CHAPTER 3

Input/Output Statements

3a. Introduction

The official ALGOL-60 language does not include input/output statements. Thus, ALGOL-60 can be used to describe computational algorithms but not the process of reading input data from punched cards, magnetic tape or disc, or the process of outputting intermediate and final answers onto printed pages, punched cards, magnetic tape or disc. Each ALGOL translator, therefore, must contain its own scheme for programming input and output operations.

ALGOL-20 includes an input/output ("I/O") system derived from the system used previously in the GATE language at Carnegie Tech.[1] The following pages contain both an introductory explanation and a complete technical description of ALGOL-20 statements for reading data cards and for printing and punching answers.

Chapter 3b is a primer on ALGOL-20 I/O which takes a particular example of printed output and builds up its solution. It is introductory in nature, and concerns only printing. Punching requires only simple extensions of the concepts used in printing. Chapter 3c is a primer on READ which includes a completely worked-out example. Chapter 3d contains a complete summary of all input/output instructions.

ALGOL-20 also contains provision for reading and updating files of information stored on magnetic tape or disc. This mechanism is related to the card reading, printing and card punching statements, but involves additional complexity. It is described separately in Chapter 6g.

---

1 The GATE input/output system is described in the manual: "20-GATE: Algebraic Compiler for the Bendix G-20", Carnegie Tech Computation Center, September 1962. The general principles of the ALGOL-20 input/output system were the subject of a paper presented by A. J. Perlis at the Working Conference on Mechanical Language Structures, August, 1963, published in Comm. A.C.M., 7 (Feb. 1964) p. 89.

CHAPTER 3b

Primer on ALGOL-20 Input/Output

Consider the task of programming a computer to print answers. To control printing, such a program must specify two distinct kinds of information:

(1) Which values are to be printed, and

(2) The format in which the values are to appear on the page.

To supply these two kinds of information, ALGOL-20 contains two types of statements: NAME statements, which select the values to be printed, and PRINT statements, which specify the printed format for these values. "NAME" and "PRINT" are reserved identifiers in ALGOL-20. In general, each NAME statement is paired with a PRINT statement and the two are used in parallel to control printing; each value specified by the NAME statement must be matched with a format specification from the PRINT statement.

The remainder of Chapter 3b is divided into sections, as follows:

A. The NAME Statement: Introduction

B. The Format Program: Introduction

C. The Print Buffer

D. An Example of Print Format

E. Replicators: Introduction

A. The NAME Statement: Introduction

A NAME statement in ALGOL-20 has the following form: The reserved identifier NAME followed by a pair of parentheses enclosing a name list. For printing (or punching), the name list is a list of values to be output and therefore is simply a list of arithmetic expressions (separated by commas):

NAME (< Arith Expr >,..., < Arith Expr >)

When a value is needed by a PRINT statement, the value of the next expression in the NAME list is computed and supplied to the appropriate PRINT instruction. Expressions in the NAME list are evaluated in left to right order, and the corresponding values are printed in the formats specified by the PRINT instructions.

For example, to print the values of the ALGOL variables A, B and C and also the value of the expression $\sqrt{B^2 - 4AC}$, the programmer may use the NAME statement:

```
NAME  (A,B,C,SQRT(B ↑ 2 - 4*A*C) )
```

along with an appropriate PRINT statement.

NAME statements may be more complicated. For example, they may contain for clauses and other forms of <u>replicators</u> which repeat the selection of values in a manner analogous to the repeated execution of an ALGOL statement by an ALGOL for clause. Replicators are discussed in Section E.

B. The Format Program: Introduction

Suppose that the value 1.7 has been computed and is to be printed by an ALGOL program. This number could be printed in any one of many different formats; for example, one of the following forms might be appropriate in a specific case:

1.7 +1.7 +00001.700 .170 $_{10}$+01 1.70 $_{10}$+00 17000 $_{10}$-04

However, there is more to format control than specification of the forms of individual numbers. Answers are generally to be printed in a readable manner: separated by blank columns and accompanied by suitable headings and titles to identify the printed results. Therefore, a PRINT statement must give the programmer control over the position of each number and title on the line, the assignment of numbers to different lines, the spacing of printed lines on the page, and the sequencing of pages, as well as the form of numbers.

To control all these aspects of format, ALGOL-20 contains a special "format language", which is used within PRINT statements. A series of instructions in this format language forms a format program. The individual instructions within a format program are separated by commas.

The format language uses some of the same characters that ALGOL uses, but with different meanings. Therefore, special brackets must be placed around each format program to set it apart from the ALGOL program in which it is embedded. Unfortunately, there are no unused symbols available in the G-20 alphabet for these format brackets, so we use "<" (less than) and ">" (greater than) for this purpose. The syntax of a PRINT statement is such that "<" and ">" symbols surrounding format programs cannot be confused with the same symbols in Boolean expressions.

The simplest form which a PRINT statement may have is the reserved word PRINT followed by a pair of parentheses which enclose a single format program, or enclose a series of format programs separated by commas. Each format program is itself enclosed in "<" and ">" brackets. The following PRINT statement, for example, contains a single format program which consists of five format instructions:

```
PRINT ( < P, 37C, 'A=', + 2D.3Z, 2E >)
```

The meanings of these instructions will be explained below. The effect of this PRINT statement would not be changed if each format instruction were enclosed in format brackets, so that the PRINT statement contained five format programs each consisting of a single format instruction:

```
PRINT (< P >, < 37C >, < 'A=' >, < +2D.3Z >, < 2E >)
```

C. The Print Buffer

Associated with the G-20 printer is a block of 120 consecutive cells in memory, called the print buffer. These cells, numbered 1, 2, 3,..., 120, correspond to the 120 physical print positions or "columns" in a line of printing.

The process of printing takes place in two steps: First, a format program in a PRINT statement places the characters to be printed into the print buffer, each character being placed into the cell corresponding to the column in which it is to be printed. In this manner, the format program builds up an "image" of the line to be printed. Second, when the entire line has been formed, a format control instruction must be executed to send all 120 characters from the print buffer to the printer and actually print the line on the paper. The format instruction which is generally used for the latter purpose is 'E', which is mnemonic for Execute. The E instruction prints the image in the print buffer and afterwards automatically "erases" the print buffer (i.e., clears it to 120 blank characters) in preparation for the next line.

The print buffer behaves like other memory cells: Storing a new character into a buffer cell replaces the character which was there previously, while sending a character to the printer to be printed does not (necessarily) erase it from the print buffer. In particular, the control instruction 'W' executes the same printing operation as 'E' but does not erase the buffer afterwards. Thus, the programmer may, if he wishes, save part (or all) of the print image for printing on successive lines.

Associated with the buffer is a pointer called the "character pointer" or "CP". The value of CP is always the number of the print buffer column into which the next character will be stored by a format instruction. As each character is stored, CP is automatically stepped ahead (to the right) by one so that successive characters are stored in left-to-right order into successive cells. Therefore, execution of a format instruction which stores characters into the print buffer automatically leaves CP set to the first column after the last character stored. For example, if CP is

47 and a format instruction stores a number requiring 5 columns, CP will be left at column 52.

Another pointer contains the "left margin" or "LM". The value of LM is the number of the left-most column into which characters may be stored. Execution of the instruction "E" leaves CP reset to the value of LM. (Execution of "W" leaves CP unchanged.) There is also a pointer which contains the "right margin" or "RM" -- the number of the right-most column into which characters may be stored. Initially, LM and RM have the values 1 and 120 respectively. Before each character is stored into the print buffer, a check is made to insure that:

$$LM \leq CP \leq RM$$

If this relation does not hold, an "E" is automatically executed: the characters already in the buffer are printed, the buffer is cleared, and CP is reset to the value of LM. The character is then stored into the buffer. The mechanism for changing LM or RM is explained in Section E of Chapter 3d.

D. An Example of Print Format

A particular print program will now be discussed in detail. Assume that an ALGOL program computes all the values in a 40 x 10 array (40 rows x 10 columns) COEF; these 400 values are to be printed along with a value of a simple variable DELTA. A sample of the desired printing is shown on page AL.3b.7.

The printing begins with a title, "ADJUSTED COEFFICIENT MATRIX", which starts in print position 37 of the first line on the page. The "1" in the next printed line is in column 17, the "2" in column 28, etc. The row numbers, down the left-hand column, are in print positions 6 and 7. Each matrix element occupies nine positions in the printed line and is separated from its neighbors by two blank spaces. The numbers to be printed are all less than 1000 in magnitude, and four digits are to be printed to the right of the decimal point. A minus sign is to be printed

immediately before the first digit if the number is negative. The value of DELTA is to be printed with two significant digits in "scientific notation", with a power of ten, as shown. No sign is to be printed for DELTA. The step by step construction of the necessary NAME and PRINT statements for printing this example follows.

First, consider printing the title. Three different types of formating operations are needed for this purpose:

(1) An instruction is needed to begin printing at the top of a page.
(2) An instruction is needed to indicate that the information is to be printed starting in column 37.
(3) Instructions are needed to specify the information to be printed.

Since the title is a fixed string of alphabetic information, it is convenient to include it entirely in the PRINT statement, with no corresponding value in a NAME statement. In fact, if only fixed information such as a title were to be printed, no NAME statement would be needed with the PRINT statement; this is an important exception to the general rule that NAME and PRINT statements come in pairs.

To specify a title or any other fixed string of alphabetic characters to be printed, we use a format instruction called an alphanumeric string instruction. This is simply the string of characters to be printed, enclosed in quote marks. Such an instruction can thus be used to print any character except the quote mark, since a quote within the string cannot be distinguished from the quote terminating the string. (A special format instruction is provided for printing a quote mark -- see page AL.3d.6) The alphanumeric string instruction used to specify the title is:

'ADJUSTED␣COEFFICIENT␣MATRIX␣␣-␣␣DELTA␣=␣'

(Here and in the sequel we use the symbol "␣" to represent a blank column, where it is necessary to emphasize that a column is to be blank.) Blank is a legitimate alphabetic character, so all blanks appearing in the alphanumeric string instruction will appear as blank columns in the title as printed.

ADJUSTED COEFFICIENT MATRIX - DELTA = 5.0 *-04

	1	2	3	4	5	6	7	8	9	10
1	1.7902	-0.0000	0.0000	0.0008	-0.0000	-0.0000	0.0034	-0.0573	-11.1610	323.8654
2	-0.0001	93.9153	0.0005	7.2796	0.0007	-16.7266	1.7892	-0.0202	-2.6406	0.0003
3	-0.6511	-0.0077	-0.0137	-0.0006	-0.0055	-79.3906	-98.8134	-4.5811	-0.0446	-0.7871
4	-1.6910	0.5956	-0.0025	0.0005	-20.9818	-0.1082	0.1210	1.8640	-0.0102	0.0004
5	-0.1198	-98.2422	-0.0000	0.0001	975.0974	0.0002	-0.0228	-45.1398	-0.0472	-17.3674
6	5.4653	0.5186	1.7492	0.0041	-0.0001	-19.0514	-0.0001	-0.0061	-0.0377	0.0591
7	-5.2393	-21.0519	-0.0180	72.8323	-0.0000	-0.1028	-0.0047	-29.7404	-196.5030	7.9580
8	0.3571	-0.0060	0.0000	-1.9352	777.2626	0.0002	-0.0692	52.0718	-0.0826	0.0000
9	45.4917	-9.7117	0.1306	-0.2042	-0.0002	0.0014	0.1669	0.0013	0.0008	0.0003
10	0.1661	0.1567	0.0002	3.6563	-0.2210	-18.2905	-22.9516	146.4738	0.0001	0.0000
11	0.0012	-0.0022	-143.7867	179.7959	0.0672	-0.0025	368.0110	-594.3002	-1.7935	-7.2878
12	-0.0345	-0.0285	-16.0073	0.0035	-0.0000	-0.0000	-0.0001	0.0003	0.2284	1.7218
13	-2.3244	-0.0100	0.0001	85.3077	0.0000	-0.0041	-0.0000	-0.0094	1.5433	0.0136
14	-0.0004	0.1944	0.2522	-0.0040	-0.0067	110.9960	-3.5136	20.5636	-201.2119	-0.4635
15	-0.1697	-66.2361	0.0017	665.2994	0.0021	0.0000	0.0115	-0.0000	-0.0804	-0.0000
16	0.0004	-481.6797	-0.0232	212.5706	0.0005	0.0954	-2.0125	3.0202	513.5410	0.9989
17	-0.3790	-814.4740	0.1218	462.6379	-19.9396	-46.4429	359.0704	0.0014	-6.7067	0.0001
18	127.0418	-742.4814	-0.0001	0.0004	-925.1035	-3.1928	0.0097	-0.0013	-0.0423	-0.9347
19	-0.0036	427.1130	-0.0659	-0.0003	0.0847	29.2204	-33.6923	-0.0219	-4.2031	-14.3640
20	0.0010	0.2172	0.3396	-3.1782	-0.0000	-0.0226	575.6602	-35.2607	2.2267	0.0000
21	0.4675	0.0039	-92.6274	0.0000	-0.0074	-14.9826	0.5947	0.3146	0.0000	1.2663
22	0.0018	6.5445	0.6704	26.1315	-0.0139	-0.0033	-112.8084	0.0000	263.9816	1.0316
23	320.5485	-0.4436	-212.2199	0.0001	1.1127	0.0805	-219.9632	226.8146	0.0001	-0.0017
24	0.1595	-0.0020	0.0012	-0.0088	77.9889	-0.0002	0.0034	0.3729	-0.0000	0.6368
25	-0.0010	-0.8177	-88.2311	-0.0094	0.0043	-1.0382	-337.7289	0.0079	0.5487	-1.5120
26	0.2388	0.0001	0.0089	-0.0001	535.3546	-0.0089	-13.3044	0.0133	0.0000	0.0467
27	-0.0002	-0.0006	-0.1319	-214.9891	-11.0611	0.0013	75.8273	228.7995	-574.2057	-0.6027
28	0.0378	-26.4622	-0.0000	0.0003	0.0005	0.2778	-0.0994	-0.0268	0.0001	0.0039
29	0.9747	0.5692	-2.4955	198.4666	-907.9086	0.0116	-0.0156	-0.0017	580.9429	-0.0142
30	0.0071	-0.0000	-0.0003	1.1572	1.2592	-484.6675	-0.0000	0.0000	-85.1370	-0.0652
31	0.0035	63.2521	-3.7464	-0.0002	-0.0197	-0.0093	-0.0033	0.0017	-0.0001	0.0004
32	-307.9847	0.0004	-632.2322	139.1301	0.0030	-810.0790	33.7184	-0.0098	0.0005	64.7952
33	0.0009	-0.0983	-496.9972	1.7182	-0.0002	0.1562	138.8615	0.0224	0.0005	0.7802
34	110.3654	0.0000	-0.0079	0.0043	0.0000	-3.9576	-0.0060	0.0002	-0.0002	0.2098
35	0.0003	-488.2567	0.0115	-843.1841	1.4976	0.2209	-0.0420	0.0068	-45.7289	0.0001
36	-0.0204	-0.0001	0.5394	-4.4282	-40.3666	0.0031	0.0000	-136.9766	-0.0018	0.0021
37	0.0002	-0.0000	-165.2746	-457.8515	-0.0001	0.0000	56.5357	0.0142	66.8979	-0.1896
38	21.5920	-0.0054	2.9670	-3.4496	0.0499	-3.6325	-1.2954	0.2277	0.0001	0.0000
39	-0.0000	-0.1920	914.2467	-839.1647	0.0000	-0.0000	378.6081	-0.0000	216.4519	7.7986
40	-26.0042	0.4164	-348.2117	-0.0083	285.7574	-112.9313	-0.1943	-548.3568	8.6184	0.0011

This string is to be stored in the print buffer starting at column 37, so CP must be set to 37 before the alphanumeric string instruction is executed. The format instruction to do this is "37C"; here "C" is mnemonic for "Column". Generally, executing an instruction of the form "nC", where n may be any integer in $1 \leq n \leq 120$, will have the effect of setting CP to column n: $CP \leftarrow n$. The format program <1C, 37R> might also have been used. 1C sets CP to column one, and 37R moves CP 37 columns to the Right. Similarly, nL moves CP n columns to the Left. To summarize:

nC	has the effect	$CP \leftarrow n$
nR	has the effect	$CP \leftarrow CP + n$
nL	has the effect	$CP \leftarrow CP - n$

Therefore, the following format program will set CP to 37, place the 40 characters of the string into print positions 37 to 76 of the print buffer, and then print the buffer:

```
<37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  ', E>
```

This could just as well have been written as three successive format programs by putting brackets around each instruction:

```
<37C>, <'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  '>, <E>
```

but the first form is easier to punch. The instruction necessary to store the value of DELTA into the print buffer is still missing. For reasons which will be discussed later, the appropriate numeric instruction is 1D.1ZL. Further, the title is to be printed at the top of the page. The format instruction used to upspace the paper to the top of the next page is "P". Thus, a complete ALGOL-20 program to print the first line of the example might be

```
NAME(DELTA);  PRINT(<P, 37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  ',
1D.1ZL, E>);
```

Equivalently, the following might be used:

```
PRINT(<P, 37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA =  '>);
NAME(DELTA);  PRINT(<1D.1ZL, E>);
```

Next consider the format for printing the number DELTA and the numbers of the matrix itself. Numeric instructions are those instructions which place numbers into the print buffer; these numbers are values which are obtained from the evaluation within the parallel NAME statement.

A numeric instruction may be regarded as giving a "picture" of the number to be printed. Generally, the following items must be specified to define a number format:

(1) The form for printing the sign, if at all.
(2) The number of places, if any, to the left of the decimal point, and whether leading zeroes are to be inserted or left as blanks.
(3) The decimal point, if any.
(4) The number of places, if any, to the right of the decimal point, and whether trailing zeroes are to be inserted or left as blanks.
(5) The "exponent part" (power of ten), if any.

Items (2), (3) and (4), defining the format of the numeric part of the number without sign or exponent, are specified by the number form portion of a numeric instruction. Item (1), the sign, is specified by the sign part, which is part of the prefix, while item (5) is specified by the suffix of the numeric instruction. The form of a numeric instruction then is given by:

<numeric instruction> ::= <prefix> <number form> <suffix>

(We will see later that the prefix includes, in addition to the sign part, a part which controls the printing of dollar signs.) The number form gives a simple picture of the basic form of the number; as an illustration, the matrix values in the example may be printed with the number form:

3D.4Z

Here "3D" indicates three Digits to the left of the decimal point, with leading zeroes replaced by blanks; the period is a picture of the decimal point which is to be printed; and "4Z" means four digits to the right of the decimal point, with trailing Zeroes printed. For example, the number 3.74 will be printed:

by 3D.4D	in the form	␣␣3.74␣␣
by 3D.4Z	in the form	␣␣3.7400
by 3Z.4Z	in the form	003.7400
by 3Z.4D	in the form	003.74␣␣
by 3Z	in the form	004.
by 3D	in the form	␣␣4

All blanks stored are shown explicitly by␣. Notice in the last two examples that the number was rounded by adding five to the first digit not printed, and then truncating the result. The syntax of number form is as follows:

<number form> ::= <integer part> | <integer part>.|
<integer part>.<fractional part> |.<fractional part>
<integer part> ::= <unsigned integer> D | <unsigned integer> Z
<fractional part> ::= <unsigned integer> D | <unsigned integer> Z

If the integer part (fractional part) appears, at least one digit will be printed before (after) the decimal point. For example, the number zero printed with the numeric primary 3D.2D appears as ' 0.0 '. The total number of digits specified must be less than 15.

In our example, DELTA is to be printed with one digit preceding and one digit following the decimal point, so it may be printed with any one of the following number forms:

1D.1D 1D.1Z 1Z.1D 1Z.1Z

The program which actually printed the sample included 1D.1Z to print DELTA.

The prefix includes the sign part to specify the form for printing the sign of the number. If no sign is to be printed, this part is left empty, as is the case for DELTA. The array elements are to be printed with a minus sign immediately preceding the first significant digit of each negative number. The sign part to use in this case is "-". If in addition plus signs were to be printed before each non-negative number, the prefix "+" would be used instead.

The suffix portion of a numeric instruction is used to supply supplementary information, such as scaling the number, printing an exponent or special spacing. The format for the array elements is completely specified by the prefix and the numeric primary portions, so the proper numeric

instruction is -3D.4Z. DELTA is to be printed in scientific notation: shifted so that the left-most digit is non-zero (if possible) and the resultant exponent printed. The suffix "L" provides such printing, so the numeric instruction 1D.1ZL is to be used to print DELTA.

E. Replicators: Introduction

In principle, everything which is necessary to print the example has now been discussed. However, writing or punching the NAME and PRINT statements for the example using only the NAME and PRINT machinery discussed so far would be very lengthy and tedious. For example, it seems as if the NAME statement would have to be a simple list of all of the 401 variable names DELTA, COEF[1,1], ..., COEF[40,10], while the PRINT statement would have to contain 401 distinct numeric instructions in addition to alphanumeric string instructions and control instructions. What is needed is a "loop" mechanism analogous to the ALGOL for statement; this mechanism is provided by replicators.

An ALGOL program which would operate in some way upon each element of each row of the matrix COEF would presumably have the form of two nested for statements:

```
FOR I ← STEP 1 UNTIL 40 DO
   FOR J ← 1 STEP 1 UNTIL 10 DO
      something with COEF[I,J] ;
```

This is essentially the form which is used in the NAME statement; the "action" to be performed on COEF[I,J] is simply "naming" its value under the control of these FOR clauses. The following NAME statement will supply all 400 values from the array COEF for printing:

```
NAME($ FOR I ← 1 STEP 1 UNTIL 40 DO $
   ($ FOR J ← 1 STEP 1 UNTIL 10 DO $
      (COEF[I,J])));
```

The "$" signs are necessary around a FOR clause when it is used as a replicator in a NAME (or PRINT) statement. Also, the phrase being replicated must be enclosed in parentheses, whether it is only a single expression like (COEF[I,J]) or a complex expression which itself contains a replicator, like:

```
($ FOR J ← ....DO $  (COEF[I,J]))
```

This accounts for the three sets of parentheses in the example above. The following is the syntax of a NAME statement:

<name statement> ::= NAME (<name list>)
<name list> ::= <name list element> | <name list>, <name list element>
<name list element> ::= <name expression> | <replicator> (<name list>)
<name expression> ::= <arithmetic expression> | <Boolean expression> |
    <logic expression>

This syntax shows that any simple or complex list of "names" may be enclosed in parentheses and replicated; such a replicated list may then be a single element in another list. The following legal name statement illustrates lists and replicated lists:

```
NAME ( A[1], $ FOR J ← 1 STEP 2 UNTIL 3 DO $
                (J, A[J], COEF[I,J]), A[7])
```

This example is equivalent to the following more simple statement:

```
NAME (A[1], 1, A[1], COEF[I,1], 3, A[3], COEF[I,3], A[7])
```

As another illustration, refer again to the example, where the row number is to be printed on every line of the matrix. The simplest way to print these numbers is to give their values in the NAME statement and use numeric instructions to place them into the print buffer. Thus, the following NAME statement will supply (in addition to the array value), the row number I just before the first element in each row:

```
NAME ($ FOR I ← 1 STEP 1 UNTIL 40 DO $
            (I, $ FOR J ← 1 STEP 1 UNTIL 10 DO $
                        (COEF[I,J])));
```

Since for clause replicators used in format programs very frequently start at one and increase in steps of one, an abbreviated notation has been provided for this special case. The replicator

<variable> → $ <arithmetic expression> $

has the same meaning as:

$FOR <variable> ← 1 STEP 1 UNTIL <arithmetic expression>$

Therefore, the NAME statement given above for the matrix with row numbers may be written more compactly as:

NAME (I → 40(I,J → 10(COEF[I,J]))) ;

One more simplification is possible in this form; in the special case that the <arithmetic expression> giving the upper limit of replication is a constant (like "40"), or a simple variable (like "N"), it need not be surrounded by "$" signs. Thus, for example, "I → N" is a correct replicator. "I → N-1" is incorrect since dollar signs are required around the arithmetic expression; the correct replicator would be "I → $N-1$".

To print the column headings in the example, the values 1, 2, ..., 10 must be supplied in a NAME statement. The simplest NAME statement for the column headings is:

NAME(I → 10(I)) ;

That is, I runs from 1 to 10, and it is the value of I itself which is to be printed.

The same forms of replicators which are used in NAME statements may also be used to execute repeatedly format programs or lists of format programs in PRINT statements. Thus, instead of writing "<2D, 2D, 2D, 2D>", we may write "J → 4 <2D>". In the case of a replicator in a PRINT statement, however, the actual value of the replicated variable frequently is not referred to; that is, the replicator is used simply as a counter. In such a case, the variable in a " → " replicator may be omitted; thus, " → 4 <2D>" may be used to get four repetitions of the format instruction "2D".

Following is the syntax of replicators:

<replicator> ::= $ <for clause> $ | <simple variable> → <limit> |
→ <limit>

<limit> ::= $ <arithmetic expression> $ | <simple variable> |
<unsigned integer>

Some examples of these forms follow:

```
$ FOR J ← 2, 3, K + 2 STEP 3 WHILE A[K] < K DO $
                J → $  (A[I] / 2) + 3 $
                J → N
                J → 3
                → $ (A[I] / 2) + 3 $
                → N
                → 3
```

If the upper limit of replication has a value such that zero or fewer replications are called for, then the phrase which is being replicated will be skipped entirely.

As an illustration, the NAME and the PRINT statement for printing the column heading of the example are:

```
NAME (I → 10(I));
PRINT (<16C>, → 10 <2D, 9R>, <E>);
```

Notice that the entire format program <2D, 9R> is replicated ten times. The replicators are <u>not</u> part of the format language, and must therefore appear <u>outside</u> the format brackets.

The variable I cannot be omitted from the replicator "I → 10" in the NAME statement, since I is referred to, and is, in fact, the value to be "named". It would definitely have been incorrect to have used the identical notation "I → 10" in our PRINT statement, since the same variable I is already being used for a different replicator in the NAME statement. Horrible confusion will result from using the same variable as a replicator at the same time in both a NAME statement and its parallel PRINT statement.

After the instructions "<16C>, → 10<2D, 9R>" have been executed, CP will be set to print position 126, past the RM of 120. However, this does not cause error printing because the two digits stored on the tenth

replication will be put into positions 115 and 116, and no attempt will be made to store characters in positions greater than RM.

Finally, we set up a PRINT statement for the matrix itself. Notice the extra blank line every five lines. To get this blank line, we need only execute an E instruction while the print buffer contains only blanks. Thus, our PRINT statement will have the form:

PRINT (→ 8(<E>, → 5 (format program for one line)))

The format program for one line could be:

<6C, 2D, 5R>, → 10 <-3D.4Z, 2R>

The entire program for the printed output of the example has now been developed:

```
PRINT (<P, 37C, 'ADJUSTED COEFFICIENT MATRIX  -  DELTA = '>);
NAME (DELTA) ; PRINT (<1D.1ZL, 4E>);
NAME (I → 10(I)); PRINT (<16C>, → 10 <2D, 9R>, <E> );
NAME (I → 40(I,J → 10(COEF [I,J])));
PRINT ( → 8(<E>, → 5 (<6C, 2D, 5R>, → 10 <-3D.4Z, 2R>, <E>)));
```

CHAPTER 3c

Introduction to READ

A useful way to visualize the process of reading alphanumeric information from cards is to consider READ to be the reverse process of PRINT. Recall that in printing, an image was formed in a buffer and then sent to the printer to be printed. In READ, however, the image originates at the input hardware and is then sent to an input buffer which is used by the READ statement in scanning string or numeric values; these are then loaded into variables named by a NAME statement. This buffer has a "CP", "LM", and "RM".

The NAME statement used with READ has the same form as with PRINT except that it supplies the names of variables rather than the values of the variables named. Therefore, the NAME statement used with READ forms a list of ALGOL variables (either simple or subscripted), not general arithmetic expressions, as are allowed with PRINT. Each numeric or alphanumeric instruction assigns a value to successive variables supplied by the NAME statement. Replicators may be used in the READ statement with the same meaning as in a PRINT statement.

The following sequence is incorrect:

```
NAME ( A + B );  READ ( < 3D > )
```

since the NAME statement names an expression which is <u>not</u> a simple or subscripted variable.

The READ format program contains a list of instructions, very similar to those in PRINT, which control the reading of new cards and which specify the location and type of information expected to be found in the READ buffer. Thus, the programmer, by using suitable READ format instructions, is free to arrange his data cards in any format he desires.

The remainder of this chapter is divided into sections:

A. Control Instructions
B. Alphanumeric Instructions
C. Numeric Instructions
D. Card Overflow
E. An Example Using READ

It is assumed that the reader has read Chapter 3b.

A. Control Instructions

Just as the user uses E or W in a print format to control printing, so does he use E or W in read format to control reading.

nE Read n card images into the current READ buffer and set CP to LM. Only the last card image read is available after executing this instruction; hence, "1E" or "E" is the most common use of the instruction.

nW The action is the same as in "nE" except that the card images are also printed on the program listing.

In a READ format program, as opposed to a PRINT program, the E or W is usually the first instruction, rather than the last. The remainder of the format program then controls the scanning of the characters read into the read buffer. As in PRINT, the user has the ability to move CP:

nC	Set CP to column n.	CP ← n
nR	Move CP to the right n columns.	CP ← CP + n
nL	Move CP to the left n columns.	CP ← CP - n
nB	Equivalent to nR.	

B. Alphanumeric Instructions

As in printing, the user has the ability to input any string information with an nA instruction:

nA Scan the next n character positions of the read buffer and store the information there into ↓((n+3)/4) words from a NAME statement. The information is stored four characters per word, with the possible exception of the last word. If the last word does not get four characters, those characters it does get are stored right-justified.

As an example, assume that the characters 'ABCDE␣' appear on a card, with the 'A' in column 15. The effect of executing the statements

NAME(L, M); READ(<15C, 6A>)

will be to store 'ABCD' into L and '␣␣E␣' into M. CP will be left at 21.

Another possibility is to supply fixed string information directly from the READ statement, rather than from the card image. This ability is particularly useful in setting successive elements of an array to contain alphanumeric string information. We have

'<string>' The n characters between the quote marks are stored into ↓((n+3)/4) words from a NAME statement, just as for nA. CP is left unchanged.

Again, an example may be useful. Executing the statements

NAME(I → 5 (A[I])); READ(<'*THIS␣IS␣A␣STRING*'>)

is equivalent to executing

A[1] ← '*THI'; A[2] ← 'S␣IS'; A[3] ← '␣A␣S'; A[4] ← 'TRIN';
A[5] ← '␣␣G*'

The number of characters between the quotes is 18, not a multiple of four. Thus, the last two characters are stored right-justified in the fifth named variable.

The last alphanumeric instruction provides the ability to read Boolean values from a card.

nT The next n columns are scanned, but only the first non-blank column is examined. If it contains 'T', the corresponding name is set to true; otherwise, the corresponding name is set to false. If the corresponding name is not of type Boolean or logic, the error situation "ILLEGAL BOOLEAN" exists and will be treated as described below in Section D of Chapter 3d.

C. Numeric Instructions

Two essentially different methods are provided for reading numbers from cards: fixed field and free field. In the former, the programmer must specify (and therefore he must know) when he writes the program the columns on data cards in which the numbers will be punched. This format information is then part of the compiled program. With free field reading, the programmer specifies in his program only the number of quantities to be read. The numbers may then be punched in any format on the cards, separated by commas. Whether fixed field or free field is selected, however, the same rules govern the actual form of the numbers read. (The distinction between fixed field and free field only has to do with the columns used.) Numbers on data cards obey the same syntax as decimal numbers in program, with one addition: If a "/" is punched before the number, either before or after the sign, the number will be treated as an octal number. If an exponent appears, it will then be treated as an octal power of eight. (In summary: / on data cards is equivalent to 8F in program, but the latter notation is not allowed on data cards. 8L and 8R are also not allowed on data cards.)

Fixed field reading will be described first. For each number, the programmer may specify the following information:

1. Number of columns to be read.
2. Treatment of blank columns. Blanks may either be ignored or may be treated as if they were punched with a zero.
3. Decimal or octal conversion. The programmer may indicate that the number is to be read as an octal rather than a decimal quantity.
4. Scaling. The programmer may indicate that the value read is to be multiplied by a power of ten (or of eight for octal conversion).
5. Alarm suppression. Normally, reading a character other than a digit, +, -, decimal point, / or $_{10}$ will cause an alarm. However, the programmer may suppress this feature and cause such illegal characters to be ignored.

The syntax for a read numeric instruction is as follows:

<read numeric instruction> ::= <unsigned integer> D <read suffix>
<unsigned integer> Z <read suffix> | <int> F
<read suffix> ::= <empty> | <read suffix> <read suffix part>
<read suffix part> ::= H | N | E <integer>
<int> ::= <empty> | <unsigned integer>

The unsigned integer gives the number of columns to be scanned, and may be as large as 127. If D is used, blank columns are ignored, while using Z causes such columns to be treated as though they were punched with a zero. The suffix H causes the number to be treated as an octal quantity, regardless of whether or not a / is punched. A suffix of the form E±n causes the number read to be multiplied by ten (or eight) raised to the ±n power. The suffix N causes illegal symbols to be ignored.

Two error conditions may be detected in reading numbers: ILLEGAL SYMBOL and IMPROPER NUMBER. (A detailed description of error messages in READ is given in Section D of Chapter 3d.) The first indicates that a character other than a digit, +, -, decimal point, / or $_{10}$ has been read. It is this error message which is suppressed by the N suffix. The second message indicates that the number is improperly formed. For example, it may have more than one decimal point, more than one $_{10}$, a decimal point after a $_{10}$,etc.

In the numeric instructions just described, the field width or number of columns to be scanned is specified by "nD" or "nZ" and is fixed. A more flexible type of numeric instruction exists in the form of "nF" or free field read. "nF" specifies that n numbers are to be read and stored into the next n names. Each number field is terminated by a comma, thus allowing the data to be punched without reference to particular card columns. Numbers may be punched in the same forms as for the fixed-field READ and may continue from one card to the next. Blanks are ignored except that if an entire field is blank, the value of the corresponding name is not altered instead of being set to zero.

An "*" may be used in place of a comma to terminate a number field. This will stop the scanning of the card. If fewer than n numbers have been read, the remaining names will be left unaltered as though the corresponding number fields were left blank. For example, executing the statements

NAME(A, B, C, D, E, F); READ(<E, 6F>)

on the data card

$$12.6,\ /14_{10}+5,\ ,\ 0\ *$$

is equivalent to executing the statements

$$A \leftarrow 12.6;\quad B \leftarrow 8F14_{10}+5;\ D \leftarrow 0;$$

It is clear, of course, that these statements leave C, E and F unaltered.

D. Card Overflow

If a READ statement attempts to scan past the right margin, a card overflow situation is said to exist. This situation is not treated as an error, but is taken care of automatically by the system. As soon as an attempt is made to read past the right margin, another card is read into the buffer using either an E or a W, depending which of these the user used last to read a card. CP is then set to LM (as usual), and the character is read from that column.

E. An Example using READ

To illustrate many of the concepts which have been discussed, a complete example follows, programmed in several ways. Assume an array A has been declared

real array A[1:80]

and that values for all 80 elements are to be read from cards. From the programmer's point of view, the simplest way to do this is the sequence

NAME(I → 80 (A[I])); READ(<E, 80F>)

Thus the numbers may be punched, as desired, on as many cards as needed, with successive numbers separated by commas. Assume instead that the data cards are already punched, without commas. Each card contains eight numbers, and each number is punched in nine columns with a column between

numbers whose contents are to be ignored. In this case, the READ statement given above might be replaced by

```
READ( → 10 ( <E>, → 8 <9D, 1R> ))
```

A more interesting possibility is the following: Suppose that the numbers are punched onto 80 cards and that each card has punched in columns 9 and 10 a subscript and between columns 12 and 30 a value. That is, the 80 cards may be placed in any order and the number in columns 9 and 10 indicates into which element of the array the value is to be stored. One way to program this is the following:

```
for i ← 1 step 1 until 80 do
begin NAME (j, A[j]); READ(<E, 9C, 2D, 1R, 19D>) end
```

This sort of construction will work since the code for naming A[j] is not executed until after a value has been read into j. The reader should satisfy himself that the following will also work:

```
NAME( → 80(j, A[j] ); READ( → 80 <E, 9C, 2D, 1R, 19D>)
```

CHAPTER 3d

A Complete Description of ALGOL-20 Input/Output

A. Introduction

Chapter 3d is a complete, detailed description of input/output statements in ALGOL-20. This material is organized to be used for reference rather than for instruction. The user unfamiliar with the concepts involved should read first Chapters 3b and 3c which are primers on printing and reading, respectively.

Chapter 3d is divided into sections, as follows:

A. Introduction
B. NAME Statements and Replicators
C. PRINT and PUNCH Statements
D. READ Statements
E. Buffer Manipulations and "|" - variables
F. Control and Execution of I/O Statements

In the following, the term "format statement" will be used to refer to either a READ statement, a PRINT statement or a PUNCH statement, since the latter three types of statement are used to indicate the format of data. The term "output statement" will be used to refer to a PRINT statement or a PUNCH statement.

B. NAME Statements and Replicators

NAME statements are used to specify values to be output in a print or punch operation or to specify locations into which data is to be stored in a read operation. The NAME statement is not executed directly: instead it becomes active and functions as a list of values or locations which are evaluated when needed by a format statement. To clarify this concept, consider the program segment:

```
I ← 7;  NAME( A[I] );  I ← 12;  PRINT(<3D, E>)
```

The value of A[12] will be printed -- not that of A[7].

Only one NAME statement may be active at any given time. If several NAME statements appear before a format statement, only the last executed NAME statement will be available to the format statement. Hence in the program segment:

```
NAME( A[1] );  NAME( A[2] );
PRINT ( <format list> )
```

only the one element, A[2], is available to the PRINT statement. This topic is discussed in detail in Section F, below.

A replicator is used in a NAME statement to indicate that an expression or list of expressions is to be used repeatedly. The replicator acts on the list of expressions in a manner analagous to a for statement acting on a statement in ALGOL. A replicator appears in one of three forms, the first of which is:

```
$ <for clause> $
```

This replicator causes the replicated name list to be used repeatedly until the for list is exhausted. An example is:

```
$ for I ← 1 step 1 until 3 do $ ( A[I], B[I] )
```

which is equivalent to

```
A[1], B[1], A[2], B[2], A[3], B[3]
```

The second form of replicator is:

```
<simple variable> → $ <arithmetic expression> $
```

This form is equivalent to

```
$ for <simple variable> ← 1 step 1 until <arithmetic expression> do $
```

with one important exception: The <arithmetic expression> is evaluated only once, when the first name is actually called for. If the arithmetic expression is a simple variable or an unsigned constant, the enclosing dollar signs may be omitted. For example:

```
I → N (J → $ 2*I $ (A[I, J]))
```

The third form of replicator is:

```
→ $ <arithmetic expression> $
```

This form functions in a manner similar to the one immediately above, except that the translator creates an internal counter to use in place of the simple variable. This form may be used whenever the controlled variable is not needed in the name list. As in the above form, the dollar signs may be omitted if the arithmetic expression is a simple variable or an unsigned integer. For example, the construction

```
I → N ( → I ( '*' ), A[I] )
```

is equivalent to

```
'*', A[1], '*', '*', A[2], '*', '*', '*', A[3], ..., A[N]
```

Syntax for NAME Statements and Replicators

```
<name statement> ::= NAME ( <name list> )
<name list> ::= <name list element> | <name list> , <name list element>
<name list element> ::= <name expression> | <replicator> ( <name list> )
<name expression> ::= <arithmetic expression> | <Boolean expression> |
        <logic expression>
<replicator> ::= $ <for clause> $ | <simple variable> → <limit> | → <limit>
<limit> ::= $ <arithmetic expression> $ | <simple variable> | <unsigned integer>
```

C. PRINT Statements and PUNCH Statements

Most of the instructions in an output statement serve to control the form and positioning of information as it is entered in the output buffer; hence, it is natural to discuss PRINT and PUNCH statements together. Because the

statements are so similar in function and in order to conserve memory locations, PRINT and PUNCH initially share a common output buffer. This means that storing characters with a PRINT statement alters any information which may have been stored by PUNCH statements, and vice-versa. In addition, PRINT and PUNCH share the same CP, LM, and RM, so that changing CP in PRINT changes it for PUNCH also. Initially CP and LM are set to 1, and RM is set to 120. Characters stored to the right of position 80 are ignored when executing an "E" or "W" instruction in PUNCH. Users may have independent buffers for PRINT and PUNCH by using the methods described in Section F of this chapter.

Instructions appearing in output statements fall into one of three classes: Control instructions to specify the position of information in the output buffer, alpha-numeric instructions to store constant information and alpha-numeric strings, and numeric instructions to specify the form in which numbers are to be stored.

Control Instructions

Associated with the output buffer are three variables: CP, LM and RM, the character pointer, the left margin and the right margin, respectively. CP points to the "next" position in the buffer into which information may be stored. LM and RM refer to the left-most and right-most positions in the buffer into which characters may be stored. The following instructions may be used to set or change CP or to output information:

nC Set CP to position n (column n). That is, CP ← n.

nR Move CP n positions to the right. That is, CP ← CP + n.

nL Move CP n positions to the left. That is, CP ← CP - n.
Moving CP to the left or right with nL or nR does not effect the contents of the positions in the buffer which are passed over.

nE Print (punch) one copy of the contents of the output buffer, output n - 1 blank lines (cards), clear the output buffer to blanks and set

CP to the left margin LM.

nW Print (punch) n identical copies of the output buffer on n successive lines (cards). The output is not cleared and CP is not moved.

P Upspace the paper to the top of the next page. (P is ignored in punch statements.) In general a message will be printed as the first line of the new page giving the date and a page number. The date is printed starting at the left margin in the form ' 04 JUL 64', and the page number is printed in the last ten columns before the right margin in the form 'PAGE nnnn ', where nnnn represents the number of pages printed since the end of compilation, in <4D> format. Printing of the page header is under the control of the programmer. He may restart the page numbering or suppress the header completely. See Section E below for details.

Executing "P" does not disturb the output buffer or CP. nP is treated as 1P or P, for any n.

In the above control instructions, and in the following alphanumeric instructions, n is assumed to be a positive, unsigned integer less than 512. If n is to be one, it may be omitted. For example, "E" is treated as "1E".

Alphanumeric Instructions

Alphanumeric instructions are used for all storage into the output buffer, except for storing of numbers. There is provision for storing strings which appear in the output statement, for storing quote marks, for storing alphanumeric information from a NAME statement, for storing blanks, and for storing Boolean quantities. Whenever a character is stored into the output buffer, it is stored into the position indicated by CP and CP is then incremented by one. However, before the storing is done, a check is made that LM $\leq$ CP $\leq$ RM. If this condition is not met, an "E" is executed and the character is then stored at the LM of the next line.

'<string>' The characters of the string appearing between the quote marks are stored. Any G-20 character except quote may be stored by this instruction.

nQ n quote marks are stored.

nA n alphanumeric characters are stored. These characters come from ↓((n+3)/4) names from a NAME statement. Each name, with the possible exception of the last, supplies four characters to be stored. The characters from the last name are taken from the right end of the word.

An example of an A primary may help. Assume that A[1] and A[2] have been named, containing 'STRI' and '**NG' respectively. Executing <6A> will cause 'STRING' to be stored into the output buffer. Had <7A> been executed instead, 'STRI*NG' would have been stored.

nB n blanks are stored. nB has the same effect as a string instruction with n blanks between the quotes.

nT A Boolean value is stored. The number of characters stored into the output buffer is min(5, n). The characters stored are taken from one of three strings, depending on the value, v, of the next NAME. If v is true, the string used is 'TRUE␣'; if v is false the string is 'FALSE'; and in all other cases the string is 'UNDEF'. (The latter may occur if the NAME is not a Boolean quantity.) The two most useful forms of this instruction are 1T, which stores 'T', 'F' or 'U', and 5T, which stores 'TRUE␣', 'FALSE' or 'UNDEF'.

Numeric Instructions

Numeric instructions are those instructions used to store numbers into the output buffer. Such an instruction may be regarded as giving a "picture" of the number to be stored. It includes the following information, some of which may be omitted if not needed:

(1) Sign control: The sign may be omitted or it may be stored. If the

latter, two more choices are available: Positive numbers may or may not have an explicit plus sign, and the sign may be either left-justified in the field or it may appear just before the left-most digit.

(2) Dollar control: Numbers may be stored as dollar amounts, with the dollar sign either left-justified or just before the left-most printed digit.

(3) Digits to the left of the decimal point: The number of such digits, if any, is specified. Leading zeros may be replaced by blanks.

(4) Decimal point: The decimal point may or may not appear, although if (5) is used it must appear.

(5) Digits to the right of the decimal point: The number of such digits, if any, is specified. Trailing zeros may be replaced by blanks.

(6) Exponent part: Several forms of "floating point" notation are available.

(7) Miscellaneous - the user may specify four more options: The number may be stored decimal or octal; special spacings may be used; alarm output may be suppressed; and the number may be truncated rather than rounded.

The syntax of a numeric instruction is as follows:

<numeric instruction> ::= <prefix> <number form> <suffix>

The prefix contains the specification of items (1) and (2); the number form contains the specification of items (3), (4) and (5); and the suffix contains the specification of items (6) and (7).

Consider first the number form, with the following syntax:

<number form> ::= <integer part> | <integer part> . |
<integer part> . <fractional part> | . <fractional part>
<integer part> ::= <unsigned integer> D | <unsigned integer> Z
<fractional part> ::= <unsigned integer> D | <unsigned integer> Z

Let the integer part be of the form νD or νZ, and the fractional part be of the form ηD or ηZ. If the integer (fractional) part is missing, let ν (η) be zero. Then the number will be stored with ν digits to the left of the decimal point and η digits to the right. If the integer (fractional) part contains a D, leading (trailing) zeros will be replaced by blanks, while the Z form causes such zeros to be stored. If ν (η) is zero, then no digits will

be stored to the left (right) of the decimal point. If ν (η) is non-zero, at least one non-blank character will be stored to the left (right) of the decimal point, even though a zero must be stored where D format would otherwise indicate a blank. The decimal point is stored whenever it is present in the number form. The number is normally rounded by adding five to the first digit to the right of the last digit stored. The sum of ν and η must be less than 15.

The prefix is the specification of sign and dollar sign. The syntax of the prefix is as follows:

```
<prefix> ::= <$ part> <sign part> | <sign part> <$ part>
<$ part> ::= <empty> | L$ | $
<sign part> ::= <empty> | L+ | L- | + | -
```

In both the sign part and the $ part, the presence of "L" indicates left-justified. A sign or dollar sign specified by "L+", "L-" or "L$" will be stored into the output buffer before any digits or blanks, while a sign or dollar sign specified by "+", "-" or "$" will be stored just before the first non-blank digit stored by the number form. The order of storing is as follows:

1. $ specified by "L$"
2. sign specified by "L+" or "L-"
3. blanks from suppressed leading zeros in D-type integer part
4. sign specified by "+" or "-"
5. $ specified by "$"
6. first non-blank character from number form

The sign part specifies one of three possible formats for storing the sign of the number. If it is empty, no sign is stored, even though the number may be negative. If it is "+" or "L+", a sign, either '+' or '-', will be stored, taking one space. If it is "-" or "L-", a '-' will be stored if the number is negative and a blank will be stored otherwise.

The suffix part of a numeric instruction is used to supply supplementary information: scaling of the number, storing the exponent, special spacing and other options. The syntax is as follows:

<suffix> ::= <empty> | <suffix> <suffix element>
<suffix element> ::= L | S <integer> | F <integer> | E <integer> |
H | K | N | T

The various suffix elements are explained below. If an exponent is stored, it takes six positions in the output buffer, in the form: '␣10±dd␣'. No more than one of the suffix elements S, L, F or E should be used on a given suffix.

L The number is left-justified in the field specified by the number form, and the resultant exponent is stored. This is "scientific notation".

E±n The number is shifted so that its exponent equals ±n and the exponent is stored.

F±n The number is shifted so that its exponent equals ±n, but the exponent is not stored.

S±n The number form portion of a numeric instruction containing this suffix must be of the form "<integer part> . ". (The decimal point must appear.) The number is shifted so that its exponent equals ±n. The resultant mantissa is then left-justified in the specified field. The two shifting operations determine the position of the decimal point, which is then inserted where needed. The resulting exponent is stored.

H The number is stored octal rather than decimal. If an exponent is stored, it is to be interpreted as a power of eight.

K One of two special spacings is used in storing the number. If a $ part appears in the prefix, the digits of the number are stored in groups of three, separated by commas. If a $ part does not appear, the digits are stored in groups of five separated by spaces. In either case, the groups are counted left and right from the decimal point. The decimal point, if present, serves as one of the spaces.

N Possible alarm output is suppressed (see the text below), and any digits which overflow the left end of the field are lost.

T The number is truncated after the last stored digit, rather than rounded as usual.

If any format other than L is used, it is possible that the magnitude of the number is such that there are more digits to the left of the decimal point than can be stored using the specified number form. In such a case (providing that the suffix "N" was not used), alarm output will take place with the use of "L" format. If E or S format was called for, no extra spaces will be taken. Otherwise the number will take six more spaces than expected. For example, the number 123 will be stored as 12 $_{10}$+01 by 2D, but as 23 by 2DN.

Examples of Numeric Instructions

The value of the number to be stored is 4673900. The numeric instructions listed on the left side of the page will cause the storing; of the corresponding strings of characters. The numbers at the right indicate the numbers of buffer positions used.

7D	4 6 7 3 9 0 0	7
8D	␣ 4 6 7 3 9 0 0	8
9Z	0 0 4 6 7 3 9 0 0	9
7D.4D	4 6 7 3 9 0 0 . 0 ␣ ␣ ␣	12
7D.4Z	4 6 7 3 9 0 0 . 0 0 0 0	12
8DL	4 6 7 3 9 0 0 0 ␣ $_{10}$ - 0 1 ␣	14
3D.1DL	4 6 7 . 4 ␣ $_{10}$ + 0 4 ␣	11
1Z.2ZE+7	0 . 4 7 ␣ $_{10}$ + 0 7 ␣	10
1Z.2ZF+7	0 . 4 7	4
3D.3ZF+7	␣ ␣ 0 . 4 6 7	7
.3ZF+7	. 4 6 7	4
+7D	+ 4 6 7 3 9 0 0	8
-7D	␣ 4 6 7 3 9 0 0	8
L$+8D	$ ␣ + 4 6 7 3 9 0 0	10
L$-8D	$ ␣ ␣ 4 6 7 3 9 0 0	10
$+8D	␣ + $ 4 6 7 3 9 0 0	10
8Z.K	0 4 6 ␣ 7 3 9 0 0 .	10
L$8D.2ZK	$ ␣ 4 , 6 7 3 , 9 0 0 . 0 0	14
3D.1DF+4	4 6 7 . 4	5
3D.1DF+4T	4 6 7 . 3	5
8Z.S+3	4 6 7 3 . 9 0 0 0 ␣ $_{10}$ + 0 3 ␣	15
8Z.S+4	4 6 7 . 3 9 0 0 0 ␣ $_{10}$ + 0 4 ␣	15
4DN	3 9 0 0	4
4D	4 6 7 4 ␣ $_{10}$ + 0 3 ␣	10 *
8DH	2 1 6 5 0 5 5 4	8
4D.2ZHL	2 1 6 5 . 0 6 ␣ $_{10}$ + 0 4 ␣	13
3ZHNTF+3	6 5 0	3

* Alarm output used.

Syntax for Print and Punch

For the purpose of this syntax, the G-20 characters "<" and ">" will be replaced by "≪" and "≫" , respectively. "<" and ">" will be reserved for meta-linguistic brackets in the Backus Naur Form syntax.

<print statement> ::= PRINT (<format list>)

<punch statement> ::= PUNCH (<format list>)

<format list> ::= <format list element> | <format list> , <format list element>

<format list element> ::= ≪ <format program> ≫ |
<replicator> ≪ <format program> ≫ | <replicator> (<format list>)

<format program> ::= <format instruction> |
<format program> , <format instruction>

<format instruction> ::= <control instruction> | <alphanumeric instruction> |
<numeric instruction>

<control instruction> ::= <int> C | <int> R | <int> L | <int> E | <int> W |
<int> P

<alphanumeric instruction> ::= <string> | <int> B | <int> Q | <int> A |
<int> T

<numeric instruction> ::= <prefix> <number form> <suffix>

<prefix> ::= <$ part> <sign part> | <sign part> <$ part>

<sign part> ::= <empty> | L+ | L- | + | -

<$ part> ::= <empty> | L$ | $

<numeric primary> ::= <integer part> | <integer part> . |
<integer part> . <fractional part> | . <fractional part>

<integer part> ::= <unsigned integer> D | <unsigned integer> Z

<fractional part> ::= <unsigned integer> D | <unsigned integer> Z

<suffix> ::= <empty> | <suffix> <suffix element>

<suffix element> ::= L | H | N | K | T | S <integer> | E <integer> |
F <integer>

<unsigned integer> ::= <digit> | <unsigned integer> <digit>

<integer> ::= <unsigned integer> | + <unsigned integer> | - <unsigned integer>

<int> ::= <empty> | <unsigned integer>

<string> ::= ' <proper string> '

<proper string> ::= <empty> |
<proper sting> <any G-20 character other than quote>

Execution of Print and Punch Statements

From the definitions of PRINT and PUNCH statements, it is evident that the forms of these statements are:

PRINT (fle, fle, ..., fle)
PUNCH (fle, fle, ..., fle)

where "fle" denotes a format list element. The fle's are executed in order of appearance, from left to right. After the rightmost fle is executed, the statement is terminated. Each fle is either a format program bracketed by "< >" and possibly replicated, or a replicated list of fle's, separated by commas. In turn each format program may be a list of format instructions (Eg., "3C, 2Q, E"). These instructions are also executed in left to right order. It should also be noted that no replicators may appear inside the "<" ">" brackets. If a format instruction requires a value, it will cause a call on the corresponding NAME statement and evaluate the next expression to obtain a value.

D. READ Statements

Most instructions in a READ statements are used to scan data which has been read into an input buffer and to store data values into variables which have been named in a NAME statement. As in PRINT and PUNCH, the instructions fall into three classes: control instructions to control the reading of data card images into the buffer and the positioning of CP, alphanumeric instructions to specify the manner in which alphanumeric data is to be scanned and stored into variables, and numeric instructions to specify the manner in which numbers are to be scanned, interpreted and stored into variables.

Control Instructions

Associated with the input buffer are three variables: CP, LM, and RM -- the Character Pointer, the Left Margin, and the Right Margin. CP points to the "next" position in the buffer which is to be scanned. LM and RM refer to the left-most and right-most positions in the buffer which may be scanned. The following instructions may be used to set or change CP:

nC	Set CP to position n (Column n). That is, CP ← n.
nL	Move CP n positions to the Left. That is, CP ← CP - n.
nR	Move CP n positions to the Right. That is, CP ← CP + n.
nB	Equivalent to "nR".

The following two instructions may be used to read data card images into the input buffer:

nE	Read n card images into the current READ buffer, and set CP to LM. At the completion of this instruction, only the last card image read is available to be scanned.
nW	The action is as in "nE", except that the card images are also printed on the program listing.

In the above control instructions, and in the following alphanumeric instructions, n is assumed to be a positive, unsigned integer less than 512. If n is one, it may be omitted. For example, "W" is treated ad "1W".

Alphanumeric Instructions

Alphanumeric instructions are used to scan alphanumeric characters and store string or Boolean values into variables named in a NAME statement:

nA	The next n character positions of the input buffer are scanned, and the string of n characters there is stored, four characters per word, into the next ↓((n + 3)/4) named variables. If n is not a multiple of four, the

characters stored in the last variable are right-justified.

'<string>' The n characters of the string are stored as in "nA". CP is not changed.

nT The next n character positions are scanned and a Boolean value is stored in the next variable named. If the first non-blank character scanned is the letter "T", the value of the variable is set to TRUE; otherwise, it is set to FALSE. CP is incremented by n. If the variable named is not of type Boolean or logic, the error condition "ILLEGAL BOOLEAN" is detected and treated as described below.

Numeric Instructions

READ numeric instructions are either fixed-field of free-field. Fixed-field instructions consist of a primary specifying field width (the number of characters to be scanned) and possibly a suffix specifying additional information, such as scaling or octal conversion.

nD (nZ) The instructions "nD" and "nZ" are used to form READ primaries. "nD" scans the next n character positions of the buffer for a real or integer number and stores it in the corresponding name. Any blanks scanned are ignored, with the exception that if the entire field of n character positions is blank, the value zero is stored. A number preceded by a "/" is treated as an octal (base eight) number. n must be a positive integer less than 128. The instruction "nZ" functions as "nD" except that blanks are treated as zeros. The forms "nD.", "nD.nD" ".nD" and the corresponding Z primaries are not correct in READ.

The suffix of the fixed-field instruction may be empty or may consist of one or more of the following suffix parts:

H — The number is converted in octal (base eight) regardless of whether or not it is preceded by a "/". If the number has an exponent, the exponent is treated as a power of eight.

E±n — The number read is multiplied by ten (or eight) to the power ±n.

N — Any character other than a digit, +, -, decimal point, /, or $_{10}$ is ignored if it is scanned. CP is incremented by one, and the next character is scanned. Normally, scanning any character other than those listed above will result in the detection of the error condition "ILLEGAL SYMBOL".

In the numeric instructions just described, the field width or number of columns to be scanned is specified by "nD" or "nZ" and is fixed. A more flexible type of numeric instruction exists in the form of "nF" or free read:

nF — n numbers are to be scanned and stored into the next n variables named. Numbers may be punched in the same forms as for fixed-field read, and each number field is terminated by a "," or a "*". Blanks are ignored, except that if an entire field is blank, the value of the corresponding variable is left unaltered instead of being set to zero.

A "*" terminates the scanning of the "nF" instruction. If fewer than n numbers have been scanned, the values of the remaining variables named are left unaltered, as though the corresponding number fields were left blank. After execution of "nF", CP points to the character position one position to the right of the last "," or "*" scanned.

Card Overflow

If a READ instruction attempts to scan character positions past the right margin, a new card image is read using a pseudo control instruction. This instruction functions as an "E" or "W" instruction, whichever has been executed most recently. Scanning continues with CP set to LM. Initially, CP = 1, LM = 1, and RM = 84.

Error Messages

Several situations are detected by the input routine as indicating an error by the user, either in his ALGOL I/O call or in his data cards. A standard error printout is provided, containing the following information:

1. The last card read is printed. (If it was read by a W, it will thus be printed twice.) The next line will contain an integer giving the present value of CP and will also have a vertical arrow (↑) pointing to the column indicated by CP. Usually, this will be the column just past the error.

2. A single line is printed identifying the particular error.

3. The standard ALGOL run error mechanism is invoked with RUN ERROR - READ. The following error messages (item 2, above) are detected:

ILLEGAL BOOLEAN An attempt has been made to read with a T instruction into a variable of type other than Boolean or logic.

$$ - CARD READ An attempt has been made to read past an end-of-file mark. Reading more card images than are in the current input file results in reading an end-of-file mark. This mark consists of special dollar signs (internal representation 165_8) in columns one and two. Attempting to read still another card image causes the error condition "$$ CARD READ" to be detected.

NO CARD READ An attempt has been made to scan information before an E or W instruction has loaded the input buffer.

IMPROPER NUMBER In scanning a number with a numeric instruction, an illegal sequence such as more than one decimal point, more than one $_{10}$, or a decimal point after a $_{10}$ has been detected.

ILLEGAL SYMBOL In scanning a number with a numeric instruction, a character other than a digit, +, -, decimal point, / or $_{10}$ has been read. This message is suppressed by the suffix N.

E. Buffer Manipulation and | - variables

As has been mentioned, an input buffer and an output buffer exist in the I/O system. Associated with each buffer are three pointers: CP, LM and RM. It is frequently convenient for the programmer to be able to make direct reference to these buffers instead of being restricted to using format instructions to refer to them. For example, in all that has been said up to this point no mention has been made of any way the programmer can change LM or RM. To permit reference to the various pointers of the I/O system, ALGOL-20 includes a special class of reserved words: the bar-variables. These variables consist of a vertical bar ("|") followed by an integer. The | and the first digit of the integer must be in successive columns of the same card, with no intervening blanks.

The format of a buffer will now be described using the print buffer for definiteness. The buffer itself consists of 120 consecutive locations in memory, corresponding to the 120 columns of the printer. Characters are stored into the buffer by placing the G-20 representation of each character in the corresponding word, right-justified. The three pointers associated with the buffer are stored in the three locations immediately before that containing column one. "Column zero" contains CP, "column -1" contains RM and "column -2" contains LM. Each of these three pointers has a name which is available to the user, the name being a bar-variable. For the print buffer, CP is in |205, RM is in |206 and LM is in |207. Thus the assignment statement

```
|205 ← 5
```

is equivalent to the format statement

```
PRINT(<5C>)
```

Similarly, the programmer may change the right margin by storing into |206 with an assignment statement.

A similar situation exists for the input buffer. 84 consecutive locations are provided for the actual read buffer. Column zero, called |200, contains CP for reading; column -1, |201, contains the read RM; and column -2, |202, contains the read LM.

Since PRINT and PUNCH share a common buffer, it follows that they share a common CP, RM and LM.

The following table may help to clarify the preceding discussion:

Location	Initial Contents	Meaning	
|202	1	LM	READ
|201	84	RM	
|200	1	CP	
next 84 words	--	the buffer	
|207	1	LM	PRINT and PUNCH
|206	120	RM	
|205	1	CP	
next 120 words	--	the buffer	

This gives the programmer convenient access to the three pointers, but it does not provide a way to refer to the words in the buffer. Since it is frequently desirable for the user to have this ability, a means has been provided for the user to cause a buffer to be in his own data area instead of in the I/O system. Again considering PRINT, the user may direct that a particular 123 element array is to be used as the buffer. The system will then use the first three locations of this array as the three pointers and the other 120 locations as the print buffer. Since the array is in the user's memory, he may refer to any column or to any pointer by the ALGOL name he has given it. For example, assume that the declaration

```
logic array BUFF[ -2 : 120 ]
```

has been used and that the procedure call

```
BUFFER.SET ('PRINT', BUFF[0] )
```

has been executed. (BUFFER.SET is a privileged identifier.) Then for any k between one and 120, column k will be in BUFF[k]. CP will be in BUFF[0], RM will be in BUFF[-1] and LM will be in BUFF[-2]. It is important to note that |205, |206 and |207 are specific machine locations and that after executing the above BUFFER.SET call they will no longer contain the pointers.

BUFFER.SET may also be used to change the READ or PUNCH buffer, using the string 'READ' or 'PUNCH' as the first parameter to the procedure. As for PRINT, the second parameter should be an array element which will be set to correspond to "column zero" of the buffer.

Before calling BUFFER.SET, the programmer should be sure that the three pointers he is about to put into effect contain reasonable values. BUFFER.SET only makes one check: it insists that the relationship

$$0 < LM < RM$$

be satisfied. If it is not, LM will be set to one and RM will be set to 84, 120 or 80 for READ, PRINT or PUNCH, respectively.

BUFFER.SET detects two error conditions which are treated as run errors: a first parameter which is not one of the three legal strings allowed, or a second parameter which is not in the user's memory.

There are certain other bar variables associated with the input/output system which are available to the user. |210 and |211 are switches for format and NAME, respectively. At any given time during the running of a program when the user has NAMEd variables which have not yet been printed, |211 will be non-zero. (Its value is the location of a routine which will supply the names to succeeding statements.) If the programmer wishes to cancel the effect of the extra names which have been supplied, he may do so by setting |211 to zero. Similarly, extra format elements which have been supplied may be cancelled by setting |210 to zero. The programmer should under no circumstances set either of these variables to non-zero values, or chaos will result.

|212 and |213 are associated with the message printed at the top of each

page. Whenever the printer is moved to the top of a new page by the execution of a P, the user may have a message and page number printed if he so chooses. The system has been set so that the page numbers will start with page one on the first after the completion of the compilation. If the user does nothing about it, each time a P is executed the first line of the new page will contain on the left the date on which the program was run, and on the right the page number. The page number is calculated by finding out from the monitor the total number of pages which have been printed since the run began and subtracting from this number the contents of |212. The contents of |212 is set on entry to the program to the number of pages used by the compiler in compiling the program. The user may change it at any time if he wishes to alter the page numbering sequence.

|213 controls the message to be printed as part of the page header. If it is negative, no page heading at all will be printed. If it is zero, the date and page number will be printed, as explained above. Positive values should not be used in this location. In the present version |213 > 0 will be treated as suppressing the header, but in planned expansion it will have a different meaning.

|214 is the up-space counter. After each line is printed, the printer is up-spaced the number of lines indicated by |214. This location is set on entry to the program to one, for single spacing. The user may set it to two for double spacing, but other values are not recommended. In particular, setting it to zero saves paper but makes it hard to read the output.

|215 is the left-justify switch. In processing number forms there are certain occasions when either blanks or zeroes will be stored depending on whether the programmer has used D or Z in his format. If a blank would have been stored and if, further, |215 is zero, then no space will be taken in the print line instead of leaving a blank. Thus setting |215 to zero permits the user to get left-justified numbers. |215 is initialized to be non-zero.

These last few bar-variables may be summarized as follows:

|210 NAME switch. $\neq 0 \Rightarrow$ there are names to be processed

|211 format switch. $\neq 0 \Rightarrow$ there are formats to be processed

|212 page count

|213 page header switch. $< 0 \Rightarrow$ suppress; $= 0 \Rightarrow$ print; > 0 (do not use)

|214 upspace counter

|215 left-justify switch. $= 0 \Rightarrow$ left-justify; $\neq 0 \Rightarrow$ don't

F. Control and Execution of I/O Statements

The relationship between NAME and format statements is given in the following description of the execution of an input/output operation.

(1) An execution of a NAME statement sets the name switch, |211, to a positive integer, and sets an internal variable δ to point to the first name expression. Whenever |211 is positive the NAME statement which set it so is said to be active. A NAME statement becomes active as encountered, cancelling any previously active name statement.

When a NAME statement becomes active, a test is made to determine if a format statement is already active ($|210 > 0$). If no format statement is active ($|210 = 0$), control passes to the successor of the NAME statement. If a format statement is active, the first name expression is evaluated and sent to the format instruction pointed to by γ. (See (2).) δ is changed to point to the next name expression, and control passes to the active format statement.

(2) An execution of a format statement sets the format switch, |210, to a positive integer. The format statement is then said to be active. Because PRINT, PUNCH, and READ statements share the switch, at most one format statement may be active at any given time. A format statement becomes active when encountered, cancelling any previously active format statement.

The value (address) of a name expression may be needed during the execution of a format statement. If so, an internal variable, γ, is set to point to the format instruction requesting the value (address), and a test is made to determine if a NAME statement is active ($|211 > 0$). If not ($|211 = 0$), control passes to the successor of the format statement. If a NAME statement is active, control passes to the expression pointed to by δ.

(3) In attempting to evaluate a name expression, a check is made to determine whether δ points to an expression or to the end of the NAME

statement. If δ points to an expression, it is evaluated and δ is set to point to the next name expression (or to the end of the NAME statement), and the value (address) of the expression is sent back to the requesting format instruction. When all name expressions have been evaluated, δ points to the end of the NAME statement. In this case, no expression can be evaluated, and |211 is set to zero indicating that no NAME statement is active. Control is passed to the common successor of the now inactive NAME statement and the active format statement. (See (5).)

(4) After the last format instruction in a format statement is executed, |210 is set to zero, and control is passed to the common successor. (See (5).)

(5) The common successor of an active statement and a statement which has just become inactive is the successor of that statement which was most recently encountered during the execution of the ALGOL program.

To clarify the above points, consider some examples of sequences of input/output operations. In the following, N(P) denotes a NAME statement with P name expressions, F(P) denotes a format statement (PRINT, PUNCH or READ) which requires P values or addresses, S denotes an arbitrary ALGOL statement, and S' denotes any ALGOL statement which is not an input/output statement.

A: N(6); S'; F(6); S;

Executing N(6) sets |211 > 0 and sets δ to point to the first name expression. S' is executed and eventually F(6) is entered. Because N(6) is already active, each request for a value or address will be filled by N(6). When the execution of the last format instruction is complete, F(6) becomes inactive, and S is executed. N(6) is still active, but δ points to the end of statement. In this state, any request for a name expression will render N(6) immediately inactive. A NAME statement followed by a format statement is the simplest and most frequently used sequence.

B: F(5); S'; N(5); S;

B illustrates an alternate sequence, in which the format statement precedes the NAME statement. Executing F(5) sets |210 > 0, but no requests for name expressions can be filled because there is no active NAME statement. γ is set to point to the first requesting format instruction, and S' is executed. When N(5) becomes active, it determines that F(5) is already active.

The first name expression is evaluated and sent to the format instruction indicated by γ. Eventually, the last format instruction in F(5) is executed and F(5) becomes inactive. As in example A, N(5) is still active but any request for a name expression will render it inactive. Control then passes to S.

```
C:      N(4); N(2); F(3); S'; N(1); S;
```

C illustrates a more complex situation which is probably a programming error. N(4) becomes active, but is cancelled by N(2). N(2) and F(3) function as in example A, except that when F(3) requests a third name expression, N(2) becomes inactive. S' is executed and N(1) encountered. N(1) now supplies the requested name expression to F(3) and F(3) becomes inactive, passing control to S. Users should be wary about using sequences such as described in C as it is very easy to produce an error which has repercussions on many other input/output operations in the program. As a safeguard, the name and format switches may be zeroed as described in Section E of Chapter 3d.