A MANUAL OF GIER ALGOL III

as developed by

Henning Christensen, Gunnar Ehrling, Jørn Jensen, Peter Kraft,
Paul Lindgreen, Peter Naur, Knut-Sivert Skog and Peter Villemoes

Second edition of A Manual of GIER ALGOL
With a page of corrections and additions
Edited by Peter Naur

A/S REGNECENTRALEN, COPENHAGEN

1965

Corrections and additions

Page 9, section 7.3. Add a paragraph:

The translator will assume that expressions of the form i↑j where i and j are of type integer are also of type integer. This is not correct if j is negative. This may cause a non-integer value to be assigned to a variable declared to be integer.

Page 12, 2nd line from below, add 4 letters to read:

```
writetext(‹<Q,=,›)
```

Page 13, line 13 from below, add (7) to read:

```
writechar(7)
```

Page 16, line 9 from below, move + one position to the left to read:

```
,+ 123.456 8,
```

Page 34, line 9 from below, change e to c to read:

```
c70, c60, ...
```

Page 38, section 11.4.4, add a paragraph:

The most serious defect of the checking is that the types of actual parameters in most cases are not checked against the specifications of the corresponding formal parameters, neither during translation nor at run time.

Page 44, section 12.4.3, add the sentence:

The variable given as parameter to gier must be of type Boolean.

Page 49, section 12.6.3, add the sentence:

During execution gierdrum will need an array in the core store big enough to hold the code read in from tape. This may cause the capacity of the core store to be exceeded.

Page 55, line 18 from above, change to) to read:

```
r[i)
```

Page 57, lines 17 to 21, change the identifiers skrvkopi, skrvml, skrvtegn, streng, sættegn, trykkopi, trykml, tryktegn, and tryktom, to read:

writecopy, writechar, setchar, outcopy, outsp, outchar

Page 57, line 22 from above, add an extra line to read:

```
624        ,<i op>,
```

Page 57, line 26 from above, change the identifiers tiltromle and fratromle, to read, respectively:

to drum and from drum

Page 58, appendix 5, add the sentence:

The emergency output may be called during the execution of any program by simulating an arithmetic overflow by transferring the control of the machine to the instruction in location 0 by manual action.

Page 61, line 16 from above, change c to e to read:

```
26+c60+e96
```

Contents.

The ALGOL 60 Report.

Throughout the present Manual reference is made to the ALGOL 60 Report or the Revised ALGOL 60 Report. The differences between these two documents are slight and do not influence the numbering of sections. The full references of these reports are as follows:

J. W. Backus, et. al., Report on the Algorithmic Language ALGOL 60 (ed. P. Naur), Numerische Mathematik 2 (1960), pp. 106-136; Acta Polytechnica Scandinavica: Math. And Comp. Mach. Ser. no. 5 (1960); Comm. ACM 3 no. 5 (1960), pp. 299-314.

J. W. Backus, et. al., Revised Report on the Algorithmic Language ALGOL 60 (ed. P. Naur), Regnecentralen, Copenhagen (1962), Comm. ACM 6 no. 1 (1963), pp 1-17; Computer Journal 5 (1963), pp. 349-367; Numerische Mathematik 4 (1963), 420-453.

INTRODUCTION.

The decision that an ALGOL compiler for the GIER should be written was made in January 1962. The work was started almost immediately and in August 1962 a preliminary version of the compiler could be distributed to all GIER installations. This version was complete except for some standard input and output procedures. The first definitive version, which also corrected a number of errors found through the extensive practical use of the preliminary version, was distributed in February 1963.

Like its predecessor DASK ALGOL the GIER ALGOL language lies sufficiently close to the ALGOL 60 reference language to make it practical to use the ALGOL 60 Report directly as the basic manual. The exact specifications of GIER ALGOL are then defined through the set of corrections and additions of the ALGOL 60 Report given in the present Manual. Because of this intimate relation to the ALGOL 60 Report the numbering of sections within the present Manual have been chosen to be a direct continuation of the section numbers of the ALGOL 60 Report.

The present second edition of the Manual describes the version of the compiler known as GIER ALGOL III and distributed by February 1964. The difference between this new version and the version described in the first edition consists in the following: (a) The new version uses English language throughout. (b) Some standard procedures have been removed. (c) Several new standard procedures have been included, to give the user the access to using machine language. (d) A system for producing an output of the variables of the program as they exists at the time of an alarm situation during program execution has been added. (e) Passes 1 and 2 have been completely rewritten. In this way the speed of pass 1 will match the 2000 characters/second tape reader developed at Regnecentralen. Likewise, the speed of pass 2 has been increased considerably. (f) The block entry administration has been speeded up somewhat. (g) There is a choice of several ways of storing the compiler on the drum, and a version which reads the translator from tape during translation is available.

The more important of these differences may be studied by comparing the following sections of the first and second editions of the Manual: 8.2, 9.1, 9.6, 11.1, 11.3.11, 11.4.5, 12, appendices 3, 5, and 6.

Those interested in the internal working of the system are referred to: Peter Naur, The Design of the GIER ALGOL Compiler, BIT Vol. 3 (1963), 124-140 and 145-166.

The new edition of the Manual was typed by Kirsten Andersen, as was the first edition.

6. 8-CHANNEL PUNCH TAPE CODE AND FLEXOWRITER KEYBOARD.

6.1. PRINTED SYMBOLS.

Lower case	Upper case	Code	Lower case	Upper case	Code
a	A	`, oo  .  o,`	w	W	`,  o  .oo ,`
b	B	`, oo  . o ,`	x	X	`,  oo .ooo,`
c	C	`, ooo . oo,`	y	Y	`,  ooo.   ,`
d	D	`, oo  .o  ,`	z	Z	`,  o o.  o,`
e	E	`, ooo .o o,`	æ	Æ	`, ooo .   ,`
f	F	`, ooo .oo ,`	ø	Ø	`, o oo. oo,`
g	G	`, oo  .ooo,`	0	^	`,  o  .   ,`
h	H	`, oo o.   ,`	1	v	`,     .  o,`
i	I	`, oooo.  o,`	2	×	`,     . o ,`
j	J	`, o o .  o,`	3	/	`,   o . oo,`
k	K	`, o o . o ,`	4	=	`,     .o  ,`
l	L	`, o   . oo,`	5	;	`,   o .o o,`
m	M	`, o o .o  ,`	6	[	`,   o .oo ,`
n	N	`, o   .o o,`	7	]	`,     .ooo,`
o	O	`, o   .oo ,`	8	(	`,    o.   ,`
p	P	`, o o .ooo,`	9	)	`,   oo.  o,`
q	Q	`, o oo.   ,`	,	⏨	`,  ooo. oo,`
r	R	`, o  o.  o,`	.	:	`, oo o. oo,`
s	S	`,  oo . o ,`	–	+	`, o   .   ,`
t	T	`,  o  . oo,`	<	>	`,  oo .  o,`
u	U	`,  oo .o  ,`	_	|	`,    o.oo ,`
v	V	`,  o  .o o,`			

The key for _| does not advance the carriage.

6.2. TYPOGRAPHICAL SYMBOLS.

LOWER CASE `, oooo. o ,` UPPER CASE `, oooo.o  ,` SPACE `,   o .   ,`
CAR RET `,o    .   ,` TAB `,  ooo.oo ,`

6.3. CONTROL SYMBOLS.

STOP CODE `,    o. oo,` TAPE FEED `, oooo.ooo,` PUNCH ADRES `,o    .   ,`
PUNCH OFF `,  o o.ooo,` PUNCH ON `, o  o.o  ,` AUX CODE `,     o.o  ,`
PUNCH ADRES and AUX CODE insert their respective codes when depressed simultaneously with any other key.

6.4. FLEXOWRITER KEYBOARD.

```
             START  STOP  PUNCH              AUX   STOP  TAPE
             READ   READ  ADRES              CODE  CODE  FEED

TAB  PUNCH    ×    /    =    ;    [    ]    (    )    ^    v    |   PUNCH
      OFF     2    3    4    5    6    7    8    9    0    1    _    ON

               Q    W    E    R    T    Y    U    I    O    P    >   CAR
               q    w    e    r    t    y    u    i    o    p    <   RET

LOWER           A    S    D    F    G    H    J    K    L    Æ    Ø    LOWER
 CASE           a    s    d    f    g    h    j    k    l    æ    ø    CASE

UPPER            Z    X    C    V    B    N    M    ⏨    :    +      UPPER
CASE             z    x    c    v    b    n    m    ,    .    -      CASE
```

6.5. NUMERICAL REPRESENTATIONS.

In the following table the characters have been arranged according to the numerical equivalent of the hole combination (after removal of the parity check hole). The first column gives the decimal value of the character, the second and third columns give the lower and upper case character, respectively, and the fourth column contains a G in the cases where the character is available only in GIER, but not on the flexowriter

	LOWER	UPPER			LOWER	UPPER	
0	SPACE			32	–	+	
1	1	∨		33	j	J	
2	2	×		34	k	K	
3	3	/		35	l	L	
4	4	=		36	m	M	
5	5	;		37	n	N	
6	6	[		38	o	O	
7	7	]		39	p	P	
8	8	(		40	q	Q	
9	9	)		41	r	R	
10	(NOT USED)			42	(NOT USED)		
11	STOP CODE			43	ø	Ø	
12	END CODE			44	PUNCH ON		
13	å	Å	G	45	(NOT USED)		
14		|		46	(NOT USED)		
15	(N̄OT USED)			47	(NOT USED)		
16	0	∧		48	æ	Æ	
17	<	>		49	a	A	
18	s	S		50	b	B	
19	t	T		51	c	C	
20	u	U		52	d	D	
21	v	V		53	e	E	
22	w	W		54	f	F	
23	x	X		55	g	G	
24	y	Y		56	h	H	
25	z	Z		57	i	I	
26	(NOT USED)			58	LOWER CASE		
27	,	$_{10}$		59	.	:	
28	CLEAR CODE			60	UPPER CASE		
29	RED RIBBON		G	61	SUM CODE		
30	TAB			62	BLACK RIBBON		G
31	PUNCH OFF			63	TAPE FEED		
				64	CAR RET		

7. THE RELATION BETWEEN GIER ALGOL AND ALGOL 60.

7.1. BASIC SYMBOLS.

7.1.1. Single character symbols.
7.1.1.1. Letters and digits. GIER ALGOL adds the letters
æ Æ ø Ø
to the reference alphabet. The appearance of all letters and digits may be seen from section 6.
7.1.1.2. Delimiters. As apparent from section 6 the following simple reference language symbols are directly available in GIER ALGOL:
$+ - \times / < = > \vee \wedge , . {}_{10} : ; ( ) [ ]$

7.1.2. Compound symbols.
Compound symbols must appear exactly as shown in this section, without additional SPACE or CARRET symbols.
7.1.2.1. Underlined words. Underlined words are produced in GIER ALGOL by depressing the underline (_) key immediately preceding each letter of the word. The symbols are the following:
true false go to if then else for do step until while comment begin end own Boolean integer real array switch procedure string label value
Boolean and boolean may be used interchangeably. Also go to, goto, and go to.
7.1.2.2. Compound symbols similar to reference language. The following compound symbols, most of which are produced by combining the underline (_) or stroke (|) with other characters, are similar to those of the reference language:
$\underline{<}$ $\underline{>}$ $\neq$ $\underline{=}$:=
7.1.2.3. Compound symbols differing from reference language. The following compound symbols show a noticable deviation from the reference language:

Reference language	↑	¬	␣	‘	’	÷	⊃
GIER ALGOL	⅄	-ˌ	⊥	⊀	⊁	$\underline{:}$	=>

7.2. USE OF comment.

Following the delimiter comment any sequence of characters specified in section 6.5 is admitted up to the first following semicolon (;). Comments have no effect in GIER ALGOL.

7.3. THE TREATMENT OF VARIABLES OF TYPES integer AND real.

Variables of types integer and real are represented by normal floating point numbers in GIER. Therefore integers must be confined to the range:
- 2↑29 = - 536 870 912 ≤ integer ≤ 536 870 911 = 2↑29 - 1
while the range of non-zero real variables is:
2⅄(-512) = $7.458_{10}-155$ < abs(real) < $1.341_{10}154$ = 2⅄512

7.3. THE TREATMENT OF VARIABLES OF TYPES integer AND real.

If in the course of a calculation an expression, which according to the rules of section 3.3.4 is of type integer, yields a result outside the range for integers, the result will be represented by too few significant figures and will therefore in general be inexact.

Round-off from type real to type integer is performed by means of the built-in machine instructions for conversion from floating form to fixed form and back again (tkf -29, nkf 39). This implies that real results in the range from 0 to 2↑29 will yield correct integers on rounding, while reals in the range from 2↑29 to 2↑39 will be rounded to an integer having too few significant figures. Real results larger than 2↑39 will yield completely erroneous results if rounded.

The integer divide operation (:) will sometimes give a result which is incorrect by unity if the absolute value of the term involved is greater than 268 435 455.

The accuracy of a real number will correspond to 29 significant binary digits. Thus one unit in the last binary place will correspond to a relative change of the number of between $2_{10}-9$ and $4_{10}-9$.

7.4. RESERVED IDENTIFIERS.

A reserved identifier is one which may be used in a program for a standard purpose without having been declared in the program. If the standard meaning is not needed in a program the identifier may freely be declared to have other meanings.

The complete list of reserved identifiers arranged alphabetically is as follows:

Identifier	Reference	Identifier	Reference
abs	3.2.4	outcr	8.6
arctan	3.2.4, 7.5	output	8.3
char	9.11	outsp	8.5
cos	3.2.4, 7.5	outsum	8.7
drum place	10.5	outtext	8.4
entier	3.2.5, 7.5	pack	12.1
exp	3.2.4, 7.5, 11.7	setchar	9.10
from drum	10.5	sign	3.2.4
gier	12.4	sin	3.2.4, 7.5
gierdrum	12.6	split	12.2
gierproc	12.5	sqrt	3.2.4, 7.5
inchar	9.9	todrum	10.5
inone	9.5	typechar	9.9
input	9.4	typein	9.8
kbon	9.6	write	8.3
ln	3.2.4, 7.5, 11.7	writechar	8.8
lyn	9.12	writecopy	9.7
outchar	8.8	writecr	8.6
outclear	8.7	writetext	8.4
outcopy	9.7		

7.5. STANDARD FUNCTIONS.

7.5.1. Accuracy.

The algorithms for calculating the standard functions arctan, cos, exp, ln, sin, and sqrt, incorporated in GIER ALGOL will all yield results having an error less than that which corresponds to about 2 units in the last place of the result or the argument, whichever gives the greater error.

7.5.2. Alarms.

Certain misuses of the standard functions will cause termination of execution of program (see section 11.7). Note, however, that ln(0) will supply the result $-9.35_{10}49$ and not call the alarm.

7.6. ARITHMETIC EXPRESSIONS.

The treatment of arithmetic types and the accuracy of <u>real</u> arithmetics is described in section 7.3. Alarms are described in section 11.7.

7.7. (This section has been deleted).

7.8. INTEGERS AS LABELS.

Integers cannot be used with the meaning of labels in GIER ALGOL.

7.9. FOR STATEMENTS.

In GIER ALGOL a subscripted variable is permitted as the controlled variable in a for clause. The identity of the variable will be established once at the beginning of each activation of the for statement and changes of the values of subscript expressions in the course of the execution of the controlled statement will have no influence on which variable is used as the controlled one.

7.10. PROCEDURE STATEMENTS.

7.10.1. Recursive procedures.

Recursive procedures will be processed fully in GIER ALGOL.

7.10.2. Handling of types.

The types <u>integer</u> and <u>real</u> will be handled according to the prescriptions of section 4.7.3 except in the case that a formal parameter, which is specified to be <u>real</u> and to which assignments are made, in the call corresponds to an <u>integer</u> declared variable. This special case will be treated incorrectly in GIER ALGOL.

7.10.3. Extended list of standard procedures.

All input and output functions are in GIER ALGOL expressed as calls of standard procedures. These calls conform to the syntax of calls of declared procedures (cf. section 4.7.1) and also should be regarded in all other respects as regular procedure calls or function designators, as the case may be. This specifically includes the activation of a standard procedure through its identifier appearing as an actual parameter of a call of a declared procedure.

7.11. ORDER OF DECLARATIONS.

In GIER ALGOL declarations may appear in any order in the block head.

7.12. Own.

In GIER ALGOL own can only be used with type declarations, not with array declarations.

7.13. PROCEDURE DECLARATIONS.

7.13.1. Recursive procedures.

Recursive procedures will be processed fully in GIER ALGOL.

7.13.2. Arrays called by value.

GIER ALGOL cannot handle arrays called by value.

7.13.3. Specifications.

The specifications for formal parameters must be complete, i.e. each parameter must occur just once in the specification part.

7.13.4. Labels called by value.

Labels cannot be called by value in GIER ALGOL (the Revised ALGOL 60 Report leaves the question unanswered).

7.14. GENERAL LIMITATIONS.

GIER ALGOL imposes a number of limitations caused by the finite size of the tables used during compilation. However, with one exception these limitations shall not be mentioned further here, partly because only very exceptional programs are likely to exceed the capacity, partly because alarm messages during compilation will indicate when they are violated (see appendix 4). The exception is the limitation that the number of variables which are active simultaneously at any time during the execution of a program must be confined to about 700. This problem is discussed in detail in section 10.

8. STANDARD OUTPUT PROCEDURES.

Output of text and results from a program will be controlled by means of output procedures permanently available to the translator (i.e. without explicit declarations). The output will be provided in the form of 8-channel punch tape or printed copy. The symbols and 8-channel code given in section 6. 8-CHANNEL PUNCH TAPE CODE AND FLEXOWRITER KEYBOARD will be used.

8.1. CONTROL OF TYPEWRITER AND OUTPUT PUNCH.

Half of the standard output procedures are available in two forms, one controlling the output punch (identifier beginning with out), the other controlling the on-line typewriter (identifier beginning with write). By operator intervention it is however possible to make a free choice of the output unit corresponding to the two sets of output procedure identifiers. See the section 11.6 CHOICE OF OUTPUT UNITS OR STOP RUN.

8.2. IDENTIFIERS AND MAIN CHARACTERISTICS.

The identifiers and main characteristics of the standard output procedures are the following:

Identifier	Example, reference	Effect
output write	write(≮+d,ddd≯,q↑2) section 8.3.	Outputs the values of an arbitrary number of arithmetic expressions in a specified layout. Other output operations may also be inserted as parameters.
outtext writetext	write(≮<Q,=,≯) section 8.4.	Outputs a specified string of symbols.

outsp	outsp(8-n) section 8.5.	Outputs a specified number of SPACEs.
outcr writecr	writecr section 8.6.	Outputs one CAR RET symbol.
outclear	outclear section 8.7.	Punches one CLEAR CODE symbol and sets an internal sum of punched symbols to zero.
outsum	outsum section 8.7.	Punches a STOP CODE, a SUM CODE and a code representing the sum of the symbols punched since program read in, last outclear or last outsum.
outchar writechar	writechar section 8.8.	Outputs the character correspon-ding to the value of the parame-ter.
outcopy writecopy	outcopy(≮</;≯) section 9.7.	Copies a section of the input tape to the output, the section being specified through a parameter.

It holds for all standard output procedures that each output operation will cause an addition to an internal variable of a number which is equivalent to the character. This may be used for checking purposes by means of the mechanisms described in sections 8.7.2 and 9.2. It should be noted, however, that for the checking to work correctly the output tape must not include any character which has been produced by a write - operation (cf. section 8.1).

8.3. STANDARD PROCEDURES: output, write.

8.3.1. Syntax.

```
<sign> ::= <empty>| - | + | ±
<exponent layout> ::= ⏨<sign>d|<exponent layout>d
<zeroes> ::= 0 |<zeroes>0 | <zeroes>,0
<positions> ::= d | <positions>d | <positions>,d
<0-positions> ::= <positions> | <0-positions>0 | <0-positions>,0
<decimal layout> ::= <0-positions>|<0-positions>.<zeroes>|
     <positions>.<0-positions>|.<0-positions>
<layout tail> ::= <decimal layout>|<decimal layout><exponent layout>
<layout> ::= <sign><layout tail>|<sign>n<layout tail>|<sign> n|
     <sign>n,<layout tail>
<general layout> ::= ≮<layout>≯|<formal parameter>|(<layout expression>)
<layout expression> ::= <general layout>|
     <if clause><general layout> else <layout expression>
<out statement> ::= <output statement>|<outtext statement>|
     <outsp statement>|<outcr statement>|<outclear statement>|
     <outsum statement>|<outcopy statement>|<outchar statement>
<output parameter> ::= <arithmetic expression>|<out statement>
<output parameter list> ::= <output parameter>|
     <output parameter list><parameter delimiter><output parameter>
<output statement> ::=
     output(<layout expression><parameter delimiter><output parameter list>)|
     write(<layout expression><parameter delimiter><output parameter list>)
```

8.3.2. Examples.

```
output(≮ddd.00≯, P, outcr, outtext(≮<=≯), w +s)
output(≮ d⏨-dd≯, epsilon/16)
output(≮dd,ddd≯, Q, outsp(5), output(≮.ddd≯, q), W, t-3)
output(if s>0 then f1 else f2, Sum)
output(m, p-q, s+t)
```

8.3.3. Semantics.

A call of the procedure output or write causes the following treatment of the parameters specified in the output parameter list:

Arithmetic expression: the value will be printed in the layout supplied in the first parameter of the call.

Out statement: the call of the statement will be executed.

8.3.4. The layout.

The layout expression will be evaluated once at the beginning of the execution of the output or write statement. The evaluation will take place in a way which is completely analogous to that of other expressions (cf. section 3.3.3). The final value must always be of the form ≮<layout>≯.

The symbols of the layout give a symbolic representation of the digits, spaces and symbols as they will appear in the printed number. Indeed, the finally printed number will have exactly the same number of printed characters as is present in the layout (except in case of alarm printing, see section 8.3.6). The various symbols of the layout have the following significance:

8.3.4.1. Sign. The four possible symbols in the sign position signify the following:
8.3.4.1.1. Empty. The number is supposed to be positive. No sign will be printed. If a negative number is encountered, an alarm printing will take place (see section 8.3.7).
8.3.4.1.2. . The sign will always be printed using SPACE for positive, and - for negative numbers. It will, if possible, move to the right, appearing as the first or second symbol to the left of the first digit (a layout SPACE may appear in between) or immediately in front of the decimal point.
8.3.4.1.3. + . The sign will always be printed using + for positive and - for negative numbers. It will, if possible, move to the right, as in 8.3.4.1.2 above.
8.3.4.1.4. $\underline{+}$. The sign will always be printed, using + for positive and - for negative numbers. It will be printed as the first symbol of the number, before any SPACE or digit.

8.3.4.2. Digits. Letters d and n represent digits. Letter n may only appear as the first symbol following the sign. The total number of letters d and n gives the maximum number of printed significant digits (cf. section 8.3.8).

If n is used in the first digit position, proper decimal fractions will be printed with a 0 in front of the decimal point and the integer 0 will be printed. If d is used these 0-digits will be replaced by SPACE.

8.3.4.3. Zeroes. Zeroes may appear at the end of a decimal layout. They influence the representation of the number in the following manner: If m zeroes are present at the end of the decimal layout the exponent printed will be exactly divisible by m+1. For this to be possible at the same time as the position of the decimal point within the complete layout is kept fixed the significant digits of the number are allowed to move to the right, using the positions of the symbols 0, depending on the magnitude of the number. If no exponent layout is included the exponent 0 is understood and the above rule holds unchanged.

8.3.4.4. Spaces. Spaces will be inserted in all positions where the symbol , appears. The symbol $\underline{,}$ may within the layout be replaced by SPACE the effect of SPACE being the same.

8.3.4.5. Decimal point. The decimal point will always be printed in a fixed position within the layout. If decimals are printed it will appear as . otherwise as SPACE.

8.3.4.6. Scale factor. The scale factor will be printed in the same way as in the language. The symbol $_{10}$ will appear immediately in front of the sign of the exponent. If the scale factor is 1 the symbols $_{10}$ and following will appear as SPACEs. Note that it is not possible to print an exponent part without a decimal part.

8.3.5. Round off.

All numbers will be correctly rounded to the number of significant digits printed.

8.3.6. Limitations.

The total number of symbols n and d in any decimal layout must be ≤ 15.

The total number of symbols n, d, and 0, written to the left of the decimal point must be ≤ 15.

The total number of symbols d and 0 written to the right of the decimal point in a decimal layout must be ≤ 15.

The number of symbols d in any exponent layout must be ≤ 7.

The symbols , and SPACE can only appear in such positions within the layout that they are preceded by fewer than 20 symbols of the kinds n, d, 0, and point (.).

8.3.7. Alarm printing.

By alarm printing is meant that the printing will consume more positions on the paper than are present in the layout. Alarm printing will occur as follows:

8.3.7.1. Negative number printed with layout having empty sign position. The correct - will be inserted, consuming one extra position.

8.3.7.2. Number too large for layout. Whenever the number to be printed is too large for the layout given, an actual layout is used which will accomodate the number by inserting an exponent layout, or by increasing the number of exponent digits.

8.3.8. Small numbers.

Printing of small numbers will never give rise to alarm printing. Instead the number of printed significant digits will be smaller than the maximum (section 8.3.4.2).

8.3.9. Examples of printed numbers.

In order to indicate the exact number of characters printed, commas are inserted immediately preceding and following each number.

```
Layout
 n,dd,dd.d0,0       +d,ddd.ddd,d          -ddd.d00⏨+d       ±dd.0⏨-dd

Normal printing
,      0.00 1,     ,      +.001 2,       ,   1.235⏨-3,     ,+12   ⏨-4,
,      0.01 2,     ,      +.012 3,       ,  12.35 ⏨-3,     ,+ 1.2 ⏨-2,
,      0.12 3,     ,      +.123 5,       , 123.5  ⏨-3,     ,+12   ⏨-2,
,      1.23 5,     ,     +1.234 6,       ,   1.235   ,     ,+ 1.2     ,
,     12.34 6,     ,    +12.345 7,       ,  12.35    ,     ,+12       ,
,   1 23.45 7,     ,   +123.456 8,       , 123.5     ,     ,+ 1.2 ⏨ 2,
,  12 34.57  ,     ,+1 234.567 9,        ,   1.235⏨+3,     ,+12   ⏨ 2,
,1 23 45.7   ,                           ,  12.35 ⏨+3,     ,+ 1.2 ⏨ 4,
                   ,      -.001 2,       ,  -1.235⏨-3,     ,-12   ⏨-4,
                   ,-1 234.567 9,        ,  -1.235⏨+3,     ,-12   ⏨ 2,
Alarm printing
,     -0.00 1,
, 1 23 45.7  ⏨3, , 1 234.567 9⏨4,
,   1 23.45 7⏨15, ,-1 234.567 9⏨14,      , 123.5  ⏨+15,
```

8.4. STANDARD PROCEDURES: outtext, writetext.

8.4.1. Syntax.

```
<general string> ::= ≮<<proper string>≯ |<formal parameter>|
     (<string expression>)
<string expression> ::= <general string>|
     <if clause><general string> else <string expression>
<outtext parameter> ::= <string expression>|<out statement>
<outtext parameter list> ::= <outtext parameter>|
     <outtext parameter list><parameter delimiter><outtext parameter>
<outtext statement> ::= outtext(<outtext parameter list>)|
     writetext(<outtext parameter list>)
```

8.4.2. Examples.

```
outtext(≮<Result is≯, a, ≮<than expected≯)
writetext(≮<Q,=,≯)
```

8.4.3. Semantics.

The execution of an outtext statement causes the following treatment of the parameters specified in the parameter list, taking them in order from left to right:

String expression: an output of the text resulting from an evaluation of the expression is performed.

Out statement: the call of the statement will be executed.

8.4.3.1. The string quote.

Note the difference between the string quotes used here

≮< ≯

and those used in layout expressions (cf. section 8.3.1).

8.4.3.2. Treatment of SPACE and CAR RET.

All characters of the proper string, including SPACEs and CAR RETs will be outputed. The symbol for space , will however be equivalent to SPACE, i.e. it will be printed, not as it stands, but as a SPACE.

8.5. STANDARD PROCEDURE: outsp.

8.5.1. Syntax.

```
<outsp statement> ::= outsp(<arithmetic expression>)
```

8.5.2. Example.

```
outsp(n + m - 7)
```

8.5.3. Semantics.

The execution of an outsp statement causes the number of SPACE symbols specified as actual parameter to be outputed.

The value of the arithmetic expression will, if necessary, be rounded to the nearest integer. If it assumes a non - positive value no symbols will be outputed.

8.6. STANDARD PROCEDURES: outcr, writecr.

8.6.1. Syntax.

```
<outcr statement> ::= outcr|writecr
```

8.6.2. Semantics.

An outcr statement causes a CAR RET symbol to be outputed. Note that this will cause the combined operation of return of carriage and line feed to take place.

8.7. STANDARD PROCEDURES: outclear, outsum.

8.7.1. Syntax.

```
<outclear statement> ::= outclear
<outsum statement> ::= outsum
```

8.7.2. Semantics.

The two output procedures described here serve to insert characters on the output tape with a view to a later use of this output tape as input tape to an ALGOL program.

The outclear statement punches the CLEAR CODE and sets the internal sum of the punched characters to zero. This prepares for the use of the checksum mechanism (cf. section 9.2.5).

The outsum statement punches a STOP CODE, a SUM CODE and a character representing the value of the internal sum of all punched characters and sets this sum to zero. During input this combination will cause an automatic sum check to take place (cf. section 9.2.5).

8.8. STANDARD PROCEDURES: outchar, writechar.

8.8.1. Syntax.

```
<outchar statement> ::= outchar(<arithmetic expression>)|
                        writechar(<arithmetic expression>)
```

8.8.2. Examples.

outchar(if upper case <u>then</u> 60 <u>else</u> 58)
writechar(49)
writechar(symbol - case)

8.8.3. Semantics.

The execution of an outchar statement causes the character corresponding to the value of the actual parameter to be outputed. The correspondence between the integers and the characters is given in the table of section 6.5. If the value of the actual parameter is not an integer it will be rounded to the nearest integer. If it is larger than 127 the value modulo 128 will be used.

The characters for UPPER CASE and LOWER CASE must be outputed explicitly where needed. Where outchar statements are used side by side with output or outtext statements it is important to note that these latter will assume the output unit to be in lower case when a call is made and will also leave it in lower case when the call is completed.

Note also that the use of outchar may produce a tape which will cause the checksum mechanism (section 9.2.5) to fail, e.g. if outchar is used to produce either of the control characters CLEAR CODE or SUM CODE.

9. STANDARD INPUT PROCEDURES.

Input of information from 8-channel punch tape may be carried out at any stage of an ALGOL program through calls of standard input procedures permanently available to the translator.

In order to provide flexibility several different kinds of standard input procedures are available. These differ both with respect to the interpretation of the single symbols supplied on the input tape and the internal effect of the input operation.

9.1. IDENTIFIERS AND MAIN CHARACTERISTICS.

The identifier and main characteristics of the standard input procedures are the following:

Identifier	Example, reference	Effect
input	input(a, b, c) section 9.4.	Reads numbers and assigns to variables or arrays.
inone typein	p×inone section 9.5, 9.8.	**real procedures** inone and typein have the next number appearing on the input tape or typed on the typewriter as their value.
kbon	bool:= kbon section 9.6.	This **Boolean procedure** supplies the current value of the manually operated KB register.
outcopy writecopy	outcopy(⊀</;⊁) section 9.7.	Cause a copying of the characters on the input tape to the output punch (outcopy) or the typewriter (writecopy).
inchar typechar	n:= typechar section 9.9.	These **integer procedures** supply the value of the next character which appears on the tape or is typed.
setchar	setchar (15) section 9.10.	Inserts an input character ahead of the ones waiting in the input.
char	p:= char section 9.11.	Supplies the value of the last character read by any input procedure.
lyn	q:= lyn + 4 section 9.12	Supplies the value of the next row of holes on the input tape.

9.2. UNIVERSAL INPUT MECHANISMS.

Certain characters on the input tape will be handled in the same way no matter which of the standard input procedures is controlling the input operation. The universal mechanisms are the following:

9.2.1. Skipping between PUNCH OFF and PUNCH ON.

All characters between PUNCH OFF and the first following PUNCH ON, these two characters included, will be completely ignored during input.

9.2.2. Ignoring of BLANK TAPE, TAPE FEED, and ALL HOLES.

The characters

.	BLANK TAPE
oooo.ooo	TAPE FEED
ooooo.ooo	ALL HOLES

will be ignored during input.

9.2.3. (This section has been deleted).

9.2.4. Input characters of wrong parity.

The machine stops when a row of an even number of holes is sensed in the tape reader. In this situation it is sufficient to place the intended symbol in the R register since the ALGOL system never makes any use of the representation stored by the input instruction itself.

9.2.5. The checksum mechanism.

When the standard input procedures read tapes which have been prepared by the standard output procedures the checksums included on this tape in consequence of calls of the outsum procedure will automatically be verified. If the check symbol does not check with the corresponding symbol as formed during previous read-in the machine will print

sum fails

and the machine will stop. If a character is typed on the typewriter the reading will continue. The internal variable which holds the current sum of the symbols which have been read in may be reset to zero by the inclusion of the CLEAR CODE on the tape. This is the symbol produced by the outclear procedure (cf. section 8.7.2). On the flexowriter use:

AUX CODE with 0

9.2.6. Stop produced by END CODE.

Whenever the END CODE appears the message

pause

will be typed and the machine will stop, waiting for a character to be typed on the typewriter. The END CODE may be produced by an ALGOL program by executing the statement outchar (12). On the flexowriter it is produced by depressing

AUX CODE with SPACE.

9.2.7. The effect of UPPER CASE and LOWER CASE.

For printed symbols (cf. section 6.1) the meaning and effect of a given hole combination depends on the most recent CASE symbol on the tape (UPPER CASE or LOWER CASE).

For typographical and control symbols (cf. sections 6.2 and 6.3) the effect is usually independent of the case.

9.3. TERMINATORS, INFORMATION SYMBOLS, AND BLIND SYMBOLS.

The effect of the input characters which do not give rise to an action of a universal input mechanism (cf. section 9.2) depends on the particular standard input procedure. In describing this effect it is convenient to make use of the following concepts:

9.3.1. Terminators. A terminator is a symbol on the input tape which indicates to the input procedure that the reading of a piece of information (e.g. a number) has been completed.

9.3.2. Information symbols. An information symbol is a symbol on the input tape supplying positive information which is transferred to the running ALGOL program by the input procedure.

9.3.3. Blind symbols. A blind symbol is a symbol on the input tape which is ignored by the input procedure.

As explained more concisely in the following sections we have for the procedures input and inone:

Terminators: <letter> TAB PUNCH ON CAR RET STOP CODE all signs except + - . $_{10}$

Information symbols: <digit> + - . $_{10}$

Blind symbols: SPACE

Each input operation will in general read three sections of the input tape:

1. Any mixture of terminators and blind symbols.
2. A legal sequence of information symbols mixed with blind symbols.
3. One terminator.

9.4. STANDARD PROCEDURE: input.

9.4.1. Syntax.

```
<input parameter> ::= <variable>|<array identifier>
<input parameter list> ::= <input parameter>|
    <input parameter list><parameter delimiter><input parameter>
<input statement> ::= input(<input parameter list>)
```

9.4.2. Examples.

```
input(P)
input(A[i,j], V, MATA)
input(k, B[1,k])
```

9.4.3. Semantics.

A call of the procedure input will cause the values of numbers supplied on the input tape to be assigned to the variables and/or arrays of subscripted variables specified as parameters. The assignments will in detail be executed as follows:

9.4.3.1. Order of assignment. The parameters will be taken in order from left to right and the assignment will be completely finished for each parameter before the next is treated. Thus the statement input(k, B[1,k]) will first assign a value from the input tape to k and this value of k will then define the particular component of B to which the next number on the tape will be assigned.

9.4.3.2. Assignment to array. If an array identifier is supplied as parameter an assignment to all the components of the array will take place. The order of assignment may be described as follows: Denoting the lower and upper subscript bounds of the array declaration by l1, l2, ... ln, u1 u2, ... un, the input operation is equivalent to

```
     for i1:= l1 step 1 until u1 do
     for i2:= l2 step 1 until u2 do
          .......

     for in:= ln step 1 until un do
      A[i1, i2, ... , in]:= input number
```

where i1, i2, ... in are internal variables.

9.4.3.3. Input tape syntax. The characters appearing on the input tape during the execution of input must conform to the following syntactic rules:

```
<input terminator>::= ∨|×|/|=|;|[|]|(|)|||∧|<|>|,|TAB|PUNCH ON|:|CAR RET|
                         STOP CODE|<letter>
<input information> ::= <digit>|.|⏨|+|-
<input blind> ::= SPACE|_
<input prelude> ::= <empty>|<input blind>|<input terminator>|
        <input prelude><input blind>|<input prelude><input terminator>
<digit sequence> ::= <digit>|<digit sequence><digit>|
        <digit sequence><input blind>|<input blind><digit sequence>
<input integer> ::= <digit sequence>|+<digit sequence>|-<digit sequence>
<input fraction>::= .<digit sequence>
<input exponent>::= ⏨<input integer>
<input decimal>::= <digit sequence>|<input fraction>|
                   <digit sequence><input fraction>
<unsigned real>::= <input decimal>|<input exponent>|
                   <input decimal><input exponent>
<input real>::= <unsigned real>|+<unsigned real>|-<unsigned real>
<input ditto>::= -|<input ditto>-|<input ditto><input blind>
<tape integer>::= <input prelude><input integer><input terminator>|
                  <input prelude><input ditto><input terminator>
<tape real>::= <input prelude><input real><input terminator>|
               <input prelude><input ditto><input terminator>
```

9.4.3.4. Examples of input tape for input.

Tape integers:

```
17 283;
i = +138,
S[25]
function(-12)
p: -/
```

Tape reals:

```
w:= 3.857_392 <
eps:= -⏨-14,
pi:= 3.141592 65;
Sæt x = 4,
q: 1.384⏨-11,
```

9.4.3.5. Semantics of input tape. Each input assignment will cause the reading of one tape real or tape integer. If these contain digits they will be interpreted according to the usual ALGOL prescriptions (cf. sections 2.5.3 and 2.5.4), ignoring all input blinds and input terminators. An input ditto, on the other hand, will cause the input assignment to be skipped for the particular variable, thus leaving its value unchanged.

9.4.3.6. Errors. The standard procedure input checks that the syntactic rules of section 9.4.3.3 are satisfied. If an error is detected one of the messages

correct input value, end in LC:

or

correct input value, end in UC:

will be typed. The operator is now expected to type one number, followed by a terminator, to be used instead of the erroneous combination appearing on the tape. The terminator must be in upper or lower case as indicated in the message since otherwise the following text on the input tape may be misinterpreted.

9.5. STANDARD PROCEDURE: inone.

9.5.1. Syntax.
<inone function designator>::= inone

9.5.2. Examples.
w:= (inone + y)/q
B[inone, inone]:= inone

9.5.3. Semantics.
Inone is a real procedure having an empty formal parameter part. Every time it is called it will read the next tape real appearing on the input tape (cf. section 9.4.3.3). This information on the input tape will define its value according to the rules of section 9.4.3.5, except that the effect of an input ditto is undefined.

9.5.3.1. Example of input tape for inone. A reasonable input tape for the second example of section 9.5.2 would be the following:
B[3,7]:= 3.847,
Note that the correct execution of this input operation is directly dependent on the strict adherence to the rules of sections 4.2.3.1 - 4.2.3.3 for assignment statements.

9.6. STANDARD PROCEDURE: kb on.

9.6.1. Syntax.
<kb on function designator> ::= kb on

9.6.2. Examples.

```
if kb on then output(≮ddd≯, outcr, Q)
if kb on ∧ i > 20 then go to finis
time is up:= kb on
```

9.6.3. Semantics.

kb on is a Boolean procedure having an empty formal parameter part. The value of the function designator is given by the current state of the manually controlled KB register of the machine; it is true when KB is on, otherwise false.

9.7. STANDARD PROCEDURES: outcopy, writecopy.

9.7.1. Syntax.

```
<outcopy statement>::= outcopy(<string expression>)|
                       writecopy(<string expression>)
```

9.7.2. Examples.

```
outcopy(≮<+/≯)
writecopy(if s>0 then w else y)
outcopy(fs)
```

9.7.3. Semantics.

A call of an outcopy statement causes a copying of characters from the input tape to the output. The section of the input tape to be copied is defined by the value of the string expression supplied as parameter. This value must have the form

≮< <proper string> ≯

where the proper string consists of one or two characters. If one character is supplied the copying will take place from the actual position of the input tape until the first occurrence of the character specified as parameter. If two characters are supplied the copying will start from the first character on the tape which is the same as the first of the two characters supplied as parameters and will continue until the first occurrence of the second of these symbols on the tape. The characters indicating the begin and end of the section of the input tape to be copied will not themselves be copied.

The copying will include all legal characters except those associated with the universal input mechanisms (cf. section 9.2) and superfluous case shifts.

9.7.3.1. Example of call, input tape, and output.
The call
outcopy(‹<[ ]›)
Operating on the following input tape:
Heading: [
Problem **number**:]
will produce as output:
Problem **number**:

9.8. STANDARD PROCEDURE: typein.

This procedure is entirely similar to procedure inone (section 9.5), but expects the input characters to be typed on the typewriter.

9.9. STANDARD PROCEDURE: inchar, typechar.

9.9.1. Syntax.
<inchar function designator> ::= inchar | typechar

9.9.2. Examples.
if typechar = 49 **then** **go to** a
symbol := inchar

9.9.3. Semantics.
inchar and typechar are **integer** **procedures** having an empty formal parameter part. Each call of an inchar function designator will activate the corresponding input unit (paper tape reader for inchar, typewriter for typechar) and will return with the value of the next proper character from the input medium as its value. By proper character is here meant a character which is not handled by the universal input mechanisms (section 9.2). The values of proper characters in lower case are given directly by the table in section 6.5. In upper case the value supplied by inchar and typechar is increased by 128. Thus the letter p will appear as 39 while P will be 167.
Note that typechar always assumes the typewriter to be in lower case when the call is made. Characters in upper case will therefore be transmitted properly only if each of them is preceded by an explicit shift to upper case.

9.10. STANDARD PROCEDURE: setchar

9.10.1. Syntax.
<setchar statement> ::= setchar(<arithmetic expression>)

9.10.2. Examples
setchar(160)
setchar(tegn)

9.10.3. Semantics.

Each call of setchar assigns the value of the expression supplied as actual parameter to an internal buffer and at the same time sets an internal Boolean variable which causes the value in the buffer to be used as the first proper input character at the first following call of any input procedure (input, inone, typein, inchar, typechar, outcopy, writecopy) ahead of the next symbol waiting in the input unit.

The values of the actual parameters supplied in calls of setchar should only be such which correspond to proper input characters, i.e. such which may appear as values of inchar.

9.11. STANDARD PROCEDURE: char.

9.11.1. Syntax.

<char function designator> ::= char

9.11.2. Examples.

```
if char < 10 then outchar(char)
if char = 133 then go to exit
```

9.11.3. Semantics.

char is an integer procedure having an empty formal parameter part. Its value is the number corresponding to the last proper character previously inputed by any standard input procedure (input, inone, typein, inchar, typechar, outcopy, writecopy) or assigned by setchar. The value corresponding to a proper character is to be understood in the same sense as for procedure inchar. Note that char does not activate any input unit, but only makes the last character supplied by any input unit available.

9.12. STANDARD PROCEDURE: lyn.

9.12.1. Syntax.

<lyn function designator> ::= lyn

9.12.2. Example

```
symbol:= lyn
```

9.12.3. Semantics.

lyn is an integer procedure having an empty parameter part supplying the value of a character from the paper tape reader, like inchar. However, the character whose value is provided by lyn is always the next one on the input tape without any intervention from the universal input mechanisms (section 9.2) or the buffer controlled by setchar (section 9.10). Likewise the case and buffer state are unaffected by calls of lyn. Thus by using lyn the programmer may interpret the input symbols having correct parity in any conceivable manner.

10. STORING VARIABLES ON DRUM.

10.1. INTRODUCTION.

ALGOL programs operating with up to about 700 variables simultaneously may be handled directly by the GIER ALGOL system. However, if programs declaring more than this number of variables simultaneously are run in the system the run will be terminated before the final <u>end</u> has been reached (cf. section 11.7, alas and array). What has happened is that the capacity of the directly available internal store of the machine, the so-called core store, has been exceeded.

This does not mean that problems involving a larger number of variables are outside the reach of the system since there is available in the machine a storage capacity on the so-called magnetic drum of more than 12 times that of the core store. What it does mean, however, is that the user must include in his program calls of the standard procedures to drum and from drum which serve to transfer variables from the core store to the drum store and back again. From the point of view of the user the magnetic drum may in this context be regarded as a new kind of input-output medium, analogous to paper tape. The two standard procedures to drum and from drum are then analogous to the standard procedures output and input.

However, the use of to drum and from drum should not be confined to the cases where it is indispensable. In fact, execution speed considerations will often make it desirable to keep the number of active variables in the program considerably lower than the admissible upper limit.

An intelligent assessment of the factors involved requires some knowledge of the storage allocation system incorporated in GIER ALGOL. This system is therefore explained in the following sections.

10.2. STORAGE OF VARIABLES.

The reservation of core storage space for a variable is made at the time of entry into the block in the head of which the variable is declared. Similarly reservations for a block are cancelled at the time of the corresponding exit from the block. For this reason the space reserved for the variables will usually change from time to time during the execution of a program, being at every moment equal to the sum of the reservations made by those blocks and procedure bodies which are active.

The reservations made at a block entry include other quantities besides variables. The total requirements may be derived from the declarations (including the implicit ones for local labels) of the block as follows:

	Number of locations required
Simple variables, local labels, local procedures, formal parameter	One for each quantity
Array segment	Number of array identifiers + 1 + number of subscripts + total number of variables.
Switch declaration	1 + number of switch elements
Working locations	Depends on structure of program, usually only a few.
Block, procedure body	2 if normal block, 3 if procedure, 4 if type procedure.

10.3. STORAGE OF PROGRAM.

GIER ALGOL incorporates a fully automatic system for handling the transfers of program drum tracks to the core store during the execution of the program. This system will at all times attempt to make the best use of that part of the core store which is not currently reserved for variables. This section of the core store will be divided into program track places, each of 41 locations. The available places will be used for those program tracks which are required as the program execution develops. Whenever the program execution calls for a transfer to another track it is investigated whether the track is available in the core store. If it is not it is transferred to that track place which for the longest time has been left unused.

10.4. BALANCING THE USE OF THE CORE STORE.

The transfer of a drum track to the core store requires 20 milliseconds. In contrast the transfer of control to a track which is already present in the core store takes between 0.7 and 1.6 milliseconds. It is therefore clear that A PROGRAM HAVING A LARGER PART OF THE AVAILABLE CORE STORE RESERVED BY VARIABLES WILL SPEND A LONGER TIME ON TRANSFERS OF PROGRAM TRACKS TO THE CORE STORE. The importance of this loss of speed for a given number of program track places depends very strongly on the loop structure of the program. It is small if most of the execution time of the program is spent in a loop which may be held completely in the available program track places.

To assist in estimating the number of program tracks involved in a loop which includes calls of standard procedures the arrangement of standard procedures on the tracks reserved for them is given below.

Standard procedure track	Used by
0	write, output
1	write, output
2	write, output
3	write, output, sqrt, outsp
4	↑ with integer exponent, abs, entier, sign, writecr, outcr, outchar
5	to drum, from drum, lyn, char
6	to drum, from drum
7	outtext, writetext
8	exp, writechar
9	outcopy, writecopy
10	input, typein, typechar, inone, inchar
11	input, typein, inone
12	outcopy, writecopy, input, typein, inone, typechar, inchar
13	(alarms of input, special storage)
14	cos, sin, setchar
15	arctan
16	ln, outclear, outsum
17	split, pack
18	gier drum, gier proc, gier, kb on
19	gier drum, gier proc, gier

These considerations indicate that in programs where the execution speed is of any concern the number of active variables in the program should be kept rather lower than the strict upper limit; a practical limit might be 500 variables. This may be achieved by using the drum as an additional store for variables.

The increase of execution speed gained by using the drum for storage of variables will be counteracted by the loss of time incurred each time these variables are transferred to or from the drum by to drum or from drum. This latter transfer time is usually of the order of 1 - 2 milliseconds per variable per transfer. Whether these transfer times are of overall significance depends on the time necessary for other processing of the variables. An estimate of such processing times may be formed on the basis of the figures given in appendix 3. It will be found that the time of even a quite moderate amount of processing will overshadow the average drum transfer time.

10.5. STANDARD PROCEDURES: to drum, from drum.
STANDARD VARIABLE: drumplace.

10.5.1. Syntax.

```
<drum transfer function designator> ::= to drum(<array identifier>)|
                                         from drum(<array identifier>)
<drumplace variable identifier> ::= drumplace
```

10.5.2. Examples.

```
Bplads := drumplace
Bshift := to drum(B)
drumplace := drumplace - Bshift
from drum(B)
```

10.5.3. Semantics.

The standard integer procedures to drum and from drum and the associated standard integer variable drumplace administer the handling of transfers of arrays of values to and from the drum memory of GIER. The procedure to drum will transfer the array of subscripted variables identified in the actual parameter to the drum and acts like an assignment of values to the drum and likewise the procedure from drum will assign values previously transferred to the drum to the array identified in the actual parameter. In either case the part of the drum involved in the transfer is defined by the value of the integer variable drumplace which enters into to drum and from drum as a non-local identifier. Thus in order to retrieve a set of values previously transferred to the drum the procedure from drum must be called with drumplace having the same value as when the corresponding call of to drum was made. The same holds if it is desired to assign new values to a previously used section of the drum. In any case the array supplied as parameter in the drum transfer function designator must be of the same type and have the same number of subscripted variables as the one used in the corresponding call of to drum. However, the two arrays need not have the same number of subscripts or the same subscript bounds. If the arrays differ in these respects the correspondence of elements is established by ordering the elements of each array in the same manner as they would be if they were read from tape by means of the standard procedure input (cf. section 9.4.3.2).

Clearly the standard variable drumplace is the key to administering values stored on the drum. In addition the programmer may use the values of the drum transfer function designators. These are closely related to drumplace as apparent from the following 3 rules which define the behaviour of the value of drumplace:

1. drumplace is initialized by the compiler to a value which is the one extreme of its permissible range of variation.

2. Every call of to drum and from drum will, as a side-effect, change the value of drumplace in a direction away from the initial value supplied by the compiler towards the other extreme of its permissible range and by such an amount that the new value is the correct one to use in transferring values to the next adjecent section of the drum.

3. The amount by which drumplace is changed through a call of to

drum or from drum will be the same whenever arrays of the same type and having the same number of subscripted variables are transferred. The amount by which drumplace is changed is available as the value of the drum transfer function designator. In other words:

new value of drumplace = old value + to drum(A)
new value of drumplace = old value + from drum(A).

However, nothing further about the dependence of the change of drumplace on the size and type of the array is defined generally (the precise meaning of drumplace will change from one edition of the compiler to another).

It will be understood from these rules that as long as no explicit assignment is made to drumplace only calls of to drum will be in order and each of these will use a new section of the drum adjecent to the one used in the last previous call of to drum. Before any call of from drum is made the programmer must make an explicit assignment to drumplace. The values assigned to drumplace can only be derived from its previous values possibly modified by integral multiples of the amount by which is has changed.

The programmer has his full freedom to overwrite sections of the drum which have previously been used as long as he makes sure to use only values of drumplace which lie within the range defined by its initial value and another extreme which marks the other end of the free section of the drum. If drumplace steps outside this range an error reaction will occur at run time and the message (cf. section 11.7)

drum alas

will by typed. The criterion for a set of values previously transferred by to drum to be still intact on the drum may be formulated as follows: Each section used on the drum by to drum will be defined by an interval of the values of drumplace, namely that defined by the value of drumplace just before to drum was called and its value just after the call was completed. The values transferred will still be intact as long as no call of to drum with an overlapping interval of drumplace has been performed.

10.5.4. The meaning of drumplace and the capacity of the drum.

The standard procedures included in GIER ALGOL III treat the drum like a linear array, the location in relative address r on track t being regarded as element number $n = r + 40\times t$. Within this array to drum and from drum will start reserving locations starting at a high element number and will successively use elements having lower numbers. The section of the drum referred to in a call made with drumplace = dp will be the locations having numbers dp, dp-1, dp-2, etc. The section of the drum involved is defined in section 11.1. This section also shows that by this arrangement to drum will first use a free section of the drum and only later use the section holding the translator, and the data given will enable the user to calculate the capacity of the drum in a given version of the compiler and in a given program.

11. OPERATING THE COMPILER.

11.1. TAPES AND STORAGE OF THE COMPILER.

The compiler will be distributed to the users in the form of 5 separate tapes. These tapes will enable the various user groups to generate their own binary versions of the compiler, as the need arises. The options provided in this manner are briefly as follows:

1. The user may choose between a permanent compiler, i.e. one which remains intact on the drum, except when overwritten by todrum (cf. section 10.5.4), and a transient compiler, i.e. one which is read in from tape piecewise between the translator passes. By using the transient compiler the user gains 78 tracks on the drum during translation.

2. The user may choose to place the compiler and system on the tracks normally occupied by the HJÆLP system and to have the HP button of the machine return directly to the ALGOL system, or he may wish to keep the HJÆLP system in the machine. By leaving out the HJÆLP system the user gains 38 tracks on the drum.

3. The user may protect certain parts of the drum from being used by the compiler. Also the compiler may directly be adapted to machines having more than 320 drum tracks.

In order to use the options the user will need the following information about the way the compiler is stored and the limits of the various alarms: During the loading of the compiler into the machine the code is placed on the drum, beginning at the track given by the initial value of c70. When the complete code has been read it is moved as a solid block of information to begin at the track given by the initial value of c60. The number of drum tracks needed to hold the compiler, N, and the part of these used for the run time system, S, are as follows:

	S=length of system	N=length of compiler	
HP-button entry to ALGOL		Permanent	Transient
No	39	145	67
Yes	41	147	69

During loading the drum must hold the HJÆLP system and the core image in addition to the compiler. Therefore the initial value of c70 should be chosen to be greater than 38. However, the final placement may overwrite any other part of the drum, with the restriction that if HP-button entry is desired the compiler must be placed from track 1. The HP-button entry is controlled by the initial value of e96. If this is 0 the entry is omitted, if it is 1 the two appropriate tracks are included.

Additional options permit the user to reserve a drum area which is not used by the translator, but which may be referred to by means of todrum and fromdrum (section 10.5), and another area which is not used by the translator and is inaccessible to todrum, but which may be referred to by fromdrum. Finally the highest track number which may be used either during translation or at run time may be specified by the user. These facilities are controlled by the values assigned to the symbolic names e86, e20, and e97. The significance of the parameters c60, e86, e20, e97, and the values of S and N, for the storage of the compiler and the translated program on the drum tracks may be derived from the following picture of the drum. In this picture the low track numbers are shown to the left. The actual tracks corresponding to a number of specific track numbers are indicated in the form of a pair of colons, pointing to the beginning and end of the track. The parameter P is the number of tracks required by the translated program. This may be derived from the pass information (appendix 1).

Track number	0	c60	c60+S	c60+N	c60+N+e86	e20-P	e20	e97
	: :	: :	: :	: :	: :	: :	: :	: :
Compiler		:		:				
Maximum program					:		:	
Normal program						:	:	
Open to fromdrum	:							:
Open to todrum without drum alas			:			:		
Open to todrum without gone				:		:		

The normal version of the compiler is defined by the SLIP definitions:
d c70=39, c60=39, e96=0, e86=0, e20=319, e97=319
This leaves the HJÆLP program on tracks 1 to 38, but does not admit the use of HJÆLP while an ALGOL program is running because the core image is used by the ALGOL program. A version which starts in track 1 and includes HP-button entry into the compiler would require the following redefinitions:
d c60=1, e96=1

The five tapes of the compiler are the following:

A: Compiler part 1.
B: Part 2 of the permanent compiler.
C: Part 2 of the transient compiler.
D: Part 3 of the transient compiler (passes 2 to 8).
E: Paper tape procedure binout, see appendix 6.

The loading of a compiler into the machine requires the following steps: Insert tape A in reader and read by means of SLIP; start by typing 1. After reading the tape for a few seconds the machine stops, waiting for input from typewriter. If the normal version of the compiler is desired (see definition above), type 1. Otherwise redefine some or all of e70, c60, e96, e86, e20, and e97, before typing 1. When the complete tape A is read, select tape B or C, as required, and start reading by typing 1. If all is well the machine reads the tape and stops with the message

algol

(cf. section 11.3). Otherwise there is a fault and the loading must be attempted anew.

The compiler is now ready to accept ALGOL programs, as described below. If the transient version is used the machine will stop after the reading of the ALGOL program tape with the message:

2. from reader
Tape D must now be inserted in the reader and the machine must be restarted by typing a SPACE. The completion of each pass will again give rise to the from-reader-message, but with no further stops.

The use of the tapes A, B and C, is not convenient for daily use. For producing complete compilers in binary form the user groups are advised to make use of the paper tape procedure binout, tape E. This procedure also provides for the production of binary output of the translated program, as described in appendix 6. See also section 12. As an example a normal complete translator in binary form will be produced by the program

<u>begin</u> gierproc(⪡<binout⪢, 5) <u>end</u>;

11.2. MANUAL JUMP TO COMPILER.

The COMPILER-READY-SITUATION may be called at any time during translation of ALGOL programs by transferring control to instruction 1 in the core store.

If the HJÆLP system is in the machine the same effect will follow if the HP button is pressed and the control words

<u>halgol</u>
<u>e</u>

are typed.

If the compiler in the machine includes direct entry from the HP-button, pressing this button will cause one of four reactions:

Message	Significance of reaction
FEJL	Sum check error on tracks 1 - 31
SUM ALGOL	- - - - other compiler tracks.
KC ALGOL	The manually controlled KA and KB registers are both L. The machine is now ready to read binary tape using the basic reading program of track 0.
algol (KA) (KB)	The machine is in the COMPILER-READY-SITUATION.

11.3. COMPILER-READY-SITUATION.

The compiler is ready to accept ALGOL programs whenever one of the messages

algol
algol KA.
algol KB.
or algol KC.

has been put out on the typewriter. In this situation the machine is waiting for symbols to be typed on the control typewriter. This leaves certain operational choices to the operator, as described below. The second part of the message reminds the operator of the state of the KA and KB registers in an obvious way.

11.3.1. Start compiling.

Typing of a SPACE (or any character other than p, w, t, o, l, n, or i) will start the compiler translating the program with output and other compiling features defined by the other characters typed previously. If SPACE is typed immediately following the algol-message and also KA and KB are 0 the compiler will produce no typed or punched output, input will be taken from the paper tape reader, and program sections between PUNCH OFF and the first following PUNCH ON will be ignored. Thus programs will be compiled at the highest possible speed. The compiler produces about 38 final machine instructions per second, except in the case of very short programs where the basic time of 4 seconds becomes prominent. Other compiling modes may be specified by typing any sequence of the letters p, w, t, o, l, n, and i, prior to the final SPACE, and by setting KA and KB at this or a later time, as described below.

11.3.2. Compilation output.

Typing of p and w selects the output unit operating during compilation, p standing for punch and w for typewriter. If both p and w are typed the output will appear on both punch and typewriter. Whenever an output unit is specified the normal compiler output is always produced. This includes:

11.3.3. Prelude to program:

All characters on the input tape up to and including the first appearance of begin are copied to the output.

11.3.4. Epilogue of program:

All characters on the input tape following the final end up to and including the first following ; (semicolon) are copied to the output.

Additional compilation output may be specified as follows (note that this presupposes a choice of output unit by typing of p or w):

11.3.5. Line output.

Typing of l causes every 10th line of the source ALGOL program to be copied to the output with its line number attached.

11.3.6. Pass information.

Typing of i causes output of the so-called pass information. This is described in appendix 1.

11.3.7. Pass output.

If KB is set to L the intermediate output from passes 1, 2, 3, 4, 5, 6, 7, and 8 will be output. The form of this output is described in appendix 2. KB may be changed at any time during compilation and pass output will be produced accordingly. The output from pass 8 (the final machine code) requires a special output program to be read in from tape. When KB is L when pass 8 is completed the message

9. from reader

is given. If the tape is not available, set KB to 0 and type a SPACE to complete the translation.

11.3.8. Program between PUNCH OFF and PUNCH ON.
If o is typed the text between PUNCH OFF and PUNCH ON is included in the program.

11.3.9. Input from typewriter.
If t is typed the compiler takes its input from the typewriter.
Input from typewriter may also be called following the pause-message (section 11.4.1).
When input is taken from the typewriter a line of text will be processed at a time and the user has the possibility of deleting the line which is being typed. Also shift to input from tape may be specified. This is achieved as follows:
11.3.9.1. A line which is terminated with the CAR RET character will be included in the program.
11.3.9.2. Whenever 4 consecutive case shifts are typed (i.e. LC, UC, LC, UC or UC, LC, UC, LC) the compiler types the message

<

(in red). If now the operator types y the compiler will complete the red message to read

<yes

and the compiler will continue to take its input from tape, including the line which has just been typed. If the operator types n the compiler will complete the red message as follows:

<no

and be ready for another line to by typed instead of the previous one, which will be ignored.

11.3.10. Stop between translation passes.
If KA is set to L the machine will stop after each of the passes 1 - 8. The compiler is restarted by typing any character on the typewriter.

11.3.11. Error message medium.
Error messages (cf. section 11.4.4) are normally typed out. However, if n is typed they will only be produced on the medium selected as specified in section 11.3.2, and may thus be suppressed altogether.

11.4. TYPED MESSAGES FROM COMPILER.

Irrespective of the choice of output from the compiler certain messages will be typed on the typewriter. These are

11.4.1. Pause message.
The message

pause

is typed and the machine stops when the END CODE is encountered on the input tape during pass 1.

If in this situation the letter t is typed the further input will be taken from the typewriter (cf. section 11.3.9). Any other character will restart the input from tape. Note that the last case shift character read from the tape will be restored correctly after shift to input from typewriter and return to input from tape.

11.4.2. Off and on messages.
Whenever the text between a PUNCH OFF and the first following PUNCH ON is ignored these two control symbols produce messages during pass 1 as follows:

line <line number> off and line <line number> on.

11.4.3. Run-message.
The message

run

indicates that the system is in the RUN-SITUATION with the program ready to be executed (cf. section 11.5).

11.4.4. Error messages.
The first 6 translation passes perform a thorough checking of the formal correctness of the program. Every error found will be reported by a suitable message typed in red. An error message consists of the text

line

followed by the number of the line where the error occurs and a short text characterizing the error. The line number is obtained by counting the CARRET symbols in the source program, line 0 being the one where the first begin appears. Line numbers may be obtained with the help of line output (cf. section 11.3.5).

When the translator has detected an error in the program the translation is discontinued after completion of pass 6 and the system returns to the COMPILER-READY-SITUATION. This means that every program is taken through the complete error detecting part of the translating process and that all errors of a program often will be detected in a single translation run.

Error messages are also produced when certain tables which are created by the compiler exceed the space allotted to them. In this case the COMPILER-READY-SITUATION will follow immediately.

Detailed explanations of the possible error messages and their meaning may be found in appendix 4.

11.4.5. Sum checking of program.
Translation pass 1 treats the characters CLEAR CODE and SUM CODE in exactly the same manner as do the universal input mechanisms (section 9.2.5). ALGOL program tapes which have been produced as output from ALGOL programs may therefore profitably include check sums. A failure of the check during input of the program will be reported in the usual manner (appendix 4).

11.5. RUN-SITUATION.

On completion of compilation and when a new execution of a program is called following a termination of execution the message

run

is typed and the machine will stop waiting for a character to be typed. If a SPACE is typed a normal run will take place. Other characters typed in this situation allow a choice of the units used for output, as explained in the following section.

11.6. CHOICE OF OUTPUT UNITS OR STOP RUN.

The running system allows a free choice of the output units associated with the standard output procedures (cf. section 8.1) or of a termination of the run. This choice must be made in the RUN-SITUATION and may be repeated at any time during the run of the program. The choice is controlled by means of the control typewriter as follows:

Symbol typed	Clue	Meaning
b	both	All output will both be typed on the typewriter and punched on tape
w	writer	All output will be typed. Nothing will be punched
p	perforator	Nothing will be typed. All output will be punched.
Any symbol than b, w, p, or e.	other	write-output goes to typewriter, out-output to punch.
e	exit	Stop run. The run will terminate with an end-message.

When a new CHOICE OF OUTPUT UNITS OR STOP RUN is desired during the execution of a program the contents of the indicator register KA should be changed. This will cause a jump to new CHOICE OF OUTPUT UNIT OR STOP RUN to be made at the first following opportunity (usually within a few seconds). When the choice has been made the execution of the program is immediately continued unless e has been typed.

An alternative way of finishing a run is to simulate an arithmetic overflow by transferring control to instruction 0 of the core store.

11.7. TERMINATION OF EXECUTION OF PROGRAM.

All regular runs of ALGOL programs terminate with a message. The possible terminating messages and their meaning are as follows:

end	The program has passed through the final end of the program.
alas	The demand on storage space exceeds the capacity of the machine. This will be caused by having too many variables of any kind (simple or subscripted, labels, for statements, etc.) in action simultaneously. See sections 10.2. and 12.5.6.
array	The program tries to declare an array too large for the machine or one with a negative number of elements.
exp	The built-in procedure for calculating exp has been called with an argument which would cause the result to exceed the range of real variables (cf. section 7.3). This may also be caused by the operation ↑ with a real exponent.
gier	One of the procedures gierproc (section 12.5.6) or gierdrum (section 12.6.3) reads a tape with a wrong identification or the sum on the tape does not check.
index	A reference to a subscripted variable having subscripts outside the bounds of the corresponding declaration is made. The test for this situation is made only on the final address, not on the individual subscripts. Therefore the alarm will not always be made when the bounds are transgressed.
ln	The built-in procedure for calculating ln has been called with a negative argument. This may also be caused by calling the operation ↑ with an exponent of real type and a negative radicand.
param	A standard procedure has been called with an improper number of arguments (cf. section 12.1).
spill	Arithmetic operation produces result outside the range of real variables (cf. section 7.3). The operation ↑ with integer exponent is first calculated with the absolute value of the exponent as exponent and may therefore cause spill even if the final result is 0.
sqrt	The built-in procedure for calculating sqrt has been called with a negative argument.
drum alas	One of the standard procedures to drum or from drum is called with a value of drumplace outside of the permitted range (capacity of drum is exceeded, cf. section 10.5.3).

Following a terminating message the machine stops waiting for a control letter to be typed on the typewriter. If

r

is typed the system returns to the RUN-SITUATION, ready for a new execution of the program (cf. section 11.5). Any of the characters b, w, p will cause an EMERGENCY OUTPUT OF THE STACK to be performed, as described in appendix 5. The typing of any other character will return the system to the COMPILER-READY-SITUATION (cf. section 11.3) ready for a new compilation, except for the case that the section of the drum which holds the compiler has been used for variables by the program just terminated (cf. section 10.5). If this is the case the message

gone

is typed. It is then necessary to perform a new loading of the compiler into the machine (cf. section 11.1).

12. USING MACHINE CODE IN ALGOL PROGRAMS.

The GIER ALGOL facilities described up to this point limit the utilization of the machine in the following ways:

1. Variables are confined to floating point numbers (integer, or real) or the sign bit of words (Boolean).
2. The machine instructions used to represent the actions of the program are confined to a subset of the complete repertoire and must conform to the segmentation rules imposed by the automatic system for handling transfers from drum (cf. section 10.3).
3. Only those peripheral units for which standard procedures have been written can be used, and only in the manner defined by the actions of these standard procedures.

The standard procedures described in the present chapter are designed to overcome all of these limitations. This is achieved by giving the user access to every bit of the stores of the machine and to executing virtually any sequence of machine instructions. In order to use this possibility the user must therefore be completely familiar with the machine coding for GIER (see Chr. Andersen and Chr. Gram: A Manual of GIER Programming, Regnecentralen, 1963). In using these facilities the programmer should be aware that the extensive checking actions performed by the system are suspended and that the result of mistakes or misunderstandings on the part of the programmer are entirely unpredictable. On the other hand, if used intelligently by an experienced programmer the procedures will probably remove the remaining obstacles to the use of GIER ALGOL for all programming on the machine while still keeping most of the advantages of the powerful language.

12.1. STANDARD PROCEDURE: pack.

12.1.1. Syntax.

```
<pack triple> ::= <arithmetic expression><parameter delimiter>
     <arithmetic expression><parameter delimiter><arithmitic expression>
<pack parameter list> ::= <variable><parameter delimiter><pack triple>|
     <pack parameter list><parameter delimiter><pack triple>
<pack function designator> ::= pack(<pack parameter list>)
```

12.1.2. Examples.

```
pack(Bool)from bit:(4) to bit: (22) the value: (i)
pack(b) from:(3) to:(9) this:(33) and from:(34) to: (41) this:(q+t)
boo:= pak(boo2, 22, 22+i, s, 23+i, 39, w)
```

12.1.3. Semantics.

This Boolean procedure serves to assign an arbitrary bit pattern to any or all of the 42 bits of a GIER machine word. The word into which the pattern is packed must be given as a variable of type Boolean in the first actual parameter. The following parameters are grouped in triples of the form:

first bit, last bit, pattern to be inserted.

The two first parameters of a triple refer to the bit numbers of the final pattern, the bits being numbered from 0, the leftmost, most significant, bit, to 41, the rightmost, least significant, bit. Each triple will cause the bits from first bit to last bit, both included, to be replaced by the pattern to be inserted. This latter must be specified in the form of the corresponding positive integer, in the following sense: The pattern, consisting of binary zeroes and ones, is obtained by expressing the integer in the binary representation and placing the units digit in the position given by last bit. If last bit - first bit > 28 the leftmost bits will always be put to zero since no positive integer in GIER ALGOL has more than 29 bits. It follows from these rules that to make sense the values of the parameters of a triple must satisfy the following relations:

0 ≤ first bit ≤ last bit ≤ 41

0 ≤ pattern to be inserted ≤ 2↑(last bit - first bit)

Before use by pack the value of each of the three arithmetic expressions of a triple will if necessary be rounded to the nearest integer.

During the execution of pack the triples are taken in order from left to right and for each triple the resulting change of the pattern will be made. Those bit positions of the given variable which do not lie within the sections defined by the triples will remain unaffected by the call.

The value of the function designator consists of the bits 0 to 39 of the resulting pattern (bits 40 and 41 are the marks which do not take part in normal assignments and transfers).

If the number of parameters of a pack function designator is not of the form 1+3×k, where k = 1, 2, ... , then the execution of the program is terminated with the message:

param

(cf. section 11.7).

12.2. STANDARD PROCEDURE: split.

12.2.1. Syntax.

```
<split triple> ::= <arithmetic expression><parameter delimiter>
     <arithmetic expression><parameter delimiter><variable>
<split parameter list> ::=
     <Boolean expression><parameter delimiter><split triple>|
     <split parameter list><parameter delimiter><split triple>
<split function designator> ::= split(<split parameter list>)
```

12.2.2. Examples.

```
split(Bool) from bit:(4) to bit:(9) into:(k) and from:(22)
     to:(33) into:(I[6])
q:= split(w[7], 2, 4, m, 33, 41, s)
```

12.2.3. Semantics.

This integer procedure serves to split the bit pattern given as the value of the first parameter into an arbitrary number of shorter patterns, which are obtained as corresponding positive integers while the given pattern is left unchanged. The numbering of bit positions and the correspondence between bit patterns and integers is the same as the one described in section 12.1.3. above.

If the first parameter, a Boolean expression defining the given pattern, is a variable, then a total of 42 positions, numbered from 0 to 41, may be employed. If it is given as a compound expression, then only the positions 0 to 39 are defined.

Each triple of parameters of the form:

first bit, last bit, variable

will assign the integer corresponding to the part of the given pattern held between first bit and last bit, both included, to the variable given as the third parameter of the triple. This variable must of type integer or real. The assignment process will proceed from left to right through the list of triples. The value of the split function designator is the same as that assigned to the last parameter. From these rules, and from the fact that positive integers in GIER ALGOL will have at most 29 bits, we can derive the following restrictions on sensible triples:

$0 \leq$ first bit $\leq$ last bit ≤ 41

last bit - first bit < 29

An improper number of actual parameters in a call of split will cause the execution of the program to be terminated, just as in the case of pack.

12.3. THE EFFECT OF BOOLEAN OPERATIONS.

Within the ALGOL text proper a Boolean variable is represented by bit 0 of the machine word holding the variable, the value 0 representing <u>true</u> while the value 1 represents <u>false</u>. However, the machine operations <u>used</u> to execute the Boolean <u>operators</u> work on the bits 0 to 39 of the words. Consequently the Boolean operations:

-, ∧ ∨ => ≡

may be used to perform parallel operations on the bit patterns generated by means of pack.
Where only the moving, combination, and masking of bit patterns in fixed bit positions within words of 40 bits are required this way of operating may replace the much more time consuming handling provided by the procedures split and pack.

12.4. STANDARD PROCEDURE: gier.

12.4.1. Syntax.
<gier function designator> ::= gier(<variable>)

12.4.2. Example.
```
gier(A[22])
a:= gier(b[2])
```

12.4.3. Semantics.
This standard, <u>real</u> type, function transfers the control to the left hand or full-word <u>machine</u> instruction located in the variable given as parameter. This instruction must be the first one of a GIER machine code program previously placed in the variable given as parameter and any other relevant locations. Normally these locations will form a one-dimensional array. In this case the normal sequencing of the machine will call an execution of the instructions in successive variables of the array, but the user may of course make use of all the facilities of the GIER order code for looping, jumping, etc., as long as the following rules are observed:

12.4.3.1. Return. The control is returned to the ALGOL text on performing the machine instruction hr s+1. When this happens the machine registers s and p must have the same contents as when the last entry from the ALGOL administration into the machine code was made. In addition, the value produced by the function designator, if used in the ALGOL text, must have been placed in RF.
12.4.3.2. Changes of contents of locations. Generally speaking the programmer must regard all machine locations in the cores or on the drum which do not hold the values of variables of the ALGOL program to be reserved for internal purposes. These contents must therefore be left unchanged by the machine instructions activated through calls of gier.
12.4.3.3. Addressing. Since the programmer has no way of knowing where the machine code will be placed in the core store all references to locations within the code itself must employ r-modified addresses.

12.5. STANDARD PROCEDURE: gierproc.

12.5.1. Syntax.
```
<gierproc function designator> ::=
     gierproc(<variable><parameter delimiter><actual parameter list>)|
     gierproc(⟨<proper string>⟩<parameter delimiter>
                                     <actual parameter list>)|
     gier proc(<number><parameter delimiter><actual parameter list>)
```

12.5.2. Examples.
```
gierproc(q[23], s, t, u)
w:= gierproc(s[2], t)
gierproc(⟨alkomp⟩, p, q, r)
```

12.5.3. Semantics.
This standard, *real* type, function extends the facilities provided by standard function *gier* by (1) admitting the machine instructions activated to refer to the quantities of the ALGOL program given as actual parameters and by (2) providing a mechanism whereby a machine coded program can be read into the machine from tape and executed once.

12.5.4. Referring to parameters.

The standard function gierproc may be called with any number of parameters. The first of these must supply the information about the first machine instruction to be executed. References to the remaining parameters from within the machine coded program are made by means of descriptions of the parameters which have been placed in the so-called formal locations. These are addressed relative to the p-register as follows:

Address	Contents
p-1	Bits 0 to 9: The number of parameters of the call. Bits 10 to 39: all zero. Marks: undefined.
p+3	Description of the 1st parameter of the call
p+4	- - - 2nd - - - -
etc.	

The meaning of the description of an actual parameter depends on the nature of this parameter, as follows:

Simple variable, label.

Bits 0 - 9: The absolute address of the location holding the variable or label. A label has the same form as the description of an expression (see below).

Bits 10 - 39 and the marks: all zero.

Numbers, logical values, strings, layouts.

Bits 0 - 39: The value of the construction. For the detailed structure of strings and layouts, see appendix 2.

Marks: b (= 01).

Array identifiers.

Bits 0 - 9: The absolute address of the dope vector. If this is denoted q and the array declaration is: array A[l1:u1, l2:u2, ... , lp:up], and we define ci=ui-li+1 then the dope vector consists of the following:

q-2:	(((.. (l1×c2 + l2)×c3 + l3)× ...)×cp + lp
q-1:	The length of the array = c1×c2×c3× ... ×cp
q:	c2
q+1:	c3
q+p-2:	cp

The constants in q-2 and q-1 are integers with the units placed in position 39, all the other coefficients (if there are any) are represented as floating point numbers.

Bits 10 - 19: The absolute address of the last element + 1 (= the absolute address of the first element + the length).

Bits 20 - 39 and the marks: all zeroes.

Switch identifiers.
Bits 0 - 9: The absolute address of the last switch element description + 1.
Bits 10 - 29: All zeroes.
Bits 30 - 39: The number of switch elements + 1.
Marks: b (= 01).

Expressions, procedure identifiers.
Bits 0 - 9: The value of the stack reference corresponding to the youngest incarnation of the lexicographically enclosing block.
Bits 10 - 19: The track relative address of the entry point.
Bits 20 - 29: All zeroes.
Bits 30 - 39: The track number of the entry point.
Marks: Entry to a left instruction: a (= 10), to a right instruction: c (= 11).

References to ALGOL text from machine coded programs should always be written as activations of actual parameter expressions, i.e. labels and procedure identifiers should not be used as actual parameters of gierproc function designators. Such references require that the contents of the registers s and p are the same as when the last entry or return from the ALGOL administration into the machine coded program took place. The reference to the expression whose description is stored in the address p+expr (this will be actual parameter no. expr-2) must be written as follows:

```
    pm p+expr, hs  s-1
```

Upon return from this reference the address of a location which holds the value of the actual parameter expression activated will be found in the so-called universal address. The address where the universal address is found is stored in the location s-2. This latter location contains the indirect addressing mark. The value of the actual parameter expression can therefore be referred to using the address (s-2). The address of the value, which is of interest in the case of a subscripted variable, can be found by using suitable address operations (e.g. it(s-2) or arn(s-2)D). It should be noted that the value of register s may not be the same before and after the reference to an expression. It is always the latest value supplied by the administration which should be used in further references and in the return instruction. Note also that this way of referring to actual parameters is correct for any expression, including constants and simple variables. After a reference to a simple or subscripted variable, the universal address contains the address of the variable itself.

12.5.5. Activation of code in array.

If the first parameter of a call of gierproc is a variable the control of the machine will be transferred to the location where this variable is stored in the same manner as in the case of standard procedure gier. In this case all the rules of section 12.4.3 hold, except that the contents of the register s to be used on return is the latest value received from the administration, which may differ from the value at entry if references to actual parameter expressions have been made.

12.5.6. Activation of code from tape.

If the first parameter of a call of gierproc is a string or a number the machine program to be executed must be supplied from the input tape. This machine program will then be executed once and subsequently overwritten by other information. The input tape should be of that form which is produced by the kompud program of the HJÆLP system, with the first information 3c, to suppress the normal first information on the tape and the second information 4c, to suppress the normal treatment of instruction hsf2 (this facility is not described in the HJÆLP manual). The machine code punched by kompud must in the first word contain an identification consisting of a string of 6 or fewer characters packed in the manner described in appendix 2, output from pass 1, or a number. The following word must be the point of entry into the machine coded program.

Example of input to SLIP:

```
i = 10
f
7̄77
m
p̄m  p4 , hs s-1  ;
arn (s-2) D      ; Radr:=  address of the subscr. var.
ar  r2 , gr (p5) ; b:= jump; (the marks are irrelevant)
hr  s1           ;
hv
h kompud
3̄c
4c
10//15
e
```

If this code is called as follows

gierproc (777, A[i], b)

it will place in b an instruction jumping to the variable A[i]. This means that a jump to the machine code placed in A[i], A[i+1], ... , may be performed by the very fast operation

gier(b)

instead of the much slower gier(A[i]).

When gierproc is called to activate a machine program from tape it will start by declaring an array to hold the machine coded program. If this array causes the storage capacity to be exceeded, the execution of the program will be terminated with an alas-message (cf. section 11.7). Otherwise the machine will start reading the tape and check the sum character at the end of the tape. Also it is checked that the string or number supplied in the first parameter of the call matches the identifying string or number appearing on the input tape except in the case that the first parameter in the call is the number 0 (zero). If either of these checks fails the message:

gier

is typed and the machine will be ready to read another tape when a SPACE is typed on the typewriter.

The general rules for writing the machine code are the ones given in section 12.4.3 with the addition mentioned in section 12.5.5.

12.6. STANDARD PROCEDURE: gierdrum.

12.6.1. Syntax.

```
<gierdrum function designator> ::=
     gierdrum(≮<proper string>≯<parameter delimiter><variable>)|
     gierdrum(<number><parameter delimiter><variable>)
```

12.6.2. Examples.

```
gierdrum(≮add≯, længde[2])
q:= gierdrum(≮mult≯) length:(multlængde)
```

12.6.3. Semantics.

This integer type function designator serves to read a machine coded program or data from tape and place it on the drum. It therefore combines some of the actions of standard procedures gierproc and to drum, more particularly as follows: Like gierproc it requires the first parameter to be a string or a number which matches the contents of the first word of the program produced in binary form by the kompud program of HJÆLP, except when the parameter is 0. However, instead of jumping to the following instruction of this program gierdrum will transfer it to the drum, including the identifying word. This transfer is entirely analogous to the transfer of an array to drum by means of to drum, i.e. the place on the drum to which the array is transferred is defined by the current value of drumplace and when the call is completed the value of drumplace has been changed by an amount which is available as the value of the function designator.

The second parameter of a call of gierdrum must be a variable of type <u>integer</u> or <u>real</u>. To this variable gierdrum will assign the number of machine words contained in the array transferred to the drum. This is the number of words produced in binary form by kompud and should be used to define the size of the array used to hold the machine instructions or data when they are transferred by fromdrum.

The procedure gierdrum includes the same checks on the size, identity, and control sum, of the machine coded program supplied from the tape as does gierproc.

The pass information is obtained as an optional output during translation (cf. section 11.3.6). It consists of the following: At the end of pass 1, just before the epilogue (cf. section 11.3.4):

1. line <number of the last line of the ALGOL program> end

Following each pass: two or three integers. The first of these, A, always gives the number of drum tracks used to hold the intermediate output from the pass. The remaining have the following meaning:

Pass 1. The figures refer to the storage of long texts on the drum:
B. The number of excess words used on the last track.
C. The number of full tracks used.

Pass 2. B. The number of different identifiers in the program, apart from standard identifiers.
C. The number of words used for storing long identifiers.

Pass 3. The number of blocks in the program.

Pass 4. B. The maximum depth in the stack used for collecting the declarations belonging to each block at the begin of the block and for rearranging procedure calls.
C. The maximum level of nesting of blocks.

Pass 5. B. The number of occurrences of identifiers in the program apart from standard identifiers and the place where the identifier is declared.
C. The number of redeclarations of identifiers.

Pass 6. The maximum number of words used in the (B) operator stack, (C) operand stack.

Pass 7. Max. depth in stack of operand descriptions.

Pass 8. B. Relative address, C. Track number, of program start.

The number of tracks of the program, P, is the sum of pass 1C and pass 8A.

Appendix 2. PASS OUTPUT.

If desired the compiler will produce printed output of the internal output produced by each pass (cf. section 11.3.7). This facility may be used as the last resort in pinning down troubles in using the compiler, whether these are due to programming errors or faulty machine operation. In any case the interpretation of the output requires some insight in the internal working of the translator. For this reason, and since the description of the output given in appendix 2 of the first edition of the present manual is still valid, except for a few inessential changes, we reprint only the description of the packing of layouts and strings. This description is of considerable interest in connection with the facility for generating arbitrary machine words provided by pack (cf. section 12.1).

PACKING OF LAYOUTS AND STRINGS.

Layouts. These are packed in one word as follows:

Bits	0 - 19	A 1 in position p indicates that character number p in the layout (not counting SPACEs) is followed by SPACE.
-	20 - 23	b = number of significant digits
-	24 - 27	h = - - digits before the point
-	28 - 29	fn = sign of number part (no sign = 0, - = 1, + = 2, ± = 3)
-	30 - 33	d = number of digits after the point
-	34	n, 0 if no n, 1 if n
-	35 - 37	s = number of digits in exponent
-	38 - 39	fe = sign of exponent (code as for fn)

Other strings. These are packed character by character. One character uses 6 bits. The numerical value of the character is the one given in section 6.5 of the Manual with the exception of CAR RET which is represented by 63. Characters for UPPER CASE and LOWER CASE are included as needed, but all strings are understood to begin and end in lower case. The end of a string is indicated by the character value 10. The strings having 6 or fewer characters are packed in one word and carried through the translation process like numbers. Longer strings are stored on the drum during pass 1 and are represented during translation and at run time by a word referring to the drum.

Packing of short strings (6 or fewer characters):

Bits	0 - 3	The constant 10	
-	4 - 9	Character no. 6	Unused character positions are set to 10
-	10 - 15	- - 5	
-	16 - 21	- - 4	
-	22 - 27	- - 3	
-	28 - 33	- - 2	
-	34 - 39	- - 1	

The word referring to a long string has the following structure:

Bits	0 - 9	The constant 0
-	10 - 19	track relative address, tr
-	20 - 29	The constant 0
-	30 - 39	track number, tn

On the drum the characters are stored in consecutive words on track tn in relative addresses tr, tr+1, tr+2, ... etc. The word following the one having relative address 39 on track tn is word 0 on track tn-1. Within each word the characters are packed in the following order:

Bits	0 - 5	Character no. 7
-	6 - 11	- - 6
-	12 - 17	- - 5
-	18 - 23	- - 4
-	24 - 29	- - 3
-	30 - 35	- - 2
-	36 - 41	- - 1 (bit 40 is mark a, bit 41 is mark b)

Appendix 3. SELECTED EXECUTION TIMES.

The execution time of a program in GIER ALGOL depends not only on its individual algorithmic constituents, but also on the loop structure and the number of variables declared at the time when each part of the program is executed (cf. section 10.4). The times given below are based on actual timings at the machine and include an average track administration time such as it may be expected in loops which may be accomodated completely in the core store. Substantially longer execution times will result under the following circumstances: a) Frequent transfers of program tracks from drum are necessary (cf. section 10.4); b) A major part of the execution time of the program is spent in a loop with a cycle time of the order of 2 millisecond or less and this loop happens to have been placed across a program track transition by the compiler. A program suffering from the latter of these calamities may be cured by insertion of a suitable amount of neutral program (r:= r or the like) before the final end.

Algorithmic entity	Example	Execution time, milliseconds
Addition	a + b	0.12
Multiplication	a × b	0.18
Division	a / b	0.21
Square	$a \uparrow 2$	0.18
Cube	$a \uparrow 3$	0.4
Power, integer exponent	$a \uparrow i$	
abs (exponent) = 1		3.8
10		5.5
100		8
1 000		10
10 000		12
100 000		14
1 000 000		16
Power, real exponent	$a \uparrow r$	12
If clause with simple relation	if a>b then	0.3
Subscripted variable		
1 subscript	A[i]	0.9
2 subscripts	B[i, j]	1.2
3 -	C[i, j, k]	1.5
Step-until element, constant step and single upper limit, each loops	step 1 until n	0.6
Block with simple variables	begin real a; end	1.4
Block with array declaration	begin array a[1:10]; end	3.0
Reference to formal parameter called by name. Actual parameter is		
simple		0.4
expression		3.2
array identifier		0.0
switch identifier		0.0
procedure identifier		0.0

Assignment statement		
a:= 0		0.05
a:= b		0.1
Go to statement		
Simple, within current block	go to A;A:	0.8
To switch designator	go to s[i]	2.1
Call of declared procedure having an empty procedure body		
No parameter	P;	3.8
1 parameter	Q(a);	4.7
2 parameters	R(a, b);	5.2
3 -	S(a, b, c);	5.5
Call of standard procedure		
abs	abs(x)	0.17
arctan	arctan(x)	6.6
cos	cos(x)	6.0
entier	entier(x)	0.4
exp	exp(x)	5.8
ln	ln(x)	5.6
sign	sign(x)	3.2
sin	sin(x)	5.8
sqrt	sqrt(x)	6.2
kbon	b:= kbon	3.5
split or pack, 1 triple	pack(b,20,25,k)	9
5 triples: split(b,0,3,k,8,11,m, 16,19,n,24,27,p,32,35,q)		24

For the general description, refer to section 11.4.4.

The pass number is typed as an integer from 1 to 8 followed by a point (.) at the beginning of the first error message belonging to the pass.

The line referred to in an error message will normally be the line in which the error occurs, but there are exceptions to this rule: a) A construction appearing near the beginning or ending of a line may have its line number changed by one unit. b) One of the error messages from pass 5 may supply a quite misleading line number (see below). c) Error messages from passes 7 and 8 will always refer to line 0.

PASSES 1 - 8.

program too big

This indicates that the capacity of the drum has been exceeded by the demands of the program text. Remedy: Use a version of the compiler which leaves more space on the drum, if such a version is available (cf. section 11.1).

PASS 1.

character

A character to which no meaning is asssigned appears on the input tape.

compound

A string of characters which represents some of the first characters of a compound symbol (cf. section 7.1.2), but not the following ones, appears in the input.

)<improper>.

The construction)<letter string> is not followed by :(

comment

The delimiter comment is not preceded by begin or ;

string

The compound symbol ⟨ is followed neither by < nor by a layout (cf. section 8.3.1)

sum

The character following a SUM CODE on the input tape does not match the corresponding value formed from the previous input (cf. section 11.4.5).

PASS 2.

too many identifiers

The program uses too many different or long identifiers. Remedy: Use the block structure to reduce the number of different identifiers.

PASS 3.

- delimiter

Two operands (i.e. identifiers, numbers, logical values, strings, or compound expressions within parentheses) follow each other. Examples:

7.3 sin(5)   4 true   r.77   r⟨<string⟩

operand

a) An operand appears in a wrong context. Examples:

7:=   begin true;

b) An operand is missing. Example:

a:= [i]

delimiter

a) The delimiter structure is impossible. Examples:

begin r/i:=   if go to   if for

b) Binary operator does not follow operand. Example:

i:= ×r;

- operand

Operand is missing at end of construction. Example:

r:= r/;

termination

Parentheses, brackets, or bracket-like structures do not match. Examples:

r[i]   begin r:= a + b,   p(i, r;

number

A construction which in its first symbols conforms to the syntax for numbers is not terminated correctly, or a number is too big for the capacity of the machine. Examples:

20.; $17.{}_{10}{-3}$ $7_{10}170$

stack

The nesting of begin's parentheses, etc. exceeds the capacity of the compiler.

PASS 4.

stack

The stack formed during the reverse pass 4 exceeds the available capacity. This stack is used to transfer the information about the type and kind of each identifier and of each switch element from the place where it is declared (for labels, where it labels a statement) to the begin of the block in which it is local, and the information about each actual parameter to the left parenthesis of the call.

PASS 5.

+ declar.

The same identifier is declared twice in the same block or appears twice in the same formal parameter list. Note that labels are considered to be declared as explained in section 4.1.3.

+ specif.

The same identifier is specified twice in the same procedure declaration heading.

- declar.
  An identifier is used at a place where it is not declared. The line number associated with this error message will be misleading in the following two cases: a) The identifier is an actual parameter. The line number will point to the line in which the left parenthesis of the call appears. b) The identifier is a switch element. The line number will point to the line which contains the <u>begin</u> of the block in which the switch is declared.

- specific.
  The specification of a formal parameter is missing.

- formal
  An identifier is specified, but does not appear in the formal parameter list.

value
  A formal parameter which according to the specification given cannot be called by value appears in a value part.

stack
  The list of the identifiers which are redeclared simultaneously exceeds the capacity of the compiler.

PASS 6.

subscript 704
  The number of subscripts given in a subscripted variable does not match the corresponding array declaration.

proc. call or ident.
  An additional integer in the message distinguishes two variants of this error:
  7?0: An identifier preceding immediately a left parenthesis, (, does not conform to the procedure call implied in the construction by being of wrong kind or having a wrong number of parameters.
  840: A procedure identifier appears in a context not consistent with its declaration.

type <error number>
  The number associated with this error message indicates from where in pass 6 the error program has been called. A more detailed description of the error associated with each integer is given in the table below. In this table the description
  <i op> ::= <inadmissible operand>
  indicates an operand which has wrong type or kind in the given context. Note that expressions are regarded as operands. The examples assume the following declarations:

integer i; real r; Boolean b; array a1[1:10], a2[2:4, 4:6];
switch s:= L, L2; procedure p0;; procedure p1(f); real f;

Error number	Error constructions	Examples	
576	+<i op>|-<i op>|×<i op>|/<i op>|↑<i op>	+s	/L
582	:<i op>	:r	
585	<Boolean operand>:= <i op>	b:= r	
590	< <i op>|≤ <i op>|= <i op>|	= b	
	≥ <i op>|> <i op>|≠ <i op>		
593	∧ <i op>|∨ <i op>|≡ <i op>|¬<i op>	∨ (i - 2)	
596	<i op><binary operator><i op>	i ∨ a1	b = s
599	<real operand>:= <i op>	r:= s	
604	<integer operand>:= <i op>	i:= p	
616	abs(<i op>)|arctan(<i op>)|cos(<i op>)|	cos(a2)	ln(r = i)
	entier(<i op>)|exp(<i op>)|ln(<i op>)|		
	sign(<i op>)|sin(<i op>)|skrvkopi(<i op>)|		
	skrvml(<i op>)|skrvtegn(<i op>)|sqrt(<i op>)|		
	streng(<i op>)|sættegn(<i op>)|		
	trykkopi(<i op>)|trykml(<i op>)|		
	tryktegn(<i op>)|tryktom(<i op>)		
630	go to <i op>|switch sw:= <i op>,	go to b	
633	;<i op>;	; r ;	
640	<i op><binary operator>	b = r∧	(i - 2)∨
646	<i op>[	i[	
649	til tromle(<i op>)|fra tromle(<i op>)	til tromle(r)	
657	then <i op>	then s	
677	<i op> else <i op>	2 - r else b	
	<i op> else if . . . then <i op>		
686	:= <i op>:=	r:= b:= b ∨ b	
690	<i op> step|<i op> until|for . . . <i op>,|	i = r step	
	for . . . <i op> do		
697	<i op>]	p0]	
714	<i op>:=	p0:=	
725	<i op> then|while <i op> do	if r then	
728	for <i op>:=	for a1:=	
732	Inadmissible subscript	a1[L]	a1[i=r]
735	<i op> : (in array declaration)	array q[b:1];	

PASS 7.

number

An arithmetic expression having only numbers as operands results in a value outside the range of the machine. Examples:
1/0 $7_{10}35\times9.2_{10}135$

PASS 8

stack

The two stacks of program points used during pass 8 exceed the capacity of the compiler. Remedy: reduce the number of labels and of nested for and conditional statements used simultaneously.

Appendix 5. EMERGENCY OUTPUT OF THE STACK.

During execution of an ALGOL program the currently active variables (cf. section 10.2) are held in a stack in the core store. The system includes a program which will produce the contents of this stack in a form which indicates the meaning of the variables within the structure of the active blocks. It should be noted that in some rare cases the interpretation of the contents is not unique so that sometimes a false picture will be given. This is mentioned in some particular cases below. The output may be obtained whenever a terminating message (section 11.7) has been typed out. The entry into the output program and the choice of output medium depends on the letter typed in this situation, as follows:

Letter	Effect
b	Emergency output on both typewriter and punch
p	- - - punch
w	- - - typewriter

During the output of the stack a new choice of output medium can be made by changing the contents of the register KA. This will stop the machine when the currently printed number has been completed. Either of the three above letters will change to their respective media. Typing the letter e will terminate the output with an end-message. Any other character will cause the output to continue on the same medium. After completion of the emergency output the machine returns to the termination situation with an end-message.

A5.1. VALUES IN THE OUTPUT.

The representations given below only hold for variables to which a value has been assigned. Before assignment numerical variables may appear as logical values.

Numerical values (<u>integer</u> or <u>real</u>) will be printed in one of the following two layouts

-nddddddd $\qquad$ -.ddd$_{10}$-ddd

depending on whether the value is integral or not.

Logical values take several forms, depending on their previous history:

	Representation of	
	<u>true</u>	<u>false</u>
Value directly writtten in ALGOL program	1	-1
Value formed by expression, either	+	-
or a numerical value, x, with	abs(x)>1	abs(x)<1

A5.2. PROGRAM POINTS.

The following kinds of program points are specified in the output:

1) Named points, i.e. entries to procedure bodies and points supplied with labels.
2) Return points, i.e. points at which the execution will be continued when a procedure call or an activation of an actual-parameter-expression is completed.

The output of a program point refers to the storage of the program on the drum and supplies a track number (2-3 digits) followed by a track-relative address (0-39). Formally:

<point> ::= <track number><track relative address>

The order of the program points on the drum is the same as the order of the same points in the original ALGOL text.

A5.3. OWN VARIABLES.

The output starts with the message

stack

followed by the values of the own variables of the program, if such are declared. These appear blockwise taking the block-begins of the program in the backward order. Within each block the own variables appear in the order in which they are declared.

A5.4. LEVEL STRUCTURE.

The following output gives the values and program points of the active parts of the program, arranged in an order which may be explained as follows: Imagine that all the copying implied in the definition of the procedure call (section 4.7.3) were actually performed on the program, then the currently active part of the program would form a nested structure of blocks, procedure bodies, and actual-parameter-expressions. The output supplies the values and program points belonging to each of these levels in turn, starting with the outermost block.

The output given for each level starts with an appropriate heading. The possible level headings and the associated output of values and points are as follows:

A5.5. LEVEL HEADING: block

A block may be an ordinary program block or the body of a procedure. Every procedure body activated will appear as a block (cf. section 5.4.3). The output in general has 3 parts:

(1) For the bodies of type procedures: the value assigned to the procedure identifier. If no value has yet been assigned: 0 (zero).

(2) Following the heading

points

appear the program points of all named points (entries to procedure bodies and labelled points) which are local to the block, in the order in which they appear in the program. These may be useful in localizing procedure calls and activations of actual-parameter-expressions, through the return points of these latter.

(3) The local variables of the block. The first few variables in the output normally are internal working variables introduced by the translator. These are followed by the local simple and subscripted variables in the order in which they are declared in the block head. Each sequence of simple variables is headed by the message

variables

whereas each array is headed by the message

array

followed by a line giving the structure of the array in the form of a sequence of integers indicating the number of values taken by the 1st, 2nd, ... , n th, subscript.

The values of the subscripted variables are given in a linear sequence, as in the input (section 9.4.3.2), printed with 5 in a line. Possible misinterpretation: the structure will include too many subscripts if in the block head the array declaration is followed immediately by an integer-valued variable which is an exact divisor of the number of values taken by the first subscript.

A5.6. LEVEL HEADING: proc. call.

This will be followed by an output of a value for each actual parameter in the order in which they appear in the call. This value will be the correct one for parameters called by value, while the value printed for other parameters is meaningless. Following this output the message:

return

announces the output of the return point, i.e. the program point immediately following the procedure statement.

Every procedure call level will be followed either by a block level, giving the local values of the procedure body, or by a value call, referring to the evaluation of an actual-parameter-expression called by value.

Standard procedures are peculiar in several ways: (1) The parameter of the procedures abs, arctan, cos, entier, exp, gier, ln, outchar, outcopy, outsp,setchar, sign, sin, sqrt, writechar, and writecopy, is evaluated before the procedure is called. (2) The procedures input, output, write, outtext, and writetext, remove their parameters from the stack as they are processed. The number of parameters shown in the output may therefore be too small. (3) Standard procedure identifiers used as actual parameters will confuse the analysis of the procedure call. However, since the rest of the stack is not affected the details need not be given here. (4) Only the following standard procedures enter a block when activated: todrum, fromdrum, split, pack, gierproc, gierdrum.

A5.7. LEVEL HEADINGS: name call AND value call.

These headings are followed by one return point. In a name call this indicates the point immediately following the formal identifier. In a value call it points to a place near the entry to the procedure body. Value calls are completed before the procedure body block is entered.

Calls of parameters from standard procedures will appear as value calls. The return point given identifies the standard procedure as shown in the following table, which refers to the storage parameters discussed in section 11.1:

Track	Relative address	Value call activated by
19+c60	6	output, write: layout
	18	- - : other parameter
26+c60+c96	9	outtext, writetext
29+c60+e96	27	input
36+c60+2×e96	8	pack, split: 1st parameter
-	13	- - : first bit
-	16	- - : last bit
-	20	- - : pattern
37+c60+2×e96	9	gierdrum, gierproc: 1st parameter
-	15	gierproc, parameter called from machine code
-	23	gierdrum: 2nd parameter

A5.8. CONDITIONS AT THE LAST LEVEL.

When the emergency is caused by an overflow of the stack (alas or array) the last level shown in the output will normally represent an incomplete state. However, the lower levels will not be affected.

Overflow (spill) may be caused by the ⋀ operation having an exponent of integer type. In this case the last level in the output will show a procedure call with one parameter = the value of the exponent.

A5.9. EXAMPLE OF EMERGENCY OUTPUT.

The following program shows most of the feature of the emergency output.

```
begin real r; Boolean t, f, bf, bt;
real array A[3:4, 1:5]; integer i,j;
real procedure P(a, b, c); value a, b; real a, b; array c;
    begin P:= c[3,4] := a; end P;
M: for i := 3,4 do for j := 1 step 1 until 3 do A[i,j]:= 10×i + j;
t:= true; f:= false; r:= 0.5; bf:= r=0; bt:= f=bf;
    begin
    real procedure Q(f); real f;
        begin own real own1 in Q, own2 in Q; integer k in Q;
K:      own1 in Q:= 3.33; own2 in Q:= 7.89; k in Q:= 55;
        Q:= f;
T:      end Q;
    own integer own in block; integer  s, w, u;
    own in block:= 888; s:= 15; u:= 7; w:= 0;
    P(u+s, Q(s/w), A);
L:  end block;
N:
end;
```

The program initializes some of the variables of the outer block, enters the inner block and initializes its variables, calls the procedure P, which evaluates its first parameter. While evaluating the second it calls the procedure Q by way of the actual parameter-expression. Within Q some variables are set and then another actual-parameter-expression is called by the reference to f. This brings the program into spill. In the emergency output which follows the meaning of the data given is shown by notes within parentheses.

```
stack
 .333$_{10}$   1   (own 1 in Q)
 .789$_{10}$   1   (own 2 in Q)
         888   (own in block)

block          (outer block)
points
  317  26      (entry to P)
  318   9      (label M)
  319  38      (label N)
```

```
variables
-.596⏨   2    (internal variable)
         -    (    -        -    )
 .500⏨   0    (real r)
         1    (Boolean t)
        -1    (    -   f)
 .500⏨   0    (    -   bf)
         +    (    -   bt)
array         (A)
         2    ×        5
        31              32             33          .626⏨    2           .626⏨   2
        41              42             43          .625⏨    2          -.405⏨ -55

variables
         4    (integer i)
         4    (   -    j)
block         (inner block)
points
  318   38    (entry to Q)
  319   37    (label L)
variables
        15    (integer s)
         0    (   -    w)
         7    (   -    u)
proc. call    (P(
        22        u+s,
         -            Q(s/w),
         -                    A))
return
  319   37
value call    (call of Q(s/w))
  317   31
proc. call    (Q(
         -        s/w))
return
  319   37
block         (body of Q)
         0    (No value yet assigned to Q)
points
  319   13    (label k)
  319   18    (  -   T)
variables
        55    (k in Q)
name call     (f, i.e. s/w)
  319   17
stack end
end
```

Tape E of the set of compiler tapes (cf. section 11.1) supplies a program for producing an output on tape of the various parts of the compiler or other sections of the drum in a binary form, suitable for later fast readback into the machine. The program is given in the form of a paper tape procedure as produced by the kompud program of HJÆLP, to be used with gierproc or gierdrum (cf. sections 12.5.6 and 12.6).

A6.1. ACTIVATION.

If executed directly from tape the program must be put in action by a statement of the form
gierproc(≮<binout≯,<variant><drum region list>)
where variant is an expression having one of the values 0, 1, 2, 3, or 5, and
<drum region list> ::= <empty>|,<drumplace value>,<length>|
<drum region list>,<drumplace value>,<length>
If the program is first placed on the drum by gierdrum and later put in action after transfer to an array, only calls of gierproc having 3 as the value of the second parameter (variant) are in order, because only this variant does not require copying from tape E after the activation.

Binout requires about 166 locations during execution.

A6.2. VARIANTS.

The value of the second parameter of the call of gierproc specifies one out of 5 variants of the program. The following table gives a brief summary of the variants. More details are given in the following sections.

Variant	Special output region	Re-input program	Exit after re-input
0	None	Track 0 or SLIP	run
1	The translated program	Track 0 or SLIP	run
2	System + transl.prog.	Track 0 or SLIP	run
3	None	gierproc,gierdrum	Program
5	Complete compiler	Track 0 or SLIP	algol

A6.3. OUTPUT REGIONS.

The output will include the special region which belongs to the variant in addition to that which is specified in the drum region list of the call of gierproc. The translated program includes the P tracks produced by the translator (appendix 1). The system includes the run-time administration, the emergency output program (appendix 5), and the standard procedures; its length is given as S in section 11.1. This section also gives the length of the complete compiler, N.

The drum region list defines regions of the drum in terms of the value of drumplace and the length of the array associated with the call of todrum which was used to transfer the values to each region. Each of these parameters may be supplied as an arithmetic expression.

A6.4. RE-INPUT OF THE BINARY TAPE.

The output of variants 0, 1, 2, and 5, starts with a special input program, which is copied by binout from the last part of tape E to the output. This output can be read back to the drum as follows: Set KA and KB to L. Press the HP-button. Depending on the message now typed, start reading as follows:

Message	To start input, type as follows
hp-knap	1
FEJL	SPACE
KC ALGOL	SPACE

This process will change some of the contents of the core store (approx. the first 400 locations), and also the part of the drum used as core image by HJÆLP.

The output of variant 3 can only be used with gierproc and gierdrum.

A6.5. EXIT AFTER RE-INPUT.

After re-input of the output of variants 0, 1, and 2, the machine transfers track 6c60 to location 0 in the core store and jumps to location 36. If this track holds the code of the ALGOL system the machine will enter the RUN-SITUATION (section 11.5). The output of variant 3 will continue with the activation of the code of the binary tape (gierproc) or with the ALGOL program where gierdrum was called. The output of variant 5 enters the COMPILER-READY-SITUATION (section 11.3).

A6.6. ALARMS.

During a call of binout the following conditions will cause termination of the execution of the program (param): (a) The number of parameters in the call of gierproc is odd. (b) A drum region with a negative length is specified. (c) A specified region exceeds the boundaries of the drum.

This re-input is checked by means of sums. Failure of a check will produce the error message

sum

In this case the re-input can be attempted again, by repositioning the tape in the reader and typing a SPACE on the typewriter.

AIPHABETIC INDEX.