A/S REGNECENTRALEN
SCANDINAVIAN INFORMATION PROCESSING SYSTEMS

SYSTEM LIBRARY

GSL NO.: 550

CLASS : 0.2.2

TYPE : Report

AUTHORS: Jørn Jensen,
Søren Lauesen,
Paul Lindgreen,
Boris Martynenko,
D.B. Wagner (ed.)

EDITION: April 1969 (E)

Pass actions, pass output, and storage organization in the Gier Algol 4 Compiler

ABSTRACT:

This report gives details of two important aspects of the Gier Algol 4 compiler:

1. Actions of the various passes; specifically the details of their interfaces.

2. Storage organization in the compiled program.

The report assumes that the reader is thoroughly familiar with Algol and in particular Gier Algol 4, and that he has some acquaintance with the operation of the compiler. It plunges immediately into technical details.

INFORMATION DEPARTMENT

DK-2500 VALBY · BJERREGAARDSVEJ 5 · PHONE: (01) 46 08 88 · TELEX: 6464 rcinf dk · CABLES: INFOCENTRALEN

Contents

I. The Tasks of the Passes

The best introduction to the design philosophy and overall operation of the compiler will be found in P. Naur, 'The Design of the GIER ALGOL Compiler', BIT 3 (1963), 124-140 and 145-166. Briefly, the compiler is divided into ten parts: the nine 'passes' and the 'General Pass Administration'. Pass 1 reads the source program and outputs to the backing store a series of 'bytes' which Pass 2 reads. Each of the following passes reads the output of the preceding pass and outputs a new sequence of bytes to be read by the next until finally Pass 8 outputs to the backing-store area 'work' the finished object program. (For historical reasons the pass named 'Pass 9' falls between Passes 6 and 7.)

We describe below the various parts of the compiler in greater detail. Appendix 2 gives the details of the interfaces between passes. Of course for the most detailed possible information about the compiler see the program listings, published by Regnecentralen as The Complete Annotated Programs of Gier Algol 4, 2 volumes, December 1967, GSL 494.

1. General Pass Administration (GPA). GPA is that part of the compiler which is common to all passes; it takes care of input and output of bytes, printing of error messages, and transition to the next pass.

The entry for output of a byte will, if wanted, print the byte as an integer. This check-out facility is a permanent part of the compiler. For details see the GA 4 Manual, section 13.4.

The entry for error message printing can identify the current place in the source program by printing the value of a common carriage return counter in front of the message. The carriage return counter is updated by all passes whenever they meet the carriage return byte in the input. Therefore this byte is carefully kept through all passes even when surrounding bytes are removed because of errors (see pass 3b and pass 4 below).

The first time GPA produces any printed output from a given pass it prints the pass number.

GPA contains a table which describes the successive passes of the compiler. This table is used during transition to a new pass.

The input and output of bytes is buffered so that the time used for drum transports during the execution of a pass is negligible.

2. Pass 1. Analysis and check of the hardware representation of the source program (micro structure). Conversion to reference language which is output as a stream of 10 bit bytes.

The input to pass 1 is the source program taken character by character from the input medium. The input medium may be paper tape, typewriter, magnetic tape, or a backing store area.

Besides the conversion to reference language, which also implies recognition of compound symbols, e.g. begin end if $<$ $\neq$:= , pass 1 performs several other tasks.

Comments and blind characters, e.g. blanks, are skipped.
Strings are packed in an internal representation.

Each n'th line of the source program may be printed.
Several non-Algol features related to the hardware representation are handled: Change of input medium, optional skipping of input between PUNCH OFF and PUNCH ON, check of character sum in the input, printing of messages to the operator, pause for insertion of new paper tape in the reader.

Pass 1 skips all input up to the first begin and terminates the processing when the corresponding end has been read.

3. Pass 2. Identifier matching.

Each identifier encountered in the input is searched for in an initially empty table in the core store. If not found the identifier is entered in the table. In any case it is output as one byte representing the serial number of the identifier in the table. The value is between 1021 and 512.

This matching is performed regardless of block structure. The generated table is kept in core for use by pass 3a after which it may be overwritten.

Pass 2 also assembles bit patterns, a non-Algol feature, and outputs them as logical values.

4. Pass 3a. Standard identifier matching.

The identifier table generated by pass 2 is searched for occurrences of standard identifiers, i.e. identifiers declared outside the source program.

Each occurrence gives rise to the output of two bytes: The serial number of the identifier in the list of standard identifiers followed by the byte representing the identifier in the pass 2 output.

5. Pass 3b. Analysis and check of delimiter structure (logical structure). Delimiters of multiple meaning are replaced by distinctive delimiters and extra delimiters may be added to facilitate the task for the following passes.

A sub-part of the logic analyzes numbers and converts them to internal machine representation which is then output as five byte constructions. Also the procedure headings are treated by a sub-part of the logic which checks for missing or double specifications and for not allowed value specifications. Furthermore the specifications are output as part of the list of formal parameters.

The main logic is performed by a finite state algorithm using a stack for holding encountered opening bracket delimiters, e.g. if begin ([.

The algorithm scans the input up to and including the next delimiter and sets the operand situation, i.e. the class of operand encountered during the scan for the delimiter.

The delimiter and the current state determine, via a matrix, the new state, and the specific delimiter meaning. This in turn determines the further actions, e.g. byte output, stacking, unstacking. Also the operand situation is checked for consistency with the delimiter.

In case of error a message is given and current state is set to a value which will insure skipping of the rest of the current construction, normally up to a semicolon or to an end.

6. Pass 4. (Backward scan): Collection of declarations at block begin.

Pass 4 stacks all declarations (labels are treated as declarations) and unstacks and outputs the top section of the stack whenever a BEGIN BLOCK byte is encountered in the input. However, to enable pass 5 to give a relevant line number in case of double declarations, the identifiers from the declarations are also transmitted to the output.

Pass 4 also counts the locations needed at run time:
- In the whole program for:
  - Display (= max block depth).
  - Own variables.
- In each block for:
  - Simple variables, array descriptions, and dope vectors.
  - Local declarations, i.e. the dynamic descriptions of labels and procedures.
- In each procedure block furthermore for:
  - Formal parameters.
  - Dope vectors for formal arrays which in the procedure body appears with subscripts. (This enables the procedure entry to move the whole actual dope vector to local cells and thereby facilitate the subscription of the formal array.)

This last counting requires that all subscripted identifiers in a procedure body are stacked together with the number of subscripts. This stacked list is then confronted with the formal list from the procedure heading and the number of subscripts is added to the array specification.

Further pass 4 tasks:
- Insertion of the bytes BYPASS LABEL and GOTO BYPASS LABEL which will enable pass 8 to generate jumps around procedure bodies.
- Insertion of the byte WHILE LABEL in front of while elements in for-lists. Insertion of the byte PREPARE ASSIGN just after the last := in assignment statements.
- Skipping of the rest of erroneous constructions found by pass 3b. As the last task, after having processed the first BEGIN BLOCK pass 4 initializes the pass 5 declaration table using the byte pairs generated in pass 3a; see pass 5.

7. Pass 5a. Storage allocation of variables. Distribution of identifier descriptions.

A table of identifier descriptions is built up, based on the declarations collected at block begin. This table is checked for double declarations by help of the identifiers left at the original place where the declaration occurred. All other occurrences of identifiers are in the output from pass 5 replaced by the description from the table.

The normal description will consist of three bytes:

< kind-type > < relative address > < block number >.

However, for a standard identifier only one byte is output. This byte refers to a table of descriptions which is built up by pass 5b, see below.

8. Pass 5b. Generation of standard identifier description table. Output of list of standard procedure code sections to be included.

A table containing the descriptions of those standard identifiers which actually have been used is built up in the top of core. This table will be used by pass 6 whenever a standard identifier is encountered in the input.

Finally pass 5b outputs a list of bytes specifying the standard procedure code segments to be included in the object program. This list is used by pass 8a.

9. Pass 6. Type checking (Global structure). Conversion to Reverse Polish Notation.

Based on a priority table for operators all expressions are converted to Reverse Polish Form. In parallel to this all kinds and types of operands are checked by means of a pseudo evaluation of the expressions. This process will also insert explicit type conversions when needed and will deliver the final type of more complicated expressions.

10. Pass 9 (between passes 6 and 7): This pass is an assembler. It interprets the text of code statements as machine code written in a subset of the SLIP assembly language. From here on each piece of user-specified machine code is taken as an indivisible sequence of machine words.

11. Pass 7. Generation of machine operations. Assignment of working locations.

By a simulation of the run time processes, with respect to where and how the operands are stored, pass 7 generates the machine code necessary to perform these processes, i.e. it determines the use of the machine registers and allocates run time working locations. However, as the internal references (jumps) can not be addressed yet, the output from pass 7 is still in the form of a byte stream.

12. Pass 8a. (Backward scan). Rearrangement of the pass 7 output on the drum. Loading of the standard procedure code sections specified in the list from pass 5b.

13. Pass 8b. Generation of final machine code including addressing of all internal references. Segmentation into backing store tracks.

14. Pass 8c. Loading of running system, i.e. the fixed set of administrative routines needed at run time.

Result of compilation: A self-contained object program stored on consecutive tracks on the drum. It is relocatable as a whole on the backing store.

II. Storage Organization

A compiled program, while it is running, makes use of three kinds of storage: the backing store (drum or disk), a core store of 1024 words, and possibly a 'buffer store' of 4096 words.

The backing store holds the entire compiled program (parts of which will also be found in core), the text of most string constants used in the program, and any files the program may explicitly make use of. (See A Manual of Gier Algol 4, section 11, for details of use of the backing store through explicit calls to standard procedures.)

Core store holds all the variables of the program (possibly excepting arrays), the running system, and some 'segments' of the program. For details of the program segmentation scheme see Naur, 'Features of the Gier Algol 4 System' Regnecentralen, November 1967).

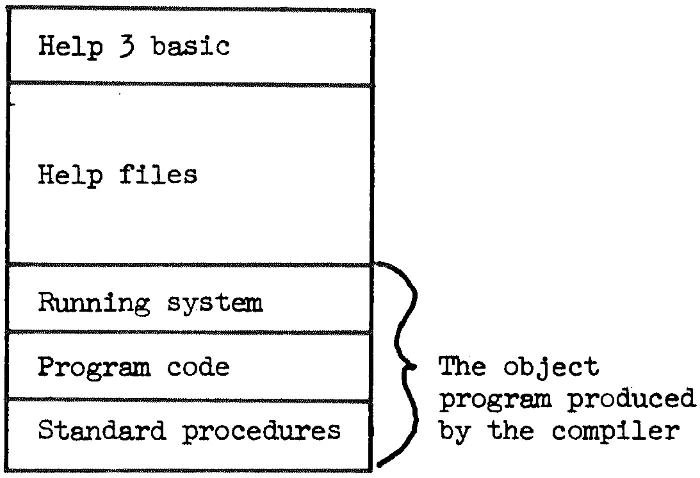
Buffer store, if available, holds all the array elements of the program. Figures 1 and 2 show the organization of core store and backing store during program execution.

Figure 1: Core Store During Program Execution

0	Some variables for Running system and Help system
e38	Program segments - - - - - - - - - - - - - - - - Stack
	Display
	Own variables
e37	Running system
1023	

Normally e37 = 1022, e38 = 15.

Figure 2: Backing Store During Program Execution



Storage of Variables

Variables are in general kept in core as long as they are active, with the single exception that array elements are kept in the buffer store if one is available.

Storage for own variables is assigned by the compiler, and can be seen on the above diagram of core store, figure 1.

Storage for any other variable is assigned during execution at entry to the corresponding block and released at exit from the block. These 'local' variables are organized in a stack which grows and shrinks during execution of the program and competes with program segments for storage. The stack is not permitted to grow so large that there is room in core for less than four program segments.

Each incarnation of each block in the program has a single section in the stack. (Wherever the word 'block' appears in this paper we include any procedure body, whether or not it is a block in the usual sense of a begin ... end structure.) The block's stack section contains:

I In the case of procedure blocks, the formal locations for the procedure.

II The program points of the block.

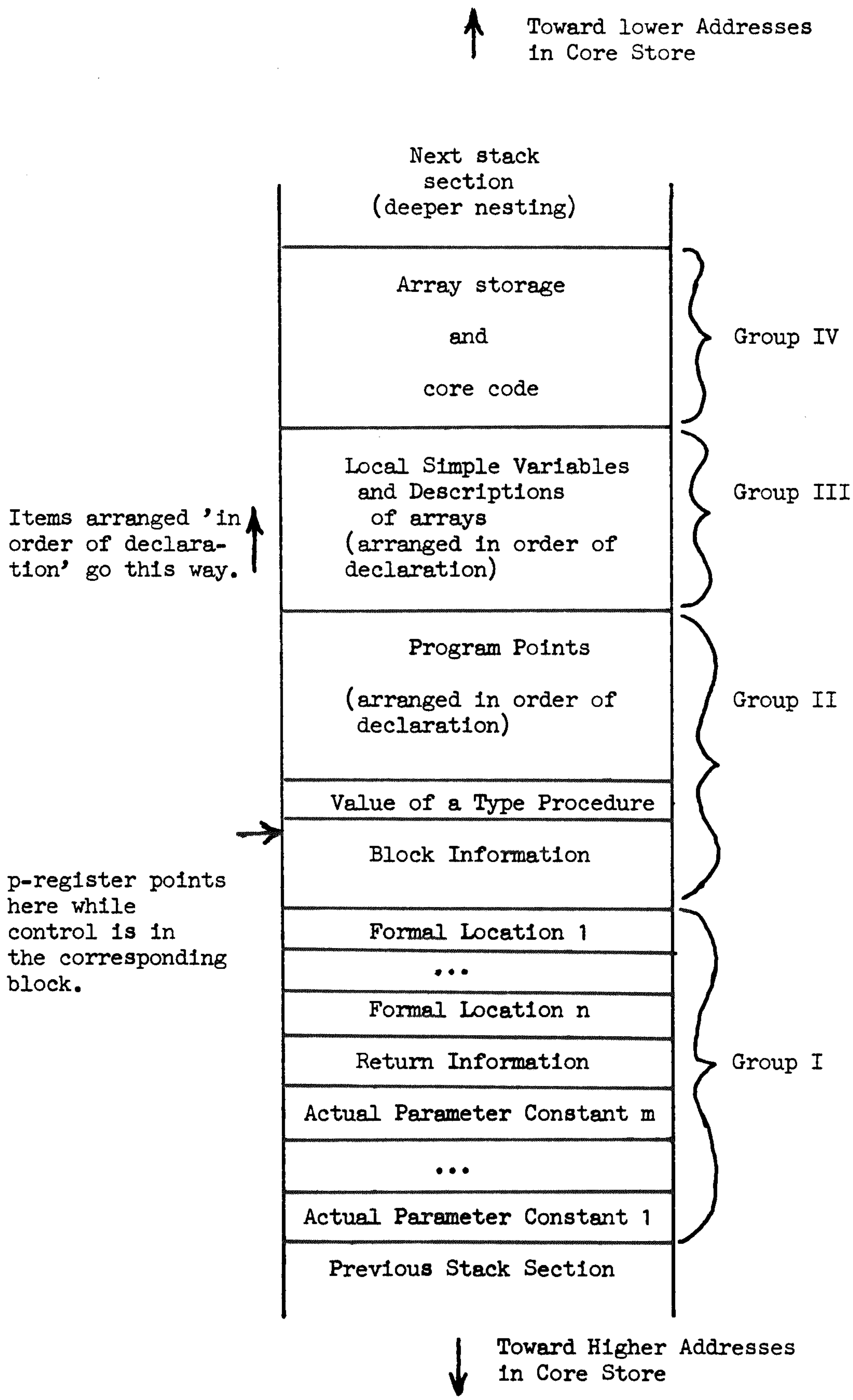
III The working locations and variables of the block.

IV Storage for arrays and for core code.

The numbering above corresponds to the common terminology of the 'groups' in a stack section. The diagram of figure 3 shows a stack section. We will go into the groups in some detail.

The 'display' is a list of pointers to stack sections. It is used for the purpose of allowing references within one block to variables in embracing blocks.

Figure 3: A Single Stack Section



Group I: The formal locations of a procedure block.

The machine code which the compiler produces for a procedure call works as follows:

1. Allocate storage for group I of the procedure block's section of the stack and create the 'block information' for that section.

2. Place into this stack section the return information and the formal words for the actual parameters of the call. The precise format of the return information and formal words is given in appendices 1.5 and 1.7. Suffice to say now that bits in the word may indicate among other things

   a. variable or constant
   b. program point
   c. 'thunk' - described below.

3. In addition place any constants referred to by these formal words into the stack section.

4. Set the p-register (which always indicates the current stack section) to point to the newly-created block information and transfer control to the procedure.

The procedure can now extend this stack section as necessary for its own storage, do its thing, and eventually return to its caller through the return information given. (At which time the p-register must of course be reset to point to the proper stack section.)

The 'thunk' is a device used to handle Algol's call-by-name convention. Its name has an obscure origin in the complex mind of Mr. Peter Z. Ingerman. If an actual parameter in a procedure call is an expression, and the corresponding formal parameter is 'by name' (that is, not declared value), then the expression must be re-evaluated every time the formal parameter is referred to in the procedure body.

A 'thunk' is a piece of code, organized somewhat like a procedure, which when invoked evaluates an actual parameter expression and places the address of the value in a standard location.

When the compiler compiles a call to a procedure it compiles a thunk for every actual parameter which is not a simple variable or constant. Then during execution, when the call is made, the call places a pointer to this thunk in the corresponding 'formal location' of the called procedure's stack section. Then a reference to the corresponding formal parameter consists of an invocation of the thunk.

In the case that the corresponding formal parameter is 'by value' rather than 'by name', thunks are not necessary but are used anyway. The actual parameter is evaluated immediately after entry to the procedure block and the value is stored in place of the formal word. Then access to this value is rather simple in the rest of the procedure.

Group II: The program points.

All labels and procedure entries are treated as local variables in the compiled code. Switches are treated as procedures, so they too must be considered here.

For each label and procedure entry in a block, a word is set aside in the stack section for the block and this word is initialized at block entry. (The format of the program point words is given in appendix 1.3.)

This convention means the compiler need not worry about the two basically different kinds of program point: Those available directly and those available by actual-formal correspondence. The formal location for a label or procedure formal parameter points to the program-point for it in the corresponding block.

Also generally included in Group II is the location set aside for the returned value of a type procedure. After the return from a type procedure as described above under Group I, the caller reaches up into the now-abandoned stack section of the procedure and picks up the returned value. After this has been done the contents of that stack section can be destroyed.

Group III: The working locations of the block.

Included in this group are all the simple variables of the program, and also descriptive information for arrays.

The descriptive information for an array is kept separate from the storage for the actual elements of the array for two reasons:

1. The storage for the array's elements must in the general case be allocated only after its limits are computed. Furthermore it may be that this storage is to be allocated in the buffer store and not in the stack. The descriptive information can keep a record of where this storage is allocated.

2. Because of efficiency considerations, descriptive information for array parameters of a procedure is copied into the stack section for the procedure at block entry.

See appendix 1.4 for the details of array descriptive information.

Group IV: Array elements and core code.

See A Manual of Gier Algol 4, section 12.7, for a discussion of core code. At entry to a block, all the pieces of core code declared in that block are copied into the stack section corresponding to the block. Then within the block they can be invoked through calls to the standard procedure gier.

At block entry the limits of arrays are evaluated, the storage is allocated for them, and appropriate information is inserted into the array descriptions mentioned in Group III. The storage allocated for arrays at this time will be in buffer store if it is available and in Group IV if not.

Storage allocated for arrays in the buffer store is assigned starting from the highest locations in the buffer and working downward.

A final use of the stack: Thunk returns. The call to a thunk is as follows:

1. Allocate one more word in the stack.

2. Place return information into this word.

3. Transfer to the body of the thunk.

After the thunk has done its evaluation, it returns by:

4. Release the topmost word of the stack.

5. Return control through this word.

It is necessary to use the stack for thunk returns because of two contingencies: The thunk may involve a call to a procedure; and it may involve a call to another thunk.

Storage of Constants

Each 'segment' of a program will contain a sequence of instructions plus all the ordinary constants which these instructions use. In most compilers it is worthwhile to group all the constants used in the program and eliminate duplications; but in Gier Algol 4, because of the segmentation scheme, such a grouping would decrease efficiency rather than increase it. No grouping of constants, therefore, is done except within individual segments.

The integer constants 0 and 1, and also any other constants with the same machine representation, are treated specially. The constant zero need never be kept in storage as instructions can do without it. The constant one is located in the running system and when needed is picked up there.

It is convenient to require that all actual parameters be continuously in core store. Therefore constant actual parameters are placed in the stack as described above in the discussion of Group I.

Strings are a little bit awkward because a string is the only kind of value that does not fit in a single word. Therefore the actual text of most strings (those longer than 6 characters) is kept on the backing store during program execution. See appendix 1.6 for a precise description of the format of strings in the machine; but here it suffices to say that when a 'string' is mentioned in connection with Gier Algol 4 implementation, the word usually means a single-word description of the string giving its location on the backing store.

A final form of constant is the layout. It is considered Boolean and its representation in storage is described in A Manual of Gier Algol 4, section 9.5.3.

Appendix 1: Details of storage formats

It will be noticed that many of these storage formats contain peculiar numerical constants in parts of various words. Generally such a constant turns a data word into an instruction or special indirect word; this makes various code optimizations possible. See appendix 2.9 for the ways in which the program store takes advantage of these storage formats.

1.1. Block information

	0–9	10–19	20–29	30–39	40	41
sr1:	<stack reference> for the surroun-ding block	display address-1	896	960	1	0

	0–9	10	11	12–24	25	26	27–39
sr+1:	<last used> in the stack for block	0	0	<last used in buffer in sur-rounding block>	0	0	<number of loca-tions used in buffer in this block>

<stack reference> for the outermost block = 1.
<last used in buffer in surrounding block> at the beginning = 4096
<number of locations used in buffer in this block> before reservations in buffer = 0.

A procedure body is always considered to be a block. The surrounding block for a procedure is the block in which it is declared.

1.2. Value of type procedure

Until a value is assigned to a type procedure the contents of the location set aside for the procedure value is as follows:

0–9	10–19	20–29	30–39	40	41
<stack reference>	0	0	0	1	0

1.3. Program point

(labels, switches, procedures)

		0 9	10 19	20 29	30 39	40	41
label		<sr> of the block where this label is local	relative address in track	0	track number	1 1	0 1
switch		ditto	ditto	40	ditto	1	0
procedure without parameters	no type	<sr>	ditto	24	ditto	1	1
	integer	ditto	ditto	488	ditto	1	0
	real	ditto	ditto	488	ditto	1	1
	Boolean	ditto	ditto	8	ditto	1	0
procedure with parameters	no type	ditto	ditto	24	ditto	1	1
	integer	ditto	ditto	16	ditto	1	0
	real	ditto	ditto	16	ditto	1	1
	Boolean	ditto	ditto	24	ditto	1	0

In the word for a label, bits 40-41 are 10 if the target is left-hand instruction in a word, and 11 if the target is the right-hand instruction.

1.4. Description of array

		0–9	10–19	20–39	40	41
Array word	arrays in core store	0	0	address of corner		
	arrays in buffer	c17	1	ditto		

bits 40-41: integer 00
real 01
Boolean 10

The 'corner' of an array is a hypothetical element with all subscripts zero (e.g. alpha[0, 0, 0, ...]).

The array word is followed by full-word integers giving lower bound 1, upper bound 1, lower bound 2, etc. These 'bound words' appear regardless of whether index checking was specified in the compilation.

1.5. Formal location

		0–9	10–19	20–29	30–39	40	41
Constants	integer	c30	absolute address	520	0	0	0
	real	c30	ditto	520	0	0	1
	Boolean	c30	ditto	520	8	0	0
	string x)	c30	ditto	520	8	0	1

		0–9	10–19	20–29	30–39	40	41	
Simple variables	integer	c30	absolute address	512	0	0	0	Absolute addresses of constants label descr. and simple variables
	real	c30	ditto	512	0	0	1	
	Boolean	c30	ditto	512	8	0	0	
	string x)	c30	ditto	520	8	0	1	
	label	c30	ditto	512	8	0	1	
Subscripted variable	integer	stack reference for block containing the expression	relative address in track	480	track number	1	0	program points for thunks xx)
	real	ditto	ditto	480	ditto	1	1	
	Boolean	ditto	ditto	0	ditto	1	0	
Other expressions	integer	ditto	ditto	488	ditto	1	0	
	real	ditto	ditto	488	ditto	1	1	
	Boolean	ditto	ditto	8	ditto	1	0	
	string	ditto	ditto	8	ditto	1	1	
	label	ditto	ditto	0	ditto	1	1	

x) The address of a string is: for a short string, the address of a word containing the string; for a long string, the address of a drum point description (see Appendix 1.6, below).

xx) Where a formal is declared value, the formal location is also used to store the value computed at block entry.

		0–9	10–19	20–29	30–39	40	41
array		address of ray word-1	dope address -addr of array word-2	0	<number of subscripts + 1>	0	0
Switch and procedure ident. other than expression	integer	stack reference for block where declared	relative address in track	16	track number	1	0
	real	ditto	ditto	16	ditto	1	1
	Boolean	ditto	ditto	24	ditto	1	0
	no type	ditto	ditto	24	ditto	1	1
	switch	ditto	ditto	40	ditto	1	0

1.6. Constants

		0 … 39
Boolean	true	1 1 1 . . . . 1 1 1
	false	0 0 0 . . . . 0 0 0

		0	1	2	3	4–9	10–15	16–21	22–27	28–33	34–39
string	short	1	0	1	0	character no. 6	character no. 5	character no. 4	character no. 3	character no. 2	character no. 1

		0–9	10–19	20–29	30–39
string	long	0	relative address	0	track number

One word of a long string on the backing store:

	0	1	2	3	4–9	10–15	16–21	22–27	28–33	34–39
not last	1	1	1	1	character	character	character	character	character	character
last	1	0	1	0	no. 6	no. 5	no. 4	no. 3	no. 2	no. 1

Integer and real constants take their 'natural' machine representations.

1.7. Return information

	0–9	10–19	20–29	30–39	40	41
for procedures	stack reference for return point	<track rel. address>	40	track number	1 1	1 0
for thunks	ditto	ditto	880	ditto	1	0

1.8. Block information

	0–9	10–19	20–29	30–39	40	41
sr:	sr for surrounding block	display address - 1	896	960	1	0

	0–9	10	11	12–24	25	26	27–39
sr+1:	last used in core store	0	0	last used in buffer in surrounding block	0	0	number of locations used in buffer in this block

<sr for the surrounding block> for the outermost block = 1
<last used in buffer in surrounding block> in the beginning = 4096
<number of locations used in buffer in this block> before reservations in buffer = 0.

Appendix 2: Details of pass output

2.1. Pass 1

The three columns give: (1) the output byte value, (2) the meaning, and (3) the pass where the value is processed.

1	a	2	37	I	2	93	begin	3	267	×	3
2	b	2	38	J	2	98	for	3	268	/	3
3	c	2	39	K	2	106	if	3	269	↑	3
4	d	2	40	L	2	109	own	3	270	:	3
5	e	2	41	M	2	116	integer	3	271	≤	3
6	f	2	42	N	2	123	real	3	272	<	3
7	g	2	43	O	2	130	boolean	3	273	≡	3
8	h	2	44	P	2	137	procedure	3	274	>	3
9	i	2	45	Q	2	144	array	3	275	≥	3
10	j	2	46	R	2	149	switch	3	276	≠	3
11	k	2	47	S	2	153	string	3	277	∧	3
12	l	2	48	T	2	155	label	3	278	∨	3
13	m	2	49	U	2	157	value	3	279	=	3
14	n	2	50	V	2	167	;	3	280	⇒	3
15	o	2	51	W	2	172	end	3	281	mod	3
16	p	2	52	X	2	176	else	3	282	shift	3
17	q	2	53	Y	2	184	(	3	287	)<let>:(	3
18	r	2	54	Z	2	194	: or ,[x]	3	291	)	3
19	s	2	55	Æ	2	196	step	3		<code>	
20	t	2	56	Ø	2	198	until	3	1008	-begcode	3
21	u	2	57	0	2,3	200	while	3	1009	end pass	2
22	v	2	58	1	2,3	202	]	3	1010	CAR RET	2
23	w	2	59	2	2,3	210	[	3		<4 bytes>	
24	x	2	60	3	2,3	220	, or :[x]	3	1011	-short str	2
25	y	2	61	4	2,3	228	:=	3	1012	-long str	2
26	z	2	62	5	2,3	231	then	3	1013	-layout	2
27	æ	2	63	6	2,3	243	do	3	1014	0	2
28	ø	2	64	7	2,3	245	abs	3	1015	$\bar{1}$	2
29	A	2	65	8	2,3	249	code	3	1016	$\bar{2}$	2
30	B	2	66	9	2,3	251	core	3	1017	$\bar{3}$	2
31	C	2	67	.	3	256	case	3	1018	$\bar{4}$	2
32	D	2	68	⏨	3	258	of	3	1019	$\bar{5}$	2
33	E	2	72	+	3	260	round	3	1020	$\bar{6}$	2
34	F	2	76	-	3	262	entier	3	1021	$\bar{7}$	2
35	G	2	82	-,	3	265	true	3	1022	$\bar{8}$	2
36	H	2	86	go to	3	266	false	3	1023	$\underline{\bar{9}}$	2

[x]) Between 249 code and the first following 1008 begcode.

<code> ::= <any number of bytes between 0 and 511>
<1024 - number of CAR RET within the machine code>
The code of the machine language representation is given in appendix 2.7.

Output byte value 1009 end pass will appear at the very end of the output from pass 1 in the following context:

... 172 1009 0

<4 bytes> ::= <short text>|<text on drum>|<layout>

<text on drum> ::= 0 <track relative> 0 <track number>

<layout> ::= <layout bits 0 - 9><layout bits 10 - 19>
<layout bits 20 - 29><layout bits 30 - 39>

2.2. Pass 2

The three columns give: (1) the output byte value, (2) the meaning

57 0	137 procedure	<code>
58 1	144 array	264-beg code
59 2	149 switch	265 true
60 3	153 string	266 false
61 4	155 label	267 ×
62 5	157 value	268 /
63 6	167 ;	269 ↑
64 7	172 end	270 :
65 8	176 else	271 ≤
66 9	184 (	272 <
67 .	194 : or ,ˣ)	273 ≡
68 ⏨	196 step	274 >
72 +	198 until	275 ≥
76 -	200 while	276 ≠
77 CARRET	202]	277 ∧
78 <4 bytes> short str	210 [	278 ∨
79 <4 bytes> long str	220 , or :ˣ)	279 =
80 <4 bytes> Boolean lit	228 :=	280 =>
82 -,	231 then	281 mod
86 go to	243 do	282 shift
93 begin	245 abs	283 end pass
98 for	249 code	287)<let>:(
106 if	251 core	291)
109 own	256 case	
116 integer	258 of	512-1021
123 real	260 round	Identifiers
130 boolean	262 entier	

ˣ) Between 249 code and the first following 264 begin code.

<code> ::= <any number of bytes between 0 and 511>
<1024 - number of CAR RET within the machine code>
The code of the machine language representation is given in appendix 2.7.

Output byte value 283 end pass will appear at the very end of the output from pass 2 in the following context:
... 172 283 0

<4 bytes> is explained in appendix 2.1.

2.3. Pass 3

The output from pass 3 is scanned in the reverse direction by pass 4. This must be remembered when interpreting the structure.

The first part of the output has the structure:

0 <standard identifier pair list><identifier limit> 20

with

<standard identifier pair> ::= <standard identifier no><identifier>

The remaining bytes are coded as follows:

```
      0   CAR RET
<4>   1f  literal
<4>   4   literal string
<il>  5f  decl simple
      8   decl label
<il>  9f  decl own
<il>  12f decl array
      15  end clean
      16  end block
      17  end bounds
<no. of parameters>
  -   18  end proc no type
  -   19  end type proc
  -   20  begin
      21  ;
      22  do
      23  then statement
      24  else statement
      25  of statement
      26  end case statement
<code>
  -   27  code end
      28  core
      29  core code end
      30  end spec
      31  end call
      32  ] one
      33  ] more
      34  call parameter
      35  comma 1
      36  comma 2
      37  bound colon
      38  begin call
      39  begin function
      40  left bracket
      41  trouble
<i>   42  decl parproc no
<i>   43f decl par proc
<i>   46  decl switch
<i>   47  decl proc no par
<i>   48f decl proc no par
<i>   51  decl undef proc
<i>   52f spec simple
<i>   55  spec string

<i>   56  spec label
<i>   57f spec value
<i>   60f spec array
<i>   63  spec proc no
<i>   64f spec proc
<i>   67  spec switch
<i>   68  spec undef
<i>   69  spec general
      70  simple for element
      71  := for
      72  step element
      73  while element
      74  while
      75  end assign
      76  :=
      77  first:=
     128  proc;
     129  ifex
     130  ifst
     131  thenex
     132  elseex
     133  delete call
     134  end else ex
     135  end else st
     136  end then st
     137  end go to
     138  for
     139  step
     140  until
     141  end do
     142  end single do
     143  mod
     144  +
     145  -
     146  ×
     147  /
     148  :
     149  ↑̄
     150  shift
     151  first bound
     152  not first bound
     153  of switch

 154 code
 155 end switch
 156 and
 157 or
 158 imply
 159 =
 160 (̄
 161 )
 162 simple for do
 163 step element do
 164 while element do
 165 case st
 166 case expr
 167 of expr
 168 end case expr
 169 case comma
 170 case semicolon
 171 end loop
 492 <
 493 <
 494 ≡
 495 >
 496 >̄
 497 ≠
 500 -,
 501 entier
 504 pos
 505 neg
 506 abs
 507 round
 508 opint
 509 opreal
 510 opbool
 511 opstring
 512-1021
     identifiers
 999 begin code
1022 internal identifier
```

<4> ::= <bits 0 - 9><bits 10 - 19><bits 20 - 29><bits 30 - 39>

<i> ::= <identifier, i.e. byte in range from 512 to 1021>

<il> ::= <i>|<il><i>

<code> is explained in appendix 2.7.

f indicates that three consecutive byte values describe types: +0 = integer, +1 = real, +2 = Boolean.

2.4. Pass 4

	0	CARRET
<base w><base var>		
- -	1	begin block
	2	end pass
<no of actuals>		
-	3	begin func
<reverse code>		
-	4	begin code
	5	end core code
<i>	6	decl switch
<i>	7	decl label
<i>	8	decl proc err
	9	formal label
	10	formal general
	11	formal unspec
	12	core store
	13	formal switch
	14	end bounds
	15	end head
	16	end decl
	17	end block
<no.of parameters>		
-	18	end proc
-	19	end type proc
<spec list>		
-	20	specifications x)
<i>	21	label colon
<no of actuals>		
-	22	begin call
<no of subscripts>		
-	23	[
	24	end bound head
<i>	28f	begin bounds
<base w><base var>		
- -	35	begin switch
- -	36	begin par proc
- -	37f	beg par proc type
- -	40	beg no par proc
- -	41f	beg no par proc type
<i>	44f	decl no par proc
<i>	47	decl no par proc
<i>	48f	decl par proc x)
<i>	51	decl par proc x)
<i>	52f	decl simple
<i>	56f	decl own
<no of subscripts>		
-	60f	decl array
-	64f	take array
<i>	68f	take value
<i>	72f	formal proc
	75	formal proc
<i>	76f	formal simple
<i>	79	formal string
<i>	80f	anon. array
<4>	84f	literal
<4>	87	literal string
	128	proc;
	129	if expression
	130	if statement
	131	then expr
	132	else expr
	133	delete call
	134	end else expr
	135	end else stat
	136	end then stat
	137	end go to
	138	for
	139	step
	140	until
	141	end do
	142	end single do
	143	mod
	144	+
	145	-
	146	×
	147	/
	148	:
	149	↑
	150	shift
	151	first bound
	152	not first bound
	153	of switch
	154	code
	155	end switch
	156	and
	157	or
	158	imply
	159	=
	160	(
	161	)
	162	simple for do
	163	step elem do
	164	while elem do
	165	case stat
	166	case expr
	167	of expr
	168	end case expr
	169	case comma
	170	case semicolon
	171	end loop
	172	do
	173	then stat
	174	else stat
	175	of stat
	176	end case stat
	177	end call
	178	] one
	179	] more
	180	call param
	181	comma 1
	182	comma 2
	183	bound colon
	184	simple for
	185	:= form
	186	step element
	187	while element
	188	while
	189	end assign
	190	:=
	191	first:=
	192	while label
	193	prep assign
	194	go to bypasslabel
	195	bypasslabel
	492	
	493	<
	494	≡
	495	
	496	‾
	497	≠
	500	not ¬,
	501	entier
	504	positive
	505	negative
	506	abs
	507	round
	508	op.integer
	509	op.real
	510	op.boolean
	511	op.string
	512 - 1022	identifiers
Specifications x)		
	1006	general
	1007	undecl
	1008	switch
	1009g	proc type
	1012	proc no type
	1013g	array
	1016g	value
	1019	label
	1020	string
	1021g	name

f and g indicate that three consecutive byte values describe types:
f: +0=integer, +1=real, +2=Boolean. g: +0=Boolean, +1=real, +2=integer
<4> ::= <bits 30-39><bits 20-29><bits 10-19><bits 0-9>

x) The declaration of a procedure with parameters or a switch (treated as a procedure having one integer value parameter) appears in the block head as:
<identifier><decl par proc><spec list> 20

<base w> ::= 1024 - number of locations used for local variables
- number of locations used for program points
<base var> ::= 1024 - number of locations used for local program points
<spec list> ::= <specification, i.e. byte between 1006 and 1023>|
<spec list><specification>

The last bytes appearing in the output refer to the outermost block and the entire program and are:

1. <base w> of outermost block
2. <base var> of outermost block
3. identifier limit = smallest identifier byte - 1
4. standard identifier having lowest standard identifier number in output from pass 3.
5. 1021 - maximum block number - number of owns
6. 2 + number of owns
7. 0

2.5. Pass 5

128 proc;
129 ifex
130 ifst
131 thenex
132 elseex
133 delete call
134 end else ex
135 end else st
136 end then st
137 end go to
138 for
139 step
140 until
141 end do
142 end single do
143 mod
144 +
145 -
146 ×
147 /
148 :
149 ↑
150 shift
151 first bound
152 not first bound
153 of switch
154 code
155 end switch
156 and
157 or
158 imply
159 =
160 (
161)
162 simple for do
163 step element do
164 while element do
165 case st
166 case expr
167 of expr
168 end case expr
169 case comma
170 case semicolon
171 end loop
172 do
173 then st
174 else st
175 of st
176 end case st
177 end call
178] one
179] more

180 call param
181 comma 1
182 comma 2
183 bound colon
184 simple for
185 := for
186 step element
187 while element
188 while
189 end assign
190 :=
191 first:=
192 while label
193 prep ass
194 goto bypasslabel
195 bypasslabel
196 CAR RET
<working base>
197 - begin block
<proc.type><working base>
198 - - begin proc
<no of subscripts>
<dope rel>
199 - - take array
200f take value
203 end bounds
204 end block
<no of formals>
205 - end proc no type
206 - end proc type
207 label colon
<no.of actuals>
208 - begin call
<no.of subscripts>
209 - [
<the code><-no.of CR>
210 - - begin code
211 end come code
212 core code
<track list> 0
213 - - end pass
<no. of actuals>
214 - begin func
<rel.adr.coef><no.arr.>
<array type>
215 --- begin bounds

<block address>
412 - formal general
416 - undeclared
420 - label
424 - switch
428 - formal label
432 - formal switch
436f- no par proc
439 - no par proc no type
440f- par proc
443 - par proc no type
int,real,bool,not
444f- simple
448f- array
<dope relative><-no of subs.>
452- -dope description
<block address>
456f- formal procedure
459 - formal proc.no type
460f- formal simple
463 - formal string
464f- anonymous array
<4 bytes>
468f- literal
471 - literal string
Specifications
472f spec simple
475 spec string
476 spec label
477f spec value
480f spec array
483 spec proc no type
484f spec proc
487 spec switch
488 spec unspec
489 spec general
492 <
493 ≤
494 =
495 ≥
496 >
497 ≠
500 not ¬,
501 entier
504 pos
505 neg
506 abs
507 round
508 opint
509 opreal
510 opbool
511 opstring
512-1023 stdproc

First byte in output
1021 - maximum block number - number of owns.

Array declarations

```
    id1
    id2
     .
     .
     .
    idn
    const
    length
    coeff
rel adr of length
no of arrays
array type           1: integer, 2: real, 3: boolean
proc type 0: no type, 1: integer, 2: real, 3: boolean
          with par 1023: boolean, 1022: real, 1021: integer, 1020: no type
          1019: switch
<block address> ::= <rel adr><block no>
```

Specifications appear following each parameter procedure identifier in the reverse order of the original formal parameter.

Byte 213, end pass, is followed by:
1. 1 + number of standard procedure tracks used.
2. 1021 - maximum block number - number of owns.
3. A list of the standard procedure tracks needed.
4. 0.

f indicates that three consecutive byte values describe types: +0 = integer, +1 = real, +2 = Boolean.

2.6. Pass 6

0 begin call
1 end call
<type><working base>
2 - - begin proc
<no.of formals>
3 - end proc
4 - end type proc
<working base>
5 - begin block
6 end block
7 go to bypasslabel
8 label decl
9 while label
10 if
11 else st
12 end else st
13 for
14 := for
15/527 simple for
16/528 simple for do
17/529 while
18 while element
19 while element do
20/532 step
21/533 until
22/534 step element
23/535 step element do
24 end do
25 first subscript
26 not first subscript
27 not last subscript
28 last subscript
/541 round top
float top
float next top
32/544 abs
/545 entier
34 -,
35/547 negative
36 proc;

<type of expr>
37/549- else Rt expr
38 - else addr expr
39/551- end else Rt expr
40 - end else addr ex
41 then
42 prepare assign
43/555 :=
44 go to
45/557 +
46/558 -
47 integer multiply
/560 real multiply
/561 /
50 :
51 mod
52 ↑ integer
/565 ↑ real
54/566 < then
55/567 ≤ then
56/568 = then
57/569 ≥ then
58/570 > then
59/571 ≠ then
60/572 <
61/573 ≤
62/574 =
63/575 ≥
64/576 >
65/577 ≠
66 ∧ and
67 ∨ or
68 ≡
<bl.rel><bl.no>
69 - - label
<bl.rel.><bl.no>
<sl.rel.dope>
<-no of subsc>
70 --- array
71 - - simple
72 - - formal
73 - - procedure
<tr.no><tr.rel.>
74 - - std. proc
<4 bytes>
75 - constant
76 end switch call
77 end single do
<type>
78/590 - param comma
<bl.rel.of dope>
<no. of arrays>
<array type>
79 - - begin bounds
80 first bound
81 not first bound
82 end bounds
83 take real value
84 take int value
<no.of subs.><array rel>
85 - - take array
86 bypasslabel
<stack ref 0 + 2>
<no.of table bytes>
<std.track tab.bytes>
87 --- end pass
88 outchar
89 lyn
90 kbon
91 shift
92 select
93 opreal
94 opintrestr
95 gier
96 CAR RET
97 case
98 begin case
<type>
99/611- caseparam
100 casest
<total type>
101/612 end case
102 5 end address case
103 end stat case
<no of groups>
<5-bytes groups>
104 - - code
105 begin switch case
106 take nonsense
107 std 2 call
108 array param
109/621 move value
110 move address
111 move short
112/624 first value
113 first address
114 first short
115 new track
116 address

150 core
151 code
152 begin code

2.7. Machine language in output from passes 1 - 6

The representation of machine language is copied without change through passes 2 to 6, using the code given below. Because pass 4 scans its input in reverse order, the bytes will appear in the output from that pass in the reverse order.

1 K
2 L
3 M
4 N
5 O
6 P
7 Q
8 R
9 S
10 T
11 U
12 V
13 W
14 X
15 Y
16 Z
17 A
18 B
19 C
20 D
21 E
22 F
23 G
24 H
25 I
26 J
27 .

28 -
29 +
30 ,
31 (
32 <illegal>
33 k
34 l
35 m
36 n
37 o
38 p
39 q
40 r
41 s
42 t
43 u
44 v
45 w
46 x
47 y
48 z
49 a
50 b
51 c
52 d
53 e
54 f

55 g
56 h
57 i
58 j
<4 text bytes>
59 - not last text word
60 - last text word
61 m
62 Ī
63 C̄AR RET
64 0
65 1
66 2
67 3
68 4
69 5
70 6
71 7
72 8
73 9
74 :
75 =
76 blind CAR RET
77 e
78 $\overline{)}$

<4 text bytes> ::= <bits 4 to 12><bits 13 to 21>
<bits 22 to 30><bits 31 to 39>

<code> ::= <a sequence of the above byte structures except 77>
77 <1024 - number of CAR RET within the machine language code>

<reverse code> ::= <code written in the reverse order>

2.8. Pass 7

```
<blockrel>
-    0 addr local
-    1 var local
-    2 var abs
<blockrel><-blockno>
- -  3 var block
     4 (UA)
<UVrel>
-    5 UV
<trackno><trackrel>
- -  6 std proc call
<blockrel><-blockno>
- -  7 begin call
<bits 30-39>...<0-9>
---- 8 constant
<trackno><trackrel>
- -  9 std 2 call
    10 begin block
<proc type>
-   11 begin proc
    12 begin case
    13 begin sw case
<no of lits>
-   14 end call
<case type>
-   15 527 end case
<appetite><-blockno>
- - 16     end proc
- - 17     end typeproc
- - 18     end block
<param inf><kind><type>
--- 19 531 call param
--- 20 532 case param
    21     if/for
    22 534 hop NT
    23 535 hop LT
    24 536 hop NZ
    25 537 hop LZ
    26 538 bypass abs
    27 539 bypass NT
    28 540 bypass LT
    29 541 do abs
    30 542 goto bypass
    31 543 bypasslabel
<type>
-   32 544 else
-   33 545 end else
<opand>
-   34     enddo
-   35     end single do
-   36 548 take forlabel
    37 549 formal assign
    38 550 take real val
    39 551 take int val
```

```
<no of indic><doperel>
--40 552 move array
  41     writecr
  42     goto computed
  43     select 1
  44     select 2
  45     tk 1
  46     lyn
  47     kbon
  48     hs mult X NZA
  49 561 mt -1 D NT
  50     il 0
  51 563 mt -1 D LT
  52     us 0
  53 565 mt neg
  54     sr eps
  55     srf half
  56     pm UV
  57     pm UA
  58     arn UA
  59     ga UA
  60     outchar var
  61     ck (addr)
  62     tk 30
  63 575 ck -10
  64     int to address
  65     ab 0 DX
  66     ar eps LT
  67     xr
  68 580 tkf -29
  69     nkf 39
<opand>
- 70 582 pm
- 71 583 gm
- 72 584 mkf
- 73 585 dkf
- 74 586 qq
- 75 587 mt
- 76 588 snn
- 77 589 ann
- 78     mb
- 79     ab
- 80 592 grn
- 81     dln
- 82     ann X
- 83     dln X
- 84     sr LT
- 85 597 hs
- 86 598 pm D
- 87 599 arn D
- 88 600 ar D
- 89     gr MA
- 90     gr MB
- 91 603 gm M
```

```
<opand>
-  92 604 grn M
-  93     mln X IZA
-  94 606 acn MA
-  95     mb X
-  96     reserve array
-  97 609 var to UA
-  98     goto local
-  99     index upper
- 100     index lower
- 101 613 move formal
- 102 614 take formal
- 103 615 contr. formal
- 104 616 take assign

 <address constant>
- 106     ck
- 108     outchar const

<no of formals>
- 109     ps p
  110     label declar
  111 623 ↑real
  112     carret
  113 625 ↑int
<5 byte words><no words>
---114    code

  115     newtrack
  116     end pass

<opand>:
- 121 633 arnt
- 122 634 art
- 123 635 srt
- 124 636 grt
- 125 637 srnt
- 126 638 annt
- 127 639 grt M
- 128 640 grt V LA
```

```
<kind>:          <type>:
 2 subscr        1019 label proc
 3 statement     1020 no type
 4 UA-expr       1021 integer
 5 expr          1022 real
 6 stdproc       1023 boolean
 7 descr            0 no type
 8 constant         1 integer
 9 simple           2 real
10 array            3 boolean
                    4 string
                    5 label
```

2.9. Pass 8

Remarks on the notation:
The word 'program' will refer to the generated machine code.

The signs * and ⊕ on instructions indicates that they are generated only in the buffer-mode or core-mode respectively.

Entry points in running system (RS) are referred to by their Slip-names i.e. c0, c1, c2, etc. For details of the running system see Asmussen, et al, 'Gier Algol 4 Library Procedures' Regnecentralen Sept. 1967, Order no. 470.

The indications -->[ref] and [ref]<-- mean that the actual instruction appears in the program immediately before or after (respectively) the instruction referred to in the brackets.

2.9.1. Segmentation

The output from pass 8 - the final machine code - is generated into segments of 40 GIER words corresponding to a backing store track. The tracks will be referred to by a relative negative number using the term <trackno>. The range of <trackno> depends on the number of standard procedure tracks used and on the size of the program.

A word on a track is referred to by a relative address ranging from 0-39 using the term <trackrel>.

Each track consists of three parts:
1) A number of words from <trackrel> = 0 and onwards containing literal constants (see ref. a7) or special jump instructions (see section 5) referred to from
2) the proper machine code which is located after possible constants until and including <trackrel> = 38.
3) In <trackrel> = 39 an exit-to-next-track instruction (ref. j7-8) or in case of the last track of the program an exit-program instruction (j17).

2.9.2. Operand addressing

Machine instructions referring to runtime locations of operands will be referred to here by the term <op> in the address part.

<op> covers the following possible ways of addressing.

ref.:	address part		meaning	dir.byte
[a1]	(p<blockrel>)	:	address in local block	0
[a2]	p<blockrel>	:	variable in local block	1
[a3]	<abs addr>	:	variable in outermost block	2
[a4]	s<blockrel>	:	variable in intermediate block	3
[a5]	(c30)	:	address in UA	4
[a6]	c17	:	UV	5
[a7]	r<rel addr>	:	constant operand on actual track	8

where:
<blockrel> = value generated in pass 7 for the corresponding variable
<absaddr> = c0+<blockrel>
<reladdr> = <trackrel of word referred to>
- <trackrel of current instruction>

The value of the p-register will at run time always be equal to the stackreference of the current block, while the s-register is set by the instruction [a8] below, which is generated in an undefined place on the current track before the variable-reference [a4] but after the last:

a) program point (see below)
b) reference to a variable in another intermediate block
c) place where s is destroyed (eg. through an entry in RS)

[a8] ps(<displ ref>)

where:

<displ ref> ::= c0-<blockno>

2.9.3. Independently generated half- or full-word instructions

The following instructions are generated independent of the surrounding input structure, direct on a corresponding directing byte. For instructions i1-29 and f1-8 this is performed in connection with following operand-bytes, for instructions c1-25 without.

The instructions f1-8 may appear in the program with an f-mark if <op> represents a real operand. This is indicated to pass 8 through the bytes ranging from 533-541.

A number of further instructions which could be classified as belonging to this section are described in section 5.

ref.	instruction				meaning / used in	caused by dir. byte
i1	pm	<op>			a:= b, array declarations	70
i2	gm	<->			- , -	71
i3	mkf	<->			×	72
i4	dkf	<->			/	73
i5	qq	<->			assign to formal variable	74
i6	mt	<->			:, mod, step var until	75
i7	snn	<->			op ≠ 0	76
i8	ann	<->			op = 0	77
i9	mb	<->			∧	78
i10	ab	<->			∨	79
i11	grn	<->			op:= 0;	80
i12	dln	<->			:	81
i13	ann	<->	X		:, mod	82
i14	dln	<->	X		mod	83
i15	sr	<->		LT	mod	84
i16	hs	<->			gier(op)	85
i17	pm	<->	D		parameter to fast std.proc	86
i18	arn	<->	D		-	87
i19	ar	<->	D		-	88
i20	gr	<->		MA	declare boolean array	89
i21	gr	<->		MB	declare real array	90
i22	gm	<->		M	initialize step element [i1]<--	91
i23	grn	<->		M	- - -	92
i24	mln	<->	X	IZA	-->[j18]	93
i25	acn	<->		MA	step (first time)	94
i26	mb	<->	X		=	95

ref.	instruction	meaning / used in	caused by dir. byte
i27	gm p<blockrel> MA	set jumpword to forlabel left	36
i28	gm p< - > MC	- - - - right	36
i29	ck <10 bit const>	shift <constant>	106
f1	arn <op>	a+op, op-a, a×op, ...	121,533
f2	ar <->	op+a	122,534
f3	sr <->	a-op	123,535
f4	gr <->	op:=	124,536
f5	srn <->	-, op	125,537
f6	ann <->	abs op	126,538
f7	gr <-> M	initialize step, decl int.array	127,539
f8	gr <-> V LA	step (not first time)	128 540
c1	tk 1	<boolean expr> then	45
c2	mt-1 D NT	a ≠ b	49
c3	mt-1 D LT	a = b	50
c4	il	A[i], buffer version	51
c5	us	- , -	52
c6	mt c44	expr > a, a < expr	53
c7	sr c41	≤ , = , ≥	54
c8	srf c43	entier	55
c9	pm c17	var:= A[i], buffer version	56
c10	pm c30	for formal var:=	57
c11	arn c30	formal var parameter to fast std.	58
c12	ga c30	A[i], as actual parameter	59
c13	ck (c33)	shift var	61
c14	tk 30	A[i], no check, core version	62
c15	ck -10	A[i], check; shift var	63
c16	ga c33	shift var [c15]<--	64
c17	ab DX	-, =	65
c18	ar c41 LT	:	66
c19	xr	A[i, j ...	67
c20	tkf -29	round real	68
c21	nkf 39	float integer	69
c22	pmf c55 , ga c55	select -->[c23]	43
c23	tln 9 , ud c55	- [c22]<--	44
c24	ps p + 2 + nform	prepare exit proc -->[j14-15]	109
c25	qq (c35) t nconst	adjust last used after call	14

nform = number of formals
nconst = number of constant formals

2.9.4. Multiple generated instructions

Each of the following instruction-groups is generated on a single directing byte with possible operands.

m1	srn c41 qqn NKB	kbon	47
m2	arnf (c30) V LB abn (c30) , nkf-39	take real value	38
m3	arn (c30) V NB arnf (c30) , tkf-29	take integer value	39

Ref.	instruction	meaning / used in	caused by dir. byte
m4	ck-10 , nc 1+<no of ind> pt c49-5 , hs c27 ga c33 , it p-3+<array rel> pa c30 , tk 10 gr c17 M pm (c17) t 1 IRC gm (c30) t 1 MRC udn c17 NZ bt (c33) t - 1 hv r-4	move array description	40
m5	ps <op> hv c2	goto local	98
m6	it <op> pa c30	variable to UA	97
m7	arn <op> V LT pt c49-5 , hs c27	index upper, check mode	99
m8	sr <op> V NT pt c49-5 , hs c27	index lower, check mode	100
m9	srn <op> , hs c20 ⊕ arn c35 , hv s2 ⊕ ck 10 ⊕	reserve array (core mode)	96

2.9.5. Program points, jumps and exits to RS

Program points are entries in the program which are referred to from other places. They may be explicit caused by labels or procedure declarations, or implicit such as those necessary in conditional statements or expressions, in the administration of for-statements or array declarations, and in evaluation of expressions as parameters.

A reference to a program point is stored in a comma-marked-instruction in the following way:

qq <trackrel>.19 + <trackno>.39 + <right>.41

where <right> = 1 (instruction f-marked) indicates that the program point is in the right-half-part of the word referenced and <right> = 0 indicates left-part.

References to implicit program points on the same track e.g. jumps in if-then-else-statements may be carried out by means of relative addressing. (cf. ref. j2-6)

Conditional jumps (ref. j3-6) to program points on other tracks are performed via a constant word containing an unconditional track jump (ref. j1)

<end track inf> ::= qq <track rel>.19 + <linecount mod 1024>.39 + <rigth>.41

ref.	instruction	meaning / used in	caused by dir. byte
j1	hs c2 , <program point>	unconditional track jump	eg.26,29,30,32
j2	hv/hh r <reladdr>	- local jump	eg.26,29,30,32
j3	hv/hh < - >	conditional - -	22,27
j4	hv/hh r < - >	- - -	23,28
j5	hv/hh r < - >	- - -	24
j6	hv/hh r < - >	- - -	25
j7	hs c1 , <end track inf>	exit to next track (normal mode)	
j8	hs c3 , < - >	- - - - (param. -)	
j9	hs c26 , <program point>	- - fast st.proc	9
j10	hvn c18	- from param. expr	19
j11	hvn c19	- - - -	19
j12	hv c18 *	- - subscr. param. expr	19
j13	hv c19 ⊕	- - - - -	19
j14	hv c21	- - type proc	17
j15	hh c22	- - no - -	16
j16	hs c9	- - block	18
j17	hhn c29 , <last line>	- - program	
j18	hs c14 X NZA	- to multiply [i24]<--	48
j19	qq (c33), hs c39	- - outchar variable	60
j20	ud c37 , hs c37	- - next in (lyn) -->[c15]	46
j21	ncn (c30), hs c13	- - goto computed	42
j22	qq 64 , hs c39	- - write cr	41
j23	qq <10 bit const>, hs c39	- - outchar constant	108
j24	hs c4 , <program point>	- - ↑ integer/real	111,113
j25	is (c38), hv/hh s<rel addr>	- - block code on same track	10
j26	hh (c38),	- - - - in same word	10
j27	arn <op> , hs c20 *	- - reserve array (buffer m.)	96
j28	ud <op> , hs c28	- - move formal	101
j29	ud <op> , hs c8	- - take formal	102
j30	ud <op> , hs c25	- - controlled formal	103
j31	ud <op> , hs c24	- - assign to formal subscr.	104
j32	hs p<blockrel>	goto for label word in stack	34,35
j33	hv/hh s<reladdr>	for label word when jump is local	34,35

2.9.6. Block entries

A block is generated with the following format:

[b1]	qq <stack appetite>, hs c7	10
[b2]	hv c11 , hh <displ ref>	16,17,18
[b3]	<block parameters>	11,110
[b4]	<goto block code>	10

<block parameters> are references to explicite program points. They appear in the program in the opposite order of that in the Algol text. The possible parameter formats are described under reference p1-11 and j8.

<goto block code> is treated by RS as a parameter, but with one of the formats j1, j25 or j26.

2.9.7. Procedure calls

The format of a procedure call is the following

```
[pc1] qq <stack appetite>,  hs c7                                          6,7
[pc2] arn              ,  <program point> ; return information          14
[pc3] <call parameters>                                                   19
[pc4] isf <displref>   ,  ps s <blockrel> ; exit call                     7
```

<call parameters> are described under reference p6-17, p21-28 and j8

ref. c4 may be replaced by j1 if the call refers to a slow standard procedure.

2.9.8. Case administration

The format of the case-administration code is the following:

```
[cs1] qq <no of param>    ,  hs c15;   [f5]<--     [c15]<--          12,13
      <case error action>                                           12,13
      <goto end case>                                               12,13
      <case parameters>                                           20(+12)
```

<case error action> is either if the case administration appears as the body of a label procedure (switch)

```
[cs2] pa    c30                     ;    indicate dummy switch action

     else

[cs3] pt    c49-4 ,  hs c27         ;    exit to case error
```

<goto end case> is an unconditional jump (j1-2)

<case parameters> are described under reference p6-11, p18-20, p22 or may be a jump to evaluation of a case parameter expression local on the track:

```
[cs4] it    (c16) ,  hv/hh <reladdr>
```

2.9.9. Reference table

ref	instruction	meaning	Used in block entries	Used in proc. call	Used in case adm.
p1	qq , <programpoint>	label	X		
p2	zq , < - >	integer proc. with parameters	X		
p3	zqf , < - >	real - - -	X		
p4	zqn , < - >	boolean - - -	X		
p5	arn , < - >	label - - - (switch)	X		
p6	zqfn , < - >	no type expr. or proc.	X	X	X
p7	psn , < - >	integer - - - no param.	X	X	X
p8	psfn , < - >	real - - - - -	X	X	X
p9	qqn , < - >	boolean - - - - -	X	X	X
p10	qqf , < - >	label - - - - -	X	X	X
p11	qqfn , < - >	string - - - - -	X	X	X
p12	ps (<displref>) , it s<blockrel>	integer simple		X	
p13	psf (< - >) X <blockrel> NTB	real -		X	
p14	ps (< - >) , it n s<blockrel>	boolean -		X	
p15	psnf (< - >) X <blockrel> NQB	string -		X	
p16	psf (< - >) X < - > NQB	label		X	
p17	isf (< - >) , pmns<blockrel>	described in stack		X	
p18	is (< - >) , < - >	float simple			X
p19	is (< - >) , < - >	non float simple			X
p20	is (< - >) X <blockrel> NTB	label			X
p21	qq <arrayrel>.9+<doperel>.19+<no.of ind>.39	array: first paramword		X	
	psf (<displ ref>), hv c12	- : second paramword		X	
p22	<anything no marks>	constant value		X	X
p23	psn (c10) , it s<stackrel>	integer constant		X	
p24	psfn (c10) X < - > NTB	real -		X	
p25	psn (c10) , itns< - >	boolean -		X	
p26	ps <programpoint>	integer subscr. var		X	
p27	psf < - >	real - -		X	
p28	ps < - >	boolean - -		X	

Appendix 3: EXAMPLE OF TEST OUTPUT FROM GIER ALGOL 4

Below follows a small ALGOL program and the test output from the compilation. The first three lines from pass 1 are given exactly as they have been printed by the compiler; after this follows the whole test output with comments inserted below each line of bytes.

```
Demonstration of factorial
begin
integer procedure fact (n); value n; integer n;
fact:= if n < one then one else n × fact (n-one);
integer n, one;
one:= 1; writecr;
for n:= one step one until 10 do write (<ddddddd>, fact(n));
end
```

```
begin
1.   93  1010   116   137     6     1     3    20   184    14
    291   167   157    14   167   116    14   167  1010     6
      1     3    20   228   106    14   272    15    14     5
```

Demonstration of factorial

```
begin
1.   93  1010    116    137     6      1      3     20   184    14
   begin   CR integer  proc     f      a      c      t     (     n

    291   167    157     14   167    116     14    167  1010     6
      )     ;  value      n     ;  integer    n      ;    CR     f

      1     3     20    228   106     14    272     15    14     5
      a     c      t     :=    if      n     ≤       o     n     e

    231    15     14      5   176     14    267      6     1     3
   then     o      n      e  else      n     ×       f     a     c

     20   184     14     76    15     14      5    291   167  1010
      t     (      n      -     o      n      e      )     ;    CR

    116    14    220     15    14      5    167   1010    15    14
integer     n      ,      o     n      e      ;     CR     o     n

      5   228     58    167    23     18      9     20     5     3
      e    :=      1      ;     w      r      i      t     e     c

     18   167   1010     98    14    228     15     14     5   196
      r     ;     CR    for     n     :=      o      n     e  step

     15    14      5    198    58     57    243     23    18     9
      o     n      e  until     1      0     do      w     r     i

     20     5    184   1013     0      0    476      0   220     6
      t     e      (  < literal      <ddddddd>       >     ,     f

      1     3     20    184    14    291    291    167  1010   172
      a     c      t      (     n      )      )      ;    CR   end

   1009     0
endpass  fill
```

2.

93	77	116	137	1021	184	1020	291	167	157
begin	CR	integer	proc	fact	(	n	)	;	value
1020	167	116	1020	167	77	1021	228	106	1020
n	;	integer	n	;	CR	fact	:=	if	n
272	1019	231	1019	176	1020	267	1021	184	1020
≤	one	then	one	else	n	×	fact	(	n
76	1019	291	167	77	116	1020	220	1019	167
-	one	)	;	CR	integer	n	,	one	;
77	1019	228	58	167	1018	167	77	98	1020
CR	one	:=	1	;	writecr	;	CR	for	n
228	1019	196	1019	198	58	57	243	1017	184
:=	one	step	one	until	1	0	do	write	(
80	0	0	476	0	220	1021	184	1020	291
<	literal	⟨ddddddd⟩		>	,	fact	(	n	)
291	167	77	172	283	0				
)	;	CR	end	endpass	fill . . .				

3.

0	6	1017	31	1018	1016	20	0	0	1021
endpass	6.std.=write		31.std.=writecr		free id.	begin	CR	CR	fact
43	1020	57	30	1021	77	129	1020	493	1019
declare	n	spec.int.	end.sp.	fact	:=	ifex	n	≤	one
131	1019	132	1020	146	1021	39	1020	145	1019
thenex	one	elseex	n	×	fact	beg.func.	n	-	one
31	134	75	1	19	0	1020	1019	5	0
endcall	endelseex	endass.	<1,	endtypeproc>	CR	n	one	declare	CR
1019	77	0	0	0	1	1	75	21	1018
one	:=	<	constant =		1	>	endass.	;	writecr
128	21	0	138	1020	71	1019	139	1019	140
proc;	;	CR	for	n	:=for	one	step	one	until
0	0	0	10	1	163	22	1017	38	0
<	constant = 10			>	stepdo	do	write	beg.call	<
0	476	0	3	34	1021	39	1020	31	31
constant = ⟨ddddddd⟩			>	call,	fact	beg.func	n	endcall	endcall
133	142	21	0	16	0				
deletecall	enddo	;	CR	endblock	fill . . .				

4.	2	17	0	142	133	177	177	1020	1	3
	endpass	endblock	CR	enddo	deletecall	endcall	endcall	n	<1, beg.func.>	
	1021	180	0	476	0	0	86	2	22	1017
	fact	,	<		constant = ‹ddddddd›		>	<2, beg.call.>		write
	172	163	10	0	0	0	84	140	1019	139
	do	stepdo	<		constant = 10		>	until	one	step
	1019	185	1020	138	0	128	1018	189	1	0
	one	:=for	n	for	CR	proc;	writecr	endass.	<	
	0	0	84	193	101	1019	0	15	1019	1020
	constant = 1		>	prepass.	:=	one	CR	endhead	one	n
	0	195	1	19	189	134	177	1019	145	1020
	CR	bypas lab.	<	(1) >	endass.	endelseex	endcall	one	-	n
	1	3	1021	146	1020	132	1019	131	1019	493
	<1, begfunc>		fact	×	n	elseex	one	thenex	one	≤
	1020	129	193	101	1021	15	16	1020	68	1021
	n	ifex	prepass.	:=	fact	endhead	enddecl.	n	value	fact
	195	1023	1023	37	1021	0	0	16	1021	48
	bypass lab.	<	(2)	>	fact	CR	CR	enddecl	fact	decl
	1018	20	1020	1019	52	1021	1023	1016	1017	1020
	int.val.	spec.	n	one	decl.int.	<		(3)		
	2	0								
	>	fill . . .								

Notes: Read the output from pass 4 backwards
(1): Number of parameters, end type proc.
(2): Base working locations, base variables, begin integ. proc. with params.
(3): These 6 bytes are:
base working locations outermost block
base variables - -
free identifier
standard identifier having lowest std. ident. no. in output pass 3
1021 - maximum block number - number of owns = relative stackref. 0
2 + number of owns.

5.	1020 sr0	197 <beg.block,	1021 basew>	196 CR	196 CR	198 <beg.proc,	1021 int.,	1023 basew>	195 byp.lab.	445 <
	2 n	1023 >	200 value	444 <	1023 fact	1023 >	191 :=	193 prepass.	129 ifex	444 <
	2 n	1023 >	493 ≤	444 <	1022 one	0 >	131 thenex	444 <	1022 one	0 >
	132 elseex	444 <	2 n	1023 >	146 ×	440 <	1023 fact,	0 integ.val.	477 >	214 <begfunc.
	1 , 1 >	444 <	2 n	1023 >	145 -	444 <	1022 one	0 >	177 endcall	134 endelseex
	189 endass.	206 <endtypepr,	1 1>	195 byp.lab.	196 CR	196 CR	444 <	1022 one	0 >	191 :=
	193 prepass.	468 <	0	0 constant = 1	0	1 >	189 endass.	1020 writecr	128 proc;	196 CR
	138 for	444 <	1021 n	0 >	185 :=for	444 <	1022 one	0 >	139 step	444 <
	1022 one	0 >	140 until	468 <	0	0 constant = 10	0	10 >	163 stepdo	172 do
	1018 write	208 <begcall,	2 2 >	470 <	0	0 constant = <ddddddd>	476	0 >	180 call,	440 <
	1023 fact,	0 integ.	477 val. >	214 < begfunc.,	1 1>	444 <	1021 n	0 >	177 endcall	177 endcall
	133 deletecall	142 enddo	196 CR	204 endblock <	213 endpass,	5 5,	1020 sr0, include	12 std.tracks	11 12,	10 11
	9 10, 9 >	0 fill . . .								

Identifiers in output from pass 5:

	kind-type	rel. addr.	block	specs
fact (as proc):	440=int.proc.w.par.	-1	0	int.val.
n (in block):	444=simple int.	-3	0	
one:	444 -	-2	0	
fact (as val.):	444 -	-1	1	
n (in fact):	444 -	2	1	

writecr: 1020
write: 1018

6.	1020	5	1021	96	96	2	1021	1023	86	71
	sr0	<beg.block,basew>		CR	CR	<beg.proc.,	int,	basew>	byp.lab.	<
	2	1023	84	71	1023	1023	116	10	71	2
	n	>	value	<	fact	>	address	if	<	n
	1023	71	1022	0	55	71	1022	0	37	1
	>	<	one	>	<then	<	one	>	< else	int.>
	71	2	1023	73	1023	0	0	71	2	1023
	<	n	>	<	fact	>	begcall	<	n	>
	71	1022	0	46	78	1	1	47	39	1
	<	one	>	-	< param,	, int.>	endcall	×int.	<endelse,	int.>
	42	43	4	1	86	96	96	71	1022	0
	prepass.	:=	<endtypepr.,	1>	byp.lab.	CR	CR	<	one	>
	116	75	0	0	0	1	42	43	106	30
	address	<		constant = 1		>	prepass.	:=	writecr	proc;
	96	13	71	1021	0	14	71	1022	0	20
	CR	for	<	n	>	:=for	<	one	>	step
	71	1022	0	21	75	0	0	0	10	23
	<	one	>	until	<		constant = 10		>	stepdo
	74	1020	0	0	75	0	0	476	0	78
	<	write	>	begcall	<	constant	=	<ddddddd>	>	<param
	3	73	1023	0	0	71	1021	0	78	1
	, bool.>	<	fact	>	begcall	<	n	>	<param,	int.>
	1	78	1	1	36	77	96	6	87	5
	endcall	<param,	int.>	endcall	proc;	enddo	CR	endblock	<endpass,	sr0
	1020	12	11	10	9	0				
	, include	std.	tracks 12,	11, 10,	9 >	fill . . .				

Identifiers in output from pass 6:

```
fact (as proc):  proc.,    reladdr.,   block
fact (as var),
n, one        :  simple,      -          -
writecr       :  106
write         :  std.proc,  trackno.,   reladdr.
```

```
7.  116      10        112   112   1021    11        31        2    1    102
    endpass  begblock  CR    CR    <int.,  begproc>  byp.lab.  <    n    formal>

    2    1    70     39       2    1    124    21   1022   2
    <    n    M:=>   intval.  <    n    :=R>   if   <      one

    121   2    1    123    23      1022   2     121    1       32
    R:=>  <    n    R- >   goif-   <      one   R:=>   <int.   else>

    2    1    70     1022   1     71     1023   0      7          2
    <    n    M:=>   <      w1    :=M>   <      fact   begcall>   <

    1    121    1022   2     123    0    5     124    0    5
    n    R:=>   <      one   R- >   <    UV    :=R>   <    UV

    1      19       0          14          0    5     70     1022   1     93
    int.   param>   <0 lits.   endcall>    <    UV    M:=>   <      w1    int.

    48      1         33         1023   1      124    1      109       1020        1023
    mult>   < int.    endelse>   <      fact   R:=>   < 1    formal,   appetite,   block

    17         31         112   112   0    0    0              0    8    70
    endproc>   byp.lab.   CR    CR    <         constant = 1             M:=>

    1022   2     71     41        112   21    1022   2     70     1021
    <      one   :=M>   writecr   CR    for   <      one   M:=>   <n:=

    2            91      1020   1     36          31         1022   2     121    1021
    M; first:=   true>   <      w2    forlab.>    byp.lab.   <      one   R:=>   <

    2    122    1021          2            128     1021    2           94       10    0
    n    R+>    <if-,first    then         n:=R>   <else   first:=     false>   <

    0    0               8    121    1021   2    123    1022   2    75
    constant = 10             R:=>   <      n    R->    <      n    xsig( >

    28         31         1020   0       6          0    476   0            0          8
    goenddo-   byp.lab.   <      write   std.cal>   <          constant = <ddddddd>

    3       19       1023   0      7          1021   0    9        1     19
    bool.   param>   <      fact   begcall>   <      n    simple   int   param>

    0          14          0    5     1     19       1         14          1020   1
    <0 lits.   endcall>    <    UV    int   param>   <1 lit    endcall>    <      w2

    35       112   1018        0       18          1020   12        11     10       9
    enddo>   CR    <appetite   block   endblock>   < sr0  include   std.   tracks   12, 11, 10

    5      0
    , 9>   fill . . .
```

Variables in output from pass 7:

n (in fact):	relative 2,	local block
one:	- -2,	outermost block
w1:	- -2	local block
fact (proc):	- -1,	block 0
UV:	- 0,	UV cells
fact (val.):	- -1,	local block
n (in block):	- -3,	outermost block

IV. Index

In the following, page references for especially important descriptive or definitional information are underlined.