Gier Algol Translator
General Pass Mechanism
3. May 1962.

THE TRANSLATION PROBLEM IN MACHINES WITH A NON-HOMOGENEOUS STORE.

The translation of an Algol program consists basically in the combination of two bodies of information, viz. the information supplied by the Algol source program and that of the translation program (the translator). Unless the machine has a sufficiently large fast homogeneous store to hold both of these bodies simultaneously some division of these bodies becomes necessary. Two ways of making this division suggest themselves: In the one we attempt to perform a purely sequential scan through the source program, taking a small section at a time. In the other we divide the translator into several functional subunits and let these operate on the complete source programs or the partially translated intermediate versions of it in turn. This latter method is then a multiscan process.

In a machine which has a sufficiently large fast store to hold the complete translator and its tables the sequential method is clearly preferable. However, if the fast store is not large enough to hold the translator the multiscan process is better. This is because in a sequential process the order in which references are made to the various parts of the translator is unknown, since this order depends on the source program. Therefore, in a sequential process, we are forced to make a simultaneous linear scan of the source program and almost random references to all parts of the translator. In a multiscan process, on the other hand, the part of the translator used during a particular scan is fixed and the only reference to a large body of information is the purely linear scan through the partially translated program. For these reasons the Gier Algol Translator will use a multiscan method.

It is interesting to note that on Gier this choice is entirely compatible with a high translation speed. In fact, the time for transferring a program occupying 150 drum tracks into the core store and back again, as it will be necessary during each pass, takes only about 6 seconds. If 10 passes are necessary the total drum transfer time will thus be about 1 minute for this rather large program. It must be expected that the actual translation process will require considerably more time than this, and thus the simultaneous drum transfer facility in Gier will work to our full advantage.

INTERMEDIATE LANGUAGES.

In a multipass translator the choice of intermediate languages becomes of prime importance. The following conflicting factors have to be considered: From the point of view of the individual scans (or translator "passes") great flexibility and a great multitude of structures of varying length is desirable. From the point of view of the programs which have to perform the packing and unpacking of information, which is necessary during each scan, uniformity of structure is desirable. Uniformity is also desirable for translator checking purposes.

In view of these considerations the following compromize has been adopted: All intermediate languages are expressed in terms of basic units of information each of 10 bits. This unit is called a "byte". Thus the input to each pass and the output from it will always consist of a uniform string of bytes. Within each intermediate language any amount of structure within this byte string may be employed in order to communicate more intricate structures. Thus, for example, a number in the original Algol program will in most of the intermediate languages be represented by 5

"number mark" while the remaining four supply the actual value of the number. However, from the point of view of the general packing and unpacking program the various intermediate languages are undistinguishable, since they all consist of a uniform byte string.

By this method the programs which will perform the packing and unpacking of the intermediate versions of the program and which will take them from the drum and store them on the drum will be the same for all passes. In addition since the output from each pass is always of the same form the same test output program can be used for all passes and can be coupled to the common pass administration. This will probably facilitate the checking of the translator greatly.

REVERSE SCANS AND ERROR REMOVAL.

In a multipass translator it is highly advantageous to let some of the scans be reverse scans, i.e. scans which start at the end of the program and move toward the begin. First of all the problem of forward references clearly is solved comple in this manner. In addition it becomes easy to eliminate syntactically incorrec sections of the program during the translation. This is important because it is highly desirable that an error which has been detected does not prevent the translation process from continuing to check the rest of the program. The way of doing this in the Gier Algol translator is as follows: During the syntactical check of the program (pass 3) a special byte "statement start" is output at every point whe an Algol statement starts in the text. If a syntactical error is found in a statement a special byte "trouble" is output and the rest of the symbols of the stateme up till the first following semicolon (;) or <u>end</u> are skipped completely. The follo scan is a backward scan. Every time this scan finds the byte "trouble" it will ski all bytes back to the first "statement start" point. In this way the complete incorrect text is removed. At the same time various parameters describing the state of the translation are reset to appropriate values to enable the translation to continue to translate the following statement correctly.

ERROR SIGNALING.

The translator includes an extensive checking of the formal correctness of the source program. Every time an error is found an error description will be produced in the output from the translator. This will include the location and kind of the error. The location will be indicated by the number of the line in the original Algol test where it occurs. In order to facilitate the identification of lines the first pass of the translation will produce a copy of every 10'th line of the program in its output, with the line number inserted at the beginning of the line. The kind of the error will be described by an appropriate error message.

STORAGE OF THE PARTIALLY TRANSLATED PROGRAM.

According to the description given above all passes use the same general pass administration. As far as the individual pass programs are concerned the general pass administration is a subroutine with two entries, one for input and one for out In fact these two entries are almost completely independent since they use two independent buffers in their communication with the drum. The general pass administration uses 4 sections of 40 words each in the core store. Two of these are used as buffers for the input to the pass programs, the other two for the output from the pass program. Normally one of the input buffers holds the bytes of that sectio of the program which is presently being processed by the pass program while the other input buffer holds the next input drum track. When all the bytes on the active buffer have been used the transfer from drum of the next following drum track is initiated. At the same time the pass program can proceed to process the input bytes waiting in the other input buffer. A simila buffering technique is used on the output side.

The bytes are packed into the Gier words with 4 bytes in a word. Thus one drum track holds 160 bytes. The unpacking (on the input side) and packing (on the output side) of the bytes into the words are performed by the general pass administration. This turns out to be a more time consuming process than the corresponding drum tran fers. In fact the average unpacking time per byte is about 220 microseconds, which means that the unpacking of the 160 bytes on a track will take about 35 millisecond This again means that except for collisions between drum transfers called from the input and output side the time for drum transfers will be negligible, owing to the parallel operation during drum transfers in Gier. The time for packing on the output side being about 260 microseconds per byte, the total time for the administration of a pass becomes about 77 milliseconds per drum track plus the time wasted due to collisions of drum transfers, about 5 milliseconds on the average. The processing time per byte of course varies greatly, the minimum being about 80 microseconds (direct copying from input to output) while a normal figure of perhaps 500 microseconds may be expected. If this holds the pass administration and processing times are comparable and the total time for 10 passes will be about 2 seconds per drum track.

On the drum the tracks holding the partially translated program are used in a cyclic manner. Probably about half the drum will be available for this, while the rest of the drum will hold the translator and the running system programs. Suppose that the available tracks are numbered from 1 to N. The first pass will then place its output in tracks no. 1 to M, say, where clearly $M \leq N$. The second pass will take its input from tracks no. 1, 2, etc. and will place its output in tracks M+1, M+2, etc. When output to track no. N has been made the administration will continue to output into track no. 1, 2, etc. which presumably have now been release by the input side administration. This process is continued smoothly from one pass to the next, except for the case that the direction of scanning is reversed. This does not cause any difficulty however, since the cyclic use of the drum tracks may with equal ease take place in either direction. Only in the programs which perform t packing and unpacking a few changes are necessary.

MACHINE ARRANGEMENT OF THE GENERAL PASS ADMINISTRATION.

Algol descriptions of the general pass administrations are found below. The following notes show the storage of the translator on the drum and the use of the core store during translation.

Drum: 320 tracks (40 words each)

SLIP address	
0-31	SLIP (symbolic input program)
32- ?	Running system
1e14	Forward pass } see core store 16 - 55
2e14	Backward pass } see core store 16 - 55
3e14	testprint, print, new line, entry to message (1 track total)
4e14	endpass (1 track)
5e14	message (output program, 1 track)
6e14	texts for message (1 track)
7e14-?	Pass n } Translator pass programs
? - ?	Pass n-1 } Translator pass programs
	. . .
? - ?	Pass 1 } Translator pass programs
? - ?	Partially translated Algol program (tracknumbers FIRST to LAST)
LAST+1 - 319	Table of strings, formed during pass 1

Core store: 1024 words

Absolute address	
0-15	Reserved for SLIP
16-54	Drum and packing administration of partially translated program (forward or backward).
55	Base in = e13
56-95	Input buffer 1 } Also used for texts by messag
96	Buffer stop } Also used for texts by messag
97-136	Input buffer 2 } Also used for texts by messag
137	Base out
138-177	Output buffer 1 } Also used for message
178	Buffer stop } Also used for message
179-218	Output buffer 2 } Also used for message
219	Buffer stop
220-259	testprint, print, new line, entry to message, universal translator parameters
261- 999	Translator pass programs and their tables
960-999	Also used for endpass
1000 -1023	Reserved for SLIP

Taking the tracks of the drum in order, this gives a general survey of the meaning of the various parts:

SLIP is the input program used for reading the translator into the machine. When the translator is completed this will of course become unnecessary and the complete translator be shifted forward on the drum.

The running system is the set of administrative programs and standard procedures used by the running Algol program, when the translation is completed. See Gier Algol Running System.

Forward pass and backward pass are the programs performing the packing, unpacking and drum transfers described above in the section on STORAGE OF THE PARTIALL TRANSLATED PROGRAM. One of these tracks will be placed from 16 to 54 in the cores. Two separate programs have been written in order to make sure that this central process runs at the highest possible speed. The pass program appropriate to each separate pass is transferred to the cores by endpass.

The track holding the programs testprint, print, new line, and entry to message, in addition to certain universal translator parameters, is permanently held in the core store 220-259. Entry to message is used for transferring the message printing program from drum. The message program will be placed in that one of the two output buffers which is not currently being used for collecting output. This output buffer will always have been transferred to drum before it is overwritten by message and therefore need not be saved. On exit message will only have to make sure that the output buffer is left with the correct marks on all words.

The next track holds the program endpass which is used to perform the transition from one pass to the next. When used this program is transferred to 960-999 in the core store. Endpass does the following: 1) The last output buffer is filled up with dummy bytes until it is transferred to the drum. 2) Information about the pass which has just been completed is printed. This will always include the number of used tracks. 3) According to a table held by endpass itself the direction of scan is reversed if necessary, the appropriate pass program (forward or backward) is transferred to 16-54 in the cores, and the program for the pass itself is transferred to locations 261 and following.

The message program is a text printing program. As described above it will be placed in one of the two output buffers when it is needed. It starts by transferring a track of text information from the following track to that one of the two input buffers which has last been transferred from the drum and which has therefore not yet been processed. At the end of its work the message program must restore the input track which was overwritten by the text track.

The next section of the drum holds the translation programs proper. This is followed by the working section of the drum (see the section on STORAGE OF THE PARTIALLY TRANSLATED PROGRAM above).

The very top end of the drum is used for holding a table of strings in the Algol program. This is formed during the very first pass and will reserved as mumch of the working section of the drum as it needs. This is possible during pass 1 because the cyclic use of the working tracks has not yet had the chance of moving to the top of store (unless the program is too big for the machine).

ALGOL PROGRAMS FOR THE PASS ADMINISTRATIONS.

The following Algol descriptions follow the machine codes sufficiently closely to be of help in deciphering these. It should be noted, first of all, that the input and output procedures are highly "sneaky" since they use and change a number of non-local variables. These variables are:

```
integer output word address, output byte number, number of used tracks, number of
        available tracks, output track, last track, input word address, byte address,
        input track;
boolean in testmode;
integer array BYTE BUFFER[1:5];
integer array WORD BUFFER[0:164];
```

The WORD BUFFER is the section in the core store from 55 to 219 (page 4). The five words placed in between and at the ends of the buffers proper are used to control the counting through their marks in Gier. The marks placed on the words in the WORD BUFFER are as follows:

		Marks	Use of buffer
WORD BUFFER	0	3	
	1-40	0	Input 1
	41	2	
	42-81	0	Input 2
	82	3	
	83-122	0	Output 1
	123	2	
	124-163	0	Output 2
	164	3	

In addition to the non-local variables a number of non-local procedures are used. Hopefully those of them which are not declared may be understood from their identifie

```
procedure output(byte); value byte; integer byte; comment This performs the packing
    and drum transfers of the output from each pass in turn. The algorithm given
works only for a forward pass. The one for backward passes is so similar that it
will not be reproduced. For further notes, see the section on STORAGE OF THE PAR-
TIALLY TRANSLATED PROGRAM on page 3;
begin switch packing := pack first, pack second, pack third, pack fourth;
    if in testmode then testprint(byte);
    output byte number := output byte number + 1;
    go to packing[output byte number];
pack first: output word address := output word address + 1;
    WORD BUFFER[output word address] := byte;
    go to output done;
pack second: WORD BUFFER[output word address] :=
             WORD BUFFER[output word address] × 2↑10 + byte;
    go to output done;
pack third:  WORD BUFFER[output word address] :=
             WORD BUFFER[output word address] × 2↑10 + byte;
    go to output done;
```

```
pack fourth: WORD BUFFER[output word address] :=
             WORD BUFFER[output word address]×2↑10 + byte;
     output byte number := 0;
     if marks of(WORD BUFFER output word address ) > 0 then
          begin number of used tracks := number of used tracks + 1;
                 if number of used tracks > number of available tracks
                      then alarm(<<program overflow>, 'stop');
                 byte := WORD BUFFER[output word address];
                 output track := output track + (if output track = lasttrack
                                        then 1 - available tracks
                                        else 1);
                 TRANSFER TO DRUM(output track)from:(output word address - 40);
                 if marks = 3 then output word address := output word address - 81;
                 go to          if this is last then exit from pass else pack first
          end;
output done:
end output;
```

integer procedure input; comment This performs the drum transfers and unpacking used for input to all passes except the first which reads the paper tape. In the machine code this is coded as an open subroutine of the following 2 long orders:

```
                ARS (el) t+1     or     PM: (el) t+1
                HS   e2  LA             HS   e2  LA;
begin byte address := byte address + 1;
      R := BYTE BUFFER[byte address];
      if R = "nonsensebyte" then UNPACK BYTES;
      input := R
end input;
```

procedure UNPACK BYTES; comment This is called every time an input word must be unpacked, i.e. once for every 4 bytes input by input;

```
begin integer marks;
      input word address := input word address + 1;
      marks := marks of(WORD BUFFER[input word address]);
      if marks > 0 then
          begin input track := input track +
                  (if input track = last track then 1-available tracks else 1);
                TRANSFER FROM DRUM(input track)to:(input word address - 40);
                if marks = 3 then input word address := input word address - 81;
                number of used tracks := number of used tracks - 1
          end;
      word := WORD BUFFER[input word address];
      R := BYTE BUFFER[1] := first part(word);
      BYTE BUFFER[2] := second part(word);
      BYTE BUFFER[3] := third part(word);
      BYTE BUFFER[4] := fourth part (word);
      comment BYTE BUFFER[5] permanently holds the value "nonsensebyte";
      byte address := 1
end UNPACK BYTES;
```

```
procedure TRANSFER TO DRUM(track number)from:(buffer location); code;
procedure TRANSFER FROM DRUM(track number)to:(buffer location); code;
comment These procedures transfer the 40 words of a track to or from the 40 words
held in the WORD BUFFER from WORD BUFFER[buffer location] and onwards;
```

The following procedures use some further non-local parameters:

```
integer rest of line, pass number, CRcounter, information 1, information 2, pass number;
boolean first print in pass, no running, [illegible] this is last;

procedure testprint(n); value n; integer n;
begin rest of line := rest of line - 1;
      if rest of line = 0 then new line;
      skrv(⟨dddd⟩, n, skrvml(2));
end testprint;

procedure print(n); value n; integer n;
begin rest of line := 0;
      skrv(⟨dddd⟩, n, skrvml(2));
end print;

procedure new line;
begin skrvvr; skrvvr;
      if first print in pass then
         begin first print in pass := false;
               skrv(⟨d⟩, pass number);
               skrvtekst(⟨<.,⟩);
         end
      else skrvml(3);
      rest of line := 10
end new line;

procedure message(n, kind); value n, kind; integer n, kind;
begin boolean give up;
      switch action := hopeless, serious error, error, line number, no line number;
      TRANSFER FROM DRUM("message track")to:(if input word address > 41 then 1
                                                                       else 42);
      new line;
      go to action kind ;
hopeless: give up := true;
serious error: no running := true;
      red output;
error: printtext( 'error');
line number: printtext( 'line');
      print(CRcounter);
no line number:          printtext(n); comment printtext is a procedure which prints
      that text in the text list which has the number given as parameter;
      TRANSFER FROM DRUM(track in)to:(if input word address > 41 then 1 else 42);
      black output;
wait: if drum transfer in progress then go to wait;
      if give up then stop
end message;
```

```
end pass:
      output(0);
      working boolean := in testmode;
      in testmode := false;
L:    output(0); output(0); comment This fills the remainder of the output track
      until it has been transferred to drum. output jumps to "exit from pass"
      when a drum transfer has been completed and "this is last" is true;
      this is last := true; go to L;

exit from pass: new line;
      print(used tracks);
      if information 1 ≠ 0 then print (information 1);
      if information 2 ≠ 0 then print (information 2);
      pass number := pass number + 1;
      in testmode := working boolean;
      if in testmode then wait;
      first print in pass := true;
      marks := marks of(passinformation[pass number]);
      R := pass information[pass number];
      comment The table "pass information" tells whether the direction of scan should
      be reversed (marks > 2). Also four addresses are given telling where to find
      the new pass program on the drum and where to enter into it;
      if marks > 1 then exchange (input track, output track);
      TRANSFER FROM DRUM(input track)to:(1);

begin integer track, store, first track, exit;
      track := part 1(R);
      first track := pa rt 2(R);
      store := part 3(R);
      exit := part 4(R);
      TRANSFER FROM DRUM(if marks = 0 ∨ marks = 2 then forward track else backward
                              track)to:(pass mechanism);
more: track := track - 1; store := store - 40;
      TRANSFER FROM DRUM(track)to:(store);
      if track > first track then go to more;
L:    if input = 0 then go to L; comment This eliminates the filler zeroes;
      byte address := byte address - 1; comment In this way the last byte will
          be repeated;
      go to instruction[exit];
end;
```

SYMBOLIC NAMES FOR THE WHOLE TRANSLATOR (e-NAMES).

Entry points and parameters held as addresses in instructions:

e1	byte address
e2	inaddress (input word address and entry to UNPACK BYTES)
e3	output
e4	(see below)
e5	entry to message
e6	testprint
e7	newline and print
e8	newline
e9	print
e10	
e11	
e12	outaddress (output word address)

General parameters for translator (are stored at the end of the track holding print etc. and are initialized to the values indicated when reading that track from drum). e4 must be defined right at beginning of loading.

	holds	initial value
e4	CRcounter	QQ 0 t1
1e4	number of used tracks	QQ 0
2e4	information 1 for output of	QQ 0
3e4	information 2 statistics	QQ 0
4e4	last track - 1	QQ e20 - 1
5e4	available tracks - 1	QQ e20 - e19
6e4	input track	QQ e19
7e4	output track	QQ e19
8e4	no running (>0 = <u>false</u>)	QQ 1
9e4	pass number	QQ 1
10e4	first track + 1	QQ e19 + 1

INTRODUCTION.

The analysis of expressions and conversion to final machine form starts in pass 6. In this pass expressions are converted into so-called inverse Polish form. In this form the operations are chosen to conform to those available in the order code of the machine, but no account is taken of the use of the available machine registers. In pass 7 this string is converted into an operation string which refers directly to machine registers.

THE FORM AND MEANING OF THE INVERSE POLISH NOTATION.

In the inverse Polish notation expressions consist of a string of operands (identifiers etc.) and operators (+ - / > etc.). However, parentheses have been eliminated. Examples:

a b + c d - /

a ¬ b c > ∧

The meaning of an inverse Polish string, i.e. the rules for evaluating the result of the expression, assumes the existence of a <u>stack</u> for holding the actual values of operands, including intermediate results. Let us denote this as follows:

<u>array</u> OPERAND STACK[1:some unknown upper limit]

The Polish string never refers directly to a given location in the OPERAND STACK. Rather the references are implied in the structure of the Polish string itself. Thus the intermediate results of the evaluation are anonymous.

Now the evaluatinn can be described as follows: Proceed through the string from left to right in a strictly sequential fashion. When encountering an operand place this at the top of the OPERAND STACK by performing the following operations:

last used := last used + 1;
OPERAND STACK[last used] := value(operand);

When encountering an operator, perform the corresponding operation on the values found at the top of the stack and place the result of the operation also at the top of the stack. The exact action depends on the nature of the operator. Unary operators ("negative", ¬ and others to be introduced later) do not change "last used". Example: ¬ produces the following action:

OPERAND STACK[last used] := ¬ OPERAND STACK [last used]

Binary operators (+ / > ∧ etc.) remove one item from the top of the stack. Example: / produces the following action:

last used := last used - 1;
OPERAND STACK[last used] := OPERAND STACK[last used] / OPERAND STACK [last used + 1]

As an illustration a step-by-step evaluation of the two above examples will be presented. In addition to the values held in the OPERAND STACK the mathematical expression for these values are given. It should be carefully noted, however, that the OPERAND STACK can only hold values (numbers, logical values) and the expressions given are only for the reader's convenience.

Example of evaluation: a b + c d - /
Assume a=1, b=2, c=3, d=4.

Input symbol	a	b	+	c	d	-	/
After processing of input symbol: OPERAND STACK [1]	1(a)	1 (a)	3 (a+b)	3 (a+b)	3 (a+b)	3 (a+b)	-3 ((a+b)/(c-d))
[2]		2 (b)		3 (c)	3 (c)	-1 (c-d)	
[3]					4 (d)		

Example 2: a ¬ b c > ∧
Assume a=<u>false</u>, b=2, c=3.

Input symbol	a	¬	b	c	>	∧
After processing OPERAND STACK [1]	<u>false</u> (a)	<u>true</u> (¬a)	<u>true</u> (¬a)	<u>true</u> (¬a)	<u>true</u> (¬a)	<u>false</u> (¬a∧b>c)
[2]			2 (b)	2 (b)	<u>false</u> (b>c)	
[3]				3 (c)		

CONVERSION OF ALGOL EXPRESSION INTO INVERSE POLISH FORM.

The conversion of a parenthesized ALGOL expression into inverse Polish form may be accomplished by means of a method which has been described by Dijkstra (APIC Bulletin no. 7, May 1961 and ALGOL Bulletin Supplement no. 10: Making a Translator for ALGOL 60). This method makes use of priority numbers associated with the operators and parentheses as follows:

Priority number	Delimiter
0	<u>begin</u> [(<u>if</u> <u>for</u>
1	<u>end</u>]) <u>then</u> <u>else</u>
2	:=
3	≡
4	⊃
5	∨
6	∧
7	¬
8	< ≤ = ≥ > ≠
9	+ -
10	"negative" × /
11	↑

The method works briefly as follows: The ALGOL expression is scanned from left to right. Operands are immediately transmitted to the output. Operators and left parentheses are entered into an OPERATOR STACK. This will at any one time hold those operators which have already occurred in the input but which have not yet been intered in the output (the Polish string) because the quantity on which the operator will operate has not yet been completed. Each new operator encountered in the input is compared with the operator waiting at the top of the OPERATOR STACK. If the priority of the operator waiting in the stack is higher than that of the new operator the operator waiting is removed from the stack and sent to

the output. This comparison of priorities is repeated until no operator having a higher priority than the new operator is found in the OPERATOR STACK. Then the new operator is placed at the top of the stack. Left parantheses are treated like other operators. Right parentheses (incl. <u>end</u>] ")" <u>then</u> <u>else</u>) on the other hand will not be placed in the OPERATOR STACK, but will remove the corresponding left parenthesis.

Example of conversion from ALGOL to inverse Polish notation: ((a+b)/(c-d))

Input	(	(	a	+	b	)	/	(	c	-	d	)	)
After processing:													
OPERATOR STACK [1]	(	(	(	(	(	(	(	(	(	(	(	(	
[2]		(	(	(	(		/	/	/	/	/	/	
[3]				+	+			(	(	(	(		
[4]										-	-		
Output (Polish)			a		b	+			c		d	-	/

A more complete discussion of the operators used in Gier Algol will be given below.

TYPE CHECKING.

The above conversion algorithm may very conveniently be extended with facilities for a complete type checking and a detection of types of the operands for each occurrence of the operators. This latter is of interest particularly for the power operator because a distinction is needed between integer and real exponents.

For this present purpose we perform a pseudo evaluation of the expression at the same time as it is generated in the inverse Polish form. This pseudo evaluation will of course not involve actual values but will only operate with the types of the values. This, however, is exactly what is necessary to perform a type checking. In order to illustrate this approach we give another example of a conversion which includes the behavior of the STACK FOR TYPE OF OPERAND:

Example of conversion with type development: ($\neg$ a $\wedge$ b > c), <u>Boolean</u> a; <u>integer</u> b; <u>real</u> c;

Input:	(	$\neg$	a	$\wedge$	b	>	c	)	
After processing									
OPERATOR STACK [1]	(	(	(	(	(	(	(		
[2]		$\neg$	$\neg$	$\wedge$	$\wedge$	$\wedge$	$\wedge$		
[3]						>	>		
Output (Polish)			a	$\neg$	b		c	>	$\wedge$
STACK FOR TYPE OF OPERAND [1]			Boo	Boo	Boo	Boo	Boo	Boo	Boo
[2]					int	int	int	Boo	
[3]							real		

Note that the final) produces the output of two operators > and $\wedge$.

By this method it is possible to check that the types of operands are consistent with the operators operating on them. The principal consistency rules may be stated as follows:

Operator	Required type of operands	Yield results of type
Arithmetic	real integer	real integer
Relational	real integer	Boolean
Boolean	Boolean	Boolean

For binary operators the handling of the STACK FOR TYPE OF OPERAND may be modified in the following manner. Instead of imitating the work of the OPERAND STACK exactly it is possible to extract and check the type information concerning the **first** operand belonging to a binary operator at the time when this operator is entered into the OPERATOR STACK. The advantage of this is that we do not have to check the consistency of an operator and both of its operands at the same time. A disadvantage is that in the case of the arithmetic operators, + - × / ↑ we have to attach the type of the first operand to that operator which is entered into the OPERATOR STACK. Thus for example there will be two possible + operators in the OPERATOR STACK, one "integer+" and the other "real+". The following example shows the use of this method:

Example of conversion to Polish notation with modified type handling.
integer i, j, k, n, m; **real** a;
((i + a)↑(j + k × n ↑ m))

Input:	(	(	i	+	a	)	↑	(	j	+	k	×	n	↑	m	)	)
After processing:																	
OPERATOR STACK [1]	(	(	(	(	(	(	(	(	(	(	(	(	(	(	(	(	
[2]		(	(	(	(		r↑	r↑	r↑	r↑	r↑	r↑	r↑	r↑	r↑	r↑	
[3]				i+	i+			(	(	(	(	(	(	(	(		
[4]										i+	i+	i+	i+	i+	i+		
[5]												i×	i×	i×	i×		
[6]														i↑	i↑		
Output (Polish)			i		a	+			j		k		n		m	"↑i" × +	"↑i"
TYPE CHECK STACK [1]			in		re	re			in		in		in		in	in	re

Note that in this example the type of n↑m is taken to be integer. Further that it is assumed that there exist distinct operators "↑i" (power to integer exponent) and "↑r" (power to real exponent). The details of the treatment of arithmetic types will be discussed below.

Gier Algol Running System
26. Jan. 1962

THE GIER ALGOL RUNNING SYSTEM.

The construction of an Algol system may be divided into 2 major sections:

1. The running system (i.e. the programs which are used by the running Algol code).
2. The translator (i.e. the program for transforming the source program into the form required by the running system).

It is now quite clear that the design of an Algol system should start with the design of the running system. The principal reason for this is that the translation process is not defined before its output has been defined.

The major problems of an Algol running system are the following:

1. Storage allocation.
2. Addressing.
3. Procedure entry.

The fourth major problem, dynamic own arrays, will be ignored here since it will not be included in Gier Algol.

Solutions of these problems, as used on DASK, have already been described (Jensen, Naur: An Implementation of ALGOL 60 Procedures, BIT Vol. 1 no. 1 (1961), Jensen, Mondrup, Naur: A Storage Allocation Scheme for ALGOL 60, BIT Vol. 1 no. 2 (1961) and Comm. ACM about Oct. 1961). However, owing to the differences in the characteristics of DASK and GIER the problems have been attacked anew.

The characteristics of Gier which are particularly relevant to the following discussion are:

The core memory is small (in DASK the wired store has effectively doubled the rapid access memory). The drum is divided into tracks of fixed length. Elaborate addressing facilities: relative, indirect, index, and subroutine marks. Built-in floating point operations. While a drum transfer is in progress other operations may be performed.

STORAGE ALLOCATION.

In designing the storage allocation the primary concern is the optimum use of the very limited core store. Further it must be considered a fact that at run time ~~rapid~~ frequent transitions from one part of the program to another, usually in a cyclic fashion, will dominate. As an illustration consider the following fragments of a program, showing the inner cycle of an iterative method for finding the root of an equation, and assume that the program is stored on the drum tracks shown to the right:

```
procedure Bisec 2 (F, . . . ); real F;                       Drum track
   begin . . .                                                ]
        for q := q/2 while abs(q) > eps do                    ]  6
                x := (if F > 0 then q else -q) + x            ]
   end;
. . . .
Bisec 2(s↑2 + (1 - (s + 1) *                                  ]  7
              ln((s + 1)/s) * chi2e2, . . . )                 ]  8
```

The for statement of this fragment will give rise to a cyclic execution of program parts placed on the following tracks (the tracks having the functions abs and ln have not been numbered):

6, abs, 6, 7, 8, ln, 8, 6, abs, . . .
(cycle)

From this and similar examples we conclude:

1. The time spent on a particular drum track is usually quite short compared with the time of a drum transfer.
2. The drum tracks needed in a particular cycle will contain widely scattered sections of the program from different blocks.

On the basis of this we further conclude that the previous solution of always having the complete program belonging to a particular block in the rapid store at the same time is poor.

The new attack is based on the following scheme:

1. The program is held completely on the drum. It is divided into sections which are completely independent, each section occupying a drum track. The independence of the sections will mean, in particular, that (a) the section may be executed from any place in the core store and (b) during execution the section will make no assumptions whatever with regard to what other sections may be present in the core store at the same time. This again means that whenever a reference from one section to another has to take place this must be handled by a central, permanent administration.
2. All variables are held in a stack (cf. Dijkstra: Recursive Programming, Num. Math. Vol. 2 (1960) 312-318). Note that this is primarily a storage saving measure. The recursiveness of procedures becomes an almost free extra facility.
3. All that part of the core store which is not used by the stack or by the administrative programs will be used to hold as many sections of the program as possible. Thus the amount of storage available for program in the core store as well as the sections held in this space will be entirely dependent on the development of the program. In other words, a close-to-optimum utilization of the core store is attempted.

The permanent administration required to handle this scheme will keep a table of the numbers of the drum tracks currently held in the available core space. Whenever a reference to another track is made from anywhere in the running program or in the administration this table will be searched to see whether the track is already in the core store. If it is not it must be transferred from the drum. The only remaining question is: where to put it? Or, rather: which of the currently available tracks should be destroyed in order to make room for the new track? In the examples and discussion below two different strategies for this choice are considered:

STRATEGY 1. The next track place to be used will be the one following the one to which a drum transfer has last been made, the word "following" understood in cyclic manner. Thus if there are 4 track places in the core store these will be filled from the drum in the following order: 1, 2, 3, 4, 1, 2, 3, 4, 1, . . . irrespective of the use which is being made of the available tracks in between two transfers from the drum.

STRATEGY 2. The next track place to be used is the one holding that track which at the moment the new track must be transferred has been left unused for the longest time.

In comparing these two strategies it should be kept in mind that clearly strategy 2 will require a table and more administration than strategy 1. Thus it will probably be realistic to compare the performance of strategy 1 having n+1 track spaces available with that of strategy 2 having only n track spaces.

It should further be noted that since a complete search of the table of the available tracks is made before a new track is transferred from drum it is clear that if there are n track spaces then any cycle of $p \leq n$ drum tracks can be held completely in the cores and no transfers from drum are necessary while this cycle is running. This again means that if strategy 1 leaves n+1 track spaces then a program of n+1 tracks will run without drum transfers with strategy 1, while some transfers will be necessary if strategy 2 with n track spaces is used. In other words there will be cases where strategy 1 will be better.

However this is counteracted by the advantages of strategy 2 in cases where the program has cycles which cannot be held completely in the available space. The following table shows the percentage of track transfers which do not require transfers from drum for some various track cycles.

Repeated track cycle	Percentage of control transfers which do not require drum transfers		
	Strategy 1 3 places	Strategy 1 4 places	Strategy 2 3 places
1, 2, 3, 4, 3, 2, ...	33	100	67
1, 2, 3, 4, 5, 4, 3, 2, ...	37	37	50
1, 2, 3, 4, 3, 5, 3, 2, ...	37	37	50
1, 2, 3, 4, 2, 3, 5, 2, 3, ...	44	44	67
1, 2, 1, 3, 4, 5, 4, ...	28	28	28

The last of these cases is the one used as illustration above.

The cases shown here have been chosen more or less at random, although with some thought to the kind of cycling which might occur frequently in Algol programs. They show that the choice of the strategy is by no means obvious, but that strategy 2 seems to offer appreciable gains at least in some cases.

The above approach works well for short track cycles, which can be held completely in the available track spaces, and for track cycles of medium length in which one or more tracks are called, repeatedly during the cycle. It is clear that nothing much can be achieved in the case of long track cycles. The remaining case is that in which the number of tracks in the cycle is slightly higher than the number of available track places and there are no or only insignificant reuse of tracks within the cycle. In this case the two above strategies will force a new drum transfer at every transition from one track to another while an appreciable gain might be achieved by reserving all track places but one for fixed tracks while using the last track space for all the remaining tracks of the cycle.

If situations of this kind are to be discovered it is clear that additional information about the track cycle must be collected. This suggestion is attractive in the present context because most of this extra work may be performed while the machine is waiting for a transfer from drum to be accomplished.

The following is a specific scheme for this purpose: In addition to the above administration a table of the frequency of use of tracks is kept. Since references to this table will be made only while the machine is waiting for a drum track to be transferred we make this table short, but inconvenient. It will consist of a fixed number of items, R say, where each item consists of a track number and a counter. The size of this table, R, will determine how long cycles we will try to handle in an improved manner. R should be greater than the average number of available track places. An upper bound for R will be imposed partly by storage capacity reasons, partly by the effect of "diminishing returns" when R becomes much larger than the number of available track spaces.

As the second part of this scheme a periodic analysis of the information in the frequency table is assumed. Basically it is assumed that this analysis will be performed so rarely that the complete program for doing the analysis is stored on drum. This suggests that the signal indicating that the analysis should be performed should depend primarily on a count of the number of actual transfers from drum.

We thus arrive at the following picture: Every time an actual transfer from drum is called the counter in the frequency table corresponding to this track is increased. When a certain total number of drum transfers has been performed the ordinary program action is interrupted, the special analysis program is called from drum and possibly a change of the parameters controlling the strategy is made.

This leaves the question of what to do if a drum track which has no entry in the frequency table is called, and the frequency table leaves no more room for new entries. The most likely answer to this seems to be that the complete table should be cleared and the counter of drum transfers reset to zero. In addition any reserved track spaces should be returned to the common pool. The reason for this is as follows: The fact that a track which does not appear in the frequency table is being transferred indicates one of two situations: (a) The track cycle has more different tracks in it than R. But this means that none of the available strategies will be of much help and we must just clear the situation to be ready to discover a new, shorter cycle as quickly as possible. (b) The program is passing from one shorter cycle into another. Again it is important not to let the obsolete information in the frequency table influence the transition into the new situation.

Finally we must decide what the periodic analysis should do. The available information is: 1) The frequency table: **integer array** hot track, frequency[1:R]; 2) T, the number of transitions between tracks which have not made a drum transfer necessary. In addition we know that the sum of the frequencies is equal to a fixed quantity, K, which is the number of actual drum transfers among a group of less than R+1 tracks which we are prepared to perform before we wish to perform the analysis. What we want to find out is whether any of the available track spaces should be used to hold certain tracks permanently. A simple and safe test for this may be performed as follows: If we fix the positions of certain tracks we must expect that certain of the advantages of the previous strategy will be lost. Actually the worst thing which may happen is that all the T simple transitions from one track to another will now have to induce a transfer from drum. This is then the maximum loss of the new scheme. On the other hand, fixing the positions of tracks no. t1, t2, ... , tp, in the available track places means that all transfers of these tracks will be saved. As long as the current cycle of tracks continuos the loss of T drum transfers must then be compared with the gain of

$$\sum_{v=1}^{p} \text{frequency}[v]$$

drum transfers. In other words, if we can find p (= n-1 where n is the number of track spaces) frequencies in the frequency table such that their sum is greater than T then the tracks corresponding to these frequencies should be held permanently in the available spaces as long as the present cycle continues.

There is no question that considerable gains may be achieved by using this technique. Consider as an illustration the administration of the following track cycle:

1, 2, 3, 4, 5, 4, 5, 6, ...

when there are 5 track spaces available. Using strategy 2 the development of a typical cycle will be handled by the following tables:

Stage of development	Track no. of space no. 1	2	3	4	5	Priority of space no. 1	2	3	4	5	Frequency of track no. 1	2	3	4	5	6
Before cycle	6	2	3	4	5	13	7	8	11	14	20	20	20	20	1	20
When in track no.																
1		1					15				21					
2			2					16				21				
3				3					17				21			
4	4					18								21		
5					v					19						
4	v					20										
5					v					21						
6		6					22									21
5					v					23						
After cycle	4	6	2	3	5	20	22	16	17	23	21	21	21	21	1	21

Here only the changed values have normally been entered. A v in the track no. table

indicates that no transfer from drum is necessary. This table shows that during one cycle 4 track transitions are "good" (i.e. require no drum transfer) while 5 are "bad". If this cycle is repeated our T will increase by 4 in each cycle while the frequencies of tracks no.s 1, 2, 3, 4, and 6 will increase by 1. If the cycling is interrupted the analysis will accept that 4 tracks be fixed as long as track no. 5 is not among those which are fixed. Depending on which additional track is not fixed the performance varies as follows:

Tracks not in fixed places	Performance	
	Good transitions	Bad transitions
(1,5), (2,5), (3,5), and (6,5)	7	2
(4,5)	5	4
Strategy 2 for comparison	4	5

It is interesting to note that the fact that the most frequently used track, no. 5, is excluded from being fixed does not have any particularly undesirable effect. In fact a cycle having only 2 "bad" transitions is as good a performance as can possibly be achieved under the given circumstances.

An additional complicating effect arises from the fact that in general it cannot be expected that the number of available track spaces stays constant during a cycle. If the cycle includes entries and exits from blocks (which normally it will) the space requirements of the stack will vary. A simple way of handling this is to keep a variable which always tells what is the minimum number of track spaces which has been available since the current cycle started. The number of tracks which may be fixed is then one less than this minimum figure. For the same reason the tracks which are fixed should be placed at the low end of the section used for the program tracks in the core store.

The above scheme for handling special cycles is, however, by no means the only possibility. Since it is possible that the simple strategy 2 will in practise turn out to be quite satisfactory, and since a detailed development of the more elaborate schemes will require a considerable amount of work such schemes will ~~ha~~ not be considered any more at present. Consequently the following Algol program only makes use of strategy 2.

DRUM ADMINISTRATION USING STRATEGY 2.

```
comment The following program has 4 entries: 1. initialize is entered once at the
beginning of the complete Algol program. 2. take item is used to take an item loca-
ted in the program  at a point described by the values of "current track" and
"relative address". 3. next track is entered from the last instruction of each
track. It will jump to the first instruction of the next following track. 4. trans-
fer jumps to the point in the program described by "current track" and "relative
address".;
integer absolute address, base address, current place, current track, highest
        priority, j, lowest priority, max number of places, number of places,
        relative address;
Boolean jump;
integer array  track number, priority [1: max number of pla ces] ;
```

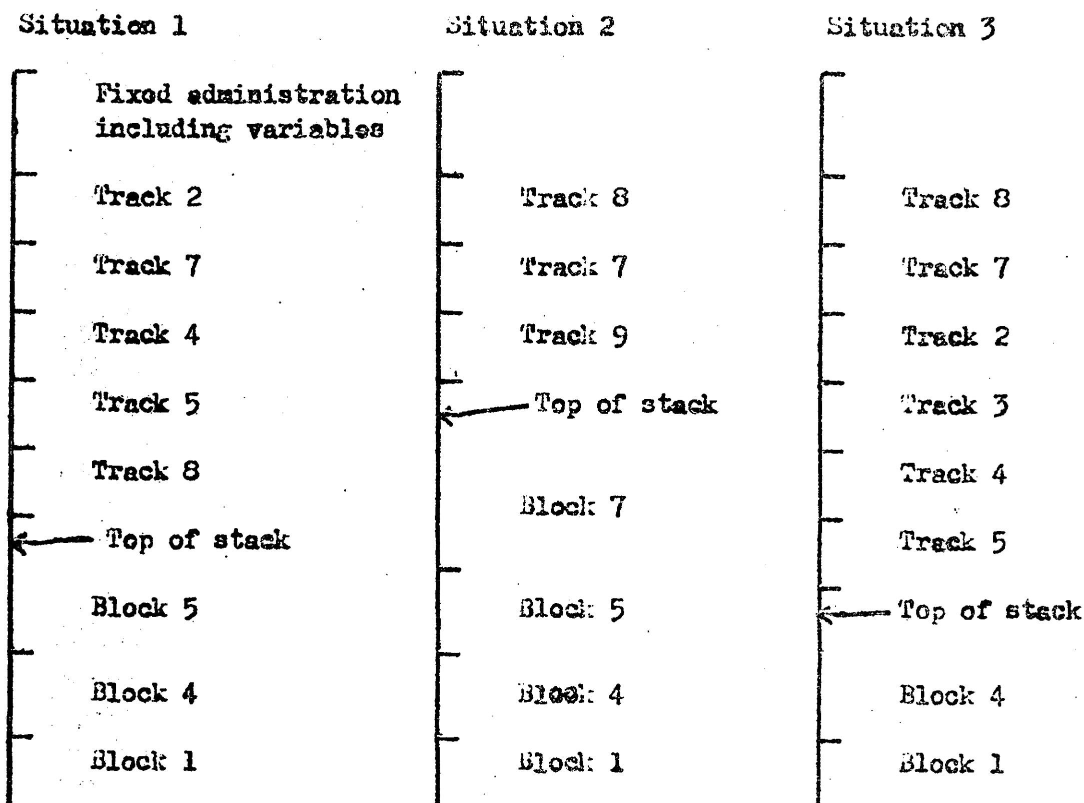
```
initialize:   highest priority := number of places := max number of places;
              for j := 1 step 1 until number of places do
                   begin track number[j] :=   0;
                         priority[j] := j
                   end;
              go to instruction [call address + 1];

take item:    jump := false;
              go to track administration;

next track:   current track := current track + 1;
              relative address := 0;

transfer:     jump := true;

track administration:
              for current place := number of places step -1 until 1 do
                   if track number[current place] = current track then go to track is
              lowest priority := highest priority - number of places;
              for j := 1 step 1 until number of places do
                   if           lowest priority ≥ priority[j] then
                        begin current place := j;
                              lowest priority := priority[j]
                        end;
              TRANSFER FROM DRUM(current track)Place in cores:(base address
                                                  + 40× current place);
              comment The following statements make sure that the quantities holding
              priorities do not overflow;
              for j := 1 step 1 until number of places do
                   priority[j] := priority[j] - lowest priority;
              highest priority := highest priority - lowest priority;
              track number[current place] := current track;
track is in:  highest priority := priority[current place]:= highest priority + 1;
              absolute address := base address + 40× current place;
wait for drum transfer:
              if drum transfer is in progress then go to wait for drum transfer;
              if jump then go to instruction[absolute address]
                    else item := store[absolute address] ;
              go to instruction[call address + 1];
```

TEST EXAMPLE: Follow the action step-by-step. After each jump to a track do next the action given in the test example.

```
base address := 160; max number of places := 3; call address := P;
P: go to initialize;
current track := 1; relative address := 0; go to transfer;
go to next track;
```

```
go to next track;
go to next track;
current track := 2; relative address := 12; go to transfer;
go to next track;
call address := S; current track := 7; relative address := 7;
S: go to take item;
```

As the final illustration of the present storage allocation scheme, here are three conceivable situations taken from the development during the run of one program. The vertical line is a picture of the core store. At any time it will be divided into three major sections: the fixed administration, the program, and the stack. The program section is divided into the track places, while the stack is divided into sections corresponding to the Algol blocks which are presently active in the program.



Note that the entry in the stack corresponding to a particular block need not be the same in the different situations because the arrays declared in the block may be of variable size.

Within each program track there are two sections: 1) Active instructions, and 2) Constants. It seems reasonable that the constants needed by a particular piece of program be stored on the same track so that a fixed table of constants is avoided at run time. Exception may possibly be taken with respect to a few of the small integers which are relatively frequent.

Within the section of the stack belonging to one block the structure will be approximately as follows:

Part 1: Internal block parameters
(fixed format, fixed order)

Part 2: Formal locations and local variables including temporaries
(the format varies from one block to the other, but is completely determined by the translator)

- Value of type procedure
- Formal locations
- Integers
- Reals
- Booleans
- Array identifiers and storage coefficients
- Switch identifiers and tables (
- Temporaries

Part 3: Components of local arrays
(the format is calculated at run time)

ADDRESSING.

The active instructions of a program track will need 4 kinds of references to variables or other program sections:

1) References within the same program track (short jumps, use of constants). These references obviously can be handled by means of relative addressing in Gier.

2) Jumps leading outside the program track, but within the same Algol block. These will make use of the drum administration described above (page 7, entries "next track" and "transfer").

3) References to variables in the stack.

4) Go to statements leading out of blocks.

In addition there will be references to a list of constants, if such a list is used.

The main problems are the kinds 3) and 4) and the associated administration of entries and exits from blocks. The solutions described below largely follow the principles described by Dijkstra (Ein ALGOL-60-Ubersetzer fur die X1, Mathematik Technik Wirtschaft, Vol. 8, Wien (1961), pp. 54-56 and 115-119, translated into English as ALGOL Bulletin Supplement no. 10, available from Mathematisch Centrum, 2e Boerhaavestraat 49, Amsterdam).

The references to variables (which all sit in the stack) must all make use of two pieces of information:

1) The block number of the variable, and
2) The relative address of the variable.

The block number of a particular block is obtained by scanning the program, countin +1 at each block <u>begin</u> and -1 at each block <u>end</u>. This kind of block number counts what Dijkstra calls the lexicographical depth of the block. For every block it indicates the number of lexicographically enclosing blocks. The basic principle of the referencing is the following: at any time the variables which may be references are the ones declared in the youngest incarnations of the lexicographicall enclosing blocks. Here the words "youngest incarnations" refer to the possible reactivations of blocks through recursive procedure calls.

The relative address is simply the position of the variable within that section of the stack which was reserved when the block in which the variable is local was entered. Both the block number and the relative address can be determined completely for all variables by the translator.

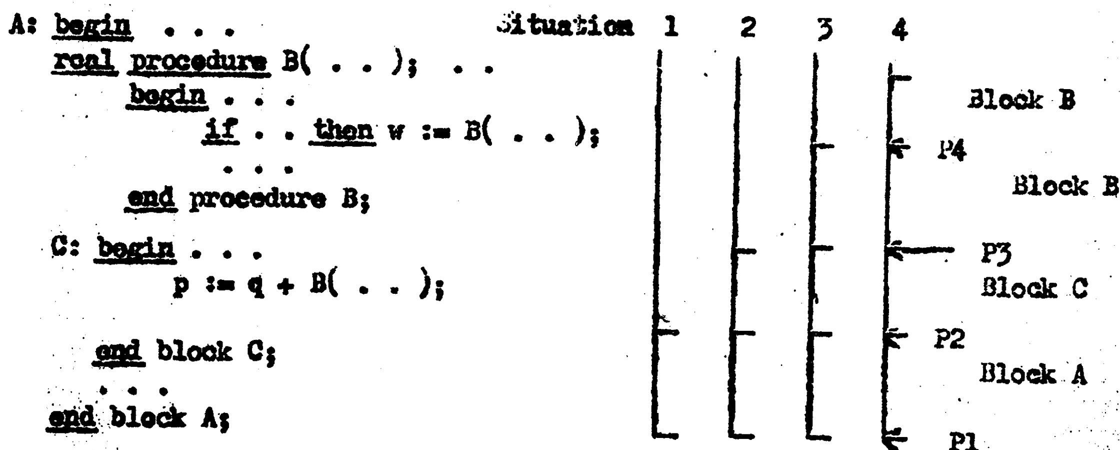
In order to calculate the absolute location of a variable when the block numbe and relative address are given it is necessary also to know the positions in the stack of all those block sections which may be referenced at the given moment. This is what Dijkstra calls the DISPLAY. It is simply a table with an entry for each of the blocks having numbers n, n-1, n-2, ... 1, 0, where n is the block number of the block in which we are now working. The entries are the absolute addresses of the first item of each of these blocks. In an obvious notation we have:

absolute address = relative address + DISPLAY [block number]

Here it is assumed that the stack is filled at small addresses first. If it starts at the high end the relative address should be subtracted.

From this it is clear that the DISPLAY will have to be changed every time blocks are entered or exited from, or even when the value of an expression called by name is evaluated.

As an illustration consider the following program structure and the corresponding development of the contents of the stack:



In this program the block numbers are:

	Block number
Block A	0
Procedure B	1
Block C	1

Thus the DISPLAY will only have two entries. The values of these two entries during the development of the program are shown here:

Situation	DISPLAY[0]	DISPLAY[1]
1. After entry into A	P1	-
2. After entry into C	P1	P2
3. After first call of B	P1	P3
4. After call of B from inside itself	P1	P4
5. After completion of 4	P1	P3
6. After return to block C	P1	P2

It is clear the the DISPLAY acts very much like a set of index registers and if a sufficient number of index registers were available they might be used for this purpose. In Gier there is only one index register and it becomes necessary to keep the DISPLAY separately in the store among the administrative parameters. When referencing a variable the contents of the index register must be set to the value held in the appropriate location in the store. If, as is often the case, several variables which all belong to the same block are referenced just after each other it is of course only necessary to set the value of the index register once. This can be taken care of by the translator. In addition the variables in the outermost block of the program will always be stored at the same place and may be addressed absolutely.

The proper values are inserted into the display with the aid of internal block parameters. These consist of 3 permanent locations holding

1. "block number", i.e. the number of the innermost currently active block.
2. "stack reference", i.e. ~~thexpositionxinxthexstackxwherexthexsectionxreserxvedxforxthexinnermostxcurrentlyxactivexblockxstartsxThisxisxgiven~~ the position in the stack where the previous values of the internal block parameters are stored.
3. "first free", i.e. the address of the first free location in the stack (the "top of the stack").
4.

The parameters located at and next to the location indicated by "stack reference" are the following:

Location	Meaning of parameter
stack reference - 4	the return point to use when the current block is exited from (track number and relative address).
stack reference - 3	"first free" belonging to the previous block
stack reference - 2	"block number" of previous block
stack reference - 1	"stack reference" corresponding to previous block
stack reference	"stack reference" corresponding to the youngest incarnation of the innermost enclosing block.

The entry into a simple block may now be described as a call of the followéng procedure:

```
procedure BLOCK ENTRY(new block number);
begin integer j;
     stack[first free - 4] := -1; comment This is a dummy return address;
     stack[first free - 2] := block number;
     stack[first free - 1] := stack reference;
     stack[first free - 3] := stack reference := first free;
     first free := first free - 4;
SHORT CIRCUIT:
     block number := new block number;
     stack[stack reference] := DISPLAY[block number - 1];
UPDATE DISPLAY:
     DISPLAY[block number] := stack reference;
     for j := block number step -1 until 1 do
          DISPLAY[j-1] := stack [DISPLAY[j]]
end BLOCK ENTRY;
```

Here the mechanisms labelled SHORT CIRCUIT and UPDATE DISPLAY should properly be represented by calls of procedures since they will be used in other contexts. Short circuit links the new block with the youngest incarnation of the innermost enclosing block. In doing this it will in general short circuit other blocks in the stack, namely any such which have block numbers greater than "new block number" -1. UPDATE DISPLAY extracts the addresses along the chain established by SHORT CIRCUIT and puts them into the DISPLAY. This mechanism was invented by Dijkstra (loc. cit.).

In simple blocks the address hold at stack reference - 3 will be stack reference itself. This does not hold for procedure bodies (see later).

The corresponding exit mechanism, to be used on exit from blocks and procedures, and for go to statements leading out of blocks, will make use of the following procedure:

```
procedure DECREASE LEVEL;
begin  block number := stack[stack reference - 2] ;
       first free := stack [stack reference - 3] ;
       return := stack [stack reference - 4] ;
       stack reference := stack[stack reference - 1] ;
UPDATE DISPLAY:
       DISPLAY[block number] := stack reference;
       for j := block number step -1 until 1 do
           DISPLAY[j-1] := stack [DISPLAY [j]]
end DECREASE LEVEL;
```

PROCEDURE ENTRY.

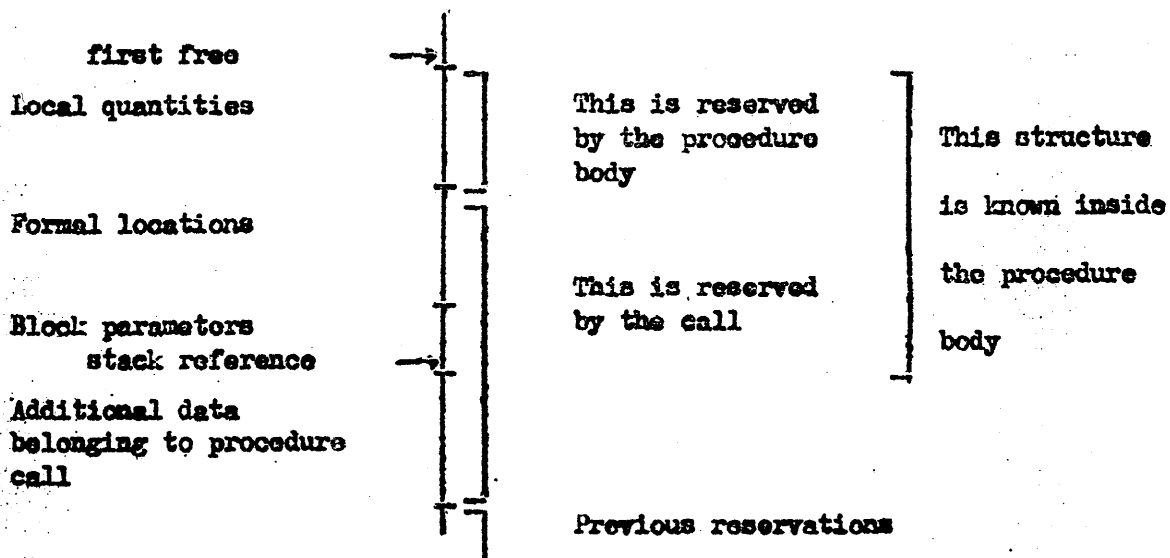
The handling of procedure entry must to some extent depend on the storage allocation and addressing. Therefore solutions, which differ from those used for DASK (J. Jensen and P. Naur: An Implementation of ALGOL 60 Procedures, BIT Vol. 1 no. 1 (1961)) must be considered, although the basic discussion of course may be taken from this earlier work.

The procedure entry is a problem of internal communication within the running program. There is a choice open as to how to distribute the information of what process has to take place between the running program code itself and the associated administrative programs. If the information in the running program is closely packed it will have to be interpreted by a more elaborate administrative program, and vice versa. In the present project we have chosen to make a choice at this point which differs from that made in DASK ALGOL. In fact, largely speaking, where there is a choice of where to save storage space we decide as follows:

Most important space to save: fixed administration.
Less - - - - : stack.
Least - - - - - : running program.

The reason for this is that the space taken by the fixed administration is permanently occupied, while the stack and running program only take space in the core store when necessary. Thus the above choice is directly a consequence of the storage allocation scheme, particularly the drum administration scheme for the program.

The following decisions follow almost necessarily from the previous design: The formal locations must be stored in the stack, like all other variable parts of the program. They must be placed in a fixed position relative to the local quantities of the procedure body, since they will be addressed from within the body with the aid of the same reference address in the DISPLAY. Whatever additional data must also be transmitted during a particular procedure call must also be placed in the stack. These considerations give us the following picture of the stack just after a procedure has been called and the procedure body has been entered:



This arrangement is consistent with the idea that most of the administrative work of the procedure entry may be done using only information available on the call side. This is desirable because the drum administration makes it undesirable to refer repeatedly back and forth between the call and the declaration. The present scheme divides the procedure entry work in two:

1) Transmit the names of all actual parameters from the call to the stack. This can be done exclusively on the basis of the information in the call.
2) Do any desired checking of the consistency of call and declaration and take value of parameters called by value. This is done as the first thing accomplished by the procedure declaration.

There is still a certain choice as to what should be considered to be "additional data" belonging to the various kinds of actual parameters. Obviously, the more information has to be transmitted to the stack the more difficult will the procedure entry become, but on the other hand the references from within the procedure body to this information will become easier. It seems fairly clear that constants in the call should be transmitted as additional information because the information in the formal location thereby gets simplified, thus compensating for the loss of trouble and storage in the stack. Compound expressions supplied as actual parameters are less obvious. It was decided not to transmit them to the stack, but to let the procedure body refer to them directly where they are in the call, for the following reasons:

1) If the full advantage of transmitting them to the stack should be gained it would be necessary to modify the addresses in them to absolute addresses in the process of transmission to the stack. This would require administration of considerable complexity, doing essentially the work which the DISPLAY mechanism has already been designed to take care of.

2) Expressions called by value are called only once at each entry. It would be an undesirable waste of storage capacity to let such parameters occupy space in the stack after it is certain that they will not be used any more.

With respect to strings (text strings, not layouts) these will be stored in a special table on the drum. They will be addressed like program sections, the beginning of a particular string being described by a "track number" and a "relative address".

In the description of the form of the procedure call given below the program will be represented as a continuous string of symbols (integers). The information in this string is given both by the single symbols and by their context. This is pictured as being written in the program itself, like program parameters following the jump to the procedure call administration.

The most complicated problem in the procedure call is the call of expressions by name. It has been decided to use, essentially, the idea of Thunk (see Comm. ACM Jan. 1961, by Ingerman), i.e. every reference from within the procedure body to a formal parameter called by name will, if necessary, evaluate the value of the corresponding actual parameter, place this value in some location and return with the address of this location placed in some universal location (a register of the machine). This approach differs from the method used in DASK ALGOL (see Jensen and Naur, loc. cit.) where either the value or the location could be obtained. The reason for this difference is twofold: a) The treatment is simpler because any kind of expression (including subscripted variables) will be treated alike in all cases (both as left parts and in expressions). b) With built-in floating point the address (placed in a register) is more generally useful inside the procedure body.

The action to take place whenever a reference to an expression called by name is made is the following:

1. Store the current "stack reference" and the calling point, "track number" and "relative address" in suitable safe locations.
2. Update the display, using the value of "stack reference" which was valid when the procedure call was made.
3. Transfer control to the program representing the expression.

After completion of its work the code for the expression must call the following action:

4. Reset "stack reference".
5. Update DISPLAY.
6. Return to place from where the reference was made.

Note that this description assumes a slight improvement of the mechanisms of pages 11 and 12 as follows: At bottom of page 11 correct to read:

stack reference - 2 "block number" of current block

In this way the number of input parameters to UPDATE DISPLAY is reduced to being "stack reference":

```
procedure UPDATE DISPLAY;
begin block number := stack[stack reference + 1];
      DISPLAY[block number] := stack reference;
      for j := block number step -1 until 1 do
          DISPLAY[j-1] := stack [DISPLAY [j]]
end UPDATE DISPLAY;
```

This improvement is due to a private communication by E. W. Dijkstra.

SWITCHES.

Switches will be handled in the following manner: On entry into the block where the switch is declared a table of the meaning of each switch element is transferred to the stack. Each item in this table has exactly the same form as the contents of a formal location. In addition one extra word tells where this table is placed and how many elements it contains. Subsequent references to the switch by means of switch designators need only refer to the information in the stack, and in fact primarily need only be supplied the location of the extra word. One advantage of this method is that when the identifier of a switch is supplied as an actual parameter in a procedure statement only this extra word need be transferred and there is no need in references to switch designators to distinguish between formal and non-formal switch identifiers.

The form of the information in the stack created by the switch declaration is as follows:

S - n: Description of n'th element
S-n+1: - - (n-1)st element
. . . .
S - 1: Description of 1st element
S: Address of 1st element = S-1, number of elements = n

This information will be stored among the other local quantities of the block.

The form of the switch declaration is similar to that of a procedure statement and will be explained below.

LABELS AND PROCEDURE IDENTIFIERS.

In analogy with the treatment of switches labels and procedure identifiers will have locations reserved for them in that section of the stack which holds the local quantities of the block in which they are local. Although this is not strictly necessary this treatment has been chosen because it simplifies the administration. In fact the difference between formal and non-formal procedure identifiers is not made in procedure statements.

The form of labels in the stack: One word is reserved having 4 components:

track number, track relative address,	} of point in program
stack reference	of section in stack where the label is local
"constant"	a mark indicating the kind.

The form of procedure identifiers in the stack: One word having 4 components:

track number track relative address	} of point in program
stack reference	of section in stack where the procedure is declared
"procedure"	a mark indicating the kind.

THE LOCAL DECLARATION.

The initialization of the locations in the stack corresponding to switch declarations, labels and procedure identifiers must be made at each entry into the blocks where any of these constructions are used. This initialization is made by a so-called local declaration. The form of this is similar to that of procedure statements, which is described below. The local declaration will in general have the following form:

1. A jump to the administration:

   call address := q;
   q: <u>go to</u> local declaration;

2. An initializer word, having 3 or 4 constituents:

   appetite of block, ie. number of words needed for local quantities
   block number
   number of formal (only for procedure bodies)
   "initialize block"
   or "initialize procedure body"

3. A list of labels, each having 3 constituents:

   track number } indicating the proper point in the program
   track relative }
   "label" a special mark

4. A list of procedure identifiers, each having 3 constituents:

   track number } indicating the position of the procedure declaration
   track relative }
   "procedure" a special mark

5. Any number of switch declarations. Each switch declaration will be described by one word of the form:

   number of elements
   "switch" a special mark

   followed by any number of words describing the switch elements. These are of one of the following two forms:

   Label as switch element:

   block number } of point in the stack
   relative address }
   "stack value" a special mark

   Expression as switch element:

   track number } of code for expression
   track relative }
   "expression" a special mark

6. A terminator word for the whole local declaration.

   track number
   relative address of the reentry point in the program
   "complete local declaration"

7. Track termination (may occur anywhere among the above words):

   "track completed" a special mark

PROCEDURE STATEMENTS, FUNCTION DESIGNATORS.

These constructions will in the running program be represented as a code having in general 3 sections as follows:

Section 1: A jump to the administration.

Section 2: Words containing coded descriptions of the actual parameters, of the procedure to be called, and of the return point to which the procedure should return.

Section 3: Codes representing those actual parameters which are not just identifiers.

The detailed descriptions of these 3 sections follow:

Section 1: call address := q; q: go to procedure entry;

Section 2: Each description normally will occupy one machine word. Only the description of the procedure to be called and the return point will occupy 2 words. If the section crosses a drum track division a special marking word will be inserted. In any case the machine word will contain an integer indicating the kind of the description as a part of it. The remaining part of the word will contain various kinds of additional information according to the following table:

Kind 1: Initialize (this must occur once before the description of parameters).
number of parameters
"initialize call"

Kind 2: Integer and real constants, layouts, strings.
value (for strings the value has two parts: 1) track number, and 2) track relative, referring to a table of strings on the drum).
"constant"

Kind 3. Array, switch, procedure identifiers, labels, formal parameters.
block number
relative address } refer to a point in the stack where the current description is stored
"stack value"

Kind 4: Simple variables.
block number
relative address } refer to a point in the stack where the current value is stored
"simple"

Kind 5: Expressions other than kinds 2, 3, 4.
track number
track relative } refer to a point in the program where the code for the expressions is stored
"expression"

Kind 6. Track completed

Kind 7. Enter body.
track number
track relative } for return point in program
block number
relative address } for point in stack containing the identifier of the procedure being called (whether formal or not).

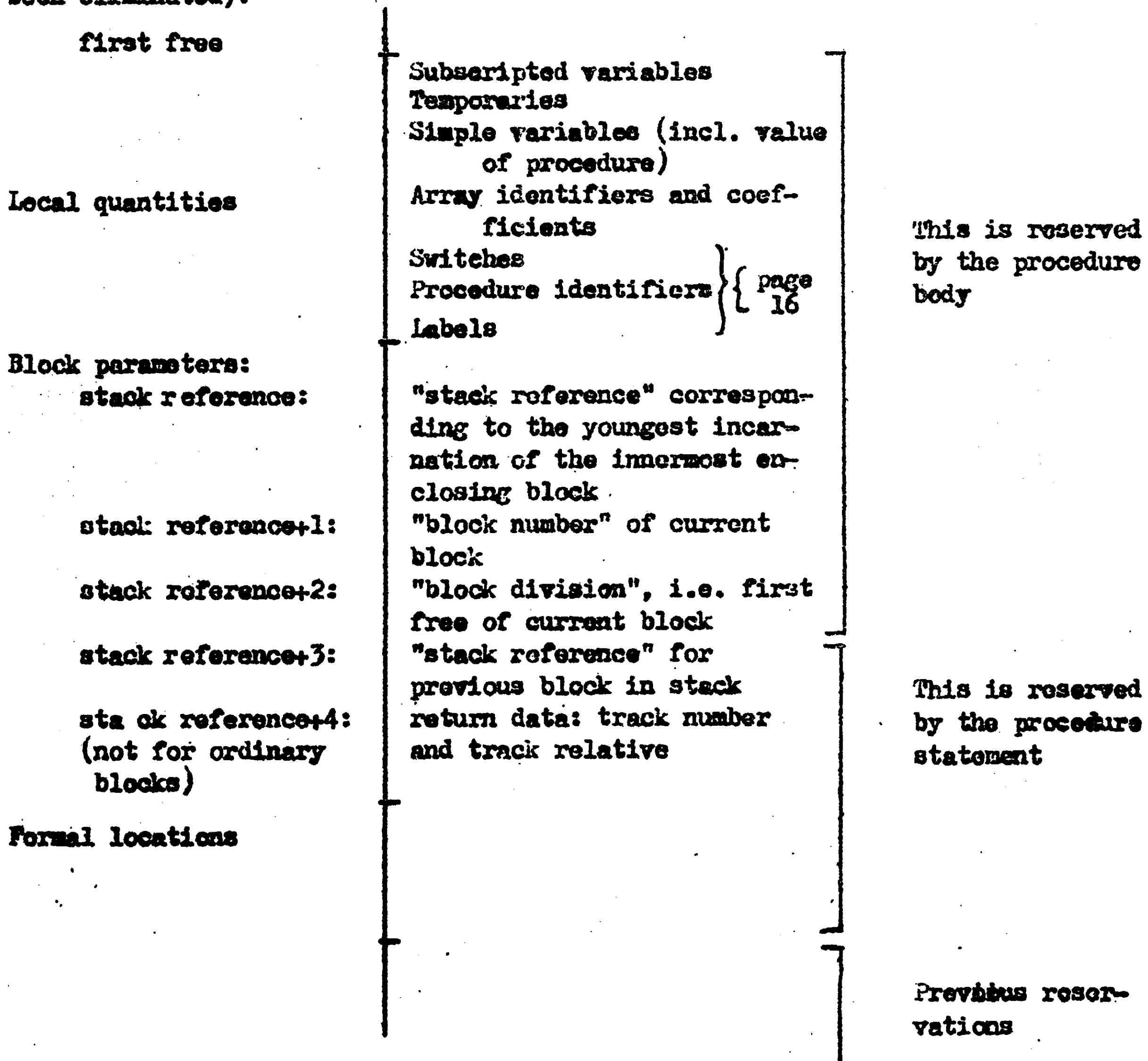
Section 3: Each actual parameter which is an expression is represented by a piece of code which if necessary evaluates the value of the expression (this is the case of a general compound expression), and in any case finally assigns the address of this value to register p (in case of a subscripted variable this will of course require some calculation in any case), followed by a jump to a fixed administration

<u>go to</u> exit from parameter expression;

The "track number" and "track relative" appearing in the description of this kind of actual parameter refer to the first instruction of this piece of code.

VARIABLES AND PARAMETERS IN THE STACK.

The picture of the stack structure given on page 13 should be changed to be as follows (note that the "additional data" belonging to a procedure call have been eliminated):



Part of this rearrangement is made in order to make the administration of expressions called by name coincide with that of procedure calls. In fact, upon calling an expression by name the items at stack reference+3 and stack reference + 4 are entered into the stack, while the remaining block parameters are unnecessary.

Each formal location occupies one full word in the machine. In some cases the marks on the GIER word are used to indicate the interpretation of the word. In any case the meaning of the contents of the word depends on the kind of the corresponding actual parameter. The various cases are the following:

Label

Mark: "constant"
track number } of point in program
track relative }
stack reference corresponding to the proper incarnation of the block in which the label is local.

Procedure identifier.

Mark: "Procedure".
track number } of point in program where declaration starts.
track relative }
stack reference corresponding to the proper incarnation of the block in which the procedure is declared.

Constant

Mark: "constant"
value

Array identifier, switch identifier

Mark: irrelevant
absolute base address (for array: where elements are stored, for switch: where table of element descriptions is stored)
second address (for array: where coefficients are, forswitch: number of elements)

Simple variable

Mark: "simple"
absolute address of value

Expression

Mark: "expression"
track number
track relative
stack reference corresponding to the proper incarnation of the block in which the expression is written in a procedure statement.

PROGRAMS FOR LOCAL DECLARATION AND PROCEDURE ENTRY.

The administrations referred to above may now be specified. Note that some labels refer to places not yet specified.

```
local declaration:
procedure entry:
    absolute address := call address; go to W;

next parameter:
    address of formal := address of formal - 1;
W:  absolute address := absolute address + 1;
    item := store[absolute address] ;

transform:
     begin switch transformation := initialize block, initialize procedure body,
                    label, procedure, switch, stack value, expression, complete local
                    declaration, track completed, initialize call, constant, simple,
                    enter procedure body;
initialize block:
     stack[first free] := stack reference;
     go to common block stacking;
initialize procedure body:
     if first free - block division = number of formals part(item) then
                    ALARM("Wrong number of parameters");
     first free := first free - 1;
common block stacking:
     stack[first free - 3] := stack reference;
     block number :=
     stack[first free - 2] := block number part(item);
     stack reference := first free - 3;
     DISPLAY[block number] := stack reference;
     first free :=

     stack[stack reference + 2] := first free - appetite part(item);
     address of formal := stack reference - 1;
     go to  reserve space in stack;
initialize procedure statement:
     address of formal := block division := first free;
     first free := first free - number of parameter part(item);
reserve space in stack:
     relative address := absolute address - current track start + 1;
     number of places := entier(first free - 5 - base address)/40);
S:   call address := Q;
Q:   go to take item;
     go to transform;
```

```
label:
    stack[address of formal] :=
         combination("constant",
                     track number part(item),
                     track relative part(item),
                     stack reference);
         go to next parameter;
procedure:
    stack[address of formal] :=
         combination("procedure",
                     track number part(item),
                     track relative part(item),
                     stack reference);
         go to next parameter;
switch:
    stack[address of formal] :=
         combination(address of formal - 1,
                     number of elements part(item));
   go to next parameter;

 stack value:
     stack[address of formal] :=
         stack[DISPLAY[block number part(item)] + relative address part(item)] ;
     go to next parameter
 expression:

     stack[address of formal] :=
         combination("expression",
                     track number part(item),
                     relative address part(item),
                     stack reference);
     go to next parameter;
 complete local declaration:
     current track := track number part(item);
     relative addres := relative address part(item);
     go to transfer;
 track completed:
     current track := current track + 1;
     relative address := 0;
     go to S;
 constant:
     stack[address of formal] :=
         combination("constant",
                     value part(item));
     go to next parameter
 simple:
     stack[address of formal] :=
         combination("simple",
                     DISPLAY[block number part(item)] + relative address part(item));
     go to next parameter;
```

PROGRAMS FOR FINAL ENTRY INTO PROCEDURES AND PARAMETER EXPRESSIONS.

The following programs are entered from the programs of the preceding sections:

```
enter procedure body:
    expr := false;
    description := store[absolute address + 1];
    description := stack[DISPLAY[block number part(description)]
                    + address part(description)];
    return description := item;
    go to common;
procedure as expression:
    expr := false; go to B;
expression:
    expr := true;
B:  return description := combination(current track,
                              call address - current track start + 1);
common:
    stack[first free] := return description;
    stack[first free - 1] := stack reference;
    stack reference := stack reference part(description);
    UPDATE DISPLAY; comment This is not necessary in case of non-formal procedure
                   identifiers. The distinction between formal and non-formal
                   procedure identifiers is lost at this stage, however;
    current track := track part(description);
    relative address := address part(description);
    if expr then
        begin first free :=              first free - 2;
              number of unused := number of unused - 2;
              if number of unused < 0 then
                  begin number of unused := number of unused + 40;
                        number of places := number of places - 1
                  end;
        end;
    go to transfer;
```

These programs are also used for entering expressions called by name, as shown below.

REFERENCE TO EXPRESSION CALLED BY NAME.

According to the convention stated on page 15 any reference to a formal parameter expression called by name will result in the machine returning with the address of the location where the value is stored placed in register p. This is accomplished as follows:

The form of the formal parameter within the procedure body is:

```
     p := DISPLAY[block number of formal] + relative address of formal;
     call address := q;
q:   go to take address of formal;
```

On return to q+1 the desired address is placed in the p register.

The administration is

```
take address of formal:
     begin switch evaluate and return with address := simple, constant, expression,
                  procedure as expression;
     description := stack[p];
     go to evaluate and return with address[mark part(description)];
simple:
     p := address part(description);
constant:
     go to instruction[call address + 1];
     comment The two remaining cases have been specified on page 23;
     end;
```

RETURN FROM PARAMETER EXPRESSIONS AND END.

The end of a simple block is represented by

```
call address := q;  q: go to end block;
```

The administration is:

```
end block:  first free := stack reference + 4;
     stack reference := stack[stack reference + 3];

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

The end of type procedures is represented as

```
first free := stack reference + number of formals + 4;
p := address of procedure value;
go to end type procedure;
```

The end of non-type procedures is represented:

```
first free := stack reference + number of formals + 4;
p:= address of procedure value;
go to end type procedure
```

The administrations used are as follows:

```
end type procedure:
     universal value := stack[p];
     p := address of universal location;
end procedure:
     return description := stack[stack reference + 4];
     stack reference := stack[stack  reference + 3];
common end:
     UPDATE DISPLAY;
     current track := track number part(return description);
     relative address := track relative part(return description);
     go to transfer;
```

The following administration has been referred to on page 19:

```
exit from parameter expression:
     return description := stack[first free + 2];
     first free := first free + 2;
     stack reference := stack[first free - 1];
     go to common end;
```