last line delete "n := n + 1;"
Page 7.
In line 5 insert extra line to read:

```
                go to assemble 3;
            n := n + 1
        end reading of first 3 characters;
```

In line 8 change to read:

```
    if class(symbol)=delimiter then go to assemble2;
```

and remove the following 4 lines.
Page 8.
Change page number to 8a.
In line 5 underline "else"
In line 19 insert two statements to read:

```
        secondary[highest number + 1]:= m;
        check list[i] := 0;
        identifier is old := false
    end i = 0
```

In the two last lines insert to read:

```
                end for k;
            identifier is old := true
        end;
    operand situation := 1;
    go to check delimiter following operand; comment On page 37;
```

Page 9.
In second line above table read: Altogether 24 independent chains . . .
Add extra line in table at bottom of page:
Entries from the checklist x
Page 10.
In table, line for array identifier, put x in column for Link.
Add item to table:
check list entry, with x for link and Other: 35-26: position in DECLARATION STACK, 15-0: block number.
Add note: The check list entries are items of the check list (page 47) which have temporarily been removed because the corresponding identifier has assumed a local significance.

Insert the attached page 8b between 8a and 9.

Page 12.
The example does not include the items belonging to the check list. Also the end of chain variables should be changed to be components of the array "last item" (the algorithms for working with the DECLARATIONS STACK are found on pages 49 -51).

Page 15.
Move , from group B into group C (because of commas following array segments). Change the numbers of members of the groups accordingly.

Page 18.
Add the number of the successors as on pages 21 ff, as given on pages 39 ff.

Page 19.
Add the number of the successors as on page 18.

```
semicolon3, correct successors  as follows: goto2  if2  begin2
                add       "        "    "   : for2  codel
semicolon7, Read:
             After statement
             Operand situation: 0 or 1.
             Successors: Depend on matching symbol in stack (see page 33-34).
semicolon9, for successors see page 33-34.
Semicolon10: Delete completely.
comma7, correct successors: if3    rightbracketl
          add       "      : thenl
```

Page 24.
rightparenthesis2, in successors in case of "(proc. statement", correct to:
semicolon7

Page 25.
In table in line for ; change as follows:
for state 20: 10 to 7, for state 26: 7 to 8, for state 31: 7 to 8.

Page 28.
Line 11: read . . . out of 28 different . . .

Page 30:
17. goto, Annihil.: read: semicolon 9, end2, else2
In 20. thenstatement and 22. elsestatement delete semicolon 10
In 23. elseexpression, Annihil., add thenl

Page 36. Change beginning of algorithm to read:

```
Initialize:     ds := block no := next symbolic := 1;
                DELIMITER STACK[ds] := "program";
                for j := 1 step 1 until 23 do last item[j] := -1;
                last localized old := -1;
                state := 11;
                highest number := current top := last stop := 0;
clear type and next:
                decl := re;
                type has appeared := false;
normal next:    . . .
```

Page 39.
In arrayl add: counter := 0;
In switchl add: decl := switchmark;
In procedurel add: go to procedure2; comment But still state := 16;

Algol translator
21. Dec. 1961

CORRECTIONS AND ADDITIONS #3.

Page 39, cont'd.
In procedure2 change to read: . . then decl+typeprocmark else . . .
In semicolon4 add: Is executed by rightparenthesis1, page 42.
In semicolon 5 delete: COMPLETE ARRAY SEGMENT;
Page 40.
In semicolon11 add: Is done by end, see page 46, line 12 from below.
In begin1 add: decl := re;
In comma2 add: counter := counter + 1;
In comma6 change to read:
comma6 (24,0): counter := 0;
Page 41.
leftparenthesis1, read:
. . . DECLARE PROCEDURE; decl := formal;
colonequal1, add: counter := 0;
rightparenthesis1, read:

```
rightparenthesis1 (unchanged or 30,1): DECLARE FORMAL; if ¬ LETTER STRING FOLLOWS then
        begin if symbol = semicolon then ALARM("semicolon missing");
              decl := re;
              type has appeared := false;
              state := 30
        end;
```

Page 43.
6 lines following comma7, read:
switchelement: counter := counter + 1; go to procedure call;
Page 44.
In line 7 read:
subscript: COMPLETE SUBSCRIPT LIST; state := 1; go to next after operand;
Page 45.
The line following colon3, read:
COMPLETE ACTUAL PARAMETER; subsc counter := subsc counter + 1;
Page 49.
In comment to procedure DECLARE, in same line as DECLARE TYPE read: DECLARE(0)
Page 50.
In procedure SPECIFY remove begin to read:
else if note = 0 then specifier := decl;
In 3rd line of procedure DECLARE PROCEDURE add factor to read:
. . . last stop × 2↑16 + decl × 2↑9;
Change 3rd line from below to read:
next symbolic := next symbolic + 1;

PAGES WHICH HAVE BEEN REVISED:

8b (new page), 31, 45, 46, 51

The loading system of the Algol system will have various tasks to perform:

1) Build-up of address modification codes
2) expansion of macros from pass 3
   a) Basic symbols such as "beginblock" etc. will be expanded into
      "calladdress:= ;
      :<u>go</u> <u>to</u> block entry:", etc.
   b) Macros procuded by the analysis of expressions will be expanded into 1105 instructions.
3) setting up of all forward reference, e.g., REFERENCE, designational expression in <u>go</u> <u>to</u> statements, linkage of <u>if</u>..<u>then</u>..<u>else</u> etc.
4) address modification of the outer block
5) Bringing running system into core
6) Addition and address modification of standard procedures to the program.

It is hoped that the length of the loader will be less than or equal to the length of the running system so that there will be no imaging of the loader and/or parts of the program. At this point in the development of the system, it is assumed that there will be no such imaging and subsequent coding is written under this assumption.

BUILD-UP OF ADDRESS MODIFICATION CODE IN LOAD PROGRAM

The partial address modification codes are built-up in a single code stack. This stack is divided into sections corresponding to the ~~sections~~ blocks of the program where respectively 1, 2, 3, ..., n independent codes are in the process of being built-up. where n is the number of (nested) blocks.

The current state of the code stack is described by five parameters:

1. DEPTH: The number of codes being built-up = the number of unclosed sections. (Initial value = 0)
2. TOP STOP: The address of the last stop code. (Initial value = 0)
3. LINES: The number of lines (complete or incomplete) for each block in the top level. (Initial value = 0)
4. BITNUMBER: The number of the last bit in the last line in the top level which has just been filled. (Initial value = 0)
5. INDEX: If one line or less is required for the code word within a section INDEX = TOP STOP. Otherwise INDEX is the address of the last complete line in the section. (Initial value = address of the first location in the code stack which is to be used for storing the address modification code words while they are being built-up)

STOP Codes. Each section ends with a stop code which contains parameters 2, 3, and 4 (above) for the section. The stop code is generated each time a new section is opened.

EXAMPLE: When the loading has proceeded as follows:

```
A: begin
        B: begin

           end;
        C: begin
                D: begin
```

the code stack will have the following structure:

Location	Block	Line No.	Last bit in line	Code Stack
6				Stop (0,0,0)
7	A	1	35	
8	A	2	17	
9				Stop (6,17,2)
10	A	1	35	
11	C	1	35	
12	A	2	35	
13	C	2	35	
14	A	3	9	
15	C	3	9	
16				Stop (9,9,3)
17	A	1	31	
18	C	1	31	
19	D	1	31	

Note that no trace of B has been left in the code stack. On meeting <u>end</u> the code a) takes out the code for the last block b) removes the last stop and c) collapses the remaining words. (see CLOSE BLOCK procedure)

Procedures. The procedures used in the address modification portion of the load program are:

1. Initalize load program
2. Open Section
3. Close Section
4. Mark (n)

<u>Procedure</u> Initialize load program;

<u>Comment</u> Is used to set parameters in the load program to the proper initial values. INDEX, which specifies the first free location in the code stack, is assumed set;

<u>Begin</u>

depth := top stop := lines := bitnumber := 0,

<u>end</u>,

```
procedure  open section;

    comment  is used when block begin, procedure body begin
    or parameter expression is encountered;

    begin

    depth := depth + 1;

    code stack [index + depth] := combination (top stop, lines,
    bitnumber);

    top stop := index + depth;

    lines := 0;

    bitnumber := 35;

  end;
```

```
procedure mark (n); value n; integer n;

   comment Marks the next bit in the n'th block and advances
   to next. For n=0 only advance;

   begin if bitnumber = 35 then

            begin bitnumber := 0;

                  index := index + depth;

                  for p:= 1 step 1 until depth do

                     code stack [index+p] := 0;

                  lines := lines + 1;
            end

         else bitnumber := bitnumber + 1;

         if n ≠ 0 then

            code stack [index + n] := code stack [index+n] +
            2↑(35-bitnumber);

   end;
```

```
procedure close section;

    comment Is used to load the completed address modifications
    code into the running code and clean up the code stack;

    begin
      integer k;

      for k:= depth step depth until lines x depth do

        compile (code stack [top stop + k] );

      old bitnumber := bitpart (code stack [top stop] );

      m := depth := depth - 1;

      if depth = 0 then go to loading finished;

      old top stop := stoppart (code stack [top stop] );

      u := old bitnumber + 1;

      v := 36 - u;

      old lines := linepart (code stack [top stop] );

      for k := top stop - depth - 1 step depth until
      topstop - 1 + depth x (lines - 2) do
           begin
             move := k < top stop-1 + depth x (lines-2) v
             bitnumber + old bitnumber > 34;

             m := m+1;

             for s := 1 step 1 until depth do
                 begin
                   codestack [k+s] := codestack [k+s] +
                   2 ↑ (-u) x codestack [k+s+m] ;

                   if move then codestack [k+s+depth] :=
                      2 ↑ v x codestack [k+s+m]
                 End
           end;

      top stop := old top stop;

      k := (lines+labels) x 2 + constant;

      comment k is required to set to zero the address
      modification words of outer blocks which refer to parameters
      and address modification words of inner blocks. Constant
      is equal to the no. of parameters at REFERENCE and following;

      lines := lines + old lines + (if move then 0 else -1);

      bitnumber := bitnumber + old bitnumber + (if move then -35
      else 1);
```

```
  index := top stop + depth x (lines - 1);
  for s := 1 step 1 until k do mark (0);
end close section;
```

Loading of conditionals

At load time there will be a stack of delimiters kept by the loading system to be used to insure the proper linkage of the if's, then's, and else's. The stack will also include a symbol marking the end of the conditional expression. In the following code "program" indicates the section of core storage in which the actual running program is stored and "location" is the index keeping trakk of this storage. "ff" is used to indicste the first free of this particular stack described above.

```
then: if stack[ff - 1] ≠ then ∧ [redacted] stack[ff - 1] ≠ else then
      begin
        ff:=ff + 3;
        stack[ff - 3]:="dummy"
      end;
stack[ff - 2]:=location;
stack [ff - 1]:="then";

else: program[stack[ff - 2]]:=location;
program[location]:=stack[ff - 3];
stack[ff- 3]:=location;
stack[ff - 1]:="else";

end of conditional: [redacted]: p:=stack[ff - 3];
for q:=p while q ≠ dummy do
      begin
        p:=program[q];
        program [q] :=location
      end;
if stack[ff - 1] = then then program[stack[ff - 2]]:=location;
ff:=ff - 3;
```

Algol running system
December 8, 1961

Representation of blocks and procedures in store

Block	Proc.		
X	X		call address:=q;
X	X	q:	go to procedure/block entry;
X	X		reference
	X		specifications and identifiers in forward order
X	X	address of 1st inst.:	array and switch declarations
X	X		other code
X	X		.
X	X		.
X	X		.
X	X		.
X	X		.
X	X		go to end;
X	X	reference	:appetite (total for variable format in fixed order)
X	X	+ 1	:number of labels (p)
X	X	+ 2	:number of integers
X	X	+ 3	:number of reals
X	X	+ 4	:number of booleans
X	X	+ 5	:number of array coefficient sets
X	X	+ 6	:depth of recursion
X	X	+ 7	:current address modifier
X	X	+ 8	:address of first instruction
X	X	+ 9	:address following declaration
X	X	+ 10	:type
X	X	+ 11	:label 1 (goal and identifier)
X	X	+ 12	: label 2
X	X		.
X	X		.
X	X		.
X	X	+ 10 + p	:label p
X	X	+ 11 + p	:address modification code

Form of specifications:

Specification 1	brief identifier of procedure
Specification 2	brief identifier of formal 1
Specification 3	Brief identifier of formal 2
.	.
.	.
.	.
"no more parameters" n	brief identifier of formal n n-1

Block information in stack

FIXED FORMAT FIXED ORDER	Stack reference current address modifier return address REFERENCE
VARIABLE FORMAT FIXED ORDER	Value of type procedure formal locations labels integers reals booleans array identifiers and coefficients switch identifiers and tables temporaries
VARIABLE FORMAT VARIABLE ORDER	expressions as actual parameters subscripted variables as actual parameters arrays called by value (components)
VARIABLE FORMAT	local arrays (components) Local switch elements being expressions

Procedure and block entry administration

```
block entry:  procedure:=false;

go to X;

procedure entry:  procedure:=true;

X:  stack[first free]:=stack reference;

REFERENCE:=store[call address2+1];

stack[first free + 1]:=store[REFERENCE + 7]; comment current address modifier
   to stack;

stack[first free + 3]:=REFERENCE;

stack reference:=first free;

first free:=first free + 4 + store[REFERENCE];

store[REFERENCE + 6]:=store[REFERENCE + 6] + 1;  comment count depth of recursion;

address of formal:=stack reference + (if store[REFERENCE + 10] defines a
   type procedure then 5 else 4);

if ¬ procedure then go to QQ;

address of actual:=call address + 1;

address of specification:=call address2 + 2;

last return:=L1;

regular return:=PE;

go to W;

PE:  address of specification:=address of specification + 1;

W:  specification:=store[address of specification];

if specification = no more parameters then

      begin

        if store[address of actual] = end mark then go to transformation

            finished

        else
```

```
        L1: begin
              printtext (#Non-agreement between number of formals and
                number of actuals#);
              new line;
              hot point:=address of actual;
              go to alarm;
            end
      end;
go to parameter  treatment;

transformation finished:  procedure:=true;
REFERENCE:=stack[stack reference + 3];
QQ:  modify addresses (stack reference + 4);
stack[stack reference + 2]:= if procedure then address of actual + 1
                             else -1;  comment this sets the return;
for k:=1 step 1 until store[REFERENCE + 1] do
   stack[address of formal + k - 1]:=combination(store[REFERENCE + 10 + k],
      stack reference);  comment this sets the labels into the stack;
go to instruction[store[REFERENCE + 8]];
```

Discussion of parameter treatment

Parameter treatment is used by
1) procedure entry
2) array declaration
3) switch declatation
4) special functions (sin, cos, etc.)

Input parameters:
address of actual - will be counted on to next parameter if exit through regular return or left at same value if exit through last return
address of formal - will be increased by 3 by regular return or left unchanged by last return
specification - must be ≠ no more parameters. Will be changed arbitrarily.
regular return - set by each action using parameter treatment and used as exit if a proper actual parameter is found
last return - set by each section using parameter treatment and used as exit when actual kind and type = end mark

Variables used:
address of actual
address of formal
address of specification
entry base
expression
value
specification
actual kind and type

```
parameter treatment:  actual kind and type:=store[address of actual];
if actual kind and type = end mark then go to instruction[last return];
actual address:=store[address of actual + 1];
if kind (actual kind and type) = formal then
      bègin
        actual kind and type:=stack[adtual address];
        actual address:=stack[actual address + 1];
        formal:=true
      end
else formal:=false;
if name or value (specification) = value then
      begin
        round:=type(specification) = integer∧type(actual kind and type )
          = real;
        float:=type(specification) = real∧type(actual kind and type) = integer
      end;
go to action[action table[actual kind and type, spcification]];


next parameter:  address of actual:=address of actual + 2 ;
next parameter after expression:  address of formal:=address of formal + 3;
go to instruction [regular return];
```

```
end:  value:=stack[stack reference + 4];
DECREASE LEVEL;
return:=stack[first free + 2];
if return>0 then go to instruction[return]
else go to instruction[store[REFERENCE + 9]];



exit from parameter expression:  first free:=first free - 1;
go to instruction[stack[first free] + 1];
```

```
procedure DECREASE LEVEL;
    begin
      REFERENCE:= stack[stack reference + 3];
      D:=store[REFERENCE + 6]:=store[REFERENCE + 6] - 1;  comment decrease
            depth of recursion;
      if D≥1 then modify addresses (stack[stack reference + 1]);
      first free:=stack reference;
      stack reference:=stack[first free];
    end;
```

Representation of procedure call in store

<u>if</u> formal procedure identifier <u>then</u> store[p]:=stack[formal + 1];

call address:=p;

p: <u>go</u> <u>to</u> store[procedure start]; <u>comment</u> this address was possibly set in the above statement;

actual parameter { kind
address

actual parameter { kind
address

.   .

.   .

.   .

"end mark"

There must be complete matching of types in all parameters called by name.

In a type procedure the procedure identifier has two meanings:
1) it calls the procedure
2) it represents the value of the procedure.
We have set the arbitrary rule that a procedure identifier can only be assigned to in the body of the procedure for which it is the identifier.

Consider the following example:

```
begin real p, q; boolean B1, B2;
  procedure P;
    begin
      Q:=p + q
    end;
  real procedure Q;
    begin
      if B1 then Q
      else if B2 then P
           else Q:=7;
    end;
  .
  .
  .
  p:=Q;
  P;
  .
  .
  end;
```

Such an example would be considered to be illegal in our system since the procedure identifier Q is assigned to from without the procedure Q. If we call P before Q has been called we nowhere have a location in which to put the value of Q.

Meaning of address in single identifier parameters

Actual parameter	Meaning of address
Simple variable	location where value of vatiable is stored
array identifier	location where representation of the identifier is stored in the stack
switch	entry point of switch declaration
procedure (ang kind)	location of start of procedure
label	location where representation of label (goal and mark) is stored
formal	formal location in the stack

Representation of expression as actual parameter im store

Subscripted variables see page 31

kind
reference
code for: value:=expression;

.

.

go to exit from parameter expression;

reference : appetite (temporaries)
+ 1: current address modifier
+ 2: address following dedlaration
+ 3: address modification code

Information in the three formal locations in the stack

Call by name

Actual parameter	f	f+1	f+2
SIMPLE variable	kind and type	address of value	not used
array identifier	kind and type	address of representation of identifier	not used
switch	kind and type	address of representation of identifier	not used
procedure	kind and type	procedure start	not used
label	kind and type	where representation is stored in stack	not used
expressions	kind and type	entry of representation in stack	not used

Call by value

Kind of value			
INTEGER, real, boolean	actual value	not used	not used
label	goal, mark	not used	not used
array	"type, array"	address of first element	address of coefficients

Table of actual parameters and associated actions

Actual parameter \ Specification:	integer	real	integer array	real array	integer procedure	real procedure	boolean	boolean array	boolean procedure	label	procedure	switch	string
Simple variable													
integer	F A	A											
real	A	F A											
Array identifier													
integer			F B	B									
real			B	F B									
Procedure identifier													
no parameters													
integer	F C	C											
real	C	F C											
parameters													
integer					F								
real						F							
subscripted variables													
integer	G E	E											
real	E	G E											
Other expression													
integer	G D	D											
real	D	G D											
Simple variable													
boolean							F A						
Array identifier													
boolean								F B					
Procedure													
boolean													
no parameters							F C		F C				
parameters									F				
subscripted variable													
boolean							G E						
Other expression													
boolean							G D						
Label										F A			
Designationalexpression										G D			
Procedure, no type											F		
Switch identifier												F	
Stringconstant													F

In case of a double entry, the left entry refers to name, the one to the right to value. A single entry always refers to name.

Blanks indicate error situations.

Action A - take value of simple variable

```
value:=stack[actual address];

assign value:  stack[address of formal]:=if float then floatf(value)
                                          else if round then entier(value + 0.5)
                                               else value;

go to next parameter;
```

Action B - take value of array

```
stack[address of formal]:=specification - value mark;

stack[address of formal + 1]:=first free;

actual address2:=stack[adtual address + 2];

actual address:=stack[actual address + 1];

stack[address of formal+2]:=actual address2;

for j:=actual address step 1 until actual address + stack[actual address2
      + 2]- 1 do

    begin

      value:=stack[j];

      stack[first free]:=if round then entier (value + 0.5)
                          else if float then floatf (value)
                               else value;

      first free:=first free + 1

    end;

go to next parameter;
```

Action C - take value of procedure without parameters

```
expression base:=first free;
STACK SITUATION;

call address:=WW;

WW:  go to instruction[actual address];

"WW + 1" "end mark"

UNSTACK SITUATION;

go to assign value; comment in Action A;
```

Action D - take value of expression

```
expression base:=first free;

entry:=if formal then actual address else STACK EXPRESSION;

if formal then address of actual:=address of actual + 2;

STACK SITUATION;

stack[first free]:=P;
 first free :=first free + 1;
P:  go to instruction[entry];

y:  UNSTACK SITUATION;

stack[address of formal]:=if formal then floatf (value)
                          else if round then entier (value + 0.5)
                               else value;

go to next parameter after expression;
```

Action E - take value of subscripted variable

```
expression base:=first free;
entry:=if formal then actual address else STACK EXPRESSION;
if formal then address of actual:=address of actual + 2;
STACK SITUATION;
stack[first free]:=P1;
first free:=first free + 1;
P1:  go to instruction[entry];
value:=stack[address];
go to y;  comment in Action D;
```

Action F - take simple name

```
stack[address of formal]:=actual kind and type;
stack[address of formal + 1]:=actual address;
go to next parameter;
```

Action G - take name of expression

```
stack[address of formal]:=actual kind and type;
stack [address of formal + 1]:=if formal then actual address else STACK
     EXPRESSION;
go to if formal then next parameter else next parameter after expression;
```

```
procedure modify addresses (modifier); value modifier; integer modifier;
    begin
      integer amount;
      amount:=modifier - store[REFERENCE + 7];
    if amount ≠ 0 then
        begin
          store[REFERENCE + 7]:=modifier;
          comment now modify addressess between store[REFERENCE + 8] and
          REFERENCE - 2 using code stored at REFERENCE + 11 + store
         [REFERENCE + 1] ;
        end
    end;
```

```
integer procedure STACK EXPRESSION;

  comment uses actual address, first free, address of actual as non-
  local parameters;

    begin

      integer j, amount;

      amount:=first free - store[actual address + 1];

      if amount ≠ 0 then

        begin

          store[actual address + 1]:=first free;

          comment now modify addresses between address of actual + 2
          and actual address - 2 using code stored at actual address + 3;

        end;

      first free:=STACK EXPRESSION:=first free + store[actual address];

      for j:=address of actual + 2 step 1 until actual address - 1 do

        begin

          stack[first free]:=store[j];

          first free:=first free + 1

        end;

      address of actual:=store[actual address + 2]

    end;
```

```
procedure STACK SITUATION;
    begin
      stack[first free]:=expression base;
      stack[first free + 1]:=address of actual;
      stack[first free + 2]:=address of formal;
      stack[first free + 3]:=address of specification;
      stack[first free + 4]:=float;
      stack[first free + 5]:=round;
      stack[first free + 6]:=regular return;
      stack[first free + 7]:=last return;
      stack[first free + 8]:=own array;
      stack[first free + 9]:=exists already;
      stack[first free + 10]:=address in stack;
      stack[first free + 11]:=n;
      first free:=first free + 12
    end;

prodedure UNSTACK SITUATION;
    begin
      n:=stack[first free - 1];
      address in stack:=stack[first free - 2];
      exists already:=stack[first free - 3];
      own array:=stack[first free - 4];
      last return:=stack[first free - 5];
      regualr return:=stack[first free - 6];
      round:=stack[first free - 7];
      float:=stack[first free - 8];
      address of specification:=stack[first free - 9];
      address of formal :=stack[first free - 10];
```

```
    address of actual:=stack[first free - 11];
    REFERENCE:=stack[stack reference + 3];
    first free:=stack[first free - 12]
end;
```

Representation of arrays in the store

```
     call address:=W3;

W3    : go to array declaration;
  +1  : A  (see below for explanation)
  +2  : number of identifiers
  +3  : kind and type
  +4  : {kind and type of first lower bound
        {address                              See page 10 for explanation of
        {kind and type of first upper bound        address
        {address
         .
         .
         .
         "end mark"
```

"A" is the first address of a three-word packet in the area reserved by block entry in the stack.

```
{A   :kind and type of array
{ +1:A0  (address of first element of array)
{ +2:A1   (address of first element in coefficient vector)
{     .
{     .    Three words for each identifier
      .
A1   : number of subscripts   (n)
  +1: s
  +2: c[0]
  + 3: c[1]
  .      .
  .      .
  .      .
  +n+2: c[n]
```

Storage of arrays

Consider the following declaration:

<u>array</u> A,B,$[l_1:u_1, l_2:u_2, \ldots, l_n:u_n]$;

These arrays will be stored row-wise in consecutive locations: (in the stack)

A0 :$A[l_1, l_2, l_3, \ldots, l_n]$
A0+1 :$A[l_1, l_2, l_3, \ldots, l_n + 1]$
.
.
.
$A0+u_n-l_n$:$A[l_1, l_2, l_3, \ldots, l_{n-1}, u_n]$
$A0+u_n-l_n+1$:$A[l_1, l_2, l_3, \ldots, l_{n-1}+1, l_n]$
.
.
.
B0 :$B[l_1, l_2, l_3, \ldots, l_n]$
.
.
.

In general the location of $A[i_1, i_2, i_3, \ldots, i_n]$ is given by:

(1) $A0 + (i_n-l_n)+(i_{n-1}-l_{n-1})x(u_n-l_n+1)+(i_{n-2}-l_{n-2})x(u_n-l_n+1)x(u_{n-1}-l_{n-1}+1) + \ldots + (i_1-l_1)x(u_n-l_n+1)x(u_{n-1}-l_{n-1}+1)x \ldots x(u_2-l_2+1)$.

The number of locations occupied by the array is given by

$L=(u_n-l_n+1)x(u_{n-1}-l_{n-1}+1)x \ldots x(u_2-l_2+1)x(u_1-l_1+1)$.

(1) may be rewritten as follows:

(2) $A0+i_n+i_{n-1}x(u_n-l_n+1)+i_{n-2}x(u_n-l_n+1)x(u_{n-1}-l_{n-1}+1)+ \ldots +i_1x(u_n-l_n+1)x \ldots x(u_2-l_2+1) - (l_n+l_{n-1}x(u_n-l_n+1) + \ldots +l_1x(u_n-l_n+1)x \ldots x(u_2-l_2+1))$

At the time of the array declaration, the coefficients of the terms in (2) are calculated and stored in A1+3 through A1+n+2 where n is the number of subscripts. The final term of (2) is calculated (it is referred to as s) and stored in A1+1. The total length of the array is stored in A1+2. (2) is then used to calculate the address of an element when necessary. For a declaration like the one above, only one set of ceofficients is calculated and A+2 and B+2 both refer to A1.

In the case of own arrays, the values of l_1, u_1, l_2, u_2, ..., l_n, u_n are stored immediately preceding the coefficient vector so that they may be used if the own array is redeclared for discerning if the old and new arrays have common elements.

```
array declaration:  own array:=exists already:=false;

own array entry:  address in stack:=call address+ 1;

n:=0;

address of actual:=call address + (if own array then 5 else 4);

regular return:=next subscript;

last return:=form coefficients;

next subscript:  address of formal:=first free;

first free:=first free + 1;

n:=n + 1;

specification:="integer value";

go to parameter treatment;

form coefficients:  n:=(n - 1) ÷ 2;

first free:=first free - 1;

if exists already then go to check for overlap;

address of last c:=store[address in stack]+ 3 x store[address in stack + 1]
  +(if own array then 3 x n + 2 else 2 + n);

stack[address of last c]:=1;

s:=0;

for p:=address of last c step -1 until address of last c - n + 1 do

      begin

         stack[p - 1]:=stack[p] x (stack[first free - 1] - stack[first free
           - 2] + 1);

         s:=s + stack[first free - 2] x stack[p];

         first free :=first free - 2

      end;

stack[address of last c - n - 1]:=s;

stack[address of last c - n - 2]:=n;

if own array then go to create new own array;
```

```
for p:=1 step 3 until store[address in stack + 1] x 3 - 2 do
    begin
      stack[store[addres in stack] + p]:=first free;
      first free:=first free + stack[address of last c - n];
    stack[store[address in stack]+ p + 1]:=address of last c - n - 2;
      stack[store[address in stack] + p - 1] :=store[address in stack + 2]
    end;
go to instruction[address of actual + 1];
```

Discussion of own quantities

We have decided to rule out the use of own variables in connection with recursion.

Simple own variables will have "absolute" locations immediately following the program and will be referenced by "absolute" addressing. They will act as if they are declared in the outermost block of the program.

Own arrays:
Own arrays will not be kept in the stack as are non-own arrays. They will be stored in an "own area" (presumed at this time to be in high end of core). The locations A, A+1, A+2, AL, etc. are not in the stack as with non-own arrays but are in the section mentioned above immediately following the program.

Referencing elements of own arrays is done exactly as referencing of non-own arrays.

When an own array is declared, various actions may be taken:
1) If the array does not already exist, the array is created in the own area and the proper addresses are supplied to the locations in the section following the program. The values of the upper and lower bounds are also stored in the section.

2) If the array already exists in the own area, a check is made to determine whether the new subscript bounds are the same as the old ones. If so, no other action is taken. Otherwise, a new version of the array is created in the own area, the new values of the coefficients and bounds are stored, and the old array is removed from the own area and the proper collapsing of the area is done. Common elements, if any, are stored in the proper locations of the new array.

Representation of own arrays in store

call address:=W6;

W6 : go to own array declaration;
+1 : A
+2 : number of identifiers
+3 : kind and type
+4 : exists already (boolean)
+5 : {kind and type of first lower bound
{address
{ .
{ .
{ "end mark"

See page 20 for explanation of A and these locations.

{A : "array, type"
{ +1 : A0 (address of first element of array in own area)
{ +2 : A1 (address of first element of coefficient vector)
{ . .
{ . .

Three words for each identifier

A1 - 2 x n: l_1
A1 - 2 x n - 1: u_1
. .
. .
. .
A1-1 : u_n
A1: number of subscripts (n)
A1+1: s
A1+2: c[0]
. .
. .
. .
A1+ n + 2: c[n]

Each own array in the own area is headed by a 3-word packet: (to be used to release locations in the own area when an array is redeclared to be of a different size)

A0 - 3: A
A0 - 2: number of identifiers
A0 - 1: address of first word of 3-word packet associated with next own array

```
own array declaration:  own array:=true;

exists already:=store[call address + 4];

go to own array entry;  comment in array declaration;

check for overlap:  no change:=overlap:=true;

address of bounds:=first free - 1;  comment address of bounds points to
   last upper limit;

address of old L1:=stack[2 + store[address in stack]]- 2 x n;

for m:=1 step 1 until n do

      begin

         current old lower:=stack[address of old L1 - 2 + 2 x m];

         current old upper:=stack[address of old L1 - 1  + 2 x m];

         current new lower:=stack[address of bounds - 1 - 2 x (n - m)];

         current new upper:=stack[address of bounds - 2 x (n - m)];

         L:=maximum lower[m]:=if current old lower < current new lower then
            current new lower else current old lower;

         U:=minimum upper[m]:=if current old upper < current new upper then
            current old upper else current new upper;

         overlap:=overlap ∧ U ≥ L;

         nochange:= no change ∧ current old lower =current new lower ∧
            current old upper = current new upper

      end;

if no change ∧ overlap then

      begin

         first free:=first free - 2 x n;

         go to instruction[address of actual + 1]

      end;

stack[first free]:=n;

address of s:=first free + 1;

first free:= first free + 3 + n;
```

```
address of last c:=address of s + n + 1;

stack[address of last c] :=1;

stack[address of s ]:=0;

for p:=address of last c step -1 until address of last c - n + 1 do

    begin

      stack[p - 1]:=stack[p] x (stack[address of bounds]- stack [address
        of bounds - 1] + 1);

      stack[address of s]:=stack [address of s]+ stack[p] x stack[address
        of bounds - 1];

      address of bounds:=address of bounds - 2

    end;

address of bounds:=address of bounds + 1; comment address of bounds now
  points to first lower limit;

number of identifiers:=store[address in stack + 1] ;

number to collapse:=stack[address of old L1 + 2 x n + 2] x number of identifiers;

comment number to collapse is three less than total;

bottom of region:=stack[store [address in stack] + 1] + number to collapse;

first free own:=first free own - stack[address of s + 1] x number of
  identifiers - 3;

if ¬ no change ∧ ¬ overlap then go to collapse;

m:=1;

current addres of old[1]:=stack[store [address in stack] + 1];

current address of new[1]:=first free own + 4;

MOVE ELEMENTS;

collapse:  number of collapse:=number to collapse + 3;

for p:= bottom of region - 1 step -1 until first free own - 1 + number to
  collapse do

     stack[p]:= stack[p - number to collapse];

first free own:=first free own + number to collapse;

stack[first free own + 1] :=store[address in stack] ;
```

```
stack[first free own + 2]:=number of identifiers;

stack[first free own + 3]:=link:=first free own + number of identifiers x
   stack[address of s + 1] + 4;

for p:=0 step 3 until(number of identifiers - 1) x 3 do

      stack[store[address in stack] + p + 1]:=first free own + 4 + p x
         stack[address of s + 1];

K: if link = bottom of region then go to move subscripts;

for p:=0 step 1 until stack[link + 1] - 1 do

      stack[stack[link] + 3 x p + 1]:=stack[stack[link] + 3 x p + 1]+ p x
        number to collapse;

stack[link + 2] :=stack[link + 2] + number to collapse;

link:= stack[link + 2];

go to K;

move subscripts:  for p:=0 step 1 until 3 x n + 2 do

      stack[address of old L1 + p]:=stack[address of bounds + p];

first free:=address of bounds;

N: store[address in stack + 3]:=true;

go to instruction[address of actual + 1];

create new own array:  for p:=0 step 1 until 2 x n - 1 do

      stack[address of last c - 3 x n - 2 + p]:=stack[first free + p];

old first free own:=first free own;

first free own:=first free own - 3 - store[address in stack + 1] x stack
  [address of last c - n];

stack[first free own + 1]:=store[address in stack];

stack[first free own + 2]:=store[address in stack + 1];

stack[first free own + 3]:=old first free own + 1;

for p:=store[address in stack + 1] x 3 - 2 step -3 until 1 do

    begin

      old first free own:=old first free own - stack[address of last c - n];

      stack[store[address in stack] + p]:=old first free own + 1 ;
```

```
        stack[store[address in stack] + p + 1] :=address of last c - n - 2;
        stack[store[address in stack] + p - 1] :=store[address in stack + 2]
    end;
go to N;
```

```
procedure MOVE ELEMENTS;

    begin

      integer j;

      for  j:=maximum lower[m] step 1 until minimum upper[m] do

        if m = n then

          for p:=0 step 1 until number of identifiers - 1 do

            stack[j - stack[address of bounds + 2 x (m - 1)]+ current
              address of new[m] + p x stack[address of s + 1]]:= stack
              [j - stack[address of old L1 + 2 x (m - 1)]+ current address
              of old[m] + p x stack[address of old L1 + 2 x n + 2]]

        else

            begin

              m:=m+1;

              current address of old[m]:=current address of old[m - 1] +
                stack[address of old L1 + 2 x n + 1 + m] x (j - stack
                [address of old L1 - 2 + 2 x (m - 1)]);

              current address of new[m]:=current address of new[M - 1] +
                stack[address of s + m] x (j - stack[address of bounds
                - 2 + 2 x (m - 1)]);

              MOVE ELEMENTS;

              M:=m - 1

            end;

    end;
```

Representation of switches

The translator produdes something very much like a procedure call. At block entry time, after address modification, this call is performed, all expressions called by name. The effect of the call is to transfer into a section of the appetite section of the stack the names of the elements of the switch declaration. Subsequent switch designators will only make use of this information in the stack.

Switch declaration

```
         call address:=W9;
W9    :  go to switch declaration;
  +1  :  address of S;    (see below for explanation of this address)
  +2  : {kind of first switch element
  +3    {address
  .     { .
  .     { .                 See page 10 for explanation of address
  .     { .
         "end mark"
```

The switch elements are representaed exactly as parameters of a procedure call. There are three possibilities:

1) label
2) designational expression
3) formal parameter.

Switch identifier in stack

```
S   : "switch"
 +1: first address of table
 +2: number of entries
first address of table:  {kind
                         {address
          .              { .
          .              { .
          .              {
```

The forma of the items in the table is the same as that of the con tents of formal lacations. Two possibilities:

1) label
2) designational expression.

switch declaration: address of switch:=store[call address + 1];

stack[address of switch]:="switch"

address of formal:=stack[address of switch + 1]:=address of switch + 3;

address of actual:=call address + 2;

n:=0;

regular return:=SW;

last return:=SW2;

specification:="label";

<u>go</u> <u>to</u> parameter treatment;

SW: n:=n+1;

<u>go</u> <u>to</u> parameter treatment;

SW2: stack[address of switch + 2]:=n;

<u>go</u> <u>to</u> instruction[address of actual + 1];

Representation of subscripted variables

```
Occurrence:   Actual parameter in stack | Left part | Expression

Running code:                    temp0:="non-integer";
                                 temp1:=exp1;
                                    .    .
                                    .    .
                                    .    .
                       "W4 - 4" : tempn:=expn;
                                 first address:=stack[A + 1];
                                 address of ceefficients:= stack[A + 2];
                                 call address :=W4;
```

```
W4: go to address of subscripted variable       | W4: go to take
                                                |     value of
                                                |     subscripted
                                                |     variable;
go to exit from             | temp0:=           |
parameter expression        |     address;      |
```

Examples of accurrences:

Actual parameter in stack	,A[exp1, . . . , expn],
In left part	A[exp1, . . . , expn]:=
In expression	. . . + A[exp1, . . . , expn] + . . .

```
take value of subscripted variable:  take value:=true;

go to t;

address of subscripted variable:  take value:=false;

t:  address of subscript:="store[call address - 4]" - stack[address of
      coefficients];

if stack[address of subscript] ≠ "non-integer then

     begin
                       number of subscripts of
       printtext(#Error in^subscripted variable#);

       new line;

       hot point :=call address;

       go to alarm1

     end;

address := - stack[address of coefficients + 1];

for m:=1 step 1 until stack[address of coefficients] do

      address:=address + stack[address of coefficients + 2 + m] x stack
        [address of subscript + m];

if address < 0 ∨ address ≥ stack [address of coefficients + 2] then

     begin

       printtext(#subscript of array element too large#);

       new line;

       hot point:=call address;

       go to alarm1

     end;

address:=address + first address;

iftake value then value:=stack[address];

go to instruction[call address + 1];
```

Representation of left parts

	done before calculation of expression	done after calculation
simple declared variable	nothing	assign directly
formal variable	calculate or take address to temporary	assign to address found in temporary
subscripted variable	calculate address to temp.	assign to address found in temporary

Representation of formal identifiers as left-part variable

```
Formal 1:=stack[formal [redacted]];
formal 2 := stack[formal+1 [redacted]];
call address:=W5;
W5:  go to take address of formal;
temp:=address;

take address of formal:  if kind(formal 1) = simple variable then
      begin
        address:=formal 2;
        go to instruction[call address + 1]
      end;
if kind(formal 1) = subscripted variable then
      begin
        stack[first free]:=call address;
        first free:=first free + 1;
        go to instruction[formal 2]
      end;
printtext(#Error in formal as left-part variable#);
```

new line;

hot point:=call address;

go to alarm1; comment kind(formal 1) = procedure identifier or other expression;

Representation of formal name parameters within procedure body

formal 1:=stack[[redacted] formal];

formal 2:=stack[[redacted]formal + 1];

call address:=u;

u: go to take value of formal; comment this jumps to the fixed administration and kind(formal 1) has one of 4 values:
1) simple variable
2) procedure identifier
3) subscripted variable
4) expression;

take value of formal: if kind (formal 1) = simple variable then

begin

value:=stack[formal 2];

go to instruction [call address + 1]

end;

stack[first free]:=call address;

first free:=first free + 1;

if kind (formal 1) = procedure identifier then

begin

call address:=W1;

W1: go to store [formal 2];

"W1+1: "end mark"

go to exit from parameter expression

end;

if kind (formal 1) = subscripted variable then

```
begin
  stack[first free]:= W2;
  first free:=first free + 1;
  W2:  go to instruction[formal 2];
  comment We now go off into the routine (placed in the stack)
  representing the subscripted variable. Thie routine
  1)  puts the address of the subscripted variable in "address"
  2)  jumps to exit from parameter expression. At this stage first
  free will always have the same value as when the routine was
  entered. From exit from parameter expression we finally return
  to the following:;
  value:=stack[address];
  go to exit from parameter expression;
end;
comment We now the case when kind (formal 1) indicates an expression;
go to instruction [formal 2];
```

Alarm output for the running system

```
alarm: ;comment This entry will be used when an actual machine fault
  (divide by 0, SCC fault, etc.) occurs. The kind of fault will be printed
  according to a bit configuration in some register set by the operator.
  Hot point will be set to indicate the actual machine location of the fault.;

alarm1: if hot point < store bottom then go to procedure or block;comment on
                                                                   page 40;
if stack[first free - 1]>first free then go to exit to administration;
Comment on page 45;
if store[stack[first free - 1]] = "go to take value of switch designator"
  then go to switch alarm; comment on page 46;

printtext(#Error in expression called by name#);

new line;

REFERENCE:=stack[stack reference + 3];

m:=first specification:=store[REFERENCE + 8];

for m:=m step -1 while store[m] ≠ "go to procedure entry" do

  first specification:=first specification - 1;

first specification:=first specification + 2;

printtext(#In body of procedure    #);

printtext(#identifier part(store[first specification]));

new line;

stack point:=stack reference + 4;

if store[REFERENCE + 10] defines a type then

     begin

       printtext(#value of procedure#);

       print(1,5,2,stack[stack point]);

       new line;

       stack point:=stack point + 1

     end;

printtext(#Formals#);

new line;
```

```
for m:=stack point step 3 while specification part(store[first specification])
  ≠ "no more parameters" do

   begin

     printtext(identifier part(store[first specification + 1]));

     if specification part(store[first specification]) = name then

       begin

         printtext(#Called by name#);

         new line;

         if stack[m + 1]≤ hot point then

           parameter in error:=identifier part(store[first specification
             + 1])

       end

     else

       if kind(store[first specification]) ≠ array then

        begin

         print(1,5,2,stack[m [illegible]]);

         new line

        end

       else

         for p:=0 step 1 until stack[stack[m + 1] + 2] - 1 do

           begin

             print(1,5,2,stack[stack[m + 1] + p]);

             new line

           end;

     first specification:=first specification + 1;

     stack point:=stack point + 3

   end;

printtext(#Parameter in error#);

printtext(parameter in error);
```

```
new line;
if store[REFERENCE + 1] ≠ 0 then
    begin
      printtext(#Labels#);
      new line;
      print label:=false;
      for m:=REFERENCE + 11 step 1 until REFERENCE + 11 + store[REFERENCE
        + 1] do
        begin
          printtext(identifier part(store[m]));
          new line;
          if m ≠ REFERENCE + 10 + store[REFERENCE + 1] then
            begin
              if goal part(store[m]) ≤ stack[first free - 1] ∧ goal
                part(store[m + 1]) > stack[first free - 1] then
                begin
                  printtext(#Error between these two labels#);
                  new line;
                  print label:=true
                end
            end;
          stack point:= stack point + 1
          if ¬ print label then
            begin
              printtext(#Error after last label#);
              new line
            end
    end;
```

```
DUMP: if store[REFERENCE + 2] ≠ 0 then
        begin
          printtext(#Integers#);
          new line;
          for m:=stack point step 1 until stack point + store[REFERENCE + 2] do
             begin
               print(1,5,2,stack[m]);
               new line
             end;
          stack point:=stack point + store[REFERENCE + 2]
        end;
      if store[REFERENCE + 3] ≠ 0 then
        begin
          printtext(#Reals#);
          new line;
          for m:=stack point step 1 until stack point + store[REFERENCE + 3] do
             begin
               print(1,5,2,stack[m]);
               new line
             end;
          stack point:=stack point + store[REFERENCE + 3]
        end;
      if store[REFERENCE + 4] ≠ 0 then
        begin
          printtext(#Booleans#);
          new line;
          for m:=stack point step 1 until stack point + store[REFERENCE + 4] do
```

```
            begin
              print(stack[m]);
              new line
            end;
          stack point:=stack point + store[REFERENCE + 4]
        end;
if store[REFERENCE + 5] ≠ 0 then
        begin
          printtext(#Arrays#);
          new line;
          for m:=0 step 1 until store[REFERENCE + 5] - 1 do
            begin
              p:=stack point + 1 + 3 x m;
              for r:= stack[p] step 1 until stack[p] + stack[stack[p + 1] + 2] do
                begin
                  print(1,5,2,stack[r]);
                  new line
                end;
              new line
            end
        end;
if store[REFERENCE + 10] indicates outer block then                  ;
first free:=stack reference;
stack reference:=stack[first free];
hot point:=if stack[first free + 2] = -1 then store[REFERENCE + 9] else stack
  [first free + 2];
go to alarm1; comment on page 36;
procedure or block:  REFERENCE:=stack[stack reference + 3];
```

```
if store[REFERENCE + 10] indicates a block then go to block;comment on page 43;
printtext(#Error in procedure body#);
m:=first specification:=store[REFERENCE + 8];
for m:=m step -1 while store[m] ≠ "go to procedure entry" do
  first specification:=first specification - 1;
first specification:=first specification + 2;
printtext(identifier part(store[first specification]));
new line;
stack point:=stack reference + 4;
if store[REFERENCE + 10] defines a type then
      begin
        printtext(#Value of procedure#);
        print(1,5,2,stack[stack point]);
        new line;
        stack point:=stack point + 1;
      end;
printtext(#Formals#);
new line;
for m:=first specification step 1 while store[m] ≠ "no more parameters" do
      begin
        printtext(identifier part(store[m+1]));
        if specification part(store[m]) = name then
          begin
            printtext(#Called by name#);
            new line
          end
        else
```

```
if type(store[m]) = array then
 begin
  for p:=stack[stack point + 1] step 1 until stack[stack point
   + 1] + stack[stack[stack point + 2] + 2] - 1 do
   begin
    print(1,5,2,stack[p]);
    new line
   end
 end
else
 begin
  print(1,5,2,stack[stack point]);
  new line
 end;
stack point:=stack point + 3;
end;
DUMP1: if store[REFERENCE + 1] ≠ 0 then
 begin
  printtext(#Labels#);
  new line;
  print label:=false;
  for m:=REFERENCE + 11 step 1 until REFERENCE + 11 + store[REFERENCE+1] do
   begin
    printtext(identifier part(store[m]));
    new line;
    if m ≠ REFERENCE + 10 + store[REFERENCE + 1] then
     begin
```

```
                if goal part(store[m]) ≤ hot point ∧ goal part(store[M + 1])
                    >hot point then
                  begin
                    printtext(#Error between these two labels#);
                    new line;
                    print label:=true
                  end
              end;
            stack point:=stack poiht + 1
            end;
        if ¬print label then
            begin
              printtext(#Error after last label#);
              new line
            end
        end;
go to DUMP; Comment on page 39;
block: Printtext(#Error in block#);
new line;
stack point:=stack reference + 4
go to DUMP1; comment on page 42;
parameter value: printtext(#Expression called by value#);
new line;
first specification:=address of specification:=stack[first free - 10];
for m:=address of specification step -1 while store[m] ≠ "go to procedure
    entry" do
    first specification:=first specification - 1;
first specification:=first specification+ 2;
printtext(#In procedure heading #);
```

```
printtext(identifier part(store[first specification]));

new line;

REFERENCE:= stack[stack reference + 3];

stack point:=stack reference + (if store[REFERENCE+ 10] defines a type then
  5 else 4);

printtext(#Formals#);

new line;

for m:=first specification step 1 until address of specification do

     begin

       printtext(#identifier part(store[m+1]));

       if specification part(store[m]) = name then

          begin

            printtext(#Called by name#);

            new line

          end

       else

          begin

            if specification part(store[m]) = array then

              begin

                for p:=stack[stack point + 1]step 1 until stack[stack point
                  + 1] + stack[stack [stack point + 2] + 2] - 1 do

                  begin

                    print(1,5,2,stack[p]);

                    new line

                  end

              end

            else

              begin

                pring(1,5,2,stack[stack point]);
```

```
                new line
                end
             end;
          stack point:=stack point + 3
        end;
printtext(identifier part(store[address of specification + 1]));
printtext(#Error in this parameter#);
new line;
hot point:=stack[first free - 12];
first free:=stack reference;
stack reference:=stack[first free];
[redacted]
go to alarm1; comment on page 36;
exit to administration: if stack[first free - 1] indicates array declaration
  then
        begin
          printtext(#Error in#);
          if stack[first free - 5] then printtext (#own#);
          printtext(#array declaration#);
          new line;
          printtext(#Error in bound number#);
          print(1,5,2,stack[first free - 2]);
          new line
        end
else go to parameter value; Comment on page 43
hot point:=stack[first free - 1];
first free:=first - 13;
go to alarm1; comment on page 36;
```

```
switch alarm:  printtext(#Error in switch [illegible] designator#);

new line;

hot point:=stack[first free - 1];

first free:=first free - 1;

go to alarm1; comment on page 36;
```

Representation of labels

A label is stored in the stack as a pair of addresses:
1) Mark: the value of stack reference at time of entry into the block in which the label is local.
2) Goal: the machine address of the first instruction representing the statement where the label is stored.

Representation of go to statements:

```
Running code: CODE FOR value:="goal and mark";
                      call address:=uu;
                  uu: go to go to;


go to:  if goal part(value) = 0 then go to instruction[call address + 1];
Q:  if mark part(value) ≠ stack reference then
      begin
        DECREASE LEVEL;
        go to Q
      end;
go to instruction[goal part(value)];
```

Representation of switch designator in running code:

Occurrence:	Actual Parameter	In expression
e.g.	,S[expression],	.., then S[expression] else ...

	CODE FOR subscript:=expression;
	first address of table:=stack[S + 1];
	number of entries:=stack[S + 2];
	call address:=W12;
	W12: go to take value of switch designator;	

	W12: go to exit from parameter expression	W12:

```
take value of switch designator: if subscript ≤ 0 ∨ subscript > number
  of entries then

     begin

       value:=0;

       go to instruction[call address + 1]

     end; comment this case is the undefined switch designator. See section
       4.3.5 of the ALGOL Report;

address of expression:=3 x (subscript - 1) + first address of table;

if kind(stack[address of expression]) = label then

     begin

       value:=stack[stack[address of expression + 1]];

       go to instruction[call address + 1]

     end;

comment We are now left with the case where kind(stack address of expression)
= designational expression;

stack[first free]:=call address;

first free:=first free + 1;

go to instruction[stack[address of expression + 1]];
```

Representation of assignment statements when the left-part list is more than one identifier or ~~xxxxxxxx~~ includes a formal identifier or a subscripted variable.

The addresses to which the value is to be assigned are assumed to be in temporaries in the stack. These temporaries have the following form:

```
        TP  value  address
```

and the last temporary of the group has the form:

```
        MJO       FILL.
```

Then the contents of the temporaries form a complete subroutine performing the assignment in the following manner:

```
        temp0:  TP  value  address1
        temp1:  TP value   address2
          .           .
          .           .
          .           .
        tempn:  TP  value  addressn
        temp(n + 1):  MJ0      FILL.
```

Then the action to be taken by the running code after evaluation of the expression is of the form:

```
        RJ  temp(n + 1)  temp0.
```

Number output

We will write the value of any ~~number~~ expression to be output as N and the resulting number printed as P_N.

Any P_N will be in the form of a mantissa and a decimal exponent, the latter being an integer. The format of P_N is described by three parameters, i,d,e:

i specifies the number of digits of the mantissa before the decimal pt.
d specifies the number of digits of the mantissa after the decimal pt.
e specifies the number of digits of the exponent.

Thus P_N is in the following form:
(sign of mantissa)(i digits)(decimal point if d≠0)(d digits)(sign of exponent if e≠0)(e digits)

The three parameters are written in the output statement:

print(i, d, e, <expression(s) to be output>)

Leading zeros of P_N are suppressed except that the integer zero as a mantissa will be output as '0'. Plus signs are printed as spaces. The mantissa is rounded to make it correct to its last digit.

When e≠0 it is evident that N, i, e, and d do not uniquely determin P_N:
If N=-6, i=3, d=1, e=1, then P_N=
(a)-600.0-2
or (b)-␣␣␣.6␣1
or (c)-␣␣6.0␣␣
etc. where ␣ indicates a space

In this case the print routine determines the format of P_N by placing the first significant digit as far to the left as possible, subject to the restrictions on the value of the exponent imposed by the fixing of e. Hence, in our example P_N=(a). If d and e are too small, it may be that no significant digits are output in the mantissa. (e.g., if N=.7x10^{-11}, i=2, d=1, e=1, P_N would be ␣␣␣.0-9. In this example when the exponent assmues its least possible value the rounded mantissa is still less than 0.1.)

Alarm printing

It may be that i and e are too small to represent a large number (e.g., N=10^{14}, i=3, e=0 or 1). In this case e will be automatically increased by 1 until the most significant digit of N can be placed in the leftmost position of P_N. An error indication ('e') is given each time e is increased by 1 (In the above example with e=0, P_N=ee␣100␣12).

Examples

N	i	d	e	P_N
70.4	3	1	0	␣␣70.4
70.4	1	2	0	e␣7.04␣1
.008	1	4	0	␣␣.0080
.008	2	0	0	␣␣0
.008	2	0	1	␣80-4
.9999	1	3	0	␣1.000
.9999	0	2	0	e␣.10␣1
0	2	2	2	␣␣␣.00␣␣

Number output

Non-local quantities
    address of actual
    call address
    address of formal
    first free
    regular return
    last return
    j
    specification
    parameter treatment

Local quantities
    label reentry, print zero, conversion, Q, SS, print finished, S,
        next value, skip, reentry exponent, OPT;
    integer i, d, e, signum, e2, e10, exponent, max exp, number of digits,
        number of zeros, digits before point, k;
    real number, f;
    boolean only spaces yet, exponent part;

Procedure print(a, ...,); comment this procedure will print the values of any number of expressions supplied as parameters. The three first parameters should be non-negative integers defining the digit layout as follows: i, the number of digits before the decimal point, d, the number of decimals, e, the number of exponent digits;

```
begin

address of actual:=call address + 1;

address of formal:=first free;

first free:=first free + 8;

regular reutrn:=SS;

last return:=print finished;

j:=0;

S:  specification:="integer value";

go to parameter treatment;

SS:  if j<2  then

       begin

         j:=j+1;

         go to S

       end;

regular return:=SSS;

next value:  specification:="real value";

go to parameter treatment;

SSS:  i:=stack[first free - 8];

d:=stack[first free - 6];

e:=stack[first free - 4];

number:=staak[first free - 2];

if number = nonsense then go to skip;

signum:=sign(number);

f:=normalized binary fraction(abs(number));

e2:=normalized binary exponent(abs(number));
```

```
comment The layout is defined by i, d, e,. The number is n in the form
n = f x 2↑e2        ½≤f<1.
This is first rewritten in the form
n = f x 10↑e'10 x 2↑e'2  where 0≥e2≥-3  and then in the form
n = f  x 10↑e10 where 0.1≤ f <1  as follows;
reentry:  exponent part:=false;
reentry exponent:  if f = 0 then go to print zero;
e10:=0;
conversion:  if e2>0 then
      begin
        e10:=e10+1;
        e2:=e2 - 3;
        f:=0.8 x f
      end
else if e2≤-4  then
      begin
        e10.=e10 - 1;
        e2:=e2 + 4;
        f:= (10/16) x f
      end
    else go to final adjustment;
if f<0.5 then
      begin
        f:=2 x f;
        e2:=e2 - 1
      end;
go to conversion;
```

```
final adjustment:  f:=f x 2↑e2;

if f<0.1 then

    begin

      f:= 10 x f;

      e10:=e10 - 1

    end;
```

comment Our object is ~~that~~ to begin the printed number ~~will begin~~ with a non-zero digit and to set the exponent accordingly. If this is not possible due to the exponent exceeding its maximum possible value we have an error. In this case an error indication('e') is given and the output format is adjusted

(e:=e+1) until the number can be output satisfactorily. If on the other hand the exponent would be less than its minimum possible value we "right shift" the number, i.e. introduce leading zeros be reducing "number of digits", until this is remedied or we are left with all zeros;

```
A:  exponent:=e10 - i;

P:  max exp:=10↑e - 1; comment form table;

if abs(exponent)≤ max exp then

  number of digits:=i + d

else if exponent <0 then

    begin

      number of digits:=i + d + max exp + exponent;

      if number of digits  <0 then

        print zero:

         begin

           ~~number of digits:=0;~~

           ~~exponent:=0;~~ exponent:=0;

           number of zeros:=i + d;

           go to OPT

         end

      else

        exponent:=-max exp
```

```
        end
      else
        begin
          output(#e#);
          e:=e + 1;
          go to P; comment this is the case of alarm printing;
        end;
number of zeros:=i + d - number of digits;
f:=f + 0.5 x 10^(number of digits); comment rounding;
if f≥1 then
        begin
          f:=0.1; comment a small tenth;
          e10:=e10 + 1;
          go to Q
        end overflow on rounding;
comment We now output the number. Leading zeros are suppressed except
the the integer zero if appearing as the mantissa is output as '0';
OPT. output(signum);
digits before point:=i;
only spaces yet:=true;
for k:=1 step 1 until i + d do
        begin
          if digits before point = 0 then
            begin
              output(#.#);
              only spaces yet:=false
            end;
          digits before point:=digits before point - 1;
```

```
        if number of zeros>0 then

           begin

             number of zeros:=number of zeroes - 1;

             output(if only spaces yet ∧(d ≠ 0 ∨ digits before point ≠
                0 ∨exponent part) then #␣# else #0#)

           end

        else

           begin

             f:=10 x f;

             output(entier(f));

             f:=f - entier(f)

           end

      end;

if e >0 then

      begin comment We now set up the exponent in the form f x 2↑e2 where
                    0.5≤ f<1 and return to the start of the conversion
        d:=0;       and output routine;

        i:=e;

        e:=0;

        f:=normalize(abs(exponent));

        e2:=power of 2(abs(exponent));

        comment Even if exponent is zero, the routine will be run in order
        to print the proper number of spaces;

        signum:=sign(exponent);

        exponent part:=true;

        go to reentry exponent

      end

skip:  address of formal:=address of formal - 2;

go to next value;
```

print finished: first free:=first free - 8;

go **to** instruction[address of actual + 1];

Output tape handler

Local variables
   character counter
   word counter
   blockette counter
   core index
   core[0:119] - psuedo-buffer
   line is full

Input parameter - symbol

```
initialize:  CR:=true;

go to initialize2;

final dump:  blockette counter:=5;

symbol:="carriage return";

output: if symbol = carriage return then

      begin

        if line is full then

           begin

             line is full:=false;

             go to instruction[call address + 1];

           end;

        CR:=true;

        go to end of line

      end;
CR:=false;
if line is full then go to overflow;

word:=word + symbol x 64↑(5 - character counter);

if character counter<5 then

   character counter:=character counter + 1

else

   end of line:

      begin
```

```
        core[core index]:=word;
        if word counter<19 ∧ ¬CR then
           begin
             word counter:=word counter + 1;
             core index:=core index + 1
           end
        else
           begin
             if blockette counter<5 then
               begin
                 blockette counter:=blockette counter + 1;
                 core index:=20 x blockette counter
               end
             else
               begin
                 TRANSFER TO BUFFER;
                 WRITE ON TAPE;
initialize2:     for k:=0 step 1 until 119 do
                    core[k]:="6 spaces";
                 blockette counter:=core index:=0
               end;
             line if full:=¬CR;
             word counter:=0
           end;
         character counter:=word:=0
       end;
   go to instruction[call address + 1];
```

"Song of the Daskerkopi"

'Twas kopi and the skrvy sluts
Did tæ and tryk in the klar;
All strengy were the læ sstreng,
And the tryktoms spild exp........

("Song of the Habberwocky" by Lewis Carroll, translated into Danish(?)
by Curt Outlaw, University of North Carolina, August 24, 1961)