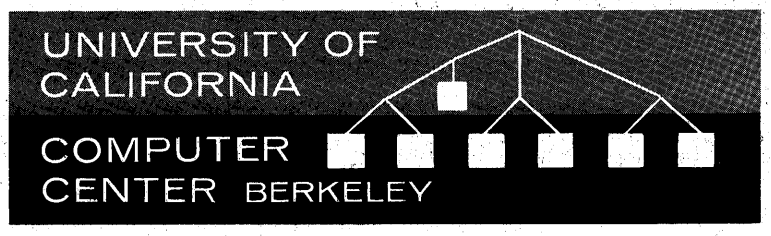


BC ALGOL Manual

October 1966

BC-ALGOL

University of California
Computer Center
Berkeley

Computer Center
University of California
Room 201, Campbell Hall
Berkeley, California 94720

BC ALGOL Manual Change Slip

Please send me changes and additions to the BC ALGOL Manual until the close of the current semester. I understand that changes will be sent to me at the address entered below unless a change of address is submitted to the Computer Center, 201 Campbell Hall.

NAME _____

ADDRESS _____

_____

Date _____

B C ALGOL MANUAL

October 1966

University of California
Computer Center
Berkeley

Table of Contents

Ralph Love, Gary Anderson, Russell Briggs, Gayne Winters, Tom Marlin, R. Sherman Lehman, Klaus Wirth

Introduction

The present manual is the third revised and enlarged edition, and describes the BC-ALGOL System as developed by Ralph Love, Russell Briggs, Ken Thompson, Ron Smith, Gary Anderson, Klaus Wirth, Gayne Winters, R. Sherman Lehman, John McConnell and David Redell. The revision was done by John McConnell, David Redell, Chr. Gram, and R. Sherman Lehman. This manual covers all the earlier writeups on the BC-ALGOL System, the input and output procedures, the print and read procedures, the comment control options, the revised diagnostics, the character and bit manipulation procedures, the use of code procedures, and the syllable string. Furthermore, the manual includes a description of tape handling procedures, the stack dump, library tape operations, new character handling facilities, and a run-time execution trace. The notation used is that of the reference language (see Appendix 1).

This manual is not intended as a textbook in ALGOL Programming. It is, rather, a reference manual describing the use of the BC-ALGOL System, and thus assumes some degree of familiarity on the part of the reader with ALGOL 60. Beginning ALGOL programmers will find sections 1.1, 1.5 - 1.7, and Chapters 2-5 most helpful. More advanced users will find the remaining sections of the manual useful in utilizing the more powerful features of the system.

The BC-ALGOL System is a program running under the FORTRAN Monitor System (FMS) on the coupled IBM 7040-7094 system at the Computer Center, University of California, Berkeley. The program consists of: a translator (or pre-processor), and an interpreter (or processor). The translator accepts a BC-ALGOL program (on punched cards) as source language and translates it into a string of operators and operands (called the syllable string) written in a modified Polish notation. The translator also performs a partial syntactical check on the source program.

If no errors are found, the interpreter executes the ALGOL program by processing the syllable string interpretively.

If any error is found, the execution phase is not entered, and when more than 8 different syntactical errors are found, translating is discontinued at that point.

1. The Relation Between BC-ALGOL and ALGOL 60

1.1 Source Language

The BC-ALGOL source language is a punched card hardware representation of ALGOL 60 as defined by the following transliteration table for the basic symbols.

Transliteration Table

ALGOL 60	BC ALGOL	ALGOL 60	BC ALGOL	ALGOL 60	BC ALGOL
a	A	÷	'DIV'	own	'OWN'
b	B	↑	**	value	'VALUE'
⋮	⋮	go to	'GO TO'	true	'TRUE'
z	Z	begin	'BEGIN'	false	'FALSE'
A	⊕ A *	(	(	if	'IF'
B	⊕ B	[	.(	then	'THEN'
⋮	⋮	`	'('	else	'ELSE'
Z	⊕ Z	;	$	end	'END'
:=	=				
≡	'EQV'	:	..	)	)
⊃	'IMP'	comment	'COMMENT'	]	).
∪	'OR'	while	'WHILE'	'	')'
∩	'AND'	for	'FOR'	,	,
¬	'NOT'	step	'STEP'	.	.
<	'LSS'	until	'UNTIL'	10	'10'
≤	'LEQ'	do	'DO'		
=	'EQL'	integer	'INTEGER'		
≥	'GEQ'	real	'REAL'		
>	'GTR	Boolean	'BOOLEAN'		
≠	'NEQ'	array	'ARRAY'		
+	+	switch	'SWITCH'		
-	-	procedure	'PROCEDURE'		
×	*	label	'LABEL'		
/	/	string	'STRING'		

* See "Capitalization Symbol", Appendix 2.

1.2 Use of Special BC-ALGOL Delimiters

BC-ALGOL includes the following four delimiters not mentioned in the ALGOL 60 report:

Notation in Manual	BC-ALGOL
assembly	'ASSEMBLY'
binary	'BINARY'
code	'CODE'
oct	'OCT'

The delimiters binary and assembly are used to allow the use of machine-coded procedures; see sections 1.6, 1.7 and chapter 7.

The delimiter code allows the use of procedures precompiled and included in the BC-ALGOL system; see section 1.6 and chapter 7.

The delimited oct is used to introduce constants written in octal rather than the conventional decimal notation; integers preceded by oct will be computed as base 8 rather than as base 10.

Furthermore the delimiter comment followed by a colon and one of several special text strings is used as an instruction to the system causing a special action at translation or execution time. (See Comment Controls, chapter 4.)

1.3 Restrictions of BC-ALGOL

Identifiers may have any length but only the first 18 characters (other than blank spaces) are significant.

Integers may not be used as labels.

The subscript bounds for arrays declared own must be constants.

1.4 Interpretations of the ALGOL Report

We distinguish the interpretations given in this section from the restrictions stated previously. In the case of the restrictions, it is recognized that the BC-ALGOL implementation does not implement ALGOL 60 fully as defined by the Revised Report. The interpretations, on the other hand, represent choices of the implementers on difficult questions where the exact meaning of the Report is not clear. The user should be warned that other implementations of ALGOL are quite likely to handle some of these questions differently. All references in this section are to the Revised Report (see Appendix 1).

a. The order of evaluation of primaries in an expression and subscripts of a subscripted variable is from left to right. (See Sections 3.1.4.2, 3.3.5.2.) The expressions occurring in array declarations will be evaluated in the order of their occurrence in the block head. (See Section 5.2.4.) The assignment of values to the actual parameters called by value occurs in the order in which they occur in the actual parameter list. (See Section 4.7.3.1.)

b. In addition to standard functions available without explicit declarations (see Section 3.2.4) there are standard procedures available without explicit declarations -- for example, input and output procedures. An identifier used for a standard function or procedure may, however, be either declared or used as a label. In that case in the corresponding block the standard function or procedure cannot be referred to since the rules of Section 4.1.3 apply.

c. Section 4.1.3 is interpreted as if the following addition were made to the second paragraph: A program may be labelled. In this context such a program should be considered as if it were enclosed by begin and end and treated as a block.

d. Paragraph 4 of Section 4.5.3.2 is interpreted to mean: If none of the Boolean expressions of the if clause is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement except for possible side effects due to the evaluation of the Boolean expressions.

e. The description of how the values assigned to the controlled variable of a for statement are obtained is interpreted literally only when the for list elements are arithmetic expressions or of the form E while F . For an element of the form A step B until C it is assumed that the execution described in section 4.6.4.2 is not dependent upon the order of evaluation of B and C nor on whether B is evaluated more than once per step.* If the controlled variable is a subscripted variable, then it is assumed that the effect of the execution corresponding to a step-until element does not depend upon whether the subscripts of the subscripted variable are evaluated more than once per step.

Section 4.6.5 is interpreted as if the following sentence were added: Upon exhaustion of each step-until element or while element of a for list the value of the controlled variable is undefined.

f. The values of own quantities are handled as if they were values of quantities local to a block enclosing the entire program. The scopes of the identifiers for own quantities are, however, determined by their declarations. (This treatment of own variables is certainly the intended one for own variables not declared within a procedure body. (See Section 5.) For own variables local to a procedure body, what the intended interpretation is, especially when the procedure is called recursively, is highly controversial.)

---

* At present B and C are evaluated, in this order, once per step.

g. A procedure whose declaration begins with a type declaration may be called by means of a procedure statement. If the procedure identifier occurs explicitly as a left part in an assignment statement, then it will be treated as in Section 5.4.4 but upon exit from the procedure body the value of the procedure identifier will be lost.

h. A designational expression may be called by value if the corresponding formal parameter is specified by the specifier label. (See Sections 4.7.3.1, 5.4.5.) In this case when the formal parameters in the value list are assigned values before entering the procedure body, the designational expression is evaluated and the resulting label substituted for the formal parameter throughout the procedure body. If the designational expression is undefined, then the effect of the procedure call is undefined.

i. Specifications of formal parameters called by name are ignored. (See Section 5.4.5.)

j. Sections 4.7.4 and 5.4.3 are interpreted as requiring that the formal parameter list of a procedure heading may not contain two or more formal parameters which are identical.

k. The exact effect of a function designator is left implicit in the ALGOL Report. We interpret it as if the following addition to Section 4.7.3 were made:

A function designator also calls for the execution of a procedure body. Where the procedure body is written in ALGOL, the effect of this execution will be equivalent to the effect of first performing the operations described in Sections 4.7.3.1 and 4.7.3.2 at the time of evaluation of the function designator. The modified procedure body is then executed as if it were located at the place where the function designator occurs, with conflicts

between identifiers handled as described in Section 4.7.3.3. Statements and expressions within the procedure body containing the procedure identifier are handled as prescribed in Section 5.4.4. If in the execution of the procedure body a go to statement leads out of the procedure body, then evaluation of the expression containing the function designator is discontinued and the next statement executed is determined as in Section 4.3.3. Otherwise, upon exit from the procedure body evaluation of the expression in which the function designator occurs is continued. If the function designator occurs in an array declaration, then according to Section 5.2.4.4. the evaluation of the function designator must be completed.

The rules in Sections 4.7.4. and 4.7.5. are interpreted as holding for function designators as well as procedure statements.

1. Section 4.3.5 is interpreted to mean the following: A switch designator will be said to be out-of-bounds if its subscript expression when evaluated according to Section 3.5.4. assumes a value other than 1, 2, ..., n where n is the number of entries in the corresponding switch list. A go to statement with a designational expressions which is an out-of-bound switch designator is equivalent to a dummy statement, except for possible side effects occurring in the process of evaluating the switch designator. If the designational expression is undefined for any other reason, then the go to statement is undefined. (Note that when the substitution for a formal parameter specified by label is made, the resulting designational expression cannot be a switch designator because it is surrounded by parentheses. See Section 4.7.3.2.)

1.5 Arithmetic Variables and Expressions

Variables of type integer and real are represented by normal integer and floating point numbers in the IBM 7094. Therefore the range for integer is

$$\text{abs (integer)} < 34\ 359\ 738\ 368 = 2\uparrow 35 .$$

and the range for non-zero reals is

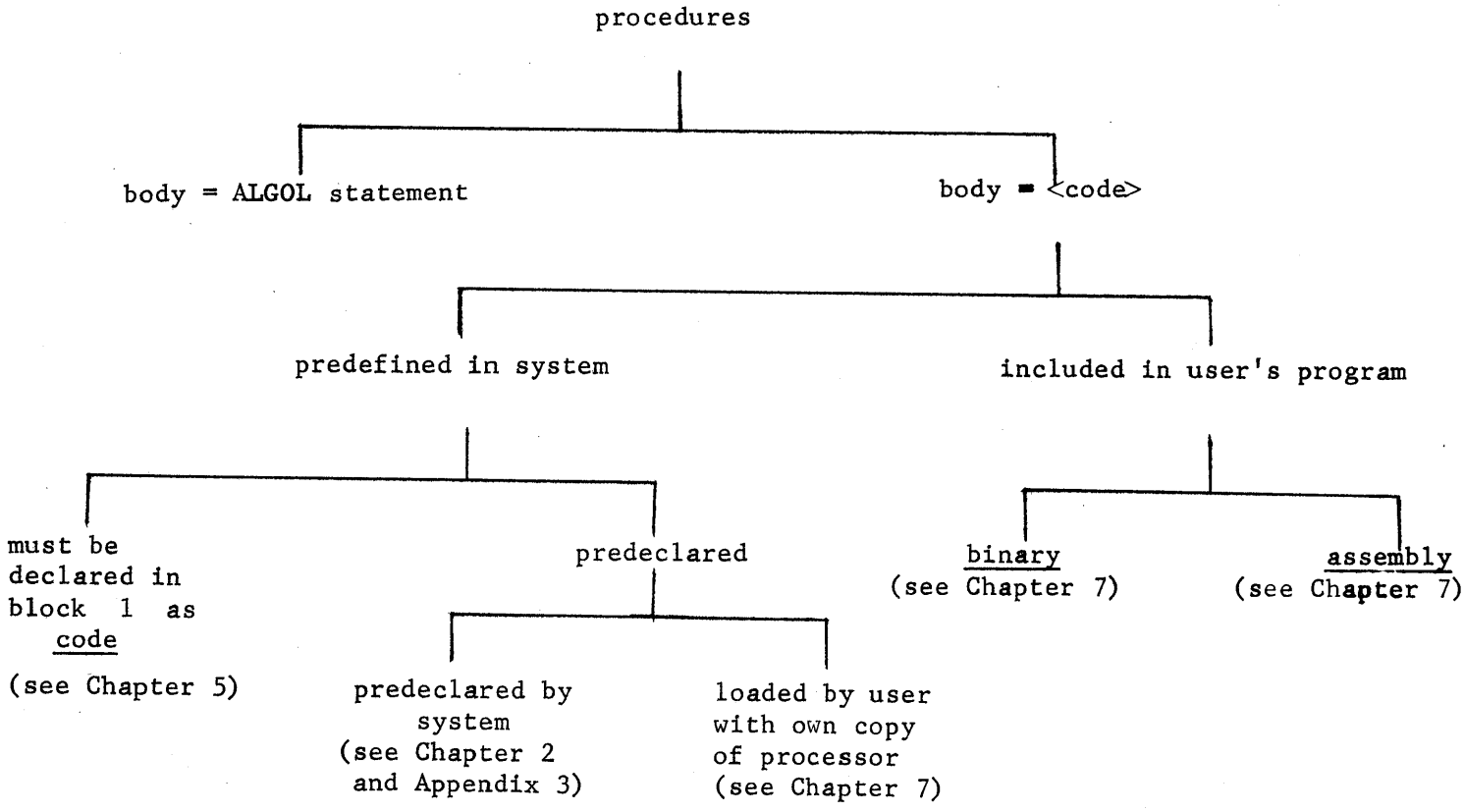
$$2\uparrow(-128) < 2.9_{10}-39 < \text{abs (real)} < 1.6_{10}38 < 2\uparrow 127.$$

The real variables are stored with 27 significant bits. Thus the last bit of a real has a relative value between $7_{10}-9$ and $15_{10}-9$.

Each arithmetic operation is rounded correctly after normalization of the result; hence, the relative error in the result has an absolute value of at most $7.5_{10}-9$.

1.6 Classes of Procedures

In keeping with the provisions of the Revised Report Sections 4.7.8, 5.4.1, and 5.4.6 (see Appendix 1), a procedure body may be either an ALGOL statement or <code> . In BC-ALGOL, <code> refers to procedures in machine language, some of which are supplied by the system, and others of which may be loaded with the user's program. The classification is as follows:



1.7 Form of Source Card Deck

BC-ALGOL programs are run under the FORTRAN Monitor System (under the DCS Monitor System) and require two standard monitor control cards before and one after the source deck. After the last end of the ALGOL program, there must follow a card having the character $ in column 1; this may be followed by cards with data to be read by the ALGOL program. The set-up is (see also [1]):

a. One normal JOB card:

cols. 1-4: $JOB
cols. 8-11: <job number>
cols. 16-36: <options>(time limit, page limit, card limit, rush, number of copies)
cols. 37-60 <name and other identification>

b. An FMS control card:

cols. 1-4: $FMS
cols. 8-12: ALGOL

c. The ALGOL source program (and binary decks, if any)

d. A card with $ in column 1

e. Data cards, if any

f. A normal EOF card:

cols. 1-4: $EOF

Binary decks may be put anywhere in the ALGOL source program; see chapter 7 for further details.

The ALGOL program must be punched in columns 1-72. The contents of cols. 73-80 are skipped by the translator (but are printed disjointly in the listing of the program and may be used for card identification purposes.) Blanks and shifting to a new card have no significance except that a $ (the transliteration of semicolon) should not appear in column 1 because of possible confusion with Monitor control cards.

1.8 Size Limitations

The finite sizes of available storage and of the tables used by the translator impose some restrictions upon the size of a program.

a. Total Storage: The total storage available for the translated program is approximately 15000 cells. In the stack, each simple variable occupies 2 cells and each array element 1 cell. The store required for the program varies greatly with the program's structure, but for estimation purposes, it may be assumed that 100 statements occupy 400-500 cells.

b. Blocks and identifiers: The maximum number of statically nested blocks allowed in a program is 32. No more than 63 identifiers may be declared in any blockhead, and a maximum of 330 may be current at any time. No more than 800 identifiers total may be declared in any program.

There are some further limitations which only very exceptional programs will exceed. Nearly all such violations give rise to appropriate error messages (see chapter 5).

1.9 Execution Times

The following table shows some selected execution times for the basic operations (timings on the standard functions are found in the next chapter). The times given are rough averages and may be used to make a first estimate of the running time for a program.

		millisec.
assignment: a := b		0.2
arithmetic operations, e.g., r1 := r2 + r3		0.3
Boolean operation, e.g.s, b1 := b2 b3		0.25
exponentiation: r1 := r2 ↑ i , where i = 1,2,3		0.5 - 0.6
r1 := r2 ↑ r3		0.8 - 0.9
transfer: go to L		0.2
if clause: if b then		0.1
for statement:		
for i1 := i2 step i3 until i4 do	initialization	0.8
	each step	0.3
for i1 := i2, i3, ... do	each list-element	0.6
procedure call:		
no parameters: P;		0.4
one simple parameter: P(1);		0.7
block head		
with simple variable: begin real r1;		0.15
with array: begin array A[1:2];		0.4
reference to subscripted variable		
with one subscript A[i]		0.1
with two subscripts B[i,j]		0.2

2. Standard Functions and Timing Procedures

2.1 Standard Functions

The predeclared standard functions include all those recommended in [2] and, in addition, the hyperbolic tangent and the decimal logarithm. For each of them the code consists of some parameter linkage and, (except for abs, entier, and sign) a call to the appropriate library subroutine under the FMS Monitor. The following table shows for each standard function 1) the library subroutine used, 2) the length of the code, 3) the average time used per call, and 4) the accuracy obtained. Further details on the algorithms are found in the write-ups for the subroutines, available from the Computer Center Library.

Standard Function	FMS Sub-routine used	Number of cells used	Execution time per call	Comments, errors
abs(x)	-	10	0.15 millisec	as described in [2]
entier (x)	-	13	0.2 millisec	as described in [2]
sign (x)	-	10	0.2 millisec	as described in [2]
sqrt(x)	BC ROOT	82	0.3 millisec	calculates $\sqrt{\lvert x \rvert}$ and terminates with a diagnostic if $x < 0$.
cos(x), sin(x)	BC SIN	182	0.4 millisec	rel. error $< 10^{-8}$ except for very large arguments
arctan(x)	BC ATAN	104	0.4 millisec	calculates the principal value, $-\frac{\pi}{2} < \arctan < \frac{\pi}{2}$ rel. error $< 10^{-8}$
exp(x)	BC EXP	94	0.4 millisec	if $x > 88.028$, an overflow message ends execution. If $x < -88.028$, the result is zero. rel. error $< 2^{i-27}$ where i = number of integral bits in x.
tanh(x)	BC TANH	114	0.4 millisec	calculates the hyperbolic tangent with rel. error $<10^{-8}$ except for x near .00034 or .17 where the rel. error may be near 3×10^{-8}

Standard Function	FMS Sub-routine used	Number of cells used	Execution time per call	Comments, errors
ln(x)	BC LOG4	107	0.4 millisec	if $x < 0$, a diagnostic ends execution. rel. error $< 3 \times 10^{-7}$
log(x)	BC LOG 4	107	0.4 millisec	calculates $\log_{10}(x)$ Errors and diagnostic as above.

With the exception of entier and sign they are all real procedures. The parameters may be of type real or integer.

2.2 Timing Procedures

Two timing procedures are available; they are predeclared and should not be declared in the user's program. One is for setting a time limit for the execution, and the other one is for obtaining the time left.

```
procedure settime(x);
```

causes execution to be interrupted x seconds later (unless a new call of settime is executed before that). When execution is interrupted the message OVERTIME TRAP and a stack dump are printed.

```
real procedure get time;
```

causes the amount of time remaining until interruption to be returned as the value of gettime. A call of settime must be executed previously.

The procedures use the automatic "clock" in cell 5 (and converts the contents of it into seconds).

3. Standard Input and Output Procedures

The input-output procedures which are described in this chapter have been designed so that they are procedures in the sense of the ALGOL Report. Although their procedure bodies must necessarily be in code because they deal with input-output, their headings can be written in ALGOL. For this reason procedures having a variable number of parameters have not been included. All procedures described in this chapter are available as predeclared procedures.

1. Input

For input the procedure input with one actual parameter is used. The parameter should be a variable which is to be assigned a value obtained from a data card. The variable may be of any type and can be either a simple variable or a subscripted variable. Numbers and logical values can be punched on the data cards separated by commas. One or two consecutive blanks spaces within a number will be treated as insignificant, but three or more blank spaces can be used in place of a comma for separating numbers. Only columns 1-72 of the card will be read. A number must not be split between two cards. The last number on a card need not be followed by a comma.

Numbers on data cards which are ALGOL numbers (see Section 2.5 of the ALGOL Report) and satisfy the following additional restrictions will be handled properly:

(1) An integer must be less than $2 \uparrow 35$ in absolute value.

(2) A number of type real may not have more than 2 digits in its exponent part.

(3) A decimal number may not have more than ten digits before the decimal point.

(4) A number of type real must be equal to 0 or have absolute value in the range from 10^{-38} to 10^{38}.

(5) Three or more consecutive blanks may not occur within a number.

(6) Either '10' or the letter E can be used as a transliteration of 10.

An ALGOL number may not end with a decimal point. However, standard output routines in local use produce such numbers as punched output. Consequently, such numbers are accepted as data by input with their conventional meanings. If a data item cannot be handled properly by input, the program will be terminated after the printing of an error message.

A number is first read into the computer and then if the type of the number differs from the type of the actual parameter, a conversion is performed before assigning it as a value to the actual parameter. For integers the number read in agrees exactly with the number represented on the data card.

For numbers of type real the decimal number part of the number on the card is truncated after 10 digits and the resulting number converted to floating point form with a relative error which will not exceed 8_{10}-9 .

Logical values may be represented on the data cards by 'TRUE' and 'FALSE' . (The alternate representations TRUE and FALSE are accepted, but their usage is strongly discouraged.)

```
procedure input (a);
comment Assigns to a as value the next number or logical value
        represented on the data cards;
```

2. Simple output

For beginning use the procedures output , line , and page will be sufficient for output. The procedure output has one parameter. The actual parameter can have a value of any one of the three types real, integer, or Boolean. The printer used with the 7094 computer has available 131 printing positions on each line. If the procedure output is used without calling the format setting procedures described in Section 5, then each number or logical value will be printed in a field occupying 20 printing positions. Each call of output will cause an additional number to be printed on the same line as long as

space remains. If insufficient space on the line remains, then a new line will be started automatically. Also, a new line can be started at any time by calling the procedure line which has no parameter. Blank lines can be obtained by additional calls of line . A new page can be started by calling page , a procedure with no parameters. To leave space between two successive fields on a line the procedure spaces with one actual parameter can be called. The actual parameter gives the number of blanks spaces to be left. Thus spaces (10) would cause 10 blanks spaces to be left.

A number printed will be adjusted to the right in the field. If the number printed is of type real, then it will be printed in floating decimal point notation with one digit before the decimal point and seven digits after the decimal point. For example, the number 12.754 will be printed in the form 1.275400E 01 with seven blank spaces before the first digit. A number of type integer will be printed as a decimal integer. If the value of the actual parameter of output is a logical value, then FALSE or TRUE will be printed right-adjusted in the field.

A line of printing which has been begun by one or more calls of the procedure output will always be finished and a new line begun when one of the procedures title, print, or printing format is called or when a dump is made.

```
procedure output (x);
comment Causes the value of x to be printed. A new line is not begun
        if space remains on the previous line;

procedure line;
comment Starts a new line;

procedure nlcr;
comment New line, carriage return. This procedure is completely equi-
        valent to line;
```

```
procedure  page;
comment  Starts a new page;

procedure  spaces (n); value n; integer n;
comment  Causes  n  blank  spaces to be left in the line being printed
         provided  n > 0.  If there are less than  n  spaces remaining
         in the line, the line will be filled with blanks and a new
         line will be started at the left margin.  If  n ≤ 0 , the pro-
         cedure has no effect;
```

Example 1. The following program will produce a table on each line of which an integer will appear with its square root and cube root. Fifty lines will appear on each page. The length of the table is determined by a parameter read in as data.

```
begin integer i, n;
        input (n);
        page;
        for i:=1 step 1 until n do
              begin output (i);
                    output (sqrt (i));
                    output (i ↑ (1/3));
                    line;
                    if i = (i ÷ 50) × 50 then page
              end
end
```

Example 2. Procedures to print a given number of values on a line can be constructed easily. The following declaration can be used to define a procedure for printing three numbers on a line.

```
procedure  print 3(a,b,c);
begin output (a); output (b); output (c); line end;
```

3. Printing alphanumeric information

The procedure text is used to place a text in the line being currently form for output. Thus it can be used to print titles, column headings, and error messages. The procedure title can be used to print a title which is read from data cards.

procedure text(s); string s;
comment Causes the text given by the string s to be placed in the line being printed. If enough space does not remain on the line for the entire text, then the text will be continued on subsequent lines;

procedure title;
comment When this procedure is called, two new data cards are read and the contents of columns 1-72 on the first card and 1-59 on the second card are copied to form a printed line with 131 characters. The remainder of the second card is skipped;

Example 3. In Example 1 a title could be printed at the top of the first page of output by inserting the following four statements after the first call of page.

```
spaces (12);
text ('table␣of␣square␣roots␣and␣cube␣roots');
line; line;
```

Note that the string blank in the hardware representation is represented by a blank space. Thus one writes

```
TEXT('('TABLE OF SQUARE ROOTS AND CUBE ROOTS')')$
```

Example 4. Suppose one wished to print three results, the values of n, x, and y. This could be accomplished by the following statements:

```
text ('n :=');  output (n);  spaces (10);
text ('x :=');  output (x);  spaces (10);
text ('y :=');  output (y);  spaces (10);
```

Example 5. The following program draws a graph.

```
begin integer i;
        page;
        for i:= 0 step 1 until 50 do
                begin spaces(sin(6.283 × i/50) × 40 + 40);
                        text('.'); line
                end
end
```

4. Punch output

The procedure punch when called has the effect that subsequent calls of the standard output procedures will cause numbers to be punched on cards rather than printed. This will be useful if the numbers are to be used as data for some other program. If they are to be read with input, it will be necessary to leave sufficient space so that numbers will be separated by at least three blank spaces. For integers a field of width 15 is sufficient. For a number of type real a field of width 17 is sufficient if the number is punched in floating point form. When punching a card for use with input only the first 72 columns of the card may be used for punching information to be read. To change back to printing after punching, a call of the procedure printer is made. After a call of punch the procedure line has the effect of starting a new card.

```
procedure  punch;
comment  Causes the card punch to be used for output rather
         than the printer.  All 80 columns of the card may be punched;

procedure  printer;
comment  Causes a return to the printer for output.  The line length is
         reset to be 131 print positions;
```

5. More flexible output

The procedures real format , integer format , and boolean format are used to change the formats according to which values of the corresponding types are printed or punched. Each of these procedures sets a format specification which remains in effect until the next call of the same procedures or until the end of the program if there is no later call. Thus, the statement integer format (5) has the effect that all numbers of type integer which are printed by means of output will be printed in fields of width five until the next call of integer format . Note that it is possible to give a new format specification with each cell of output , but usually this will not be necessary.

```
procedure integer format (n); value n; integer n;
comment This procedure has the effect that number of type integer
        printed or punched subsequently by output will appear right-
        adjusted in fields of width n provided 1 ≤ n ≤ 72;

procedure boolean format (n); value n; integer n;
comment logical values printed or punched by output will occupy fields
        of width n positions provided 5 ≤ n ≤ 72. Either TRUE fol-
        lowed by a blank space or FALSE will be printed right-adjusted
        in the field;

procedure real format (fixed n, d); value n,d;
Boolean fixed; integer n,d;
comment Numbers of type real will be printed or punched by output in
        fields of width n with d digits after the decimal point pro-
        vided 2 ≤ n ≤ 72 and 0 ≤ d ≤ 9 . If the parameter fixed has
        the value true, then the numbers will be printed in fixed point
        notation without an exponent. If fixed has the value false, then
        the number will be printed in floating decimal point notation with
        an exponent and with one digit before the decimal point; A num-
        ber of type real having absolute value greater than 2^27-1(1.3x10^8)
        will be printed incorrectly if the parameter FIXED has the value
        true. If real numbers this large are to be printed, then the para-
        meter FIXED should have the value false;
```

The above procedures have restrictions for their parameters of type integer. If an erroneous call is made with an actual parameter having a value outside the permitted range, then a return is made to the standard format with a field of width 20 for values of the corresponding type.

Example 5. If the number to be printed will not fit in the field specified for it, then the sign and most significant digits of the number will be lost. As an example of how this can be avoided, consider the following statements:

```
real format (abs(x) < 1000, 10, if abs(x) < 1000 then 5 else 3);
output (x);
```

which will assure that x can be fitted in a field of width 10.

Example 6. The output given for the example in Section 1.4 of A Guide to ALGOL Programming by D. McCracken could be handled by the following statements:

```
real format (true, 10, 3);
integer format (6);
output (a); output (b); output (c);
output (x 1 real); output (x 1 imag);
output (x 2 real); output (x 2 imag);
output (roots); line
```

6. Character handling procedures

The procedures described in this section can be used in connection with the procedures input , output , and text for input or output of a single character.

```
procedure in character (c); integer c:
comment This procedure assigns to c as value the BCD representation of the
        next character on the data cards. The BCD representation will be an
        integer between 0 and 63 inclusive. If in character is called
        immediately after column 72 of a data card is read by a previous call
        of in character, then the value 58 representing an end of line
        character will be assigned to c . Another call of in character
        will result in the first character on the next data card being read.
        If in character is used after a call of input , the character
        read will be the one immediately after a comma terminating the num-
        ber, the first character after three successive blank spaces ter-
        minating the number, or the first character on the next data card;
```

```
procedure  new input line;
comment  This procedure causes the remainder of the current data card
         to be skipped.  A subsequent call of  input  or  in character
         will read data from the next data card;

procedure  out character (c); value c; integer c;
comment  This procedure places the character whose BCD representation
         is the integer  c  on the current output line.  If the charac-
         ter will not fit on the current line, then a new line is started;

integer procedure character (s); string s;
comment  This procedure assigns to the function designator the BCD repre-
         sentation of the first character in the BC-ALGOL hardware repre-
         sentation of the first basic symbol in  s  after the initial left
         string quote.  The BCD representation is an integer between 0
         and 63;
```

Example 7. The following program reads in arithmetic expressions containing the variables x and y, the binary operators + and ×, and parentheses. Each expression is supposed to be on a separate data card and is terminated by the first blank space on the card. The program transforms these expressions into Polish notation and prints out the resulting Polish strings. The priority scheme used is similar to that currently employed by the preprocessor in the BC-ALGOL system.

```
begin integer c, i, j, n;
      integer array stack [0:50];
      integer procedure priority(c); value c; integer c;
      priority := if c = character('(') then 0
                  else if c = character(')') then 1
                  else if c = character('+') then 2
                  else if c = character('×') then 3 else -1;
start: input(n);
       for j := 1 step 1 until n do
       begin  new input line; line; i := 0;
```

```
next:           in character(c);
                if c = character('x') ∪ c = character('y') then
                begin out character(c); go to next end;
                  if c = character('(') then go to put on stack;
check:            if i > 0 then
                  begin   if priority(c) ≤ priority(stack[i]) then
                          begin   out character(stack[i]); i := i - 1;
                                  go to check
                          end
                  end;
                if c=character(')') then begin i:=i-1; go to next end;
                if priority(c) ≥ 0 then
put on stack:    begin i := i+1;  stack[i] := c; go to next end;
                 line
          end
end
```

4. Comment Control Options

The use of the delimiter comment allows the programmer to add notation to his program for easier understanding. The comment has no effect on the processing of the program except as described in this chapter.

The following comments when inserted in the ALGOL source program will perform the described control functions during the translating (pre-processing) phase of computation of the ALGOL program. Comments may also appear outside of the ALGOL source program. When comments follow the last end, a semicolon must follow the delimiter end before the delimiter comment.

Comment Control Card	Function
Listing Controls:	
comment: page;	The listing of the source program will start on a new page, with a heading specified by the title control card.
comment: title;	The next card in the source is the title to be put on the top of a page when the page control card is used.
comment: stop print source;	This is used to delete printing of the source program. The comment controls, name list, page, and title are disabled when this is used.
comment: start print source;	This is used to resume printing of the source program.
comment: name list;	The translator will print out the current name list at the point where the control card is located.
comment: print string;	The Syllable String (see chapter 8) will be printed after the source program has been completely translated or at a terminating diagnostic.
comment: loader list;	This is used to obtain the storage map, entry points, transfer vector, and a listing of the relocated object program for each code procedure loaded by the Pre-processor Loader.
comment: sts list;	This is used when operating in the STS system to speed the printing of source programs by the 1050 terminal. The identification field is deleted.

Comment Control Card	Function
Execution Time Options	
comment: no array tests;	This is used to delete the testing for correctness of bounds and dimensions in the pre-processor and in the processor.
comment: skip array print;	This is used to delete the printing of arrays when a stack dump is made by the processor.
comment: variable number of parameters;	This is used to allow the next procedure declared to be used with any number of parameters.
comment: no parameter tests;	When this is used the pre-processor and the processor do not make any tests as to correctness of the number of parameters in any procedure statement.
comment: common;	This is used to allow externally coded procedures to use COMMON. This has the disadvantage that the source program's name list is destroyed and cannot be used by the processor with the stack dumps.
comment: dump;	This will cause the processor to give a stack dump at the point in the program where the control card is placed. This control card may not be put in the blockhead, or before the first begin.
comment: dump number := c;	This is used to obtain a stack dump c locations past the accumulator pointer at the time the dump is requested (where c is a constant).
Semicolon Trace Option (see Appendix 5)	
comment: semicolon trace; or comment: semicolon trace := c;	This informs the system that the semicolon trace will be used.
comment: on semicolon trace;	Turns on the semicolon trace.
comment: off semicolon trace;	Turns off the semicolon trace.
Library Tape Options (see Chapter 10)	
comment: library A5;	This will assign unit A5 as a library tape.
comment: library B5;	This will assign unit B5 as a library tape.
comment: <filename>;	This will insert the file named <filename> at this point in the source program.

Comment Control Card	Function
Input Tape Options (see Chapter 10)	
comment: scratch tape;	Causes cards to be read from unit A4 rather than from the source card deck.
comment: source tape;	Resets the system to resume reading cards from the source card deck.

5. Error Messages

The system performs a nearly complete syntactical check of the source program. Most of this is performed by the translator (pre-processor) and the error messages from this phase will appear in the listing of the program immediately after the statements causing the messages. Some syntactical and some semantic check is performed at execution time (by the processor) and an error messages from this phase will appear among the output from the program.

During translation, the translator will continue working on the source program until 8 different error types are recognized, but the execution phase will never be entered when an error is found.

An error found at execution time results in a message and a dump, and further execution is deleted.

5.1 Translator Messages

Below follows a list of the error messages from the translator. Many of them are followed by a reference to [2], not shown here. In most cases, a short explanation or an example of an ALGOL text which could cause that particular message is given.

5.1.1 Excess of Capacity

** THE MAXIMUM NUMBER OF DECLARATIONS HAS BEEN EXCEEDED IN THE PRESENT BLOCK.

The maximum number allowed is 63.

** THE MAXIMUM NUMBER OF FOR LIST ELEMENTS IN A FOR LIST HAS BEEN EXCEEDED.

The maximum number allowed in 100.

** OPERATOR STACK SIZE EXCEEDED.

An expression requires stacking of more than 100 operators (or parentheses) during translation.

** LENGTH OF NAME LIST EXCEEDED.

The maximum number of names allowed at once is 330.

5.1.2 Errors in Bracket-like Structures

** THE END OF YOUR DECK WAS REACHED WHILE PROCESSING A STRING. THE NUMBER OF MISSING RIGHT STRING QUOTES IS <integer>. THIS STRING BEGAN AT WORD ADDRESS <integer>.

Ex: begin text ('UHA) end

** THE END OF YOUR DECK HAS BEEN REACHED. THERE ARE <integer> "'BEGIN'S" FOR WHICH NO "'END'S" HAVE BEEN FOUND.

Ex: begin; begin; end

** A PROGRAM MUST BE A BLOCK OR A COMPOUND STATEMENT.

Ex: A program not enclosed between begin and end .

** THERE ARE <integer> MORE ")" THAN "(" IN THE LAST STATEMENT.

Ex: a := (b-c) d + f) ;

** THERE ARE <integer> MORE "(" THAN ")" IN THE LAST STATEMENT.

** THERE ARE <integer> MORE). THAN .(IN THE LAST STATEMENT.

Ex: a := M (i,j] + 2 ;

** THERE ARE <integer> MORE .(THAN). IN THE LAST STATEMENT.

5.1.3 Other Errors in Delimiter Structure

** THE NUMBER OF "'IF'S" AND "THEN'S" DOES NOT BALANCE.

Ex: if a > b do x := 0 ;

** AN "'ELSE'" OCCURS IN A CONDITIONAL STATEMENT WHICH CONTAINS NO "'THEN'".

Ex: if (a > b) then a := 0 else b := 1 ;

** "'ELSE'" DOES NOT OCCUR IN A CONDITIONAL STATEMENT. POSSIBLY THERE IS AN UNWANTED "$" AFTER THEN "'THEN'".

Ex: if a > b then a := 0; else b := 1 ;

** THE CURRENT "'DO'" IS NOT IN A FOR STATEMENT.

Ex: for i := 1; do s := s + b ;

** A "'DO'" IS IN A FOR STATEMENT BUT HAS NOT BEEN PRECEDED BY AN "=".

Ex: for a, b, c do s := s + t ;

** THE SEPARATOR "'WHILE'" IS NOT IN A FOR STATEMENT.

Ex: for i := i + 1 ; while b < 0 do s := s + t ;

** THE SEPARATOR "'WHILE'" OR THE SEPARATOR "'UNTIL'" IS NOT IN A FOR STATEMENT.

```
Ex:  ; i := 1 step 1 until 10 do s := s - N[i] ;
```

** AN "'EQL'" HAS BEEN USED AFTER A "'FOR'" WHERE AN "=" WAS EXPECTED.

```
Ex:  for i = 1 do a := 0 ;
```

** AN "=" WAS EXPECTED AFTER THE LAST "'FOR'" PERHAPS AN "'EQL'" WAS USED.

** THE SEPARATOR "'UNTIL'" IS NOT PRECEDED BY A "'STEP'".

```
Ex:  for i := i + 1 until 10 do t := t   M[i] ;
```

** THE SEPARATOR "'STEP'" IS NOT IN A FOR STATEMENT.

```
Ex:  for i := 1; step 1 until 10 do s := s + M[i] ;
```

** ILLEGAL SYMBOL IN LAST FOR STATEMENT.

```
Ex:  for i := 1 then x := 0 ;
```

** ILLEGAL DELIMITER USED IS "'<string>'" ... POSSIBLY A MISSING ' OR MIS-SPELLED DELIMITER

```
Ex:  begin ream a, b ;
```

** THE LAST COLON IS NOT PRECEDED BY A LABEL.

```
Ex:   a := b ;
     : x := 0 ;
```

** A "," HAS OCCURRED IN AN ILLEGAL LOCATION. A "$" WAS EXPECTED HERE.

```
Ex:  A := B + C, B := C + D ;
```

** AN "=" HAS BEEN USED WHERE AN "'EQL'" WAS EXPECTED.

```
Ex:  if  a := b then s := 0 ;
```

** ILLEGAL SYMBOL ORDER FOLLOWING "'BEGIN'".

```
Ex:  begin ,
```

** ERROR IN SYNTAX IN THE LAST STATEMENT.

This may occur in different contexts.

** INCORRECT LEFT PART OF ASSIGNMENT TO FUNCTION DESIGNATOR.

```
Ex:  real procedure x(a);
     x(a) := a ↑ 2;
```

5.1.4 Operand Errors

** AN INTEGER LARGER THAN 2**35-1 IS NOT ALLOWED.

Ex: x := 34359738368 ;

** A DECIMAL POINT MUST BE FOLLOWED BY A DIGIT.

Ex: y := 4. ;

** A DECIMAL POINT OR LEFT BRACKET OCCURS ILLEGALLY.

Ex: z := 45 [B] ;

** AN EXPONENT PART IS IMPROPERLY FORMED.

Ex: a := 5_{10}N ;

** A NUMBER OUTSIDE THE RANGE OF FLOATING POINT NUMBERS IS NOT ALLOWED.

Ex: b := 5_{10}200 ;

** A LETTER OF A "(" CANNOT LEGALLY FOLLOW A NUMBER.
IMPLICIT MULTIPLICATION IS NOT ALLOWED.

Ex: c := 5A ;

Ex: d := 16(B) ;

** SYSTEM OR MACHINE ERROR. PLEASE REPORT THIS TROUBLE.

See your instructor or a Computer Center consultant.

5.1.5 Errors in Declarations or Specifications

** THE PRESENT IDENTIFIERS <identifier> IS DOUBLY DEFINED. THE ALGOL PREPROCESSOR ONLY RECOGNIZES THE FIRST 18 SIGNIFICANT CHARACTERS.

```
Ex: begin real a,b, c;
    a: b := 0;
```

```
Ex: procedure P(A, B, C, A) ;
```

** DECLARATIONS MUST OCCUR IN A BLOCKHEAD, NOT IN THE MIDDLE OF STATEMENTS.

```
Ex: begin real a; a := 0 ;
    procedure P;
```

** <identifier> IS AN UNDEFINED NAME.

```
Ex: begin real a1; a2 := 0 ;
```

** THERE IS AN ILLEGAL DELIMITER IN THE ARRAY DECLARATION BOUNDED PAIR LIST FOLLOWING ".(" OR "," .

Ex: begin array A[1 = 10];

** THERE IS AN ILLEGAL DELIMITER IN THE ARRAY DECLARATION BOUND PAIR LIST FOLLOWING ".." .

Ex: begin array A[1:,1:10] ;

** AN IDENTIFIER HAS BEEN CALLED BY VALUE BUT IS NOT IN THE PARAMETER LIST.

Ex: procedure P(a,b); value ab ;

** AN IDENTIFIER IN THE VALUE LIST HAS NOT BEEN SPECIFIED.

Ex: procedure P(a,b); value a; real b; b := 2 x a ;

** ILLEGAL SYMBOL IN VALUE LIST.

Ex: value a[1:10] ;

** AN IDENTIFIER IN THE SPECIFICATION PART IS NOT IN THE FORMAL PARAMETER LIST. THE CURRENT IDENTIFIER IS <identifier>.

Ex: procedure P(a,b); real ab;

** AN IDENTIFIER IN THE SPECIFICATION PART IS A GLOBAL VARIABLE. THE CURRENT IDENTIFIER IS <identifier>.

Ex: procedure P(a,b); real a, P;

** AN IDENTIFIER IN THE SPECIFICATION PART HAS OCCURRED MORE THAN ONCE. THE CURRENT IDENTIFIER IS <identifier>.

Ex: procedure P(a,b); real a,b; Boolean a;

** A .(CANNOT OCCUR IN THE SPECIFICATION PART.

Ex: procedure P(a,b); array a [1:10] ;

** A DELIMITER OTHER THAN "," OR "$" HAS OCCURRED IN THE SPECIFICATION PART.

Ex: procedure P(a,b); real a + b ;

** AN ACTUAL PARAMETER LIST OF A PROCEDURE STATEMENT DOES NOT HAVE THE SAME NUMBER OF ENTRIES AS THE FORMAT PARAMETER LIST OF THE PROCEDURE HEADING. THE PROCEDURE WAS DECLARED WITH <integer> PARAMETERS, AND USED WITH <integer> PARAMETERS.

Ex: procedure P(a,b); a := b ;
P(x + y) ;

** THERE IS AN ERROR IN SYNTAX IN THE BLOCK HEAD.

5.1.6 Type Errors

** A FORMAL PARAMETER CORRESPONDING TO A PROCEDURE IDENTIFIER MAY NOT BE USED AS A LEFT PART. THE CURRENT IDENTIFIER IS <identifier>.

```
Ex:  procedure P(A); procedure A;
     begin A := 1 ;
```

** A PROCEDURE IDENTIFIER IS NOT FOLLOWED BY "(".

```
Ex:  x := sqrt x (a-b) ;
```

** THE IDENTIFIER PRECEDING THE CURRENT "(" HAS NOT BEEN SPECIFIED OR DECLARED AS A PROCEDURE IDENTIFIER. THE CURRENT IDENTIFIER IS <identifier>.

```
Ex:  real a,b,c; a := b(c + 1) ;
```

** THE IDENTIFIER PRECEDING THE CURRENT ".(" HAS NOT BEEN DECLARED OR SPECIFIED AS AN ARRAY OR SWITCH IDENTIFIER.

```
Ex:  begin real r ; array rr[1:F]; r[1] := 0 ;
```

** AN ARRAY OR SWITCH IDENTIFIER IS NOT FOLLOWED BY ".(" . THE CURRENT IDENTIFIER IS <identifier>.

```
Ex:  begin array M[1:110];
       M := 0 ;
```

** THE NUMBER OF SUBSCRIPTS OF A SUBSCRIPT LIST IS NOT EQUAL TO THE DIMENSION OF THE ARRAY AS DECLARED. IT WAS DECLARED AS <integer> DIMENSIONAL BUT USED WITH <integer> SUBSCRIPTS.

```
Ex:  begin array A [1:2,1:5];
                  A[1,3,5]; = 0;
```

** INTEGER LABELS ARE NOT ALLOWED IN BC-ALGOL.

```
Ex:  502 : X := 0 ;
```

** A LABEL IS BEING USED AS THE CONTROL VARIABLE IN A FOR STATEMENT.

```
Ex:  for L := 1,2,3 do s := s + f ;
     L : a := 0 ;
```

5.1.7 Pre-processor Loader Diagnostics

** PROCEDURE <identifier> WILL OVERFLOW COMMON AND INTERFERE WITH THE PROCESSOR.

Ex: The code procedure used too much common storage.

** PROCEDURE <identifier> REQUIRES MORE SPACE THAN ALLOTTED BY PROGRAM CARD, POSSIBLY A MISSING PROGRAM CARD.

** LOADING OF CODE PROCEDURE <identifier> WILL EXCEED SYLLABLE STRING SIZE.

Ex: The code procedure contains too many instructions and will not fit into the syllable string.

** CODE PROCEDURE NOT FOUND <identifier>.

** LOWER LEVEL ROUTINES NOT FOUND <identifier>.

** CARD <integer> DOES NOT HAVE A 12 OR AN 11 PUNCH IN CARD COLUMN 1.

Ex: Trying to load a card which is not a binary card.

** THE FIRST BINARY DECK DOES NOT HAVE A PROGRAM CARD.

** PROCEDURE <identifier> CALLS FOR COMMON STORAGE WHICH IS UNDEFINED.

Ex: A code procedure using common was loaded without using the control card comment: common;

** CHECKSUM ERROR ON CARD <integer>.

** CHECKSUM DELETED ON CARD <integer>.

5.2 Interpreter Diagnostics

The following diagnostics are detected and listed on the program output during the interpretive phase of processing. Any of the diagnostics will terminate the program.

5.2.1 Excess of Capacity

** THE ALGOL PROCESSOR STACK SIZE HAS BEEN EXCEEDED.

** AN OVERFLOW OCCURRED IN PERFORMING INTEGER ARITHMETIC. THE RESULT OF AN OPERATION WHEN IT IS OF TYPE INTEGER MUST BE LESS THAN 2**35 IN ABSOLUTE VALUE.

** THE ALGOL PROCESSOR PROGRAM ADDRESS STACK SIZE HAS BEEN EXCEEDED.

** AN OVERFLOW OCCURRED IN AN EXPONENTIATION OPERATION OF THE FORM A**B WHERE A = <number>, B = <number> .

** AN OVERFLOW OCCURRED IN PERFORMING REAL ARITHMETIC. EITHER THE RESULT OF AN OPERATION OR A NUMBER OBTAINED AT AN INTERMEDIATE STEP IS GREATER THAN OR EQUAL TO 2**127 IN ABSOLUTE VALUE.

5.2.2 Errors in bracket-like structures

** THE UPPER SUBSCRIPT BOUND OF BOUND PAIR NUMBER <integer> OF AN ARRAY DECLARATION IS SMALLER THAN THAT OF THE CORRESPONDING LOWER BOUND. SEE ALGOL 60 REPORT SECTION 5.2.4.3.

5.2.3 Errors in Input-Output

** ERROR IN A CALL OF AN INPUT OR OUTPUT PROCEDURE WHICH USES FORMATS. NO FORMAT CORRESPONDS TO THE FIRST PARAMETER.

** ERROR MESSAGE BY EXEM, THE MONITOR'S ERROR SUBROUTINE.

** ERROR IN INPUT. AN INTEGER REPRESENTED ON A DATA CARD IS GREATER THAN 2**35-1 IN ABSOLUTE VALUE. THE DATA CARD IS ...

** ERROR IN INPUT. A NUMBER OF TYPE 'REAL' REPRESENTED ON A DATA CARD IS TOO LARGE IN ABSOLUTE VALUE. THE DATA CARD IS ...

** ERROR IN INPUT. THE INCORRECT DATA CARD IS ...

** A CALL TO SAVE HAS BEEN MADE WITHOUT A CORRESPONDING CALL TO PRINTING FORMAT.

** A CALL TO READING INTEGER, READING REAL, OR READING BOOLEAN HAS BEEN MADE WITHOUT A CORRESPONDING CALL TO READING FORMAT.

5.2.4 Type Errors

** THE VALUE OF THE CONTROLLED VARIABLE OF A FOR STATEMENT IS UNDEFINED DUE TO THE EXHAUSTION OF A FOR LIST ELEMENT. IT HAS NOT BEEN REINITIALIZED BEFORE ITS USED IN AN EXPRESSION. SEE ALGOL 60 REPORT SECTIONS 4.6.5 AND 4.6.

```
Ex:  for i := 1 step 1 until 100 do A := 1; b := A[i];
```

** THE ALGOL PROCESSOR DID NOT FIND A STACK ADDRESS TO STORE THE VALUE OF THE ARITHMETIC OR BOOLEAN EXPRESSION. A VALUE OR INCORRECT QUANTITY HAS BEEN USED AS A LEFT PART VARIABLE. SEE ALGOL 60 REPORT SECTION 4.2, 2.7, AND 2.8.

```
Ex:  B + 1 := C;
```

** A SIMPLE VARIABLE HAS NOT BEEN INITIALIZED BEFORE IT IS USED IN AN EXPRESSION. SEE ALGOL 60 REPORT SECTIONS 3.3.3, 3.4.3, 2.7, 3.1.1, AND 3.1.3.

```
Ex:  real C; A := C ;
```

** THE ALGOL PROCESSOR HAS EXECUTED AN ILLEGAL SYLLABLE. ILLEGAL SYLLABLE = <letter> (OCT = <integer>).

** A FORMAL PARAMETER WHICH IS CALLED BY VALUE CORRESPONDS TO A SWITCH DESIGNATOR WITH AN UNDEFINED VALUE. SEE ALGOL 60 report SECTION 4.3.5.

** AN ILLEGAL ACTUAL PARAMETER WAS USED IN A PROCEDURE STATEMENT CALLING A CODE PROCEDURE.

** BOOLEAN CONVERSION REQUESTED IN <identifier>.

5.2.5 Arithmetic Errors

** THE RESULT OF AN EXPONENTIATION OPERATION OF THE FORM A**B IS UNDEFINED WHERE A = <number>, B = <number>. SEE ALGOL 60 REPORT SECTION 3.3.4.3.

** DIVISION BY 0 IS UNDEFINED.

5.2.6 System Diagnostics

** CODE PROCEDURE DID NOT EXIT CORRECTLY.

** SUBROUTINE <identifier> WAS CALLED WITH <integer> ARGUMENTS.

```
Ex:  Erroneous call of standard functions such as a := ln (b,c) ;
```

** OVERTIME TRAP.

** SUBROUTINE <identifier> NOT IN CODE LIST.

6. Predefined Code Procedures

6.1 Code Procedures

Code procedures are allowed in ALGOL 60 as stated in section 4.7.8 of the ALGOL 60 report. The predefined code procedures described in this chapter are to allow greater flexibility in using BC-ALGOL.

Note: Each of these procedures, if used, must be declared in the first block of the program.

The parameters must be the same as the expected type; outside of this restriction, they may be any general ALGOL parameter.

These procedure identifiers are not reserved for use unless they are specifically declared. If the code procedure of a given name is not declared, that identifier may be used in any manner desired.

6.2 Input-Output Code Procedures

The BC-ALGOL system contains some predefined code procedures for use along with the standard input-output procedures described in Chapter 3. For each of these procedures the declaration (which should appear in the heading of the first block of the user's program if he wishes to use the code procedure) is given below together with a comment explaining the procedure. Many of these procedures occur naturally in pairs, with one for input and one for output.

```
procedure  set input line length(n); value n; integer n; code;
comment    This procedure sets the length of subsequent input lines to
           be  n  characters provided  1 ≤ n ≤ 132.  All characters after
           the first  n  are ignored.  A call of this procedure automa-
           tically causes a new line to be started;

procedure  set output line length(n); value n; integer n; code;
comment    This procedure sets the length of subsequent output lines to
           be  n  characters.  It does not cause a new line to be started.
           Thus the length of an output line can be adjusted depending on
           the contents even after the line is partially constructed;
```

integer procedure read position; code;

comment The function designator is assigned as value the number of the next position to be read on the current input line. The first position on a line is numbered 1;

integer procedure print position; code;

comment The function designator is assigned as value the number of the next position on the output line for which a character has not been constructed. The first position on a line is numbered 1;

procedure in line (a); integer array a; code;

comment A new input line is always started when this procedure is called. Then, if possible, an entire line of input is read into the one-dimensional array a with each element of a containing the BCD representation of one character.

The representation of the first character of the line is placed in the element of a with least subscript, the second in the element of next higher subscript, etc. If the array contains more elements than there are position on the input line, then the number 58^{*}_{10} is stored in the element immediately following the one containing the representation of the last character of the line. The remaining elements of the array, if there are any, are not assigned values. If the array is not long enough to hold the entire line, then only enough characters are read to fill the array;

procedure out line (a); integer array a; code;

comment A new output line is started. This procedure is essentially the inverse of in line . It causes output of the line of characters whose BCD representations are in a . Characters are taken from a until the end-of-line character corresponding to the integer 58^{*}_{10} is reached or until the whole array a is used. Additional characters may be placed on the end of the line by using out character , text , or output if space remains;

---

* $58_{10} = 72_8$ = standard end-of-line character.

procedure octal format (n); value n; integer n; code;

comment This procedure has the effect that numbers of type integer printed or punched subsequently by output will appear right-adjusted in fields of width n in octal notation provided $1 \leq n \leq 72$;

integer procedure bcd(s); string s; code;

comment This procedure assigns to the function designator the BCD representation of the first 6 characters in the hardware representation of the open string obtained by stripping off the outer string quotes from s . If there are less than 6 characters in the open string, then enough blanks are added at the end to make up a full word of 6 BCD characters.

6.3 Tape Selection Procedures

procedure in unit (n); value n; integer n; code;

comment This procedure causes the logical tape unit from which data are to be read by means of input , in character , etc. to be changed to the unit with number n . This change is also effective for the input procedures which use FORTRAN formats. A call of in unit causes a new input line to be started;

procedure out unit (n); integer n; code;

comment This procedure causes the logical tape unit for output by means of output , text , out character , etc. to be the unit with number n . This procedure has no effect upon whether the output will be suitable for punching on cards or listing on a line printer. Thus it should be preceded by a call of punch or printer which will determine whether a carriage control character is included on each record or not. The change of output unit is also effective for the output procedures which use FORTRAN formats. A new output line is started automatically;

6.4 Tape Handling Procedures

These procedures will handle all tapes except those used by the monitor; these are the monit input tape A2, logical -1; the monitor punch tape B4, logical -2; the monitor print tape A3, logical -3. In the following descriptions, the parameter log is the logical number of the tape unit, and count, where used, determines the number of records or files to be skipped.

```
procedure  bksprecord (log, count); integer log, count; code;
comment    The tape specified by log is backspaced count records;

procedure  fdsprecord (log, count); integer log, count; code;
comment    The tape specified by log is forward spaced count records.

NOTE: In bksprecord and fdsprecord , and end of file gap and mark is
      considered to be the same as an end of record gap;

procedure  bkspfile (log, count); integer log, count; code;
comment    The tape specified by log is backspaced count files;

procedure  fdspfile (log, count); integer log, count; code;
comment    The tape specified by log is forward spaced count files;

procedure  endfile (log); integer log; code;
comment    The tape specified by log has an end of file written on it;

procedure  rewind (log); integer log; code;
comment    The tape specified by log is rewound;
```

6.5 Rescan Procedure

procedure rescan; code;

comment rescan causes the monitor to resume reading control cards at the current location on the input tape (A2). Note that inunit cannot be used to assign this function to a unit other than A2;

Example Both of the ALGOL programs in the following job will execute because of the call to rescan:

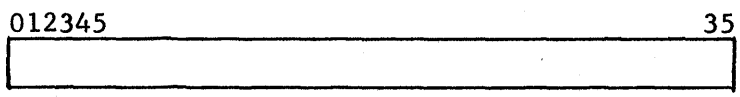
```
$JOB...
$FMS ALGOL
begin
     procedure rescan; code;
     integer i,j,k; real x;
     i := 1; j := k := 2;
     x := i/k; text ('program one');
     rescan
end
$
*USE(ALGOL)
*DATA
begin
     real y;
     y := 5; output (y);
     text ('program two')
end
$
$EOF
```

5 spaces after the *

6.6 Bit Manipulation Procedures

procedure getptw (full word, low bit number, high bit number, output); code;

comment getptw (get partword) extracts the binary bits between the word including low bit number and high bit number. The type of all parameters must be integer. Bits are numbered from 0 to 35 from left to right;



To take the high order octal character, which takes 3 bits, the call would be

```
getptw (full, 0, 2, out);
```

To take the high order alphanumeric (BCD) character, the call would be

```
getptw (full, 0, 5, syllable);
```

Output is type integer with the value right-adjusted and the higher order bits zeroed.

procedure stoptw (input, full word, low bit number, high bit number); code;
comment stoptw (store partword) stores the rightmost high-low + 1 bits of input into bits between and including low bit number and high bit number of full word. The type of parameters must be integer;

To store the character "S" (octal 62) into the third character of full word the call would be

```
stoptw (oct 62, full word, 12, 17);
```

procedure type shift (input, output); code;
comment type shift takes an input of either type integer or type real and changes the type to real or integer, respectively, without changing the representation (that is, without floating or fixing the input);

To take the real number 1.0 and store it in the integer array stack, the call would be

```
type shift (1.0, stack [1]);
```

The integer number stored in stack would be (in octal) 201400000000, which is the octal number which represents the floating point 1.0 in the 7094.

procedure convrt (number, exponent, output); code;
comment convrt takes the integers, number and exponent, and forms the indicated real representation, which is returned through output an integer. (Exponent is taken as a power of 10);

procedure shift (argument 1, argument 2, shift number, output); code;
comment shift shifts the "double precision" integer words argument 1 and argument 2 'shift number' bits to the left. The high order 'shift number' bits from argument 1 are returned to output (type integer) right-adjusted with the extra bits zeroed;

6.7 Logical Arithmetic

```
procedure compl (in, out); integer in, out; code;
comment    comp 1 takes the 1's complement of in and returns it in the
           variable out.  This is a bit-wise complementing procedure;
           i.e., each bit of the word is complemented;

procedure  comp2 (in, out); integer in, out; code;
comment    comp 2 takes the 2's complement of the parameter in and returns
           it in the parameter out.  This is like the 1's complement with
           an additional 1 added in bit number 35;

procedure  and (word 1, word 2, out); integer word 1, word 2, out; code;
comment    and takes the Boolean product of word 1 and word 2 and returns
           the result in out.  This is a bit-wise Boolean product; each
           corresponding bit of the two words is operated on by these rules:
           if both are 1's, then a 1 is stored in that bit in out; otherwise,
           a 0 is stored there;

procedure  or (word 1, word 2, out); integer word 1, word 2, out; code;
comment    or performs a Boolean addition of word 1 and word 2 and returns
           the result in out.  This is again a bit by bit operation with
           the following rule:  if either bit is a 1, then a 1 is stored
           in out; otherwise, a 0 is stored in out;
```

6.8 Random Number Generator

Two random number generators are available as predefined procedures. The first, random, is a real procedure and generates floating point numbers between 0 and 1 with a flat probability distribution. The second, lehmer, is an integer procedure and generates integers between 1 and $2^{35} - 1$. The procedure lehmer has the additional property that the bit pattern in the word containing the generated integer is also random.

If used, they must be declared in the outermost block of the program as:

```
real procedure random; code;
```

and/or

```
integer procedure lehmer; code;
```

Of course the word "random" is used here in the sense of pseudo-random. The sequences generated have extremely long periods -- random's is 2^{33}, and that of lehmer is slightly less -- so large that a user would generate only a small fraction of the possible numbers in any problem.

The numbers generated by lehmer are those relatively prime to the modulus $2^{35} - 1$. They are obtained as power residues: $X_{n+1} + X_n * K (\text{mod } 2^{35} - 1)$. K and X_0 are of course also relatively prime to $2^{35} - 1$.

The numbers generated by random are obtained by considering the integers obtained from $X_{n+1} = X_n * K \ (\text{mod } 2^{35})$ as binary fractions, i.e., with the binary point at the extreme left. Here the X_n are odd integers and K is of the form $8 * F \pm 3$ (to ensure maximum period).

Additional procedures are available to change the constants K and X_0 in random:

```
procedure start random (N); integer N; code;
```

This procedure assigns to X_0 the value of N. Hence following calls of random create a sequence of numbers with N as "base".

```
procedure lodrandom (K); integer K; code;
```

This procedure replaces the multiplier in random with the value of K.

```
procedure getrandom (N); integer N; code;
```

get random assigns to N the value of the last integer generated within random.

Associated with lehmer are similar procedures, startlehmer, getlehmer, and lodlehmer which can be declared similarly. If the additional procedures are not used, they need not be declared.

It is recommended that if the multiplier K is changed in either routine, it be of the order of the square root of the modulus, i.e., $2^{17} < K < 2^{18}$. The suitability of either generator to a particular problem should be tested by the user.

6.9 Examples of ALGOL procedures using predefined code procedures

GET CHARACTER

This procedure puts the next BCD character on the input cards into the parameter char , and keeps in filled with new characters. The code procedures used are shift , and bcd procedure incard reads a card.

```
procedure get character (char);
     begin integer temp, blank, nothing; own integer cc, wc;
          own integer array inbuffer [1..12];
          blank := bcd ('␣␣␣␣␣␣');
          shift (in, inbuffer[wc], 6, temp);
          shift (temp, blank, 30, nothing);
          char := temp;
          if cc = 6 then
               begin cc := 1;
                    if wc = 12 then
                         begin wc := 1; incard (inbuffer) end
                    else wc := wc + 1
               end
          else cc := cc + 1
     end;
```

REAL NUMBER ROUTINE

This procedure takes numbers from cards (in BCD representation) and returns them through out in real representation, but integer mode. The actual parameter replacing out could be an element of an integer array. The code procedures used are convrt and bcd , which are to be declared in block 1.

A non-local integer array called alpha list is used which holds a code number for each type of BCD symbol (for example, 1 for number, 2 for letters, etc.).

```
procedure real number (out); integer out;
     begin integer char, number, exponent, add;
          add := exponent := number := 0;
          return 1:
               get character (char);
               if char = bcd ('␣') then go to return 1;
               if alpha list [char] ≠ number code then go to end;
               if char = bcd ('.') then
                    begin add := -1; go to return 1;
                    end;
               number := number × 10 + char;
               exponent := exponent + add;
               go to return 1;
          end;
               convrt (number, exponent, out);
     end;
```

REAL MULTIPLY

The following routine performs a "stack" type multiply operation. This method is similar to that used in the BC-ALGOL processor.

```
procedure real mult (accumulator pointer, stack);
     begin real real 1, real 2;
          type shift (stack [accumulator pointer], real 1);
          type shift (stack [accumulator pointer-1], real 2);
          type shift (real 1 x real 2, stack [accumulator pointer-1]);
          accumulator pointer := accumulator pointer-1;
     end;
```

7. Implementation and Use of Code Procedures

7.1 Introduction

The advantages of using Code Procedures in the BC-ALGOL System are (a) speed of execution, and (b) availability of machine code.

A major problem in using Code Procedures is the transference of parameters from the ALGOL Proceeor's Stack to the Code Procedure. There are six routines to facilitate the transferring of parameters. These are discussed in Section 2. Two methods of returning from a Code Procedure to the ALGOL processor are discussed in Section 3. Routines to fix and float numbers are discussed in Section 4. Examples of the use of the routines discussed in Sections 2, 3, and 4, are found in Section 5. Two methods of linking a Code Procedure with the ALGOL source program are discussed in Sections 6, 7, and 8. The method discussed in Section 8 is a new method. It utilizes a loader in the ALGOL preprocessor. A partial library is provided but a double transfer vector must be used.

It is assumed that the reader is familiar with the Syllable String[1], the Processor's Stack[1], and FAP.

7.2 Transferring of Parameters

The six routines to facilitate the transference of parameters are:

7.2.1 INTERP

The call of INTERP from a code procedure causes the ALGOL Processor to execute a syllable string that is in the Code Procedure at location -1, 4. Associated with INTERP is the Escape Syllable, 35_8, which when executed by the processor causes:

---

[1] See chapters 8 and 9.

a. The top of stack[2], i.e., STACK [AP] and STACK [AP + 1] to be put in the MQ and AC[3], respectively;

b. The ALGOL processor to return control to the Code Procedure by transferring to the word immediately preceding the word containing the Escape Syllable.

7.2.2 LOADST

The call of LOADST will place the MQ[4] and AC on the top of the stack at STACK [AP + 2] and STACK [AP + 3] and set AP := AP + 2.

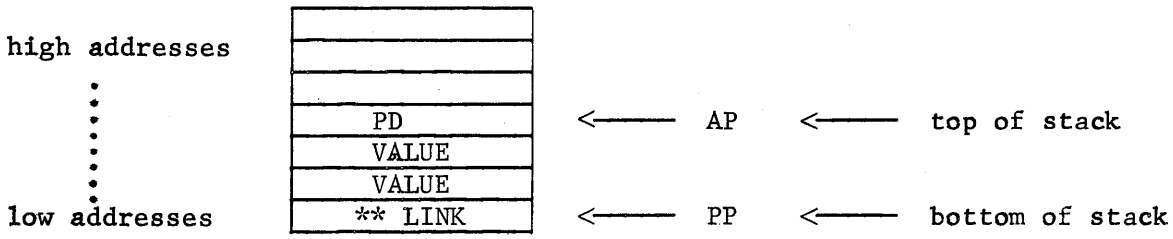
7.2.3 FUNCTN

The transfer to FUNCTN places the AC and MQ[4] in the last Program Description (STACK [PP-1] and STACK [PP-2], respectively) generated by the ALGOL Processor and returns control to the ALGOL Processor.

---

2 In this report STACK is considered as an add on stack, i.e., when an entry is put on the top of STACK, it is placed in the first empty location up from the bottom of STACK. Note that this is the reverse of the order in the Stack dump (see Chapter 9).

example:



3 The following notations are used in this chapter:

Vn	=	The value of the variable N
SAn	=	The Stack Address of N
PAn	=	The Program Address of N
PD	=	Program Descriptor
PL	=	Procedure Link
WSP	=	Working Space Pointer
AP	=	Accumulator Pointer
PP	=	Parameter Pointer
AC	=	7090-7094 Accumulator
MQ	=	7090-7094 Multiplier-Quotient

4 When storing a value in stack, the information word must be all zeros except for the type (the tag).

7.2.4 FETCH

By using INTERP only the top value in STACK may be brought to a Code Procedure. The routine FETCH will bring a value from any location in Stack to a Code Procedure. FETCH is used with one full word integer parameter, for example, NUMBER. The call FETCH(NUMBER) will bring STACK [AP-NUMBER] and STACK [AP=1-NUMBER] to the MQ and AC, respectively.

7.2.5 REPLAC

By using LOADST (or FUNCTN) a Code Procedure places a value only on the top of Stack (or in the last Program Descriptor). The routine REPLAC is used with one full word integer parameter, e.g., NUMBER. The call REPLACE(NUMBER) places the MQ and AC into STACK [AP-NUMBER] and STACK [AP+1-NUMBER], respectively.

7.2.6 GETADD

The call of GETADD will return the addresses of the ALGOL Processor's variables:

STACK
ACCUMULATOR POINTER
PARAMETER POINTER
ARGUMENTS
SYLLABLE STRING
WORD ADDRESS
SYLLABLE COUNT

in locations:

1,4
2,4
3,4
4,4
5,4
6,4
7,4

respectively.

7.3 Returning from a Code Procedure

There are two logically equivalent methods to return from a Code Procedure to the ALGOL Processor:[3]

a. Transfer to $RETURN.

b. Use INTERP to have the ALGOL Processor execute a period ($=33_8$).

7.4 Routines to Fix and Float Numbers

The following routines, which must be used via INTERP, will fix or float (if it is not already integer or real) the value on the top of STACK, i.e., STACK [AP+1], and leave it on the top of STACK.

7.4.1 To fix the value on the top of Stack, one must cause the ALGOL Processor, via INTERP, to execute the syllables 5715_8.

7.4.2 To float the value on the top of Stack, one must cause the ALGOL Processor, via INTERP, to execute the syllables 5716_8.

7.5 Examples

Consider two logically equivalent Code Procedures for computing the square of an integer number:

7.5.1 Example 1. Use of INTERP and FUNCTN

ALGOL Source Program

```
begin
    integer m,n;
    procedure square (A); binary;
        n := 2;
        m := square(n);
end;
```

[3] Also see FUNCTN, Sec. 7.2.3.

Code Procedure SQUARE written in FAP

```
            COUNT       20
            ENTRY       SQUARE
            TRA         *+3
            OCT         650201350000
SQUARE      TSX         $INTERP,4
            STQ         INFO
            STO         N
            XCA
            MPY         N
            XCA
            LDQ         INFO
            TRA         $FUNCTN
INFO        PZE
N           PZE
            END
```

Note that the parameter is in block 2. This implies the procedure square must be declared in block 1. The syllable 35_8 brings STACK[AP], STACK[AP+1] to the MQ and AC, respectively.

7.5.2 Example 2. Use of INTERP and LOADST

ALGOL Source Program

```
begin
    integer m,n;
    procedure square (a,b); binary;
        n := 2;
        square (m,n);
end;
```

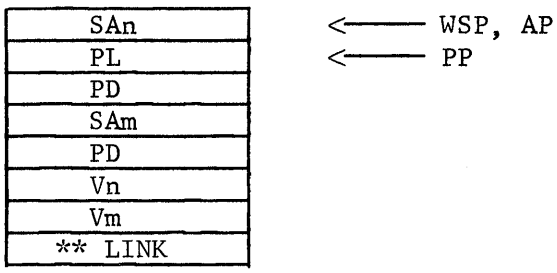
Code Procedure SQUARE written in FAP

```
            ENTRY       SQUARE
            COUNT       20
            TRA         *+3
            OCT         650202350000
SQUARE      TSX         $INTERP,4
            STQ         INFO
            STO         N
            TRA         *+3
            TRA         *+3
            OCT         210201350000
            TSX         $INTERP,4
```

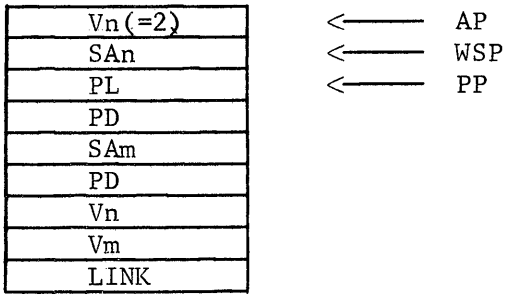
```
                LDQ             N
                MPY             N
                XCA
                LDQ             INFO
                TSX             $LOADST,4
                TRA             *+2
                BCI             1,=$.
                TSX             $INTERP,4
INFO            PZE
N               PSE
                END
```

Discussion of Example 1:

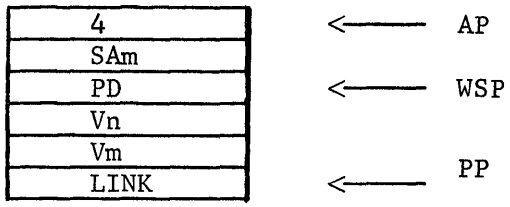
When the code procedure is entered, at SQUARE, the status of the Stack is:



The execution of 35_8 through the call of interp brings the value of N to the AC and the associated information word to the MQ. The top of Stack is now:



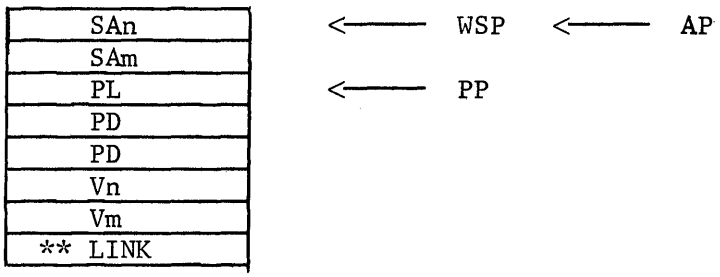
The transfer to FUNCTN overlays the last program descriptor in STACK with the AC and MQ (in STACK [PP-1] and STACK [PP-2], respectively), and transfers control back to the ALGOL Processor. Stack is now:



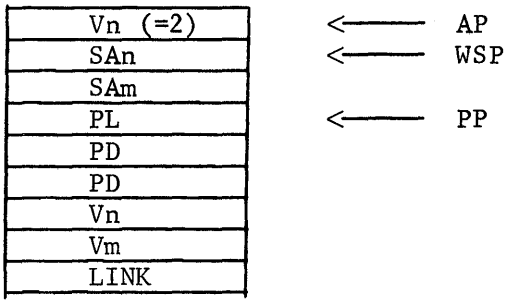
The processor now completes the execution of the Syllable String which puts the value of 4 into M and cleans up Stack, i.e., sets the AP equal to the WSP.

Discussion of Example 2:

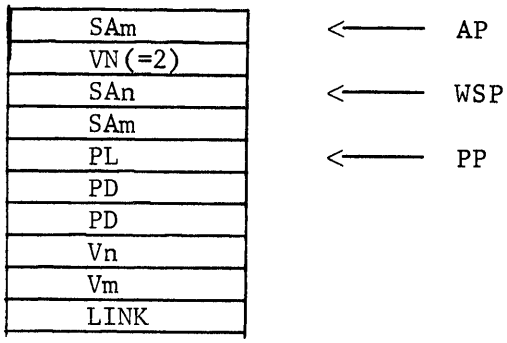
When the code procedure is entered, at SQUARE, the top of Stack is:



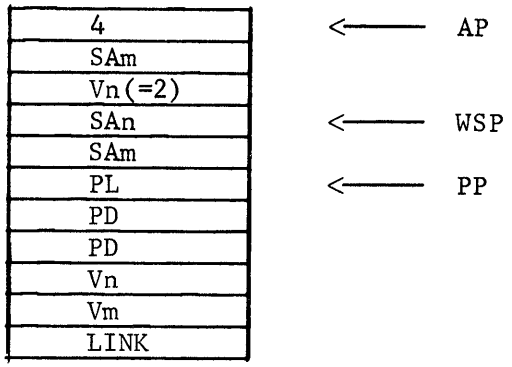
The execution of 35_8 through the first call of interp brings the value of N to the AC and the information word to the MQ. The top of Stack is now:



The execution of 35_8 through the second call of interp brings up the Stack Address of M, i.e., STACK:



The call of LOADST puts the value of N^2 on the top of Stack:

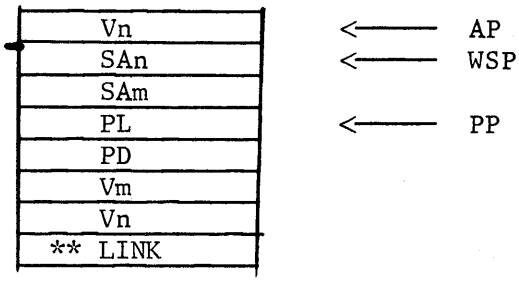


The third call of INTERP causes three things to happen:

1. = 4 is stored in M.
2. $ The AP is set equal to the WSP.
3. This is the Return Syllable, execution of which returns control to the ALGOL Processor.

7.5.3 Use of FETCH

If STACK is in the following configuration:



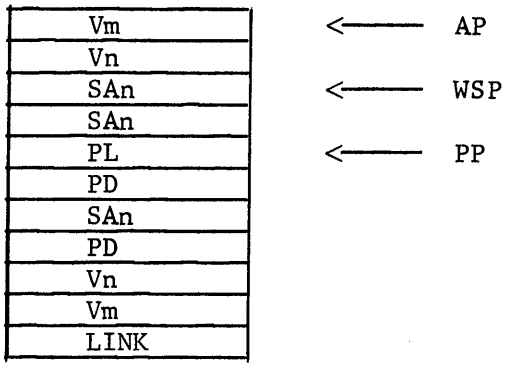
The following coding will suffice to bring the stack address of M, i.e., STACK [AP-4] and STACK [AP-3], to the MQ and AC, respectively:

```
.
.
.
TSX    $FETCH,4
PZE    =4
.
.
```

Remember that every entry in Stack, except array elements and switch program addresses, is two 7090-94 words.

7.5.4 Use of REPLAC

If STACK is in the following configuration:



The following coding will suffice to overlay the last program descriptor in STACK with the MQ and AC

```
.
.
.
TSX      $REPLAC,4
PZE      =10
.
.
.
```

The MQ would be put int STACK[AP-10], and the AC in STACK[AP-9].

7.5.5 Use of GETADD

Suppose that the following coding exists in a code procedure:

```
.
.
.
TSX     $GETADD,4
CLA*    2,4
.
.
.
```

after the execution of CLA* 2,4 the value of the Accumulator Pointer will be in the AC.

7.5.6 Use of GETADD

Suppose the following coding exists in a code procedure:

```
.
.
.
TSX      $GETADD,4
CLA      5,4
.
.
.
```

after the execution of CLA 5,4 the AC will contain the absolute address of syllable string in the form

```
TSX Syllable String, 0
```

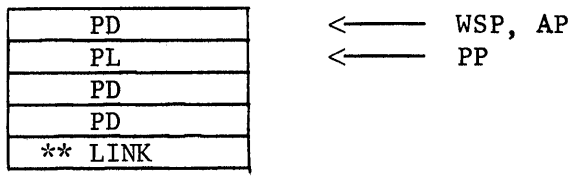
7.5.7 Use of INTERP, GETADD

As an illustration of how INTERP and GETADD can be used to obtain information from syllable string, consider:

ALGOL Source Program

```
begin
     procedure online (s); binary;
     online ('INFORMATION TO BE TRANSMITTED TO A CODE PROCEDURE
              BY USING INTERP AND GETADD');
end;
```

After execution of the call of the procedure **online** , the Stack looks like:



where the last PD contains the absolute word address of Syllable String containing the character 75_8 which precedes the message INFORMATION ... GETADD.

The code procedure to transmit this information from Syllable String to the code procedure

```
            .
            .
            .
            TRA   SETUP
            OCT   350000000000        RETURN THE PROGRAM ADDRESS OF THE
ONLINE      TSX   $INTERP,4           MESSAGE TO THE AC(14—>30) IN BINARY
SETUP       ANA   =0001777740000
            ARS   14
            ADD   =1                  INDEX 1 CONTAINS THE SUBSCRIPT OF
            PAX   0,1                 SYLLABLE STRING
            TSX   $GETADD             RETURNS ABSOLUTE ADDRESS OF
            CLA   5,4                 SYLLABLE STRING
            STA   *+3
AGAIN       AXT   12,2
            AXT   6,4
            LDQ   **,1
            STQ   MESSE,2
            TX1   *+1,1,1
            .
            .
            .
```

Each time the LDQ **,1 command is executed, a word of the message, INFORMATION ... GETADD, is brought to the MQ from Syllable String.

7.5.8 An Example of Using an Assembled Procedure

Sample Deck:

```
Col. 1    Col. 7
*         DECKS
*         FAP
          ENTRY     SQUARE
          TRA       *+3
          OCT       650201350000
SQUARE    TSX       $INTERP,4
          STQ       INTO
          STO       N
          XCA
          MPY       N
          XCA
          LDQ       INFO
INFO      PZE
N         PZE
          END

Col. 1    Col. 7
*         USE(ALGOL)  )
*         DATA        )  Both cards are required

    begin
        integer m,n;
        procedure square(n); assembly;
        n := 2;
        m := square(n);
    end;
$EOF
```

7.6 Methods of Linkage

There exist two methods of linking code procedures with ALGOL Programs; one is advantageous to the systems programmer while the other is advantageous to the general user.

Code Procedures Loaded in Front of the ALGOL Processor

7.6.1 Method 1

Method 1 is advantageous to the systems programmer but not to the user. The method consists of

1. Loading the code procedure in FRONT of the ALGOL Processor;
2. Entering the entry point(s) in the subroutine XCODE.

XCODE is an ALGOL Processor subroutine that contains a list of all code procedures linked by this method (plus other predefined I/O routines). The ALGOL Processor searches the XCODE list when a code procedure, linked by this method is to be executed. The entry in XCODE consists of the first six non-blank characters to the ALGOL Source Declaration immediately followed by a transfer to the appropriate entry point.

Code procedures linked by this Method 1 must be declared as:

procedure <procedure identifier><formal parameter part>; <specification part> **code**;

Example of the entry in XCODE: Consider the code procedure square discussed in Section 4.1. To link it by method 1, the declaration must be:

```
procedure square(n); code;
```

The entry in XCODE must be:

```
.
.
.
BCI   1,SQUARE
TRA   $SQUARE
.
.
.
```

These two names need not be the same; the name in the address field of the BCI command is the name used in the ALGOL Source Program. The name in the address field of the TRA command is the name of the entry point in the code procedure. If the name in either of these cases contains less than six (6) characters, it is to be left-adjusted and filled with blanks.

Example

ALGOL Source Program

```
.
begin
procedure AAA(B,c); code;
.
end
.
```

Entry in XCODE

```
.
.
.
BCI   1,AAA
TRA   $ZZZZ
.
.
.
```

WARNING ! ! ! !

CODE PROCEDURES LINKED BY THIS METHOD WILL AFFECT THE LINKAGE OF CODE PROCEDURE BY METHOD 2. SEE SECTION 7.7 FOR EXPLANATION.

Advantages of Method 1

1. The standard single transfer vector is used.
2. If an absolute core load of the processor is used, the code procedures only have to be relocated once.
3. Common is defined.[4]
4. Library is available.

---

[4] If common is used, the ALGOL Source Program's name list, which is used in a Stack dump, will be destroyed.

Disadvantages of Method 1

1. The code procedures always remain in core thereby limiting the size of stack and string.

2. The user must have his own copy of the processor which takes approximately twenty (20) seconds to load.

7.6.2 Code Procedures Loaded by the Preprocessor Loader

Method 2. Method 2 is advantageous to the general user. The method consists of placing the binary deck(s) anywhere in the source program before the last $.

Code procedures linked by this method must be declared as:

```
procedure <procedure identifier> <formal parameter part>;
          <specification part> binary;
```

Code procedures may also be loaded from the assembly tape. An *DECKS card must be included as one of the monitor control cards. Code procedures linked by this method must be declared as:

```
procedure <procedure identifier> <formal parameter part>;
          <specification part> assembly;
```

Code procedures linked by Method 2 are loaded into syllable string by a loader in the ALGOL Preprocessor (hereafter referred to as PP Loader) at preprocessing time. Code procedures may be linked to a partial library. The loader does not know where the entry points to this library are. To solve this problem a double transfer has been created. In core locations 144_8 to 271_8 at execution time are the addresses of all entry points in the partial library.

A partial library is available consisting of all the routines that the processor requires. Other routines required must be provided. NOTE: If a code procedure requires a lower level routine, for example routine AA, and this entry point appears in the library and is also loaded, the linkage will be set up with the routine that is loaded.

The entry points provided in the partial library are:

ALOG - 14
ALOG10 - 14
ATAN - 15
ATAN2 - 15
COS - 16
DATE - 8
DUMP - 7
EMPTY - 1
ENDFYL - 9
EXECUT - 1
EXEMDP - 7
EXIT - 7
EXP - 13
FETCH - 1
FUNCTN - 1
GETADD - 1
INPUT - 9
INTERP - 1
LOADST - 1
LOG - 14
LOG10 - 14
OUTPUT - 9
PDUMP - 7
REPLAC - 1
REWYND - 9
RETURN - 1
SDUMP - 7
SIN - 16
SQRT - 17
TANH - 18
TENLOG - 14
TIME - 8
TIMEH - 8
TIMREM - 8
TSHGCY - 6
TSHSET - 6
WRITE3 - 25[5]
(BSF) - 2
(BSR) - 2
(BUF) - 4
(CSH) - 6
(EFT) - 17
(ETT) - 2
(EXB) - 4
(EXIT) - 7
(EXPF) - 13
(FIL) - 3
(IOB) - 4
(IOH) - 3
(IOS) - 2
(IOU) - 20
(LCH) - 2
(PRTN) - 21
(RCH) - 2
(RDC) - 11
(RDS) - 2
(RER) - 11
(RERN) - 7
(REW) - 2
(RLR) - 10
(RTN) - 3
(RUN) - 2
(RWT) - 22
(SCH) - 5
(SET) - 4
(SPH) - 5
(SPHN) - 5
(STC) - 2
(STH) - 5
(STHD) - 5
(STHM) - 5
(TCO) - 2
(TEF) - 2
(TES) - 23
(TRC) - 2
(TSB) - 10
(TSH) - 6
(TSHM) - 6
(UNIT) - 2
(WEF) - 2
(WER) - 12
(WRS) - 2
(WTC) - 12
(XCED) - 7
(XCDE) - 7
(XCDT) - 7

The number beside the entry point indicates, with the table below, what routine they are in:

1 - ALGOL Processor
2 - I9 BC IOS
3 - M2 BC IOH
4 - I1 BC IOB
5 - I1 BC STH
6 - I4 BC TSH
7 - P2 BC EXEM
8 - Z0 BC TIME
9 - I9 BC IONE
10 - I9 BC TSB
11 - I9 BC RER
12 - 19 BC WER
13 - B3 BC EXP
14 - B3 BC LOG4
15 - B1 BC ATAN
16 - B1 BC SIN
17 - B4 BC ROOT
18 - B2 BC TANH
19 - Q3 BC EFT
20 - L2 BC IOU
21 - J0 BC PRTN
22 - Q3 BC RWT
23 - I9 BC TES
24 - L3 BC RERN
25 - WRITE

The Double Transfer Vector

Code procedures loaded by Method 2 using routines in the partial library go through a double transfer vector.

---

5 WRITE3 must be called once before the code procedure does any I/O or calls EXIT.

Example

If one has the logical number of a tape unit, N, and wants to use (IOU) to obtain the physical number, in the case of a single transfer vector the following coding would be sufficient to bring the physical number to the AC:

```
.
.
.
CLA $(IOU)
ADD N
STA *+1
CLA **
.
.
.
```

For a block diagram see Figure 7.1.

In the case of a double transfer vector an additional level of indirect addressing is needed, i.e.,:

```
.
.
.
CLA* $(IOU)
ADD  N
STA  *+1
CLA  **
```

For a block diagram see Figure 7.2.

Example

Consider the case in which one wishes to use (IOS) to test for an end-of-file on the tape that has just been read. In the case of a single transfer vector, the following coding would suffice:

```
.
.
.
AXC  EOFROU,4
XEC* $(TEF)
.
.
.
```

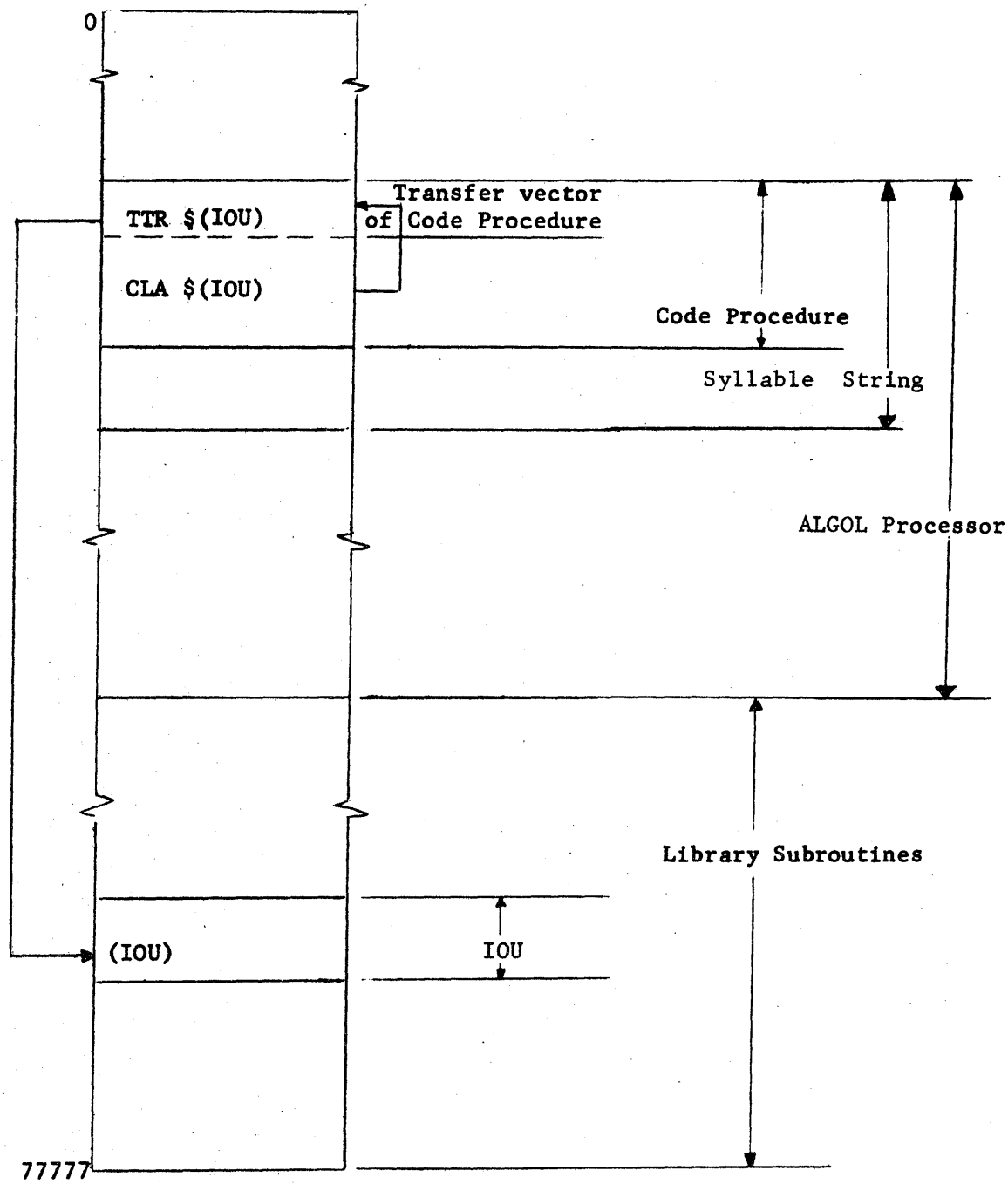


Fig. 7.1. A map of core showing the single transfer vector.

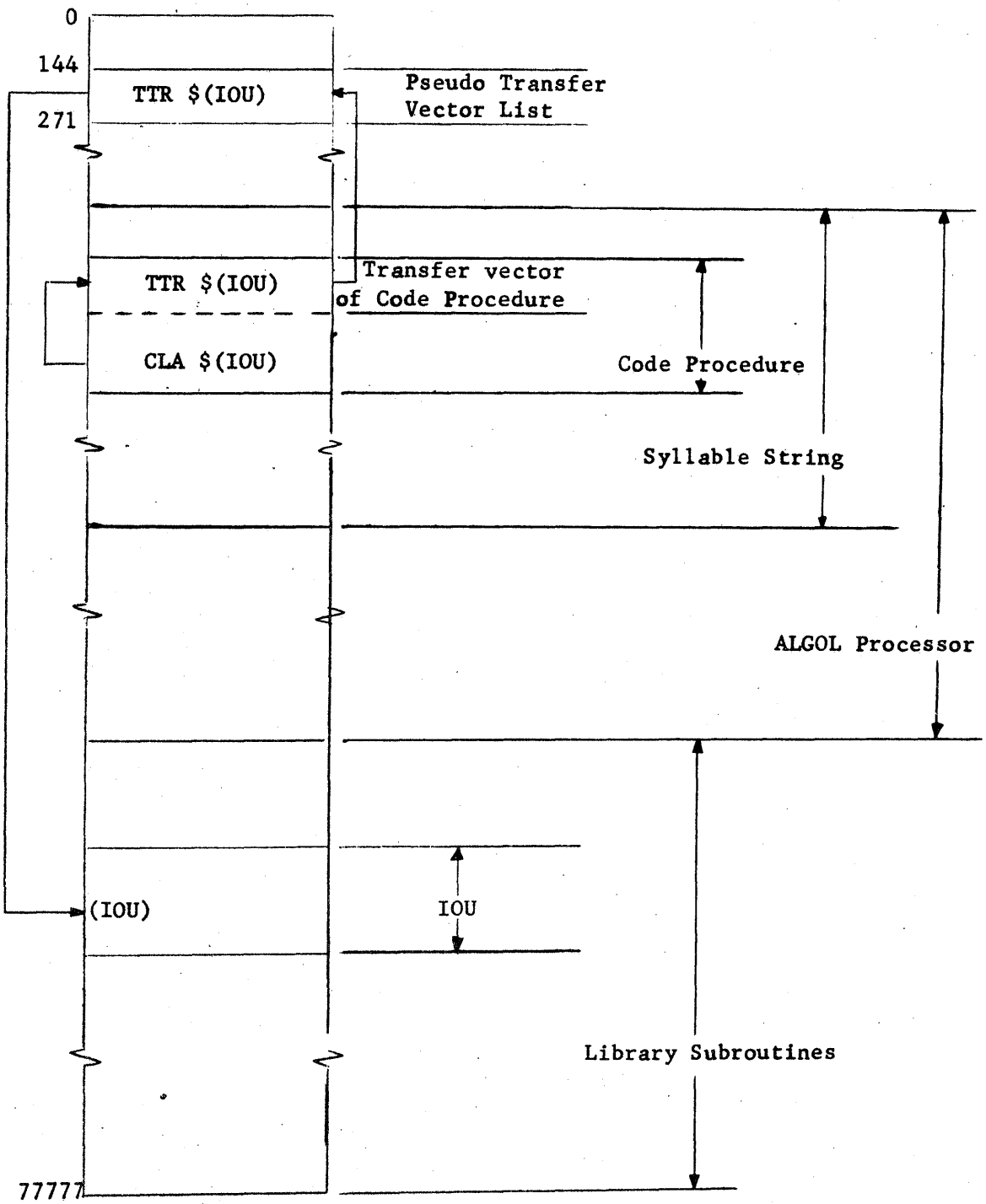


Fig. 7.2. A map of 7094 core showing the double transfer vector linkage between code procedures and external routines

In the case of a double transfer vector, an additional level of indirect addressing is needed:

```
.
.
.
AXC  EOFROU,4
CLA* $(TEF)
STA  *+1
XEC* **
.
.
.
```

Note that this double transfer vector does not apply to any transfers or lower level routines loaded by the PP Loader and linked to code procedures.

WARNING ! ! ! THE PP LOADER MUST KNOW THE ABSOLUTE ADDRESS OF SYLLABLE STRING AT EXECUTION TIME. ANY CHANGES THAT CAUSE THIS ADDRESS TO CHANGE NECESSITATE A RECOMPILATION OF THE PREPROCESSOR.

There are two types of changes that cause this address to change:

1. Any change in any of the subroutines loaded in FRONT of the ALGOL Processor such that the load point of the ALGOL Processor is changed, i.e., use of Method 1 for linking code procedures.
2. Any changes in the declaration list of the first flow chart of the ALGOL Processor.

If the absolute address of Syllable String changes, the method to compute the new address is:

1. Using the storage map find the location at which the ALGOL Processor is loaded.
2. From the ALGOL Processor Name List find the location at which the Syllable String is located.
3. The new absolute address of Syllable String is the sum of (1) and (2); (all three of these are octal numbers).

Initialize the variable BASE ADDRESS OF SYLLABLE STRING in the ALGOL Preprocessor (located about card number 59000) and recompile the ALGOL Preprocessor. The PP Loader also expects to find at OCT 144 the second transfer vector list. The entries in this transfer vector list must be in the same order as the entries in the PP Loader's variable: PSEUDO TV LIST.

Advantages of Method 2:

1. Only the code procedures needed are loaded.
2. Binary decks are loaded with the source program or from the assembly tape[6] (Use the *DECKS card.)

Disadvantages of Method 2:

1. There is a need for double transfer vector.
2. The complete library is not available.
3. Common is undefined.[7]

7.7 Specific Information About the Preprocessor Loader

7.7.1 Declarations

1. When binary deck(s) are included in the ALGOL source program use:

   procedure <procedure identifier> <formal parameter part>;
   <specification part> binary;

2. When the assembly tape is to be loaded use:

   procedure <procedure identifier> <formal parameter part>;
   <specification part> assembly;

   (Also use *DECKS card.)

   NOTE: Both can be used in the same program.

7.7.2 Restrictions

1. No more than 50 code procedures can be loaded.
2. No more than 500 entry points can be loaded.
3. Common is undefined.[7]

---

6 See Section 7.2.

7 See chapter 4 on comment control.

7.7.3 Error Diagnostics and Messages

See chapter entitled "Error Messages".

7.7.4 Printout

7.7.4.1 Printout from the PP Loader should only be used as a debugging aid. To obtain the printout, use the comment control card: comment: LOADER LIST;

7.7.4.2 The printout obtained by using the comment control card LOADER LIST consists of:

a. from where each code procedure was loaded;

b. a list of all the entry points in each code procedure and their core location at execution time

c. the transfer vector for each code procedure

d. a comment that the checksum was deleted on any card

e. the location in core used by each code procedure both absolute core locations and word addresses of Syllable String

f. a listing of each relocated code procedure

7.7.5 General

1. The PP Loader loads all code procedures after all of the ALGOL Source Program has been pre-processed.

2. The PP Loader refers to all code procedures that are loaded by the first entry point in the code procedures.

3. The PP Loader card count is recycled to 1 when a program card is encountered.

4. The PP Loader will accept overlays.

7.7.6 The Assembly Tape

To load code procedures from the assembly tape:

1. the assembly tape must be logical -2;

2. the assembled decks on the tape must be preceeded by an end-of-file mark;

3. the read/write heads must be positioned AFTER the assembled decks when the PP Loader is called.

The above conditions are satisfied on the present BC Monitor System at Berkeley. If the assembled decks are written on the assembly tape starting from the load point, i.e., without a preceeding end-of-file mark, the following changes must be made in the subroutine LDRIO:

1. Replace Card

TJM01940	with:	AXC *+2,4
TJM01960	with:	TRA REWB4+2

2. Merge the following cards:

SLQ	RESETA+3	TJM01671
TRA	SATRTN	TJM01721
TLOB	*	TJM01951

7.8 An Interface Routine for FORTRAN II Binary Decks

Introduction

The only requirement on the compiled subprogram which is to be linked to the BC ALGOL system by this method is that it accept a standard FORTRAN II calling sequence. This is the case for all SHARE subprograms not designed to run under IBSYS.

The user must write a buffering subprogram in FORTRAN II whose name will be used in the ALGOL program and which will call both the linking subroutine and the subprogram to be linked. For example, if the user wishes to use a matrix inversion routine INVERT(A,N), his buffering program would be:

```
SUBROUTINE   INV(A,N)
CALL LINK
CALL INVERT(A,N)
CALL RETURN
RETURN
END
```

The subroutine LINK effects the parameter linkage; the subroutine RETURN returns control to the ALGOL program; and the FORTRAN verbs RETURN and END are present only because the FORTRAN II compiler requires them.

In the user's ALGOL program, it is the buffering routine which is declared and used. In the above example, INV would be declared as a procedure.

```
begin
procedure  INV(A,n); binary;
.....
.....
INV(B,n);
.....
end
```

The binary decks for both the buffering routine and the linked routine (in the above example INV and INVERT respectively) are loaded with the ALGOL source.

The above example illustrates the method which is to be used in linking external routines to an ALGOL main program. It must be remembered, however, that the full power of the ALGOL procedure statement is restricted by the nature of the routine to be linked. In particular, the following restrictions hold:

1. An expression or subscripted variable may not be called by name.
2. A parameter may not be a designational expression (e.g., a label).
3. A parameter may not be a procedure or switch identifier.
4. An integer expression must be less than 2^{17} in absolute value.

If 2, 3 or 4 occur, processing will be halted and a diagnostic will be given. An exception to 3 is a function designator without parameters; in which case the identifier is an expression itself.

Expressions and subscripted identifiers are handled according to the following:

1. If the formal parameter corresponding to an expression or constant appears in the value list of the buffering routine, then it is evaluated and converted to the specified type. (This occurs before the linking routine is called.) Linkage is made to the storage cell which contains the value of the variable created.

2. If the actual parameter is the identifier of a simple variable then linkage is made to the storage cell which contains the current value of the variable.

3. If the actual parameter is an array identifier, then linkage is made to the first element of the array. (This corresponds to FORTRAN II requirements.)

4. If the actual parameter is a string, then linkage is made as in (3) to an integer array which contains the Hollerith characters six to a word. (If the number of characters in the string is a multiple of six, then the first character of the next word in the array will be an octal 75. If not, the last character of the string will be an octal 75.) In no case will trailing blanks be supplied.

5. If the actual parameter is an expression whose corresponding formal parameter does not appear in the value list, then the expression is evaluated, and linkage is made to the storage cell which contains the result. No type conversion is made.

BC ALGOL stores integers differently than FORTRAN II does.

The Boolean quantitites true and false are stored in BC ALGOL as the the Hollerith TRUE-- and FALSE- and will be linked as if the quantities were integers. It is sometimes useful to note that true is a negative integer and false is a positive integer.

If the subprogram to be linked is a function, then the return part of the buffering subprogram is slightly different. A call to FUNCT I, FUNCT J, FUNCT R, or FUNCT B is made instead of a call to RETURN. For example, if F(X,Y) is the function to be linked, then the buffering would be

```
SUBROUTINE F1(X,Y)
CALL LINK
CALL F(X,Y)
CALL FUNCTR
RETURN
END
```

The corresponding ALGOL program:

```
begin
real procedure F1(X,Y); binary;
......
......
Z := X + Y * F1(X,Y);
......
......
end
```

To link INTEGER FUNCTIONS, use the subroutines FUNCTI and FUNCTJ instead of FUNCTR. With FUNCTI, the storage readjustment of integers is automatic; with FUNCTJ no storage adjustment is made.

FUNCTB is used to link Boolean functions.

8. The Syllable String

8.1 Introduction

The syllable string is the link between the two processing programs in the BC-ALGOL system. Being the output of the pre-processor and the commands for the processor, it contains information as to how the user's ALGOL program is being processed. The syllable string represents the logical processing pattern for the computational processes which the ALGOL source describes. The string itself is composed of syllables (six bit binary numbers - printed as BCD characters) which are interpreted by the processor as operations on the stack - the processor's tool for organizing and ordering the computations.

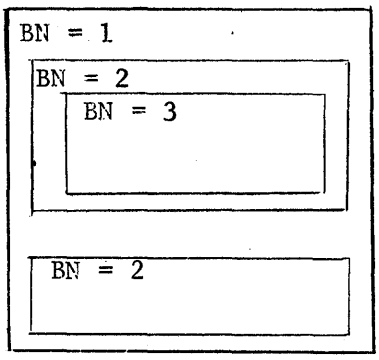
The string is written in modified Polish notation. This notation's main advantage is that arithmetic and logical expressions can be written without parentheses. This is achieved by placing the operator after the variables instead of between them. For example X + Y is written XY+ and (X-Y)*Z becomes XY-Z*. All operations are thus strictly from left to right.

8.1.1 Stack Addresses. Variables are identified in the stack by the block in which they were declared (block number) and by the order in which they appear in that block (order number). For example, the stach address 38 refers to the variable which was declared eighth in the third block. An A or a V precedes the stack address to indicate whether the address or the value of the variable does onto the stack.[1] The string segment A12 V31 tells the processor to "stack" the address of the second variable in the first block then to stack the value of the first variable in the third block.

Statically nested blocks increase in block number; while parallel blocks have the same block number. For example, a program may have the following block structure:

---

[1] The syllable for a formal parameter value call is W not V .



The parameters of a procedure have a block number one higher than the procedure itself. Variables local to the procedure have a block number two higher than that of the procedure. Block zero is reserved for predeclared procedures and own variables.

8.1.2 Program Addresses. As syllables are six bit binary numbers, six syllables fit into a thirty-six bit computer word. In the syllable string, these words are numbered sequentially from zero as are the syllables in each word (left to right). A particular syllable in the string is located by its word address - the number of the word in which it is contained - and by its syllable count - the number of the syllable in the word. Together these two numbers form a program address. When reading a stack dump or when referring to a particular syllable or location in the string, one uses the program address as an absolute reference. However, word addresses appear relative to the position in which they occur in the string. (The syllable count is always an absolute reference.) A word address in the string can be thought of as the number of words one must move from one syllable (present location) to another (designated location) increasing word addresses. There is a simple rule which one may use to compute program addresses for absolute references. The program address in the string is comprised of four syllables. The first is the syllable count; the next three are the word address. Let W denote this word address. Add the decimal equivalent of W to the word address of the first syllable in W; the result, modulo 2^{15}, is the new word address. There is an exception to this rule

however; if the syllable count (preceding W) is in a different word than is the first syllable of W then 1 must be subtracted from the above result. For example, consider the word address 3C+ in figure 2. Its decimal equivalent is 227, adding the word address of the syllable 3, i.e. 10, yields 237 and this is correct since the syllable count, 4, is in the same word.

0 0 ǂ = 10
4 3 C +

Fig 2. Sample Word with Heading

8.1.3 The Edited String. As some syllables do not have a unique printing character, the string is edited prior to printing so that a syllable appears on the page as a BCD character or as a period followed by a BCD character. Typical words might be:

+ = $.F4 or ..F402.V

A period appearing in a word implies that it and the next character combine to name the syllable. Table 1 gives a complete list of how the edited syllables print.[2]

To obtain the edited string, use the following comment control:

comment: PRINT STRING;

Above each word of string, the word address appears both in a BCD representation and decimal value. As an example:

045 = 306
[syllable string]

---

[2] With the exception of .*, .., and .-, a period before a character adds octal 10 to the BCD value of the character in the internal representation.

8.1.4 Form of the syllable string. Where it is necessary for easier comprehension of the structure of the syllable string, a diagram will be included. Some symbols which will be used with be a circle enclosing a letter - this designates an actual syllable in the string; a rectangle represents a complex structure which will explained in a later section.

8.2 Syllable String Examples

8.2.1 Declaration of Variables

8.2.1.1 Simple Variables

form: Ⓑ [NUM] [TYPE] [NO. TYPE] ... Ⓢ

where NUM = total number declared
TYPE = the type of variable declared,
R = real, J = integer, H = Boolean, S = switch,
Y = array, P = procedure. For arrays and procedures numbers also specify types: 0 = real,
1 = integer, 2 = Boolean.
NO. TYPE = the number of that type declared

reference: begin real a,b,c,x,y,3; integer i; boolean good, bad, poor;

source: 'BEGIN' 'REAL' A,B,C,X,Y,Z $ 'INTEGER' I $ 'BOOLEAN' GOOD,
BAD, POOR $

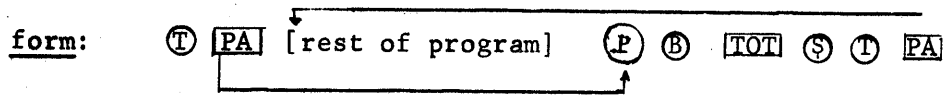
string: 02T = 179 02U = 180 02V = 181
B ≠R6J1H 3$

comment:

B	begin syllable
≠	10 variables declared in this block
R6	6 are of type real
J1	1 is of type integer
H3	3 are of type boolean
$	end of declarations for this block (NOTE - no other declarations are followed by a $)

8.2.1.2 Own Declarations

form:



where TOT = the total number of owns.

comment: Own variables are handled in the BC ALGOL system by making it look like they have been declared in block 0. This means that there will be no string generated at the point where the variable was declared, but the string will appear at the end of the program. The first word of every program is a transfer to the end to check to see if there were any own declarations.

reference: own real r; own integer i;

source: 'OWN' 'REAL' R $ 'OWN' 'INTEGER' I $

string:

```
02T = 179     ... [rest of ALGOL program] ... 033 = 195
T300AB        ...                              .P

034 = 196              035 = 197
BOR1J1                 $T57.X.0
```

comment:

T300A	transfer to end of the program to check for own declarations
B	begin of ALGOL program
.PB	own begin
0	$= 46_8 = 38_{10}$, BCD representation of total number of variables declared as own. This includes the predefined procedures, 36 or (M) + those declared (2).
R1J1	1 real, 1 integer declared
$	end of declarations
T57.X.0	transfer to beginning of program

8.2.1.3 Array Declarations

form: Ⓨ TYPE LB UB ... LB UB ⓘ TOT

where TYPE = the type of the array
LB = lower bound
UB = upper bound
TOT = total number of arrays declared in this statement

reference: real array xyz[A:7, B:1];

source: 'REAL' 'ARRAY' XYZ.(A..7, B..1). $

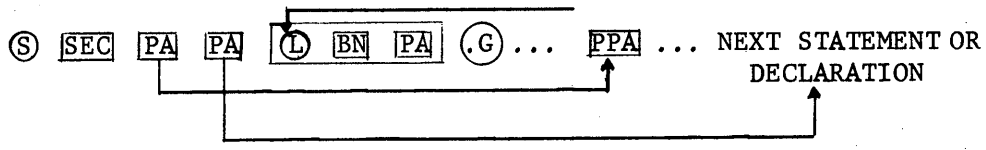
string:
```
02U = 180    02V = 181    02W = 182
Y0V11C       000007       .0V12'1
```

comment:

Y	array declaration syllable
0	type is real
V11	value of A, lower bound
C000007	integer constant 7, upper bound
.0	integer constant 1, lower bound
V12	value of B, upper bound
'	end of declaration
1	1 array declared with these bounds

8.2.1.4 Switch Declarations

form:



where SEC = switch element count

[ ] = a label structure explained in 8.2.4.1.

PPA = pseudo program address. This consists of 3 syllables instead of the usual 4. The left most 3 bits of the left most syllable contain the syllable count, the remaining 15 bits contain the word address.

example: P.X, = 477773_8

word address = 77773_8 = -5

syllable address = 4^8

reference: switch s := $\ell 1$, $\ell 2$, $\ell 3$, $\ell 4$;

source: 'SWITCH' S = L1, L2, L3, L4 $

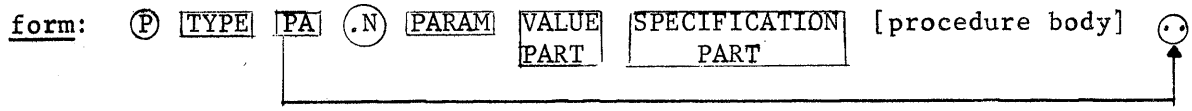
string:
```
035 = 197   036 = 198   037 = 199   038 = 200   039 = 201   031 = 202
S           000004      600500      0≠L100      0.X.GL11    019.GL1

03 = 203    03' = 204   03.5 = 205  03.6 = 206  03.7 = 207
301.7.GL    1301F.G     G.X,        .G.X,       P.X,

03+ = 208
.P.X,
```

comment:

```
S          switch declaration syllable
000004     4 elements in this switch
6005       PA which points to first PPA
000≠       PA which points to next declaration or statement
L1000X     first label in switch
.G         switch go to syllable
... etc.
G.X,       first PPA pointing to first label
... etc.
```

8.2.1.5 Procedure Declarations

form:



where TYPE = the type of the procedure declared
PARAM = number of parameters
VALUE PART)
SPECIFICATION PART) as defined in ALGOL 60 report

reference:
```
procedure sam (a,b,c,d,e); value a,b; real a; integer array b;
        [procedure body]
```

source:
```
'PROCEDURE' SAM(A,B,C,D,E) $ 'VALUE' A,B, $ 'REAL' A $
      'INTEGER' 'ARRAY' B $ [procedure body]
```

string:
```
046 = 262   047 = 263   048 = 264  [procedure body]  04.7 = 271
P0100       9.N5U21     0.PGU221                     ..
```

comment:
```
P          procedure declaration syllable
0          type real
1009       PA pointing to next statement or declaration
.N         check parameter count syllable
5          5 parameters
U21        first parameter called by value
0          type is real
.PG        array called by value
U22        second parameter called by value
1          type is integer
[procedure body]
..         procedure return syllable
```

8.2.2 Assignment Statements

8.2.2.1 Assigning Constants

form: [T] [CONSTANT]

where T = Type of Constant. If the constant is not a special one, the constant occupies the next full word. In this case C is the syllable for an integer constant; D is for a real constant. Special constants appear as any other syllable would. These are 0 for integer; .0 for integer 1; .W for boolean true; and .X for boolean false.

number conversions

To convert a number in the string, first convert it to octal, then to decimal. Conversion to octal merely requires replacing each syllable by its octal equivalent; e.g., + .P - 000 is 20 57 13 00 00 00. Note that .P represents one syllable. Note also that two octal digits replace on syllable.

An octal integer is easily converted. For example:

$$\text{octal } 4\ 2\ 1\ 7\ 1 = 4 * 8^4 + 2 * 8^3 + 1 * 8^2 + 7 * 8 + 1 = 22529.$$

Floating point numbers are stored as $2^n * M$ where n is an integer and M is an octal fraction.

To obtain n: subtract 200 (octal arithmetic) from the first three octal digits.

To obtain M: divide each digit beginning with the fourth by increasing powers of eight, and sum these.

Example: $205715000000 = 2^5 * \left(\frac{7}{8} + \frac{1}{8}2 + \frac{4}{8}3 + \ldots\right) = 30.0$

8.2.2.2 Assigning Variables

reference: x := y;

source: X = Y $

string:
```
02X = 183   02Y = 184
A14V15      =$
```

comment:

A14	place the address of X on the top of the processor's stack
V15	place the value of Y on the top of the processor's stack
=	store the value on the top of the stack into the next address, i.e. value of Y onto X
$	end of statement

8.2.2.3 Arithmetic Assignments

reference: x := -(b - c ↑ a) ;

source: X = -(B-C**A) $

string:

02.V = 189	02.W = 190	02.X = 191
A11V11	V13V11	.* - . - = $

comment:

A11	Stack the address of x
V12V13V11	Stack the values of c, b, and a, respectively
.*	Raise the next-to-top value in the stack to the power of the top value in the stack. Store the result in the next-to-top location in the stack, make this the top of the stack. (c↑ a).
-	Subtract the top value in the stack from the next-to-top value and store in the next-to-top location. Make this the new top of the stack. (b - c↑ a).
.-	unary minus
=	store into x
$	end of statement

8.2.2.4 Array Assigments

form: [SUB 1] [SUB 2] ... [SUB n] Ⓐ [BN] [ON] [D-1] Ⓥ [BN] [ON] ⊜ Ⓢ

where SUB 1, SUB 2, ..., SUB n = the subscripts of the array
BN = block number
ON = order number
D-1 = number of dimensions of the array -1.

reference: data [1,3] := a;

source: DATA .(1,3). = A $

string:

04 = 294	04P = 295	04Q = 296	04R = 297
.OC	000003	A131V1	2=$

comment:

.0	integer constant 1, first subscript
C000003	integer constant 3, second subscript
A13	stack the address of data
1	dimensions of data -1
V12	stack the value of a
=	store a into data [1,3]
$	end of statement

8.2.2.5 Boolean Assigments

reference: good := good ∨ bad ⊃ poor;

source: GOOD = GOOD 'OR' BAD 'IMP' POOR $

string:

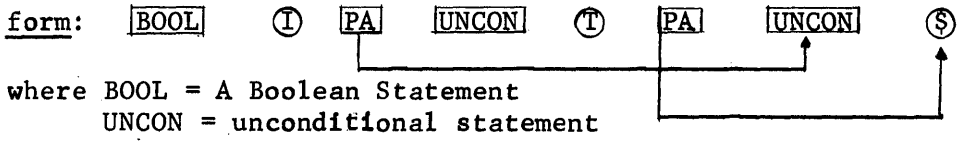
041 = 257	042 = 258	043 = 259
A18V18	V193V1	≠M=$

comment:

A18	stack the address of good
V18,V19	stack the values of good and bad
3	logical or syllable
V1≠	stack the value of poor
M	logical implication operation
=	store the result in good
$	end of statement

8.2.3 Conditional Statements

form:



where BOOL = A Boolean Statement
UNCON = unconditional statement

reference: if a > b then c := a else c := b ;

source: 'IF' A 'GTR' B 'THEN' C = A 'ELSE' C = B $

string:

041 = 257	042 = 258	043 = 259	044 = 260	045 = 261
V11V12	8I3003	A13V11	=T4002	A13V12

046 = 262
=$

comment:

V11 V12	stack the values of A and B
8	Logical operator, greater than
I	If syllable
3003	PA which indicates transfer to be made if statement is false
A13 V11=	C = A
T 4002	transfer around the false statement to the end of the conditional statement
A13 V12	C = B
$	end of statement

8.2.4 Transfers

8.2.4.1 Labels

form: Ⓛ BN PA Ⓖ Ⓢ [1]

reference: here: c := a; [other statements] go to here;

source: HERE .. C = A $ [other statements] 'GO TO' HERE $

string:
```
010 = 64    011 = 65  ... [other statements] ...
      A     13V11

015 = 69      016 = 70
      L2      57.X.SG$
```

comment:
```
A13 V11=  C = A, note no syllables are generated
          at the labeled point
[other statements]
  L2          label in block 2
 57.X.5       PA which points to location of label in
              the syllable string
  G           go to the location of the designational
              expression just processed
  $             end of statement
```

8.2.4.2 Switches

form: SUB Ⓩ BN ON D-1 Ⓖ

A switch is processed as a one-dimensional array of type label.

SUB = the subscript of the switch
BN) = the stack address of the switch
ON)
D-1 = number of dimensions -1

reference: go to S[5];

source: 'GO TO' S.(5).$

string:
```
02U = 180    02V = 181    02W = 182
C            000005       Z160G$
```

---

[1] where BN = the block number of the label, this is used to maintain the block structure of ALGOL after the transfer is made.

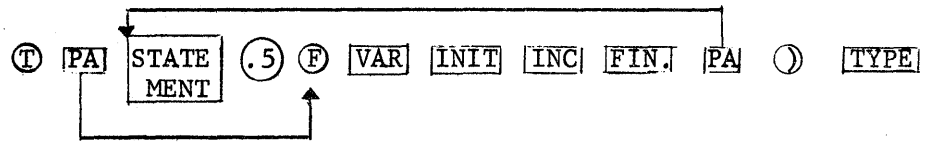
PA = the Program Address which points to the location of the labeled point in the string.

comment:	C000005	integer constant 5, the subscript of the switch
	Z	switch transfer syllable
	16	the stack address of switch S
	0	dimension -1 of switch S
	G	go to syllable
	$	end of statement

8.2.5 FOR Statements

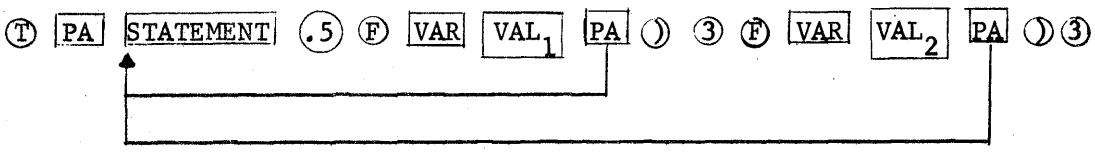
As there are different types of 'for' statements, different forms will be shown. The type of for statement is specified as follows: 0 = simple for, 1 = general for, 2 = for while, 3 = for list.

a. the for step until statement: e.g. for a := 1 step 3 until 22 do ...;



where STATEMENT = the statement to be executed
VAR = the controlled variable
INIT = the initial value of the variable
INC = the value by which it is incremented each time
FIN = the final value the variable is to have
TYPE = the type of for statement; in this case 1; unless the comment control simple for has been used, then it would be 0.

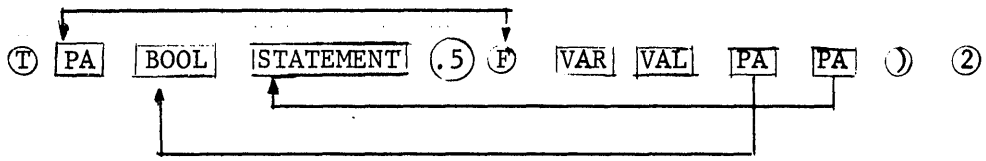
b. the for list statement: e.g. for a := 1,2,3, ... do ... ;



Note 3 = the type this time

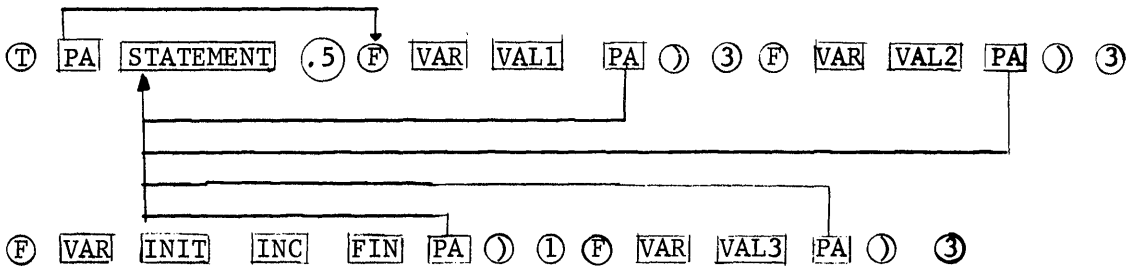
VAR = the variable in the for statement
VAL_1, VAL_2, etc. = the successive values taken on by the variable

c. the for while statement: e.g. for a := 10 while b < 50 do ... ;



where BOOL = a Boolean statement such as b < 50

d. nested fors: e.g. for a := 1,1.5,2 step 2 until 14, 17 do ...;



where VAL1, VAL2 =1, 1.5 values in a for list type = 3
INIT)
INC) the initial value, increment, and final
FIN) value for the for step until stement, type = 1
VAL3 = 17 value in a for list type = 3.

a would take the successive values, 1,1.5,2,4,6,8,10,12,14,17.

reference: for a := 1 step 1 until 10 do sam[a] := a;

source: 'FOR' A=1 'STEP' 1 'UNTIL' 10 'DO' SAM .(A). = A $

string:

02X = 183	02Y = 184	02Z = 185	02.5 = 186	02, = 187
T20	03V11A	120V11	=.5FV11	.0.0C
02(= 188	02.V = 189			
00000≠'	27.X,)1			

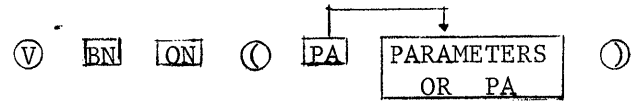
comment:

T2003	transfer to the start of the for statement
V11A120V11=	SAM[a] := a the statement
.5	end of statement
F	for syllable
V11	value of a
.0	integer constant 1, the initial value
.0	the increment
C00000≠	integer constant 10, the final value
27.X,	PA which points to the statement
)	end of for statement
1	type of for statement

8.2.6 Procedure Statements

Comment: A procedure statement is effected in the same manner as a value call, i.e. getting a value to the top of the stack. The parameter list, if not empty, follows the value call and is enclosed within parentheses. A program address (which points to the parameter list) follows the left parenthesis.

form:



parameters = the string giving the parameters which compose the procedure statement

reference: p(x,y,z);

source: P(X,Y,Z) $

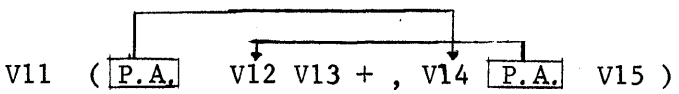
string:

04+ = 272	04A = 273	04B = 274	04C = 275
V11(20	01V12V	13V11)	$

comment:

V11	call of procedure p
(	parameter list follows
2001	program address which points to the start of the parameters
V12V13V14	X,Y,Z = the parameters
)	end of the parameter list

Comment: When a parameter is more complicated than an unsigned number or a simple variable, the parameter is handled as follows: The syllable string for this "complex" parameter appears within the parentheses surrounding the parameter list between the program address which points to the parameter list and the parameter list itself. A comma follows this string. In place of the value call which would have been generated for a "simple" parameter is a program address which points back to the string generated for the complex parameter. For example, the procedure statement P(A,X+Y,B) has a string with the following structure:



If a simple parameter is itself a formal parameter, a procedure identifier, an array identifier, or a switch identifier, then the W syllable is used instead of the V syllable. Signed constants are treated as complex parameters while unsigned constants are simple parameters and appear in the parameter list with the usual structure (C syllable etc.). Labels are considered complex parameters if they appear in the source program after their use as a parameter; otherwise, they are treated as a simple parameter and appear in the parameter list. If the label is a formal parameter, then the W syllable is used in the parameter list; otherwise, the L structure (see 8.2.4.1.) is used.

8.2.7 Handling of Strings

Comment: When strings are used as constants, they appear in the string as any other constant would.* When used as a parameter in a procedure, they have a special syllable.

reference: text ('ABC');

source: TEXT('(' ABC'')') $

string:
```
04( = 316   04.V = 317   04.W = 318   04.X = 319
V0.6(10     02.VABC      .V,27.X.X    )$
```

comment:

V0.6	call of text
(	parameters follow
1002	program address which points to end of parameters
.V	string quote
ABC.V	ABC string quote
,	end of string
27.X.X	program address which points to start of string
)$	end of parameters

* See chapter 6 on character handling and bit manipulation routines.

8.2.8 The Continuation Syllable

To allow the processor to operate with 127 different syllables instead of the 64 corresponding to the normal 64 BCD characters, the syllable .P is used as a continuation syllable. This has the effect that the syllable following .P is interpreted as shown in Table II instead of Table I (see below).

TABLE I

BCD CODE (octal)	DECIMAL EQUIVALENT	PRINTING REPRESENTATION	CARD CODE	SYLLABLE STRING MEANING
00	00	0	0	fault
01	01	1	1	$\div$ integer divide
02	02	2	2	$\wedge$ logical "and"
03	03	3	3	$\vee$ logical "or"
04	04	4	4	$<$ less than
05	05	5	5	$\leq$ less than or equal to
06	06	6	6	= equal to
07	07	7	7	$\geq$ greater than or equal to
10	08	8	8	$>$ greater than
11	09	9	9	= not equal to
12	10	≠	2-8	transfer to word address + 1 for entry into external procedure . (E.g. code)
13	11	=	3-8	:= store
14	12	'	4-8	end of array or switch declarations
15	13	.5	5-8	end of for body
16	14	.6	6-8	ML* LINE
17	15	.7	7-8	ML PAGE
20	16	+	12	add
21	17	A	12-1	address call

* ML = machine language

TABLE I (continued)

BCD CODE (Octal)	DECIMAL EQUIVALENT	PRINTING REPRESENTATION	CARD CODE	SYLLABLE STRING MEANING
22	18	B	12-2	begin
23	19	C	12-3	integer constant
24	20	D	12-4	floating point constant
25	21	E	12-5	end
26	22	F	12-6	for
27	23	G	12-7	go to
30	24	H	12-8	Boolean
31	25	I	12-9	if
32	26	.*	12-0	↑ exponentiation
33	27	..	12-3-8	procedure return
34	28	)	12-4-8	ML OUTPUT
35	29	.E	12-5-8	escape character - used to return to the processor from a code sub-routine
36	30	.F	12-6-8	function identifier
37	31	.G	12-7-8	switch go to
40	32	-	11	subtract
41	33	J	11-1	integer
42	34	K	11-2	≡ equivalent
43	35	L	11-3	label
44	36	M	11-4	⊃ implies
45	37	N	11-5	¬ not

TABLE I (continued)

BCD CODE (Octal)	DECIMAL EQUIVALENT	PRINTING REPRESENTATION	CARD CODE	SYLLABLE STRING MEANING
46	38	0	11-6	integer constant 0
47	39	P	11-7	procedure
50	40	Q	11-8	ML SPACES
51	41	R	11-9	real
52	42	.-	11-0	unary minus
53	43	$	11-3-8	semicolon
54	44	*	11-4-8	multiply
55	45	.N	11-5-8	check parameter count
56	46	.O	11-6-8	integer constant 1
57	47	.P	11-7-8	continuation syllable (see Table II)
60	48	BLANK	BLANK	skip to next syllable
61	49	/	0-1	real divide
62	50	S	0-2	switch declaration
63	51	T	0-3	then or transfer
64	52	U	0-4	value
65	53	V	0-5	value call
66	54	W	0-6	value call, formal parameter
67	55	X	0-7	fault
70	56	Y	0-8	array
71	57	Z	0-9	switch transfer

TABLE I (continued)

BCD CODE (Octal)	DECIMAL EQUIVALENT	PRINTING REPRESENTATION	CARD CODE	SYLLABLE STRING MEANING
72	58	.S	0-2-8	fault
73	59	,	0-3-8	comma (in a procedure call a comma ends a parameter expression)
74	60	(	0-4-8	external code procedure call or left parenthesis
75	61	.V	0-5-8	string quotes
76	62	.W	0-6-8	Boolean true
77	63	.X	0-7-8	Boolean false

TABLE II. THE INTERPRETATION OF SYLLABLES FOLLOWING THE CONTINUATION SYLLABLE

BCD CODE (Octal)	DECIMAL EQUIVALENT	PRINTING REPRESENTATION	CARD CODE	SYLLABLE STRING MEANING
0	0	0	0	fault
1	1	1	1	ML ABS
2	2	2	2	ML SIGN
3	3	3	3	ML LN
4	4	4	4	ML ARCTAN
5	5	5	5	ML EXP
6	6	6	6	ML LOG
7	7	7	7	ML SIN
10	8	8	8	ML COS
11	9	9	9	ML SQRT
12	10	≠	2-8	ML TANH
13	11	=	3-8	ML ENTIER
14	12	'	4-8	ML SET TIME
15	13	.5	5-8	ML FIX ARGUMENT
16	14	.6	6-8	ML FLOAT ARGUMENT
22	18	B	12-2	OWN BEGIN
24	20	D	12-4	STACK DUMP
27	23	G	12-7	VALUE CALL OF ARRAY
30	24	H	12-8	LIBRARY
31	25	I	12-9	ML GETTIME

All other syllables are faults.

TABLE III. STANDARD FUNCTIONS

Standard functions are declared in block zero; otherwise, they are addressed in the normal manner.

Order Number	Function	Reference	Description
1	PRINT	3	--
2	READ	3	--
3	ABS	1	Absolute value of an argument
4	SIGN	1	Sign of an argument
5	LN	1	Natural logarithm
6	ARCTAN	1	Arc Tangent
7	EXP	1	Exponential
8	LOG	1	Common logarithm (base 10)
9	SIN	1	Sine
10	COS	1	Cosine
11	SQRT	1	Square Root
12	TANH	1	Hyperbolic tangent
13	ENTIER	1	Greatest integer function
14	TEXT	2	
15	SETTIME	1	Sets running time limit
16	GETTIME	1	Indicates time remaining
17	PRINTING FORMAT	3	--
18	SAVE	3	--
19	WRITE	3	--
20	READING FORMAT	3	--
21	READING REAL	3	--
22	READING EXIT	3	--
23	READING INTEGER	3	--
24	READING BOOLEAN	3	--
25	LINE	2	New line
26	REAL FORMAT	2	--
27	INTEGER FORMAT	2	--
28	BOOLEAN FORMAT	2	--
29	PAGE	2	--
30	SPACES	2	--

Table III (Continued)

Order Number	Function	Reference	Description
31	TITLE	2	--
32	PUNCH	2	--
33	PRINTER	2	--
34	INPUT	2	--
35	OUTPUT	2	--
36	FORMAT	3	--

9. The Stack

9.1 Introduction

The ALGOL Processor's Stack is used in carrying out the computational processes as indicated by the syllable string generated by the pre-processor.

The stack is a large one dimensional array stored backwards in the memory. On the printed page, the top, or the last entry, of the stack is on the bottom of the page.

There is another smaller stack known as the display. It is used to keep account of dynamic entries to blocks and procedures. Several registers are also used for communication within the stack; these are described as they are encountered in the examples.

A stack dump may be obtained by using the comment control card - comment: dump; . This will cause a stack dump to be made at the place in the ALGOL source program where the comment occurs. A stack dump will also be made when an error condition causes the Processor to terminate its execution of a program.

9.1.1 Form of the stack dump

The first things printed are some of the variables used for communication within the stack.

The accumulator pointer, abbreviated AP, is used to index the stack and gives the location of the top of the stack. The display pointer, abbreviated DP, is used to index the display and indicates the number of dynamically entered blocks and procedures. The parameter pointer, abbreviated PP, is used to allow returns from dynamically nested blocks. It points to the location of the beginning of the last dynamic block. The syllable count, word address, and syllable have alredy been defined; they indicate the location in the string when the dump occurred. The block number and order number have also been defined; they give the variable being processed at the time the stack dump occurred. The value address, abbreviated VA, is used to give the location in the stack of the variable being processed when the dump occurs.

The next block of information printed is the display. This is printed with the display pointer to give the history of the dynamic block structure of the program. The value of the display for each display pointer is the beginning of that block in the stack.

The stack itself is then printed. The left hand column contains the value of the accumulator pointer while the content of the stack at that location is printed to the right.

```
..   STACK DUMP
       VARIABLES
          ACCUMULATOR POINTER =     158
          DISPLAY POINTER =           1
          PARAMETER POINTER =       156
          SYLLABLE COUNT =            1
          WORD ADDRESS =            198
          SYLLABLE =                  0
          BLOCK NUMBER =              0
          ORDER NUMBER =              0
          VALUE ADDRESS =             0

          DP    DISPLAY
           1      156
```

9.2 Stack Dump Examples

These examples will be similar to the examples of Chapter 8. The stack dump will contain the structures created by these examples when executed by the Processor.

9.2.1 Declarations of Variables

```
begin real a,b,c,d,e,f; integer x,y,z; boolean good, bad, poor;
```

9.2.1.1 Simple Variables

```
 AP STACK
136      **LINK          BN= 1            PP= 1        WP= 160
138        VALUE         UNINITIALIZED (a)
140        VALUE         UNINITIALIZED (b)
142        VALUE         UNINITIALIZED (c)
144        VALUE         UNINITIALIZED (d)
146        VALUE         UNINITIALIZED (e)
148        VALUE         UNINITIALIZED (f)
150        VALUE         UNINITIALIZED (x)
152        VALUE         UNINITIALIZED (y)
154        VALUE         UNINITIALIZED (z)
156        VALUE         UNINITIALIZED (good)
158        VALUE         UNINITIALIZED (bad)
160        VALUE         UNINITIALIZED (poor)
```

The execution of the begin syllable B causes the Processor to create a block link to indicate the beginning of a block. The location of the link is entered in the display. The link contains information on the dynamic block number BN „ the parameter pointer, and the working space pointer, WSP .

The amount of storage to be allocated for the variables declared in each block is determined by the syllable following the begin syllable. Two locations in the stack are set aside for each declared variable. In the case of simple variables, one word contains informations on the type and whether the variable has been initialized.

The other word contains the actual value of the identifier. The working space pointer gives the position of the last location used for this allocation. The computational processes are carried out above the working space pointer to protect the information and the value of the identifiers.

9.2.1.2 Own Variables

```
begin own real a; own integer i;

          AP STACK
          75        VALUE              UNINITIALIZED
          77        VALUE              UNINITIALIZED
         136      **LINK               BN=  1       PP=     1  WSP=    138
         138        VALUE              UNINITIALIZED
```

Storage is allocated for the own variables before the block link for block number 1 is added to the stack. Storage for declared variables is handled in the same manner.

9.2.1.3 Array Declarations

```
begin array a[1:7,-1:3];

140       **LINK          BN=  1       PP=     1     WP=   184
142        VALUE ARRAY    REAL      DIM=     2     MP=   145   FEP= 151 AL=35 ADD=-20
             AP           MAPPING DATA
           145            BS=         1 SB=     7
           147            BS=        -1 SB=     5
             AP           ARRAY VALUES
```

The execution of the array declaration syllable Y causes information on the dimensions and the allocation of array storage to be stored in the two cells corresponding to that identifier. This information contains the number of dimensions DIM , the mapping pointer MP , the first element pointer FEP , the array length AL , and the ADD which is used to compute the actual address of array elements.

The mapping pointer gives the location of further information on the array. There is one more cell set aside for each dimension of the array. This cell contains the basis subscript BS and the subscript bound SB . These are the lower bound and the width of each dimension of the array.

The first element pointer has the address of the first element of the array. All other elements are stored in order with the first dimensions increasing the most rapidly. The array elements are stored above the working space pointer, and another register known as the declaration pointer is used to protect them; it is not printed however.

When arrays are printed, the two cells in which the initial information is stored are printed. The mapping information then follows, with the array elements following last. Array elements having a value of 0 are not printed. Printing of the stack is then returned to the usual sequential order.

9.2.1.4 Switch Declarations

switch s=: ℓ1, ℓ2, ℓ3, ℓ4;

```
151     SWITCH          PA=6,  205    SEC=     4          PP=    149
          WA                SWITCH LIST PA
        205                 PA=2,  200
        206                 PA=3,  201
        207                 PA=4,  202
        208                 PA=5,  203
```

The execution of the switch declaration syllable S causes information on that switch identifier to be stored in the two cells allocated for it. The program address gives the location in the

syllable string of the first pseudo program address. The switch element count SEC gives the number of elements in the switch list. The parameter pointer gives the location of the last block or procedure entered. This is required by the local and global nature of ALGOL variables. The locations in the syllable string of the pseudo program addresses is given next by the word address WA . The location which the pseudo program address is referencing is given by the corresponding switch list PA .

9.2.1.5 Procedure Declarations

```
procedure sum(a,b,c,d,e); value a,b; real a; integer array b;
176     PROGRAM DESCRIPTOR      BN= 1     PP=  168   PA=1,  263      REAL
```

The execution of the procedure declaration syllable P causes a program descriptor to be stored in the two cells allocated for the procedure identifier.

The information contained is the block number in which the procedure is declared, a parameter pointer performing its usual function, a program address giving the location in the string where execution of the procedure begins, and the type of the procedure.

The location where the actual execution begins is usually the location of the check parameter count syllable .N . The exceptions are when the comment control cards comment: no parameter tests; or comment: variable number of parameters; are in effect.

9.2.2 Procedure Statements

A procedure statement causes the program descriptor associated with that procedure identifier to be brought to the top of the stack. A procedure link is then added above the program descriptor. The procedure link is similar to a block link in that it contains information on the location of the last block or procedure link. It contains the working space pointer which performs the same function as it does for a block. The location in the string of the procedure statement is also included.

9.2.2.1 Parameters

Parameters are brought to the top of the stack in the order in which they appear in the procedure statement.

If the parameter is a simple variable, a stack address is placed on the top of the stack for this parameter. It contains a value address VA , which gives the actual location of the paramter in the stack. The type of parameter is also given. If the parameter is a constant, its value is placed in the location for the parameter. The description DES is given which has the property of the parameter.

Parameters which are not simple identifiers are handled differently. Parameters which are array or switch elements, procedure identifiers, or expressions, cause a program descriptor to be placed in that parameter's location. This program descriptor contains the block member of the procedure statement, the parameter pointer, and the location in the string where the parameter appears. The type of the parameter is also given.

The following examples illustrate calls of three parameters. In the first one the parameters are all simple identifiers which cause three stack addresses to be placed on the stack. In the second, the first parameter is an array element which causes a program descriptor to be placed in the corresponding location.

```
      sum (a,b,c);
241   PROGRAM DESCRIPTOR   BN=43     PP=  191   PA=5,   181   REAL
244  *PROCEDURE LINK       BN= 2     PP=  191   PA=2,   217   WP= 250  RP=  191
246   STACK ADDRESS        VA=  199  INTEGER    DES=0
248   STACK ADDRESS        VA=  201  INTEGER    DES=0
250   STACK ADDRESS        VA=  203  INTEGER    DES=0
      sum (x[i],b,c);
241   PROGRAM DESCRIPTOR   BN=43     PP=  191   PA=5,   181   REAL
244  *PROCEDURE LINK       BN= 2     PP=  191   PA=4,   221   WP= 250  RP=  191
246   PROGRAM DESCRIPTOR   BN= 1     PP=  191   PA=5,   218   REAL
248   STACK ADDRESS        VA=  201  INTEGER    DES=0
250   STACK ADDRESS        VA=  199  INTEGER    DES=0
```

If the formal parameter is called by value, the same mechanism is used. However, before execution of the procedure body the actual value of the parameter is placed on the top of the stack instead of the stack address. This has the effect of making a copy of the actual parameter for the procedure so that the procedure cannot change the actual parameter in the stack. This is how calls by value are specified in the ALGOL 60 Report Section 4.7.3.1.

If arrays are called by value, the information word for the array is placed in that parameter's location and the actual elements are placed above the working space pointer.

If the procedure body itself is a block, then a block link is added after the spaces for the parameters have been filled with the appropriate information. Execution of the procedure body begins, and storage is allocated for the declared variables as described before.

```
sam (10,b,7,7,2);
241  PROGRAM DESCRIPTOR          BN=43        PP= 191  PA=2,  189 REAL
244 *PROCEDURE LINK              BN= 2        PP= 191  PA=1,  236 WP= 279  RP= 191
246  VALUE       REAL            0.10000000E 02
248  VALUE ARRAY INTEGER         DIM=    1    MP= 212  FEP=  256 AL=  5  ADD=    1
       AP        MAPPING DATA
      212        BS=         1   SB=    5
       AP        ARRAY VALUES
      256                    1
      257                    2
      258                    3
      259                    4
      260                    5
250  VALUE       INTEGER              7
252  VALUE       INTEGER              7
254  VALUE       INTEGER              2
```

9.3 FOR Statements

For statements are treated as implicit procedures in the BC-ALGOL system. Examples illustrating the various types are given.

9.3.1 for step until

for i := 1 step 1 until 3 do <statement>;

```
252  STACK ADDRESS          VA= 204  INTEGER    DES=0
254  VALUE        INTEGER       1
256  VALUE        INTEGER       1
258  VALUE        INTEGER       3
260  PROGRAM DESCRIPTOR     BN=1     PP=  202   PA=1, 215  REAL
263 *PROCEDURE LINK         BN=2     PP=  202   PA=6, 220  WP=  263   RP=   1
```

The controlled variable's stack address is placed on the top of the stack. Then the initial value, the increment and the final value are placed on the stock. A program descriptor is placed on the top of the stack with a program address which contains the location in the string of the statement to be executed. A procedure link is then constructed in which the program address contains the location in the string of the for list. The actual computations described by the statement are performed above the procedure link.

for list

for i := 1,2,3, do <statement>;

```
252  STACK ADDRESS          VA=   204   INTEGER    DES=0
254  VALUE        INTEGER       1
256  PROGRAM DESCRIPTOR     BN= 1       PP=  202   PA=0,  222   REAL
259 *PROCEDURE LINK         BN= 2       PP=  202   PA=2,  226   WP=  259   RP=   3
```

The controlled variable's stack address is again brought to the top of the stack. The value that the controlled variable is to assume is placed above it. The program descriptor and procedure link are then placed on the top of stack as described before.

for while

for i := 1 while b < 10 do <statement>;

```
252  STACK ADDRESS          VA=   204   INTEGER    DES=0
254  VALUE        INTEGER       1
256  PROGRAM DESCRIPTOR     BN= 1       PP=  202   PA=0,  234   REAL
258  PROGRAM DESCRIPTOR     BN= 1       PP=  202   PA=2,  236   REAL
261 *PROCEDURE LINK         BN= 2       PP=  202   PA=6,  242   WP=  261  RP=   2
```

The controlled variable is again brought to the top of the stack. The value it is to assume is then brought to the top of the stack. A program descriptor giving the location in the string of the Boolean expression is then placed on the top of the stack. The program descriptor and the procedure link are then added to the stack.

If the parameters of a for statement are not simple identifiers or constants, then a program descriptor will appear with the location in the string of the parameter.

If the for statement is nested, then there will be a series of stack addresses, values, program descriptors and procedure links on the top of the stack as each for statement is executed. The example following is a stack dump of a statement of the form

```
for a:= 1 step 2 until 5 do
    for b:= 1 step 1 until 3 do
        <statement>;
```

```
229  STACK ADDRESS            VA=  181  INTEGER    DES=0
231  VALUE         INTEGER         1
233  VALUE         INTEGER         2
235  VALUE         INTEGER         5
237  PROGRAM DESCRIPTOR       BN= 1     PP=  179   PA=1,  209    REAL
240 *PROCEDURE LINK           BN= 2     PP=  179   PA=6,  219    WP=  240  RP=   1
242  STACK ADDRESS            VA=  189  INTEGER    DES=0
244  VALUE         INTEGER         1
246  VALUE         INTEGER         1
248  VALUE         INTEGER         3
250  PROGRAM DESCRIPTOR       BN= 2     PP=  240   PA=0,  210    REAL
253 *PROCEDURE LINK           BN= 3     PP=  240   PA=6,  215    WP=  253  RP=   1
```

9.4 Composition of the Stack's Special Structures

For ease in writing machine code procedures, a more detailed knowledge of the structure of the stack may be useful. This section is intended for that purpose.

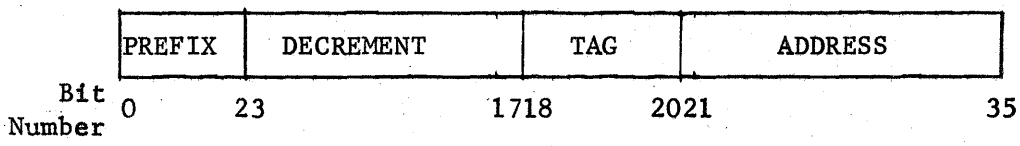
The stack storage assignments for the stack quantities are:

Description	Stack Type	Type of Stack Element
0	0	REAL value
0	1	INTEGER value
0	2	BOOLEAN value
0	3	SWITCH DESIGNATOR
0	4	LABEL
0	5	STRING ADDRESS
1		ARRAY DECLARATION
2		STACK ADDRESS
3		PROGRAM DESCRIPTOR
4		BLOCK LINK
5		PROCEDURE LINK
6		SWITCH DECLARATION

The abbreviations used are:

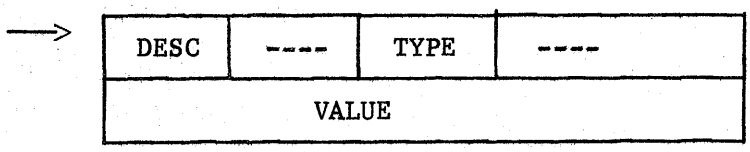
BN for block number
DESC for description
INFOWORD for the IBM 7094 word containing the description of the stack quantity
PROCRTN for processor return
STKDS for stack address description
SC for syllable count
---- for zeros normally expected in that part of a word.

The representation of the IBM 7094 word is as follows:



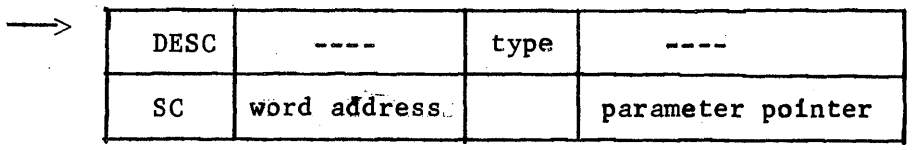
The various structures will now be given; the actual IBM 7094 word referenced by the accumulator pointer is designated by an arrow on the left.

9.4.1 Value.. real, integer, or boolean



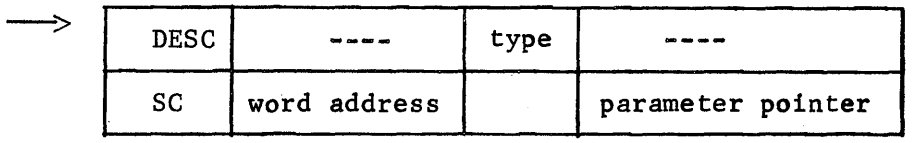
If the low order bit of the info word is a 1, then the variable is uninitialized. If next bit to the low order bit is a 1, then the variable is undefined.

9.4.2 A Switch designator



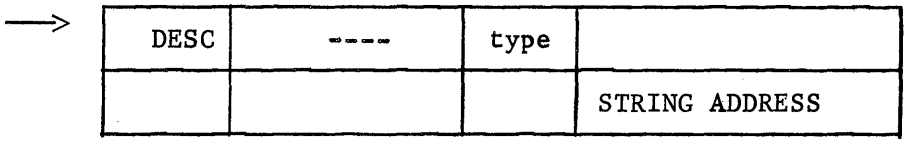
DESC	----	type	----
SC	word address		parameter pointer

9.4.3 A label



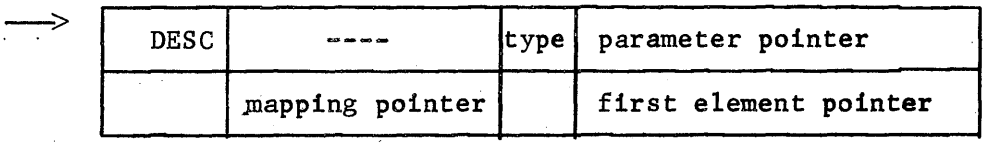
DESC	----	type	----
SC	word address		parameter pointer

9.4.4 A string address



DESC	----	type	
			STRING ADDRESS

9.4.5 An array declaration



DESC	----	type	parameter pointer
	mapping pointer		first element pointer

array mapping data

- ADDER
LOWER BOUND, FIRST SUBSCRIPT
WIDTH, FIRST SUBSCRIPT

other subscripts

DIMENSION
ARRAY LENGTH

9.4.6 A stack address

DESC	----	TYPE	
STKDS			STACK ADDRESS

9.4.7 A program descriptor

DESC	BN, PROC RTN	TYPE	PROCESSOR ADDRESS
SC	WORD ADDRESS		PARAMETER POINTER

9.5.8 A block link

DESC	BN	TYPE	working space pointer
	declaration pointer		parameter pointer
SC	word address		

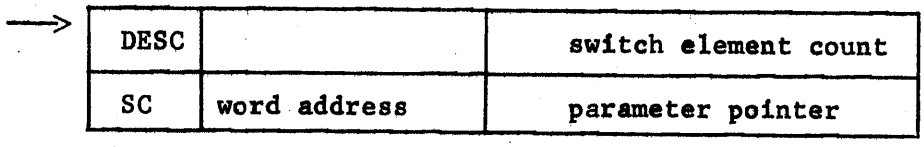
Note that this structure contains three IBM 7094 words instead of the usual two. The declaration pointer is used in case arrays are declared. It is set at the end of the storage allocation for the array elements and protects them in the same manner the working space pointer protects the declared variables information and values.

9.4.9 A procedure link

DESC	BN	TYPE	working space pointer
	declaration pointer		parameter pointer
SC	word address		return pointer

Note that the procedure link also contains three IBM 7094 words. The return pointer is used to restore the processor to its original state after the completion of a procedure statement. The link for a for body is similar except that the for type replaces the return pointer.

9.4.10 A switch declaration



DESC		switch element count
SC	word address	parameter pointer

10. Library Tape Operations

10.1 Introduction

The BC-ALGOL library tape operations will allow a user a fair degree of flexibility in generating a source language BC-ALGOL program. Provisions are made for selecting various input-output units, as well as providing for scratch storage.

10.2 Input-Output Selection

The selection of the various input-output units is achieved by using comment-control cards. The cards and their uses are also listed in Chapter 4.

To inform the BC-ALGOL system of the user's wish to use the library facilities two comment controls are available comment: library A5; or comment: library B5; Either one or both of these may be used in the same program. They must appear before the first begin of the program. Accompanying these cards must be a $SETUP card for each unit used. The purpose of the $SETUP card is to instruct the computer operator to mount a given tape reel and to assign to it a logical name (see Appendix 4). These cards must appear after the $JOB card.

The $SETUP card has the following format:

```
column 1       8          16
       $SETUP unit       reel,NORING
```

NORING is used to protect the library tapes from accidental destruction and must be used.

Once the BC-ALGOL system is using the library facilities, input files may be selected with the following control card comment: <filename>; . The input file is a logical grouping of source language statements ended by an end of file character. The control card instructs the BC-ALGOL system to read the file designated by <filename> and insert it as part of the source program to be executed. This card must be in the source program deck; if found elsewhere, it is ignored. If there is no file with that name on the library units used, the card is ignored. For obvious reasons, when using the library facilities, file names corresponding to the comment control words listed in Chapter 4 should not be used.

10.3 Scratch Storage

Provisions have been made for the use of a scratch tape. The method is to again use a comment control comment: scratch tape; . This must appear in the source deck or it will be ignored. The action taken is to transfer input control to the scratch tape unit A4. One of the uses could be to save intermediate text generated by one ALGOL program as the input to another segment of an ALGOL job. The device used will be a disk simulation unless a $SETUP card for this option was included.

To return control to the normal input mode by using comment: source tape; It must be found on unit A4 to be executed. If found anywhere else, it is ignored. In addition to returning the input mode to the normal one, an end of file is written on unit A4 without rewinding it.

10.4 Additional Facilities

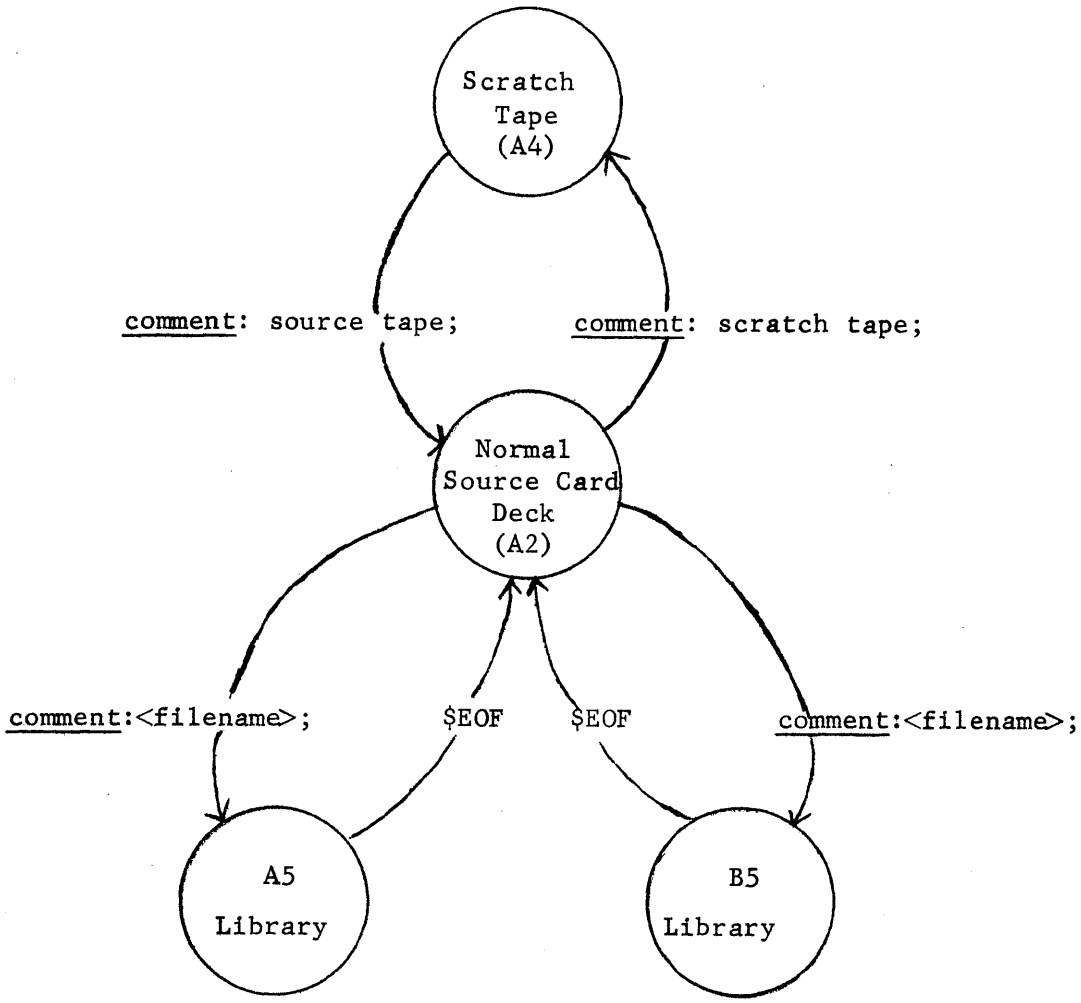
There exists a group of code procedures which will allow the library facilities to be used with a great deal of flexibility. As they are all described in Chapter 6, they will be briefly mentioned here.

The procedure rescan is used to return control to the monitor for segmentation of a long job into separate programs. With the scratch storage facilities, this could be useful.

The procedures inunit and outunit are useful for generating or reading from tapes at execution time.

There are quite a few other tape handling facilities available. The user is referred to Chapter 6 for their descriptions.

A figure to show the interactions of the library facilities along with their means of entry and exit is given. Note that these facilities are not recursive and that everything is channelled through the normal source card deck. The $EOF is the end of file character on the tape.


Scratch
Tape
(A4)
comment: source tape;
comment: scratch tape;
Normal
Source Card
Deck
(A2)
comment:<filename>;
$EOF
$EOF
comment:<filename>;
A5
Library
B5
Library

Reprinted by the
ASSOCIATION FOR COMPUTING MACHINERY
From *Communications of the ACM* 6 (Jan. 1963), 1-17.

Revised Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, F. L. BAUER, J. GREEN, C. KATZ, J. MCCARTHY, A. J. PERLIS, H. RUTISHAUSER, K. SAMELSON, B. VAUQUOIS, J. H. WEGSTEIN, A. VAN WIJNGAARDEN, M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

Revised Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, F. L. BAUER, J. GREEN, C. KATZ, J. MCCARTHY, A. J. PERLIS, H. RUTISHAUSER, K. SAMELSON, B. VAUQUOIS, J. H. WEGSTEIN, A. VAN WIJNGAARDEN, M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

SUMMARY

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition, the notions, reference language, publication language and hardware representations are explained.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

CONTENTS

gramming languages will lead to better resolution:

1. Side effects of functions
2. The call by name concept
3. **own:** static or dynamic
4. For statement: static or dynamic
5. Conflict between specification and declaration

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

REFERENCE LANGUAGE

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

PUBLICATION LANGUAGE

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

HARDWARE REPRESENTATIONS

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference Language	*Publication Language*
Subscript bracket [ ]	Lowering of the line between the brackets and removal of the brackets
Exponentiation ↑	Raising of the exponent
Parentheses ( )	Any form of parentheses, brackets, braces
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign

DESCRIPTION OF THE REFERENCE LANGUAGE

> Was sich überhaupt sagen lässt, lässt
> sich klar sagen; und wovon man nicht
> reden kann, darüber muss man schweigen.
> LUDWIG WITTGENSTEIN.

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets **begin** and **end** to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an

a program the following "comment" conventions hold:

The sequence of basic symbols:	*is equivalent to*
; **comment** ⟨any sequence not containing ;⟩;	;
begin comment ⟨any sequence not containing ;⟩;	**begin**
end ⟨any sequence not containing **end** or ; or **else**⟩	**end**

By equivalence is here meant that any of the three structures shown in the left-hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4. Identifiers

2.4.1. Syntax

⟨identifier⟩ ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩⟨digit⟩

2.4.2. Examples

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3. Semantics

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

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.5. Numbers

2.5.1. Syntax

⟨unsigned integer⟩ ::= ⟨digit⟩|⟨unsigned integer⟩⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩|+⟨unsigned integer⟩|
−⟨unsigned integer⟩
⟨decimal fraction⟩ ::= .⟨unsigned integer⟩
⟨exponent part⟩ ::= $_{10}$⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩|⟨decimal fraction⟩|
⟨unsigned integer⟩⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩|⟨exponent part⟩|
⟨decimal number⟩⟨exponent part⟩
⟨number⟩ ::= ⟨unsigned number⟩|+⟨unsigned number⟩|
−⟨unsigned number⟩

2.5.2. Examples

0	−200.084	$-.083_{10}-02$
177	$+07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
+0.7300	$2_{10}-4$	$+_{10}+5$

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types

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

2.6. Strings

2.6.1. Syntax

⟨proper string⟩ ::= ⟨any sequence of basic symbols not containing ' or '⟩|⟨empty⟩
⟨open string⟩ ::= ⟨proper string⟩|'⟨open string⟩'|
⟨open string⟩⟨open string⟩
⟨string⟩ ::= '⟨open string⟩'

2.6.2. Examples

'5k,,−'[[['/\=/:'Tt''
'.. This ␣ is ␣ a ␣ 'string''

2.6.3. Semantics

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

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

2.7. Quantities, Kinds and Scopes

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

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8. Values and Types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

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

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩ ::= ⟨arithmetic expression⟩|⟨Boolean expression⟩|
⟨designational expression⟩

3.3.2. Examples

Primaries:

$7.394_{10}-8$
sum
$w[i+2,8]$
$cos(y+z\times 3)$
$(a-3/y+vu\uparrow 8)$

Factors:

$omega$
$sum\uparrow cos(y+z\times 3)$
$7.394_{10}-8\uparrow w[i+2,8]\uparrow(a-3/y+vu\uparrow 8)$

Terms:

U
$omega\times sum\uparrow cos(y+z\times 3)/7.394_{10}-8\uparrow w[i+2,8]\uparrow (a-3/y+vu\uparrow 8)$

Simple arithmetic expression:

$U-Vu+omega\times sum\uparrow cos(y+z\times 3)/7.394_{10}-8\uparrow w[i+2,8]\uparrow (a-3/y+vu\uparrow 8)$

Arithmetic expressions:

$w\times u-Q(S+Cu)\uparrow 2$
if $q>0$ **then** $S+3\times Q/A$ **else** $2\times S+3\times q$
if $a<0$ **then** $U+V$ **else if** $a\times b>17$ **then** U/V **else if** $k\neq y$ **then** V/U **else** 0
$a\times sin(omega\times t)$
$0.57_{10}12\times a[N\times(N-1)/2,\ 0]$
$(A\times arctan(y)+Z)\uparrow(7+Q)$
if q **then** $n-1$ **else** n
if $a<0$ **then** A/B **else if** $b=0$ **then** B/A **else** z

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4. Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4. Operators and types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer** (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators $+$, $-$, and $\times$ have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2. The operations ⟨term⟩/⟨factor⟩ and ⟨term⟩ ÷ ⟨factor⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

$$a/b\times 7/(p-q)\times v/s$$

means

$$((((a\times(b^{-1}))\times 7)\times((p-q)^{-1}))\times v)\times(s^{-1})$$

The operator / is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator ÷ is defined only for two operands both of type **integer** and will yield a result of type **integer**, mathematically defined as follows:

$$a\div b=\ sign\ (a/b)\times entier(abs(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation ⟨factor⟩↑⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

$2\uparrow n\uparrow k$ means $(2^n)^k$

while

$2\uparrow(n\uparrow m)$ means $2^{(n^m)}$

Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

$a\uparrow i$ If $i>0$, $a\times a\times\ \ldots\ \times a$ (i times), of the same type as a.
If $i=0$, if $a\neq 0$, 1, of the same type as a.
if $a=0$, undefined.
If $i<0$, if $a\neq 0$, $1/(a\times a\times\ \ldots\ \times a)$ (the denominator has $-i$ factors), of type **real**.
if $a=0$, undefined.

$a\uparrow r$ If $a>0$, $exp(r\times ln(a))$, of type **real**.
If $a=0$, if $r>0$, 0.0, of type **real**.
if $r\leqq 0$, undefined.
If $a<0$, always undefined.

3.3.5. Precedence of operators

The sequence of operations within one expression is

Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression

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

3.5.5. Unsigned integers as labels

Unsigned integers used as labels have the property that leading zeros do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound Statements and Blocks

4.1.1. Syntax

⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩|⟨go to statement⟩|⟨dummy statement⟩|⟨procedure statement⟩
⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩|⟨label⟩:⟨basic statement⟩
⟨unconditional statement⟩ ::= ⟨basic statement⟩|⟨compound statement⟩|⟨block⟩
⟨statement⟩ ::= ⟨unconditional statement⟩|⟨conditional statement⟩|⟨for statement⟩
⟨compound tail⟩ ::= ⟨statement⟩ **end** |⟨statement⟩ ; ⟨compound tail⟩
⟨block head⟩ ::= **begin**⟨declaration⟩|⟨block head⟩ ; ⟨declaration⟩
⟨unlabelled compound⟩ ::= **begin** ⟨compound tail⟩
⟨unlabelled block⟩ ::= ⟨block head⟩ ; ⟨compound tail⟩
⟨compound statement⟩ ::= ⟨unlabelled compound⟩|⟨label⟩:⟨compound statement⟩
⟨block⟩ ::= ⟨unlabelled block⟩|⟨label⟩:⟨block⟩
⟨program⟩ ::= ⟨block⟩|⟨compound statement⟩

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

L: L: ... **begin** S ; S ; ... S ; S **end**

Block:

L: L: ... **begin** D ; D ; .. D ; S ; S ; ...S ; S **end**

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

```
a := p+q
go to Naples
START: CONTINUE: W := 7.993
```

Compound statement:

```
begin x := 0 ; for y := 1 step 1 until n do
        x := x+A[y] ;
      if x>q then go to STOP else if x>w-2 then
        go to S ;
      Aw: St: W := x+bob end
```

Block:

```
Q: begin integer i, k ; real w ;
     for i := 1 step 1 until m do
     for k := i+1 step 1 until m do
     begin w := A[i, k] ;
           A[i, k] := A[k, i] ;
           A[k, i] := w end for i and k
end block Q
```

4.1.3. Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets **begin** and **end** enclose that statement. In this context a procedure body must be considered as if it were enclosed by **begin** and **end** and treated as a block.

Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.2. Assignment Statements

4.2.1. Syntax

⟨left part⟩ ::= ⟨variable⟩ := |⟨procedure identifier⟩ :=
⟨left part list⟩ ::= ⟨left part⟩|⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩|⟨left part list⟩⟨Boolean expression⟩

ment following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

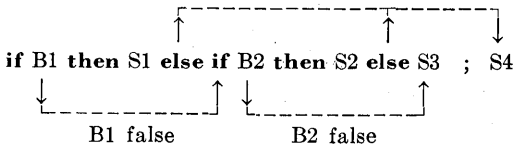
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

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

For further explanation the following picture may be useful:



4.5.4. Go to into a conditional statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6. For Statements

4.6.1. Syntax

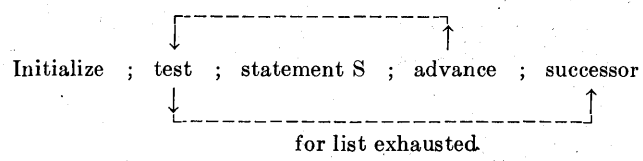
⟨for list element⟩ ::= ⟨arithmetic expression⟩|
⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until** ⟨arithmetic expression⟩|⟨arithmetic expression⟩ **while** ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩|⟨for list⟩ , ⟨for list element⟩
⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for list⟩ **do**
⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩|
⟨label⟩:⟨for statement⟩

4.6.2. Examples

for q := 1 **step** s **until** n **do** $A[q] := B[q]$
for k := 1, $V1\times2$ **while** $V1<N$ **do**
for j := $I+G$, L, 1 **step** 1 **until** N, $C+D$ **do**
$A[k,j] := B[k,j]$

4.6.3. Semantics

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



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

4.6.4. The for list elements

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

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

V := A ;
L1: **if** (V−C)× *sign*(B)>0 **then go to** *element exhausted*;
statement S ;
V := V+B ;
go to *L1* ;

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

4.6.4.3. While-element. The execution governed by a for list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional Algol statements as follows:

L3: V := E ;
if ¬F **then go to** *element exhausted* ;
Statement S ;
go to *L3* ;

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

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

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.7. Procedure Statements

4.7.1. Syntax

⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨array identifier⟩|⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩ ::= ⟨letter⟩|⟨letter string⟩⟨letter⟩

same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code

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

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin**, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a go to statement) all identifiers which are declared for the block lose their local significance.

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

Syntax.

⟨declaration⟩ ::= ⟨type declaration⟩|⟨array declaration⟩|
⟨switch declaration⟩|⟨procedure declaration⟩

5.1. TYPE DECLARATIONS

5.1.1. Syntax

⟨type list⟩ ::= ⟨simple variable⟩|
⟨simple variable⟩ , ⟨type list⟩
⟨type⟩ ::= **real** | **integer** | **Boolean**
⟨local or own type⟩ ::= ⟨type⟩|**own** ⟨type⟩
⟨type declaration⟩ ::= ⟨local or own type⟩⟨type list⟩

5.1.2. Examples

integer *p,q,s*
own Boolean *Acryl,n*

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

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

For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. ARRAY DECLARATIONS

5.2.1. Syntax

⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩:⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩|⟨bound pair list⟩,⟨bound pair⟩
⟨array segment⟩ ::= ⟨array identifier⟩[⟨bound pair list⟩]|
⟨array identifier⟩,⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩|⟨array list⟩,⟨array segment⟩
⟨array declaration⟩ ::= **array** ⟨array list⟩|⟨local or own type⟩
array ⟨array list⟩

5.2.2. Examples

array *a, b, c*[7:*n*,2:*m*], *s*[−2:10]
own integer array *A*[**if** *c*<0 **then** 2 **else** 1:20]
real array *q*[−7:−1]

5.2.3. Semantics

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

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions

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

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

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

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

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. How-

block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6. Code as procedure body

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

Examples of Procedure Declarations:

EXAMPLE 1.

```
procedure euler (fct, sum, eps, tim) ; value eps, tim ;
integer tim ; real procedure fct ; real sum, eps ;
comment euler computes the sum of fct(i) for i from zero up to
infinity by means of a suitabley refined euler transformation. The
summation is stopped as soon as tim times in succession the abso-
lute value of the terms of the transformed series are found to be
less than eps. Hence, one should provide a function fct with one
integer argument, an upper bound eps, and an integer tim. The
output is the sum sum. euler is particularly efficient in the case
of a slowly convergent or divergent alternating series ;
begin integer i, k, n, t ; array m[0:15] ; real mn, mp, ds ;
i := n := t := 0 ; m[0] := fct(0) ; sum := m[0]/2 ;
nextterm: i := i+1 ; mn := fct(i) ;
        for k := 0 step 1 until n do
            begin mp := (mn+m[k])/2 ; m[k] := mn ;
              mn := mp end means ;
        if (abs(mn)<abs(m[n]))∧(n<15) then
            begin ds := mn/2 ; n := n+1 ; m[n] :=
              mn end accept
        else ds := mn ;
        sum := sum + ds ;
        if abs(ds)<eps then t := t+1 else t := 0 ;
        if t<tim then go to nextterm
end euler
```

EXAMPLE 2.[8]

```
procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi) ; value x,y ;
integer n ; Boolean fi ; real x,eps,eta,xE ; array
y,yE ; procedure FKT ;
comment: RK integrates the system y_k'=f_k(x,y_1, y_2, ... , y_n)
(k=1,2, ... ,n) of differential equations with the method of Runge-
Kutta with automatic search for appropriate length of integration
step. Parameters are: The initial values x and y[k] for x and the un-
known functions y_k(x). The order n of the system. The procedure
FKT(x,y,n,z) which represents the system to be integrated, i.e.
the set of functions f_k . The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the
integration interval xE. The output parameter yE which repre-
sents the solution at x=xE. The Boolean variable fi, which must
always be given the value true for an isolated or first entry into
RK. If however the functions y must be available at several mesh-
points x_0 , x_1 , ... , x_n , then the procedure must be called repeat-
edly (with x=x_k , xE=x_k+1, for k=0, 1, ... , n-1) and then the
later calls may occur with fi=false which saves computing time.
The input parameters of FKT must be x,y,n, the output parameter
z represents the set of derivatives z[k]=f_k(x,y[1], y[2], ... , y[n])
for x and the actual y's. A procedure comp enters as a nonlocal
identifier ;
begin
  array z,y1,y2,y3[1:n] ; real x1,x2,x3,H ; Boolean out ;
  integer k,j ; own real s,Hs ;
  procedure RK1ST(x,y,h,xe,ye) ; real x,h,xe ; array
          y,ye ;
    comment: RK1ST integrates one single RUNGE-KUTTA
          with initial values x,y[k] which yields the output
          parameters xe=x+h and ye[k], the latter being the
          solution at xe. Important: the parameters n, FKT, z
          enter RK1ST as nonlocal entities ;
    begin
      array w[1:n], a[1:5] ; integer k,j ;
      a[1] := a[2] := a[5] := h/2 ; a[3] := a[4] := h ;
      xe := x ;
      for k := 1 step 1 until n do ye[k] := w[k] := y[k] ;
      for j := 1 step 1 until 4 do
      begin
        FKT(xe,w,n,z) ;
        xe := x+a[j] ;
        for k := 1 step 1 until n do
        begin
          w[k] := y[k]+a[j]×z[k] ;
          ye[k] := ye[k] + a[j+1]×z[k]/3
```

[8] This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step-by-step integration of differential equations in an automatic computing machine, [*Proc. Camb. Phil. Soc. 47* (1951), 96]; and E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, [*Fysiograf. Sällsk. Lund, Förhd. 20*, 11 (1950), 136–152]. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

END OF THE REPORT

APPENDIX 2

Capitalization Symbol ⊕

Lower case letters are transliterated into capital letters. Capital letters are transliterated into a special capitalizing symbol followed by the capital letter. The capitalizing symbol is punched on cards by multiple punching of + and 0 . It is printed on the card, usually, as a + inside a 0 . When the card is reproduced, the printing symbol is a stylization of a steer's head. It is printed on the 407 as 0 on the main computer's printer as + and on the 1401 printer under 930 program control as a 2 and , superimposed. It prints as a / on the sts terminals (IBM 1050).

APPENDIX 3

Input and Output Using FORTRAN Format

The eleven procedures described in this chapter allow for input and output governed by FORMAT's nearly as in FORTRAN programs. The procedures are available as predeclared procedures; two of them, READ and PRINT, do not adhere strictly to the rules of ALGOL 60 since they can be called with any number of parameters.

1. The procedure FORMAT

FORMAT is called with two actual parameters; the first must be an integer variable and the second must be a string which is a FORTRAN Format (see [3]) with the following conventions:

The Format written between the string quotes includes the outer parenthesis of the FORTRAN Format.

The Format character for integers is J, not I.

The Format character for Booleans is A5.

The call causes the Format string to be assigned to (stored in) the variable given as the first parameter and then later calls of READ or PRINT can use this format, see below. Naturally, the value of the variable used as the first parameter should not be changed by other assignments before the READ/PRINT statements referring to this variable.

2. The procedures READ and PRINT

READ and PRINT allow for an arbitrary number (≥ 1) of actual parameters. The first may be an integer variable containing the format wanted (must be assigned in a previous call of FORMAT), or it may be a string with the format itself.

In a call of READ the following parameters must be names of simple or subscripted variables of any type to which values will be assigned according to the format and the data cards read. Each call of READ initiates the reading of a new data card.

In a call of PRINT the following parameters may be expressions of any type, and the value of each expression is evaluated and printed according to the format. Each call of PRINT causes printing to start on a new line.

Reading or printing is continued until the list of variables or expressions is exhausted; if the list of specifications in the format is exhausted before this, the format list is repeated as described in [3].

It is impossible to read or print all elements of an array just by giving the array identifier as a parameter.

The conversion between decimal and binary representations are carried out by means of the input output routines in the FORTRAN Monitor System, see [3]. If the type of the actual parameter does not fit with the corresponding part of the format, the outcome may be a nonsense value but no warning is given.

Example 1

```
begin integer i, j, fmt, fmt 3; Boolean b;
      FORMAT (fmt, '(2F5, A5)');
      FORMAT (fmt 3, '(1H0, F4, F7, A5)');
      READ (fmt, i, j, b);
      PRINT (fmt 3, j, (i+j)*2, b);
end
```

Example 2

```
begin integer fmt, i; array A[1:20];
      for i := 1 step 1 until 20 do
            READ ('(F10.5)', A[i]);
      FORMAT (fmt, '(2E15.5)');
      for i := 1 step 2 until 20 do
            PRINT (fmt, A[i], A[i+1]);
end
```

This program will read 20 numbers from 20 cards and print them out with 2 numbers per line.

3. Supplementary Procedures for Input

The following 5 procedures may be used for reading several values, one at a time, according to a specified format:

READING FORMAT with one parameter, a string or an integer variable, supplies the format or the reference to the format which governs the following data input by any of the three procedures below. If READING FORMAT is called with an integer variable as the parameter, a previous call of FORMAT must have assigned a format to that variable.

The real procedure READING REAL (without parameters) has as its (real) value the next item read from the data cards according to the format defined by the last call of READING FORMAT.

The integer procedure READING INTEGER and the Boolean procedure READING BOOLEAN correspondingly reads the next integer or Boolean value.

The procedure READING EXIT (without parameters) must be used to signal the termination of input with the given format using the above procedures, see the note and an example in 4 below.

Each call of READING FORMAT starts input from a new data card but apart from that the card control is governed by the specified format.

4. Supplementary Procedures for Output

The following 3 procedures may be used for printing many values, one at a time, according to a specified format:

PRINTING FORMAT with one parameter sets up the wanted format for output, exactly as does READING FORMAT for input.

SAVE is called with one parameter which may be an expression of any type. Each call sets aside the value of the parameters and all these values will be printed by one subsequent call of the procedure WRITE (without parameters). Each call of PRINTING FORMAT starts printing on a new line but otherwise the line control is governed by the specified format.

Use of the procedures described in 3 and 4 adheres furthermore to the following

NOTE:

READING FORMAT activates the input mechanism.

READING EXIT deactivates the input mechanism.

PRINTING FORMAT activates the output mechanism.

WRITE deactivates the output mechanism.

Only one mechanism may be activated at a time.

No dumps or other input/ouput procedures may be called while a mechanism is activated as above.

At most calls of SAVE may be made before WRITE is called.

Example 3

```
begin integer i, data; array x [1 : 100];
     format (data, '(F10, E20.8)');
     READING FORMAT (data);
     for i := READING INTEGER while i ≤ 100 do
                x [i] := READING REAL;
     READING EXIT;
     PRINTING FORMAT ('(5E20.8)');
     for i := 2 step 2 until 100 do  save (x[i]);
     WRITE;
end
```

This program will read pairs of one integer and one real number from each data card (as long as the integer is ≤ 100), and then print 50 numbers in 10 lines with 5 numbers per line.

APPENDIX 4

Tape Unit Correspondence Table

Logical No.	Unit	Normal ALGOL Use
-3	A3	PRINTER
-2	B4	PUNCH
-1	A2	READER
0	illegal	illegal
1	B1	
2	B2	
3	B3	
4	A4	Scratch
5	A5	library tape
6	B5	library tape
7-16	See BC Users Manual	

APPENDIX 5

Semicolon Trace

The semicolon trace is useful for debugging programs which execute but fail to give the expected results.

The comment control "comment: semicolon trace;" (or, alternately "comment: semicolon trace := <constant>;") should appear before the first begin. It sets up the semicolon trace by numbering all semicolons, and printing, after each line in the listing, the number of the first semicolon on that line.

The comment control "comment: on semicolon trace;" turns on the trace: as each semicolon is reached during execution, the number of that semicolon is printed out. Up to 2^{15}-1 semicolons may be printed, unless the alternate form "comment: semicolon trace := <constant>;" was used, whereupon the constant replaces 2^{15}-1 as the limit.

The comment control "comment: off semicolon trace;" turns off the trace. The trace may be turned on and off any number of times during a program.

REFERENCES

1. Users Manual. Computer Center, University of California, Berkeley. First revision, September 1965.

2. P. Naur (ed.): Revised Report on the Algorithmic Language ALGOL 60. Regnecentralen, Copenhagen 1962.

3. IBM Reference Manual. 709/7090 FORTRAN Programming System. Form No. C 28-6054.

210 West Union Street, #20
Fullerton, California

October 10, 1966

Computer Center
University of California
Room 201, Campbell Hall
Berkeley, California

Gentlemen:

I have read with interest the BC ALGOL Manual sent to me. I note that the contributors to the manual have been given credit at the beginning of the manual, however, my name has been omitted. I would appreciate it if you could include my name among those of the authors when the manual is revised.

Sincerely,

Tom Marlin