A Proposal For Input-Output Conventions In ALGOL 60
A Report of the Subcommittee on ALGOL of the ACM Programming Languages Committee.

S. Gorn, Editor; R. W. Bemer, Asst. Editor, Glossary & Terminology
J. Green, Asst. Editor, Programming Languages
E. Lohse, Asst. Editor, Information Interchange.
D. E. Knuth[*], Chairman
L. L. Bumgarner, P.Z. Ingerman, J.N. Merner
D. E. Hamilton, M.P. Lietzke, D.W. Ross

May 1964

The ALGOL 60 language as first defined made no explicit reference to input and output processes. Such processes appeared to be quite dependent on the computer used, and so it was difficult to obtain agreement on those matters. As time has passed, a great many ALGOL compilers have come into use, and each compiler has incorporated some input-output facilities. Experience has shown that such facilities can be introduced in a manner which is compatible and consistent with the ALGOL language, and which (more importantly) is almost completely machine-independent. However, the existing implementations have taken many different approaches to the subject, and this has hampered the interchange of programs between installations. The ACM ALGOL committee has carefully studied the various proposals in an attempt to define a set of conventions for doing input and output which would be suitable for use on most computers. The present report constitutes the recommendations of that committee.

The input-output conventions described here do not involve extensions or changes to the ALGOL 60 language. Hence they can be incorporated into existing processors with a minimum of effort. The conventions take the form of a set of procedures[1], which are to be written in code for the various machines; this report discusses the function and use of these procedures. The material contained in this proposal is intended to supplement the procedures *in real*, *out real*, *in symbol*, *out symbol* which have been defined by the international ALGOL committee; the procedures described here could, with trivial exceptions, be expressed in terms of these four[2].

The first part of this report describes the methods by which formats are represented; then the calls on the input and output procedures themselves are discussed. The primary objective of the present report is to describe the proposal concisely and precisely, rather than to give

[*]California Institute of Technology, Pasadena, Calif.

[1]Throughout this report, names of system procedures are in lower case, and names of procedures used in illustrative examples are in UPPER CASE.

[2]Defined at meeting of IFIP/WG2.1–ALGOL in Delft during September, 1963. The definition of these procedures is not available for publication at this time.–J. G., Ed.

a programmer's introduction to the input-output conventions. A simpler and more intuitive (but less exact) description can be written to serve as a teaching tool.

Many useful ideas were suggested by input-output conventions of the compilers listed in the references below. We are also grateful for the extremely helpful contributions of F. L. Bauer, M. Paul, H. Rutishauser, K. Samelson, G. Seegmüller, W. L. v. d. Poel, and other members of the European computing community, as well as A. Evans, Jr., R. W. Floyd, A. G. Grace, J. Green, G. E. Haynam, and W. C. Lynch of the USA.

1 Formats

In this section a certain type of string, which specifies the format of quantities to be input or output, is defined, and its meaning is explained.

1.1 Number Formats (cf. ALGOL Report 2.5)

1.1.1 Syntax

Basic components :

```
<replicator> ::= <unsigned integer> | X
<insertion> ::= B | <replicator> B | <string>
<insertion sequence> ::= <empty> | <insertion sequence> <insertion>
<Z> ::= Z | <replicator> Z | Z <insertion sequence> C |
     <replicator> Z <insertion sequence> C
<Z part> ::= <Z> | <Z part> <Z> | <Z part> <insertion>
<D> ::= D | <replicator> D | D <insertion sequence> C |
     <replicator> D <insertion sequence> C
<D part> ::= <D> | <D part> <D> | <D part> <insertion>
<T part> ::= <empty> | T <insertion sequence>
<sign part> ::  = <empty> | <insertion sequence> + |
     <insertion sequence> −
<integer part> ::  = <Z part> | <D part> | <Z part> <D part>
```

Format structures:

```
<unsigned integer format> ::= <insertion sequence> <integer part>
<decimal fraction format> ::= . <insertion sequence> <D part> <T part> |
     V <insertion sequence> <D part> <T part>
<exponent part format> ::= 10<sign part> <unsigned integer format>
<decimal number format> ::= <unsigned integer format> <T part> |
     <insertion sequence> <decimal fraction format> |
     <unsigned integer format> <decimal fraction format>
<number format> ::= <sign part> <decimal number format> |
     <decimal number format> + <insertion sequence> |
     <decimal number format> - <insertion sequence> |
     <sign part> <decimal number format> <exponent part format>
```

Number format	Result from −13.296	Result from 1007.999
+ZZZCDDD.DD	-013.30	+1,008.00
+3ZC3D.2D	-013.30	+1,008.00
-3D2B3D.2DT	-000␣␣013.29	001␣␣007.99
5Z.5D-	13.29600-	1007.99900
‘integer␣part␣’-4ZV	integer␣part␣␣␣-13	integer␣part␣1007
‘,␣fraction’B3D	,␣fraction␣296	,␣fraction␣999
-.5D$_{10}$+2D‘...’	-.13296$_{10}$+02...	.10080$_{10}$+04...
+ZD$_{10}$2Z	-13	+10$_{10}$␣2
+D.DDBDDBDDB$_{10}$+DD	-1.32␣96␣00␣$_{10}$+01	+1.00␣79␣99␣$_{10}$+03
XB.XD$_{10}$-DDD	(depends on call)	(depends on call)

Figure 1: Examples of number formats

Note. This syntax could have been described more simply, but the rather awkward constructions here have been formulated so that no syntactic ambiguities (in the sense of formal hmguage theory) will exist.

1.1.2 Examples

Examples of number formats appear in Figure 1.

1.1.3 Semantics

The above syntax defines the allowable strings which can comprise a “number format.” We will first describe the interpretation to be taken during *output*.

1.1.3.1 Replicators An unsigned integer `n` used as replicator means the quantity is repeated n times; thus `3B` is equivalent to `BBB`. The character `X` as replicator means a number of times which will be specified when the format is called (see Section 2.3.1).

1.1.3.2 Insertions The syntax has been set up so that strings, delimited by string quotes, may be inserted anywhere within a number format. The corresponding information in the strings (except for the outermost string quotes) will appear inserted in the same place with respect to the rest of the number. Similarly, the letter `B` may be inserted anywhere within a number format, and it stands for a blank space.

1.1.3.3 Sign, zero, and comma suppression The portion of a number to the left of the decimal point consists of an optional sign, then a sequence of `Z`'s and a sequence of `D`'s, with possible `C`'s following a `Z` or a `D`, plus possible insertion characters.

The convention on signs is the following: (a) if no sign appears, the number is assumed to be positive, and the treatment of negative numbers is undefined; (b) if a plus sign appears, the sign will appear as + or − on the external medium; and (c) if a minus sign appears, the sign will appear if minus, and will be suppressed if plus.

The letter `Z` stands for zero suppression, and the letter `D` stands for digit printing *without* zero suppression. Each `Z` and `D` stands for a single digit position; a zero digit specified by `Z` will be suppressed, i.e. replaced by a blank space, when all digits to its left are zero. A digit

specified by D will always be printed. Note that the number zero printed with all Z's in the format will give rise to all blank spaces, so at least one D should usually be given somewhere in the format.

The letter C stands for a comma. A comma following a D will always be printed; a comma following a Z will be printed except when zero suppression takes place at that Z. Whenever zero or comma suppression takes place, the sign (if any) is printed in place of the rightmost character suppressed.

1.1.3.4 Decimal points The position of the decimal point is indicated either by the character "." or by the letter V. In the former case, the decimal point appears on the external medium; in the latter case, the decimal point is "implied," i.e., it takes up no space on the external medium. (This feature is most commonly used to save time and space when preparing input data.) Only D's (no Z's) may appear to the right of the decimal point.

1.1.3.5 Truncation On output, nonintegral numbers are usually rounded to fit the format specified. If the letter T is used, however, truncation takes place instead. Rounding and truncation of a number X to d decimal places are defined as follows:

Rounding $10^{-d} \cdot entier(10^d X + .5)$

Truncation $10^{-d} \cdot sign(X) \cdot entier(10^d abs(X))$

1.1.3.6 Exponent part The number following a "10" is treated exactly the same as the portion of a number to the left of a decimal point (Section 1.1.3.3), except if the "D part" of the exponent is empty, i.e. no D's appear, and if the exponent is zero, the "10" and the sign are deleted.

1.1.3.7 Two types of numeric format Number formats are of two principal kinds:

(a) Decimal number with no exponent. In this case, the number is aligned according to the decimal point with the picture in the format, and it is then truncated or rounded to the appropriate number of decimal places. The sign may precede or follow the number.

(b) Decimal number with exponent. In this case, the number is transforlned into the format of the decimal number with its most significant digit nonzero; the exponent is adjusted accordingly. If the nmnber is zero, both the decimal part and the exponent part are output as zero.

If in case (a) the number is too large to be output in the specified form, or if in case (b) the exponent is too large, an overflow error occurs. The action which takes place on overflow is undefined; it is recommended that the number of characters used in the output be the same as if no overflow had occurred, and that as much significant information as possible be output.

1.1.3.8 Input A number input with a particular format specification should in general be the same as the number which would be output with the same format, except less error checking occurs. The rules are, more precisely:

(a) Leading zeros and commas may appear even though Z's are used in the format. Leading spaces may appear even if D's are used. In other words, no distinction between Z and D is made on input.

(b) Insertions take the same amount of space in the same positions, but the characters appearing there are ignored on input. In other words, an insertion specifies only the number of characters to ignore, when it appears in an input format.

(c) If the format specifies a sign at the left, the sign may appear in any Z, D or C position as long as it is to the left of the number. A sign specified at the right must appear in place.

(d) The following things are checked: The positions of commas, decimal points, "10", and the presence of digits in place of D or Z after the first significant digit. If an error is detected in the data, the result is undefined; it is recommended that the input procedure attempt to reread the data as if it were in standard format (Section 1.5) and also to give some error indication compatible with the system being used. Such an error indication might be suppressed at the programmer's option if the data became meaningful when it was reread in standard format.

1.2 Other Formats

1.2.1 Syntax

```
<S> ::= S | <replicator> S
<string format> ::= <insertion sequence> <S> | <string format> <S> |
     <string format> <insertion>
<A> ::= A | <replicator> A
<alpha format> ::= <insertion sequence> <A> | <alpha format> <A> |
     <alpha format> <insertion>
<nonformat> ::= I | R | L
<Boolean part> ::= P | 5F | FFFFF | F
<Boolean format> ::=
     <insertion sequence> <Boolean part> <insertion sequence>
<title format> ::= <insertion> | <title format> <insertion>
<alignment mark> ::= / | ↑ | <replicator>/ | <replietor>↑
<format item 1> ::= <number format> | <string format> |
     <alpha format> | <nonformat> | <Boolean format> | <title format> |
     <alignment mark> <format item 1>
<format item> ::= <format item 1> | <alignment mark> |
     <format item> <alignment mark>
```

1.2.2 Examples

```
↑5Z.5D///
3S'='6S4B
AA'='
```

```
↑R
P
/'Execution.'↑
```

1.2.3 Semantics

1.2.3.1 String format A string format is used for output of string quantities. Each of the S-positions in the format corresponds to a single character in the string which is output. If the string is longer than the number of S's, the leftmost characters are transferred; if the string is shorter, ␣-symbols are effectively added at the right of the string.

The word "character" as used in this report refers to one unit of information on the external input or output medimn; if ALGOL basic symbols are used in strings which do not have a single-character representation on the external medium being used, the result is undefined.

1.2.3.2 Alpha format Each letter A means one character is to be transmitted; this is the same as S-format, except the ALGOL equivalent of the alphabetics is of type integer rather than a string. The translation between the external and internal code will vary from one machine to another, and so programmers should refrain from using this feature in a machine-dependent manner. Each implelnentor should specify the maxinmm number of characters which can be used for a single integer variable. The following operations are undefined for quantities which have been input using alpha format: arithmetic operations, relations except "=" and "≠", and output using a different number of A's in the output format. If the integer is output using the same nmnber of A's, the same string will be output as was input.

A programmer may work with these alphabetic quantities in a machine-independent manner by using the transfer function *equiv(S)* where S is a string; the value of *equiv(S)* is of type integer, and it is defined to have exactly the same value as if the string S had been input using alpha format. For example, one may write

```
if X = equiv('ALPHA') then go to PROCESS ALPHA;
```

where the value of X has been input using the format "AAAAA".

1.2.3.3 Nonformat An I, R or L is used to indicate that the value of a single variable of integer, real, or Boolean type, respectively, is to be input or output from or to an external medium, using the internal machine representation. If a value of type integer is output with R-format or if a value of type real is input with I-format, the appropriate transfer function is invoked. The precise behavior of this format, and particularly its interaction with other formats, is undefined in general.

1.2.3.4 Boolean format When Boolean quantities are input or output, the format P, F, 5F or FFFFF must be used. The correspondence is defined as follows:

Internal to ALGOL	P	F	5F = FFFFF
true	1	T	TRUE␣
false	0	F	FALSE

On input, anything failing to be in the proper form is undefined.

1.2.3.5 Title format All formats discussed so far have given a correspondence between a single ALGOL real, integer, Boolean, or string quantity and a number of characters in the input or output. A title format item consists entirely of insertions and alignment marks, and so it does not require a corresponding ALGOL quantity. On input, it merely causes skipping of the characters, and on output it causes emission of the insertion characters it contains. (If titles are to be input, alpha format should be used; see Section 1.2.3.2)

1.2.3.6 Alignment marks The characters "/" and "↑" in a format item indicate line and page control actions. The precise definition of these actions will be given later (see Section 2.5); they have the following intuitive interpretation: (a) "/" means go to the next line, in a manner similar to the "carriage return" operation on a typewriter. (b) "↑" means do a /-operation and then skip to the top of the next page.

Two or more alignment marks indicates the number of times the operations are to be performed; for example, "//" on output means the current line is completed and the next line is effectively set to all blanks. Alignment marks at the left of a format item cause actions to take place before the regular format operation, and if they are at the right they take place afterwards.

Note. On machines which do not have the character ↑ in their character set, it is recommended that some convenient character such as an asterisk be substituted for ↑ in format strings.

1.3 Format Strings

The format items mentioned above are combined into format strings according to the rules in this section.

1.3.1 Syntax

```
<format primary> ::= <format item> |
     <replicator> (<format secondary>) | (<format secondary>)
<format secondary> ::= <format primary> |
     <format secondary>, <format primary>
<format string> ::= ‘<format secondary>’ | ‘’
```

1.3.2 Examples

‘4(15ZD),//’
‘↑’
‘.5D$_{10}$+D, X(2(20B.8D$_{10}$+D), 10S)’
‘‘...This␣is␣a␣peculiar␣‘format␣string’’’

1.3.3 Semantics

A format string is simply a list of format items, which are to be interpreted from left to right. The construction "`<replicator> (<format secondary>)`" is simply an abbreviation for "replicator" repetitions of the parenthesized quantity (see Section 1.1.3.1). The construction "`(<format secondary>)`" is used to specify an infinite repetition of the parenthesized quantity.

All spaces within a format string except those which are part of insertion substrings are irrelevant.

It is recommended that the ALGOL compiler check the syntax of strings which (from their context) are known to be format strings as the program is compiled. In most cases it will also be possible for the compiler to translate format strings into an intermediate code designed for highly efficient input-output processing by the other procedures.

1.4 Summary of Format Codes

A	alphabetic character represented as integer
B	blank space
C	comma
D	digit
F	Boolean TRUE or FALSE
I	integer untranslated
L	Boolean untranslated
P	Boolean bit
R	real untranslated
S	string character
T	truncation
V	implied decimal point
X	arbitrary replicator
Z	zero suppression
+	print the sign
−	print the sign if it is minus
10	exponent part indicator
()	delimiters of replicated format secondaries
,	separates format items
/	line alignment
↑	page alignment
‘’	delimiters of inserted string
.	decimal point

1.5 “Standard” Format

There is a format available *without* specification (cf. Section 2.5) which has the following characteristics.

(a) On input, any number written according to the ALGOL syntax for `<number>` is accepted with the conventional meaning. These are of *arbitrary* length, and they are delimited at the right by the following conventions:

(i) A letter or character other than a decimal point, sign, digit, or “10” occurring to the right of a decimal point, sign, digit, or “10” is a delinfiter.

(ii) A sequence of k or more blank spaces serves as a delimiter as in (i); a sequence of less than k blank spaces is ignored. This number $k \geq 1$ is specified by the implementor (and the implementor may choose to let the programmer specify k on a control card of some sort).

(iii) If the number contains a decimal point, sign, digit, or "10" on the line where the number begins, the right-hand margin of that line serves as a delimiter of the number. However, if the first line of a field contains no such characters, the number is determined by reading several lines until finding a definfiter of type (i) or (ii). In other words, a number is not usually split across more than one line, unless its first line contains nothing but spaces or characters which do not enter into the number itself. (See Section 2.5 for further discussion of standard input format.)

(b) On output, a number is given in the form of a decimal number with an exponent. This decimal number has the amount of significant figures which the machine can represent; it is suitable for reading by the standard input format. Standard output format takes a fixed number of characters on the output medium; this size is specified by each ALGOL installation. Standard output format can also be used for the output of strings, and in this case the number of characters is equal to the length of the string.

2 Input and Output Procedures

2.1 General Characteristics

The over-all approach to input and output which is provided by the procedures of this report will be introduced here by means of a few examples, and the precise definition of the procedures will be given later.

Consider first a typical case, in which we want to print a line containing the values of the integer variables N and M, each of which is nonnegative, with at most five digits; also the value of $X[M]$, in the form of a signed number with a single nonzero digit to the left of the decimal point, and with an exponent indicated; and finally the value of $cos(t)$, using a format with a fixed decimal point and no exponent. The following might be written for this case:

```
output 4 (6, '2(BBBZZZZD),3B,+D.DDDDDD10+DDD,3B,-Z.DDDDBDDDD/',
     N, M, X[M], cos(t))
```

This example has the following significance. (a) The "4" in output 4 means four values are being output. (b) The "6" means that output is to go to unit number 6. This is the logical unit number, i.e., the programmer's number for that unit, and it does not necessarily mean physical unit number 6. See Section 2.1.1, for further discussion of unit numbers. (c) The next parameter, '2(BBB ... DDDD/', is the format string which specifies a format for outputting the four values. (d) The last four parameters are the values being printed. If $N = 500$, $M = 0$, $X[0] = 18061579$, and $t = 3.1415926536$, we obtain the line

␣␣␣␣␣500␣␣␣␣␣␣␣␣0␣␣␣+1.80615810+007␣␣␣-1.0000␣0000

as output.

Notice the "/" used in the above format; this symbol signifies the end of a line. If it had not been present, more numbers could have been placed on the same line in a future output statement. The programmer may build the contents of a line in several steps, as his algorithm proceeds, without automatically starting a new line each time output is called. For example, the above could have been written

```
output 1 (6, 'BBBZZZZD', N);
```

```
output 1 (6, 'BBBZZZZD', M);
output 2 (6,'3B,+D.DDDDDD10+DDD,3B,-Z.DDDDBDDDD',X[M],cos(t));
output 0 (6, '/');
```

with equivalent results.

In the example above, a line of 48 characters was output. If for some reason these output statements are used with a device incapable of printing 48 characters on a single line, the output would actually have been recorded on two or more lines, according to a rule which automatically keeps from breaking numbers between two consecutive lines wherever possible. (The exact rule appears in Section 2.5)

Now let us go to a slightly more complicated example: the real array $A[1:n, 1:n]$ is to be printed, starting on a new page. Supposing each element is printed with the format "BB-ZZZZ.DD", which uses ten characters per item, we could write the following program:

```
output 0 (6, '↑');
for i := 1 step 1 until n do
begin for j := 1 step 1 until n do
          output 1 (6, 'BB-ZZZZ.DD', A[i, j]);
      output 0 (6, '//') end.
```

If $10n$ characters will fit on one line, this little program will print n lines, double spaced, with n values per line; otherwise n groups of k lines separated by blank lines are produced, where k lines are necessary for the printing of n values. For example, if $n = 10$ and if the printer has 120 character positions, 10 double-spaced lines are produced. If, however, a 72-character printer is being used, 7 values are printed on the first line, 3 on the next, the third is blank, then 7 more values are printed, etc.

There is another way to achieve the above output and to obtain more control over the page format as well. The subject of page format will be discussed further in Section 2.2, and we will indicate here the manner in which the above operation can be done conveniently using a single output statement. The procedures *output O*, *output 1*, etc. mentioned above provide only for the common cases of output, and they are essentially a special abbreviation for certain calls on the more general procedure *out list*. This more general procedure could be used for the above problem in the following manner:

```
out list (6, LAYOUT, LIST)
```

Here LAYOUT and LIST are the names of procedures which appear below. The first parameter of out list is the logical unit number as described above. The second parameter is the name of a so-called "layout procedure"; general layout procedures are discussed in Section 2.3. The third parameter of *out list* is the name of a so-called "list procedure"; general list procedures are discussed in Section 2.4.

In general, a layout procedure specifies the format control of the input or output. For the case we are considering, we could write a simple layout procedure (named "LAYOUT") as follows:

```
procedure LAYOUT; format 1 ('↑,(X(BB-ZZZZ.DD),//)', n)
```

The 1 in *format 1* means a format string containing one `X` is given. The format string is

```
↑,(X(BB-ZZZZ.DD),//)
```

which means skip to a new page, then repeat the format `X(BB-ZZZZ.DD),//` until the last value is output. The latter format means that `BB-ZZZZ.DD` is to be used X times, then skip to a new line. Finally, *format 1* is a procedure which effectively inserts the value of n for the letter `X` appearing in the format string.

A *list procedure* serves to specify a list of quantities. For the problem under consideration, we could write a simple list procedure (named "LIST") as follows:

```
procedure LIST(ITEM);
for i := step 1 until n do
    for j := 1 step 1 until n do
        ITEM(A[i, j])
```

Here "`ITEM(A[i, j])`" means that $A[i, j]$ is the next item of the list. The procedure *ITEM* is a formal parameter which might have been given a different name such as *PIECE* or *CHUNK*; list procedures are discussed in more detail in Section 2.4.

The declarations of *LAYOUT* and *LIST* above, together with the procedure statement `out list (6, LAYOUT, LIST)`, accomplish the desired output of the array A.

Input is done in a manner dual to output, in such a way that it is the exact inverse of the output process wherever possible. The procedures *in list* and *input n* correspond to *out list* and *output n* ($n = 0, 1, \ldots$).

Two other procedures, *get* and *put*, are introduced to facilitate storage of intermediate data on external devices. For example, the statement *put(100, LIST)* would cause the values specified in the list procedure named *LIST* to be recorded in the external medium with an identification number of 100. The subsequent statement *get(100, LIST)* would restore these values. The external medium might be a disk file, a drum, a magnetic tape, etc.; the type of device and the format in which data is stored there is of no concern to the programmer.

2.1.1 Unit numbers

The first parameter of input and output procedures is the logical unit number, i.e. some number which the programmer has chosen to identify some input or output device. The connection between logical unit numbers and the actual physical unit numbers is specified by the programmer *outside* of the ALGOL language, by means of "control cards" preceding or following his program, or in some other way provided by the ALGOL implementor. The situation which arises if the same physical unit is being used for two different logical numbers, or if the same physical unit is used both for input and for output, is undefined in general.

It is recommended that the internal computer memory (e.g. the core memory) be available as an "input-output device," so that data may be edited by means of input and output statements.

2.2 Horizontal and Vertical Control

This section deals with the way in which the sequence of characters, described by the rules of formats in Section 1, is mapped onto input and output devices. This is done in a manner which is essentially independent of the device being used, in the sense that with these specifications the programmer can anticipate how the input or output data will appear on virtually any device. Some of the features of this description will, of course, be more appropriately used on certain devices than on others.

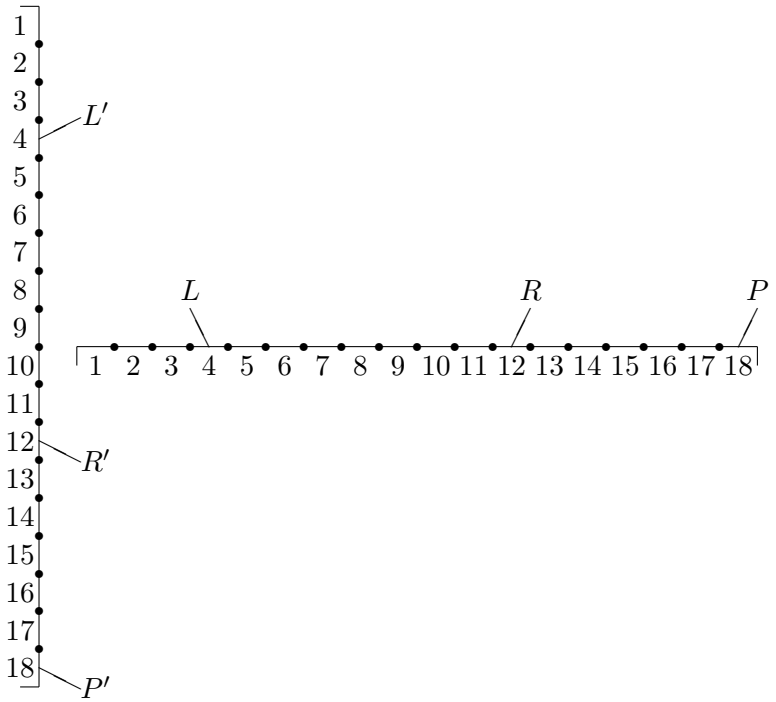

Figure 2: Margins in horizontal and vertical control

We will begin by assuming we are doing *output* to a *printer*. This is essentially the most difficult case to handle, and we will discuss the manner in which other devices fit into the same general framework.

The page format is controlled by specifying the horizontal and the vertical layout. Horizontal layout is controlled in essentially the same manner as vertical layout, and this symmetry between the horizontal and vertical dimensions should be kept in mind for easier understanding of the concepts in this section.

Refer to Figure 2; the horizontal format is described in terms of three parameters (L, R, P), and the vertical format has corresponding parameters (L', R', P'). The parameters L, L' and R, R' indicate left and right margins, respectively; Figure 2 shows a case where $L = L' = 4$ and $R = R' = 12$. Only positions L through R of a horizontal line are used, and only lines L' and R' of the page are used; we require that $1 \leq L \leq R$ and $1 \leq L' \leq R'$. The parameter P is the nmnber of characters per line, and P' is the number of lines per page. Although L, R, L', and R' are chosen by the programmer, the values of P and P' are characteristics of the device and they are usually out of the programmer's control. For those devices on which P and P' can vary (for example, some printers have two settings, one on which there are 66 lines per page, and another on which there are 88), the values are specified to the system in some manner external to the ALGOL program, e.g. on control cards. For certain devices, values of P or P' might be essentially infinite.

Although Figure 2 shows a case where $P \geq R$ and $P' \geq R'$, it is of course quite possible that $P < R$ or $P' < R'$ (or both) might occur, since P and P' are in general unknown to the programmer. In such cases, the algorithm described in Section 2.5 is used to break up logical lines which are too wide to fit on a physical line, and to break up logical pages which

are too large to fit a physical page. On the other hand, the conditions $L \leq P$ and $L' \leq P'$ are insured by setting L or L' equal to 1 automatically if they happen to be greater than P or P', respectively.

Characters determined by the output values are put onto a horizontal line; there are three conditions which cause a transfer to the next line: (a) normal line alignment, specified by a "/" in the format; (b) R-overflow, which occurs when a group of characters is to be transmitted which would pass position R; and (c) P-overflow, which occurs when a group of characters is to be transmitted which would not cause R-overflow but would pass position P. When any of these three things occurs, control is transferred to a procedure specified by the programmer in case special action is desired (e.g. a change of margins in case of overflow; see Section 2.3.3).

Similarly, there are three conditions which cause a transfer to the next page: (a') normal page alignment, specified by a "↑" in the format; (b') R'-overflow, which occurs when a group of characters is to be transmitted which would appear on line $R' + 1$; and (c') P'-overflow, which occurs when a group of characters is to be transmitted which would appear on line $P' + 1 < R' + 1$. The programmer may indicate special procedures to be executed at this time if he wishes, e.g. to insert a page heading, etc.

Further details concerning pages and lines will be given later. Now we will consider how devices other than printers can be thought of in terms of the ideas above.

A typewriter is, of course, very much like a printer and it requires no further comment.

Punched cards with, say, 80 columns, have $P = 80$ and $P' = \infty$. Vertical control would appear to have little meaning for punched cards, although the implementor might choose to interpret "↑" to mean the insertion of a coded or blank card.

With paper tape, we might again say that vertical control has little or no meaning; in this case, P could be the number of characters read or written at a time.

On magnetic tape capable of writing arbitrarily long blocks, we have $P = P' = \infty$. We might think of each page as being a "record," i.e., an amount of contiguous information on the tape which is read or written at once. The lines are subdivisions of a record, and R' lines form a record; R characters are in each line. In this way we can specify so-called "blocking of records." Other interpretations might be more appropriate for magnetic tapes at certain installations, e.g. a format which would correspond exactly to printer format for future offline listing, etc.

These examples are given merely to indicate how the concepts described above for printers can be applied to other devices. Each implementor will decide what method is most appropriate for his particular devices, and if there are choices to be made they can be given by the programmer by means of control cards. The manner in which this is done is of no concern in this report; our procedures are defined solely in terms of P and P'.

2.3 Layout Procedures

Whenever input or output is done, certain "standard" operations are assumed to take place, unless otherwise specified by the programmer. Therefore one of the parameters of the input or output procedure is a so-called "layout" procedure, which specifies all of the nonstandard operations desired. This is achieved by using any or all of the six "descriptive procedures" *format*, *h end*, *v end*, *h lim*, *v lim*, *no data* described in this section.

The precise action of these procedures can be described in terms of the mythical concept of six "hidden variables," H1, H2, H3, H4, H5, H6. The effect of each descriptive procedure is

to set one of these variables to a certain value; and as a matter of fact, that may be regarded as the sum total of the effect of a descriptive procedure. The programmer normally has no other access to these hidden variables (see, however, Section 2.7). The hidden variables have a scope which is local to *in list* and to *out list*.

2.3.1 Format procedures

The descriptive procedure call

```
format (string)
```

has the effect of setting the hidden variable H1 to indicate the string parameter. This parameter may either be a string explicitly written, or a formal parameter; but in any event, the string it refers to must be a format string, which satisfies the syntax of Section 1.3, and it must have no "X" replicators.

The procedure *format* is just one of a class of procedures which have the names *format n*, $(n = 0, 1, \ldots)$. The name *format* is equivalent to *format O*. In general, the procedure *format n* is used with format strings which have exactly n X-replicators. The call is

```
format n (string, X1, X2, ..., Xn)
```

where each $\texttt{X}_i$ is an integer parameter called by value. The effect is to replace each X of the format string by one of the $\texttt{X}_i$, with the correspondence defined from left to right. Each $\texttt{X}_i$ must be nonnegative.

For example,

```
format 2 ('XB.XD10+DD', 5, 10)
```

is equivalent to

```
format ('5B.10D10+DD').
```

2.3.2 Limits

The descriptive procedure call

```
h lim (L, R)
```

has the effect of setting the hidden variable H2 to indicate the two parameters L and R. Similarly,

```
v lim (L', R')
```

sets H3 to indicate L' and R'. These parameters have the significance described in Section 2.2. If *h lim* and *v lim* are not used, $L = L' = 1$ and $R = R' = \infty$.

2.3.3 End control

The descriptive procedure calls

```
h end (PN, PR , PP);
v end (PN', PR', PP');
```

have the effect of setting the hidden variables H4 and H5, respectively, to indicate their parameters. The parameters P_N, P_R, P_P, $P_{N'}$, $P_{R'}$, $P_{P'}$ are names of procedures (ordinarily dummy statements if *h end* and *v end* are not specified) which are activated in case of normal line alignment, R-overflow, P-overflow, normal page alignment, R'-overflow, and P'-overflow, respectively.

2.3.4 End of data

The descriptive procedure call

```
no data (L);
```

has the effect of setting the hidden variable H6 to indicate the parameter L. Here L is a label. End of data as defined here has meaning only on input, and it does not refer to any specific hardware features; it occurs when data is requested for input but no more data remains on the corresponding input medium. At this point, a transfer to the statement labeled L will occur. If the procedure *no data* is not used, transfer will occur to a "label" which has effectively been inserted just before the final **end** in the ALGOL program, thus terminating the program. (In this case the implementor may elect to provide an appropriate error comment.)

2.3.5 Examples

A layout procedure might look as follows:

```
procedure LAYOUT;
begin format('/');
      if B then begin format 1('XB', Y + 10);
                      no data (L32) end;
      h lim(if B then 1 else 10, 30) end;
```

Note that layout procedures never have formal parameters; this procedure, for example, refers to three global quantities, B, Y and $L32$. Suppose Y has the value 3; then this layout accomplishes the following:

Hidden variable	Procedure	**if** B = **true**	**if** B = **false**
H1	*format*	'13B'	'/'
H2	*h lim*	(1, 30)	(10, 30)
H3	*v lim*	(1, ∞)	(1, ∞)
H4	*h end*	(, ,)	(, ,)
H5	*v end*	(, ,)	(, ,)
H6	*no data*	*L*32	*end program*

As a more useful example, we can take the procedure *LAYOUT* of Section 2.1 and rewrite it so that the horizontal margins (11,110) are used on the page, except that if P-overflow or R-overflow occurs we wish to use the margins (16, 105) for overflow lines.

```
procedure LAYOUT;
begin format 1 (‘↑,(X(BB-ZZZZ.DD),//)’, n);
      h lim (11,110); h end (K, L, L) end;
procedure K; h lim (11,110);
procedure L; h lim (16,105);
```

This causes the limits (16, 105) to be set whenever overflow occurs, and the "/" in the format will reinstate the original margins when it causes procedure K to be called. (If the programmer wishes a more elaborate treatment of the overflow case, depending on the value of P, he may do this using the procedures of Section 2.6)

2.4 List Procedures

2.4.1 General characteristics

The concept of a list procedure is quite important to the input-output conventions described in this report, and it may also prove useful in other applications of ALGOL . It represents a specialized application of the standard features of ALGOL which permit a procedure identifier, L, to be given as an actual parameter of a procedure, and which permit procedures to be declared within procedures.

The purpose of a list procedure is to describe a sequence of items which is to be transmitted for input or output. A procedure is written in which the name of each item V is written as the argument of a procedure, say *ITEM*, thus: *ITEM(V)*. When the list procedure is called by an input-output system procedure, another procedure (such as the internal system procedure out item) will be "substituted" for *ITEM*, V will be called by name, and the value of V will be transmitted for input or output. The standard sequencing of ALGOL statements in the body of the list procedure determines the sequence of items in the list.

A simple form of list procedure might be written as follows:

```
procedure LIST (ITEM);
begin ITEM(A); ITEM(B); ITEM(C) end
```

which says that the values of A, B and C are to be transmitted. A more typical list procedure might be:

```
procedure PAIRS (ELT);
for i := 1 step 1 until n do
begin ELT(A[i]);
      ELT(B[i]) end
```

This procedure says that the values of the list of items $A[1]$, $B[1]$, $A[2]$, $B[2]$, ..., $A[n]$, $B[n]$ are to be transmitted, in that order. Note that if $n \leq 0$ no items are transmitted at all.

The parmneter of the "item" procedure (i.e. the parameter of *ITEM* or *ELT* in the above examples) is called by name. It may be an arithmetic expression, a Boolean expression, or a string, in accordance with the format which will be associated with the item. Any of the ordinary features of ALGOL may be used in a list procedure, so there is great flexibility.

Unlike layout procedures which simply run through their statements and set up hidden variables H1 through H6, a list procedure is executed step by step with the input or output procedure, with control transferring back and forth. This is accomplished by special system

procedures such as *in item* and *out item* which are "interlaced" with the list procedure, as described in Sections 2.4.2 and 2.5. The list procedure is called with *in item* (or *out item*) as actual parameter, and whenever this procedure is called within the list procedure, the actual input or output is taking place. Through the interlacing, special format control, including the important device-independent overflow procedures, can take place during the transmission process. Note that a list procedure may change the hidden variables by calling a descriptive procedure; this can be a valuable characteristic, e.g. when changing the format, based on the value of the first item which is input.

2.4.2 Other applications

List procedures can actually be used in many ways in ALGOL besides their use with input or output routines; they are useful for manipulating linear lists of items of a quite general nature. To illustrate this fact, and to point out how the interlacing of control between list and driver procedures can be accomplished, here is an example of a procedure which calculates the sum of all of the elements in a list (assuming all elements are of integer or real type):

```
procedure ADD(Y, Z);
begin procedure A(X); Z := Z + X;
      Z := 0; Y(A) end
```

The call *ADD(PAIRS, SUM)* will set the value of *SUM* to be the sum of all of the items in the list *PAIRS* defined in Section 2.4.1. The reader should study this example carefully to grasp the essential significance of list procedures. It is a simple and instructive exercise to write a procedure which sets all elements of a list to zero.

2.5 Input and Output Calls

Here procedures are described which cause the actual transmission of input or output to take place.

2.5.1 Output

An output process is initiated by the call:

```
out list (unit, LAYOUT, LIST)
```

Here unit is an integer parameter called by value, which is the number of an output device (cf. Section 2.1.1). The parameter *LAYOUT* is the name of a layout procedure (Section 2.3), and *LIST* is the name of a list procedure (Section 2.4).

There is also another class of procedures, named *output n*, for $n = 0, 1, 2, \ldots$, which is used for output as follows:

```
output n (unit, format string, e1, e2, ..., en)
```

Each of these latter procedures can be defined in terms of out list as follows:

```
procedure output n(unit, format string, e1, e2, ..., en);
begin procedure A; format (format string);
      procedure B(P); begin P(e1); P(e2); ...; P(en) end;
      out list (unit, A, B) end
```

We will therefore assume in the following rules that out list has been called.

Let the variables p and p' indicate the current position in the output for the unit under consideration, i.e. lines 1, 2, ..., p' of the current page have been completed, as well as character positions 1, 2, ..., p of the current line (i.e., of line $p'+1$). At the beginning of the program, $p = p' = 0$. The symbols P and P' denote the line size and page size (see Section 2.2). Output takes place according to the following algorithm:

Step 1. The hidden variables are set to standard values:
H1 is set to the "standard" format ''
H2 is set so that $L = 1, R = \infty$.
H3 is set so that $L' = 1, R' = \infty$.
H4 is set so that P_N, P_R, P_P are all effectively equal to the $DUMMY$ procedure defined as follows: "**procedure** `DUMMY;;`".
H5 is set so that $P_{N'}$, $P_{R'}$, $P_{P'}$ are all effectively equal to DUMMY.
H6 is set to terminate the program in case the data ends (this has meaning only on input).

Step 2. The layout procedure is called; this may change some of the variables H1, H2, H3, H4, H5, H6.

Step 3. The next format item of the format string is examined. (*Note.* After the format string is exhausted, "standard" format, Section 1.5, is used from then on until the end of the procedure. In particular, if the format string is '', standard format is used throughout.) Now if the next format item is a title format, i.e. requires no data item, we proceed directly to step 4. Otherwise, the list procedure is activated; this is done the first time by *calling* the list procedure, using as actual parameter a procedure named *out item*; this is done on all subsequent times by merely returning from the procedure *out item*, which will cause the list procedure to be continued from the latest *out item* call. (*Note.* The identifier out item has scope local to out list, so a programmer may not call this procedure directly.) After the list procedure has been activated in this way, it will either terminate or will call the procedure *out item*. In the former case, the output process is completed; in the latter case, continue at step 4.

Step 4. Take the next item from the format string. (*Note.* If the list procedure was called in step 3, it may have called the descriptive procedure *format*, thereby changing from the format which was examined during step 3. In such a case, the new format is used here. But at this point the format item is effectively removed from the format string and copied elsewhere; so that the format string itself, possibly changed by further calls of *format*, will not be interrogated until the next occurrence of step 3. If the list procedure has substituted a title format for a nontitle format, the "item" it specifies will not be output, since a title format consists entirely of insertions and alignment marks.)

Set "*toggle*" to **false**. (This is used to control the breaking of entries between lines.) The alignment marks, if any, at the left of the format item, now cause process A (below) to be executed for each "/", and process B for each "↑". If the format item consists entirely of alignment marks, then go immediately to step 3. Otherwise the size of the format (i.e. the number of characters specified in the output medium) is determined. Let this size be denoted by S. Continue with step 5.

Step 5. Execute process C, to ensure proper page alignment.

Step 6. Line alignment: If $p < L - l$, effectively insert blank spaces so that $p = L - 1$. Now if *toggle* = **true**, go to step 9; otherwise, test for line overflow as follows: If $p + S > R$, perform process D, then call P_R and go to step 8; otherwise, if $p + S > P$, perform process D, call P_P , and go to step 8.

Step 7. Evaluate the next output item and output it according to the rules given in Section 1; in the case of a title format, this is simply a transmission of the insertions without the evaluation of an output item. The pointer p is set to $p+S$. Any alignment marks at the right of the format item now cause activation of process A for each "/" and of process B for each "↑". Return to step 3.

Step 8. Set *toggle* to **true**. Prepare a formatted output item as in step 7, but do not record it on the output medium yet (this is done in step 9). Go to step 5. (It is necessary to re-examine page and line alignment, which may have been altered by the overflow procedure; hence we go to step 5 rather than proceeding immediately to step 9.)

Step 9. Transfer as many characters of the current output item as possible into positions $p+1$, ..., without exceeding position P or R. Adjust p appropriately. If the output of this item is still unfinished, execute process D again, call P_R (if $R \leq P$) or P_P (if $P < R$), and return to step 5. The entire item will eventually be output, and then we process alignment characters as in step 7, finally returning to step 3.

Process A. ("/" operation) Check page alignment with process C, then execute process D and call procedure P_N

Process B. ("↑" operation) If $p > 0$, execute process A. Then execute process E and call procedure $P_{N'}$.

Process C. (Page alignment)
If $p' < L' - 1$ and $p > 0$: execute process D, call procedure P_N , and repeat process C.
If $p' < L' - 1$ and $p = 0$: execute process D until $p' = L' - 1$.
If $p' + 1 > R'$: execute process E, call procedure $P_{R'}$, and repeat process C.
If $p' + 1 > P'$: execute process E, call procedure $P_{P'}$, and repeat process C.

Process D. Skip the output medium to the next line, set $p = 0$, and set $p' = p' + 1$.

Process E. Skip the output medium to the next page, and set $p' = 0$.

2.5.2 Input

The input process is initiated by the call:

```
in list (unit, LAYOUT, LIST)
```

The parameters have the same significance as they did in the case of output, except that *unit* is in this case the number of an input device. There is a class of procedures *input n* which stand for a call with a particularly simple type of layout and list, just as discussed in Section 2.5.1 for the case of output. In the case of input, the parameters of the "item" procedure within the list must be variables.

The various steps which take place during the execution of *in list* are very much the same as those in the case of *out list*, with obvious changes. Instead of transferring characters of title format, the characters are ignored on input. If the data is improper, some standard error procedure is used (cf. Section 1.1.3.8).

The only significant change occurs in the case of standard input format, in which the number S of the above algorithm cannot be determined in step 4. The tests $p + S > R$ and $p + S > P$ now become a test on whether positions $p + l$, $p + 2$, ..., $min(R, P)$ have any numbers in them or not. If so, the first number, up to its delimiter, is used; the R and P positions serve as delimiters here. If not, however, overflow occurs, and subsequent lines are searched until a number is found (possibly causing additional overflows). The right boundary $min(R, P)$ will not count as a delimiter in the case of overflow. This rule has been made so that the process of input is dual to that of output: an input item is not split across more than one line unless it has overflowed twice.

Notice that the programmer has the ability to determine the presence or absence of data on a card when using standard format, because of the way overflow is defined. The following program, for example, will count the number n of data items on a single input card and will read them into $A[1]$, $A[2]$, ..., $A[n]$. (Assume unit 5 is a card reader.)

```
procedure LAY; h end(EXIT, EXIT, EXIT);
procedure LIST(ITEM); ITEM(A[n + 1]);
procedure EXIT; go to L2;

     n := 0;
L1:  in list(5, LAY, LIST);
     n := n - 1;
     go to L1;
L2:  comment mission accomplished;
```

2.5.3 Skipping

Two procedures are available which achieve an effect similar to that of the "tab" key on a typewriter:

```
h skip (position, OVERFLOW)
v skip (position, OVERFLOW)
```

where position is an integer variable called by value, and $OVERFLOW$ is the name of a procedure. These procedures are defined only if they are called within a list procedure during an *in list* or *out list* operation. For *h skip*, if $p <$ *position*, set $p =$ *position*; but if $p <$ *position*, call the procedure $OVERFLOW$. For *v skip*, an analogous procedure is carried out: if $p' <$ *position*, effectively execute process A of Section 2.5.1 (*position* $- p'$) times; but if $p' <$ *position*, call the procedure $OVERFLOW$.

2.5.4 Intermediate data storage

The procedure call

```
put (n, LIST)
```

where n is an integer parameter called by value and $LIST$ is the name of a list procedure (Section 2.4), takes the values specified by the list procedure and stores them, together with the identification number n. Anything previously stored with the same identification number is lost. The variables entering into the list do not lose their values.

The procedure call

```
get (n, LIST)
```

where n is an integer parameter called by value and $LIST$ is the name of a list procedure, is used to retrieve the set of values which has previously been put away with identification number n. The items in $LIST$ must be variables. The stored values are retrieved in the same order as they were placed, and they must be compatible with the type of the elements specified by $LIST$; transfer functions may be invoked to convert from real to integer type or vice versa. If fewer items are in $LIST$ than are associated with n, only the first are retrieved; if $LIST$ contains more items, the situation is undefined. The values associated with n in the external storage are not changed by *get*.

2.6 Control Procedures

The procedure calls

```
out control (unit, xl, x2, x3, x4)
in control (unit, xl, x2, x3, x4)
```

may be used by the programmer to determine the values of normally "hidden" system parameters, in order to have finer control over input and output. Here unit is the number of an output or input device, and xl, $x2$, $x3$, $x4$ are variables. The action of these procedures is to set xl, $x2$, $x3$, $x4$ equal to the current values of p, P, p', P', respectively, corresponding to the device specified.

2.7 Other Procedures

Other procedures which apply to specific input or output devices may be defined at certain installations, e.g. *tape skip* and *rewind* for controlling magnetic tape, etc. An installation may also define further descriptive procedures (thus introducing further hidden variables); e.g., a procedure might be added to name a label to go to in case of an input error. Procedures for obtaining the current values of hidden variables might also be incorporated.

3 An Example

A simple example follows, which is to print the first 20 lines of Pascal's triangle in triangular form:

```
   1
  1 1
 1 2 1
1 3 3 1
```

These first 20 lines involve numbers which are at most five digits in magnitude. The output is to begin a new page, and it is to be double-spaced and preceded by the title "PASCALS TRIANGLE". We assume that unit number 3 is a line printer.

Two solutions of the problem are given, each of which uses slightly different portions of the input-output conventions.

```
begin integer N, K, printer;
      integer array A [0:19];
      procedure AK(ITEM); ITEM(A[K]);
      procedure TRIANGLE; begin format('6Z');
           h lim(58 - 3 ×N, 63 + 3 ×N) end;
printer := 3;
output 0 (printer, '↑'PASCALS␣TRIANGLE'//');
for N := 0 step 1 until 19 do
begin A[N] := 1;
      for K := N - 1 step -1 until 1 do
           A[K] := A[K - 1] + A[K];
      for K := 0 step 1 until N do
           out list (printer, TRIANGLE, AK);
      output 0 (printer, '//') end end
```

```
begin integer N, IK, printer;
      integer array A [0:19];
      procedure LINES; format 2 ('XB,X(6Z),//', 57 - 3 ×N, N+1);
      procedure LIST(Q); for K := 0 step 1 until N do Q(A[K]);
printer := 8;
output 1 (printer, '↑20S//', 'PASCALS␣TRIANGLE');
for N := 0 step 1 until 19 do
begin A[N] := 1;
      for K := N - 1 step -1 until 1 do A[K] := A[K - 1] + A[K];
      out list (printer, LINES, LIST) end end
```

4 Machine-dependent Portions

Since input-output processes must be machine-dependent to a certain extent, the portions of this proposal which are machine-dependent are summarized here.

1. The values of P and P' for the input and output devices.
2. The treatment of I, L, and R (unformatted) format.
3. The number of characters in standard output format.
4. The internal representation of alpha format.
5. The number of spaces, K, which will serveto delilnit standard input format values.