leads to a much better tree than making them from bottom to top. However, the bottom-to-top method is much more convenient to program, and it can be used if the new branches from a given node are attached from right to left. The tree begins looking as in Figure 3.

The preceding criterion of excellence was only the number of total multiplications. Clearly if n is an *unknown* variable, the Binary Method is the best to use. In fact this method would be quite suitable to incorporate in the hardware of a binary computer, as an exponentiation operator.

If y is floating point, there is of course a point of diminishing returns when n gets large, since it will eventually be better to take logarithms and exponentials. Since the Tree Method uses r multiplications at level r it is possible to stop generating the tree at a certain point and we then have a set of all the "interesting" values of n. The tree in Figure 3, for example, shows all n for which it is known that y^n needs 6 or less multiplications. The Factor Method, on the other hand, is valuable when n is large and an application requires frequent calculation of y^n.

A system manual (which will not be mentioned here by name) has a subroutine for "float to fix exponentiation" of $y \uparrow n$ which uses $n-1$ multiplications since it says "there is small probability of this routine being used with a very large n." This may seem to be a valid point at first; but it didn't take long before a user had to calculate y^{70} a very large number of times, and so the user rewrote the subroutine. This remark is included here to offer some justification for having a good power method.

REFERENCES

1. See Todd, John. *A Survey of Numerical Analysis*, pp. 3-4. McGraw-Hill, 1962.
2. Ostrowski, A. M. *Studies in Mathematics and Mechanics Presented to R. von Mises*, pp. 40-48. Academic Press, Inc., 1954.
3. Floyd, R. An algorithm for coding efficient arithmetic operations. *Comm. ACM 4* (Jan. 1961), 50-51.

Editor's Note. Since much usage may be made of these methods in the floating-point mode, one could consider also the relative timing of multiplication and division. In some machines these have even been equal. Thus, for y^{31} other possibilities exist, using "D" to indicate a division:

Binary	(8)	$SXSXSXSX \equiv$	$X_1X_1X_3X_1X_7X_1X_{15}X_1$
Factor	(7)		$X_1X_2X_2X_6X_{12}X_6X_1$
Division	(6)		$X_1X_2X_4X_8X_{16}D_1$

The reader is invited to try n^{127} and see if some algorithm could be derived for mixed operations.

A Decision Matrix as the Basis for a Simple Data Input Routine

G. J. Vasilakos
Datatrol Corporation, Silver Spring, Md.

Currently a great deal of time and effort is being spent on the development of bigger and better compiler languages, multiprogram executive systems, etc. Since the implementation of of new methods and procedures is not instantaneous, but rather occurs by an evolutionary process, we should be concerned also with the problem of maintaining, improving and incorporating new ideas into existing systems. It is with this somewhat neglected area that the author is interested. A method employing a decision matrix is presented for the handling of a standard systems programming problem, that of providing a data input routine.

Introduction

Motivation for this project came from an analysis of several current systems which revealed that the routines for handling input character data strings had been coded in an ad hoc manner, brute-force, do-it-any-way-you-can method. The technique to be outlined may either suggest that recoding of these programs could be worthwhile or it may at least provide some useful ideas for people designing their own input routines. However, it is not our purpose to suggest a format for a general data input routine. Thus the details of the program to be described, which in itself is fairly simple-minded and somewhat restrictive, are not to be construed as recommended specifications but are provided only for the purposes of illustration.

We are concerned with the analysis of character strings which conform to a fairly universally accepted format for defining input data. Basically this involves categorizing data items as falling into one of three general classes: alphanumeric, hollerith or numeric. Alphanumeric and hollerith information consists of character strings which are translated into the corresponding internal machine representation by the input routine and placed in the computer memory. The distinction is usually made to identify character strings which can be manipulated by the program (alphanumeric), and strings which are fixed (hollerith) and can only be altered by re-assembling or recompiling. Numeric data is converted to an appropriate binary representation (we will restrict ourselves to binary machines) and then placed in memory. An input string can contain any of the three data types mixed in any order. A format statement can be used to communicate information about the organization of the data string to the input routine, or the routine can determine this from the context of the input string.

	1 ⟨alpha⟩ not E	2 ⟨digit⟩	3 , | ƀ	4 + | −	5 E	6 .	7 (	8)	9 $
1. ⟨terminator⟩	2,M	7,MN	1	6,SS	2,M	5,SD	3	ERR	END
2. ⟨a-element⟩	2,M	2,M	1,TAH	ERR	2,M	ERR	ERR	ERR	TAH
3. ⟨h-signal⟩	4,M	4,M	4,M	4,M	4,M	4,M	ERR	ERR	ERR
4. ⟨h-element⟩	4,M	4,M	4,M	4,M	4,M	4,M	ERR	1,TAH	ERR
5. .	ERR	8,MN	ERR	ERR	ERR	ERR	ERR	ERR	ERR
6. ⟨sign⟩	ERR	7,MN	ERR	ERR	ERR	5,SD	ERR	ERR	ERR
7. ⟨integer⟩	ERR	7,MN	1,TN	ERR	9	8,SD	ERR	ERR	TN
8. ⟨fixed point number⟩	ERR	8,MN	1,TF	ERR	9	ERR	ERR	ERR	TF
9. E	ERR	11,ME	ERR	10,SE	ERR	ERR	ERR	ERR	ERR
10. E ⟨sign⟩	ERR	11,ME	ERR	ERR	ERR	ERR	ERR	ERR	ERR
11. ⟨exponent⟩	ERR	11,ME	1,TF	ERR	ERR	ERR	ERR	ERR	TF

FIG. 1. Decision matrix for input scan

It is not the purpose of this article to get deeply involved in a detailed discussion of a well-known method of specifying the construction of input information. For the reader unfamiliar with this type of representation, the input/output section of reference [1] is suggested reading.

Our intent is to describe the actual analysis of the input character string, and we are not concerned with the methods used for getting the data from input devices or the way in which a programmer can request allocation of the translated or converted data to memory locations. The technique to be outlined is applicable to any binary computer. However, a program using this approach has been written and checked out for an IBM 7090 and therefore some references in the discussion are directly related to the hardware characteristics.

The Program

The program processes a continuous input character string of any length. The basic syntactic component of any string is a data field. Fields can be one of three types; alphanumeric, hollerith or numeric. Any number of data fields can be grouped to form a logical record. Any number of logical records can comprise a complete input file.

Alphanumeric fields can contain alphabetic characters, digits and some special characters. However, they must always begin with an alphabetic character. Hollerith fields are enclosed within left and right parentheses and can contain any characters except (,), or the special end-of-logical record character $. Numeric fields can be either integer, fixed point or floating point. They conform to the normal rules of formation for numeric data.

The program analyzes input information by context. That is, *no format statement is used to specify the input data construction.* The syntax rules are such that the characters of any input string uniquely define the types of fields and their grouping into logical records.

In order to save further lengthy description of field, record and file organization, and yet to uniquely define their construction, the syntax for input string formation is given below. A blank is denoted by the character "ƀ". The notation is consistent with that used in the report on ALGOL 60 [2].

Definition of Elementary Characters

⟨alpha⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | = | / | *
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨sign⟩ ::= + | −
⟨terminator⟩ ::= , | ƀ | ⟨terminator⟩ ⟨terminator⟩

Definition of Alphanumeric and Hollerith Fields

⟨*a*-element⟩ ::= ⟨alpha⟩ | ⟨*a*-element⟩ ⟨alpha⟩ | ⟨a-element⟩ ⟨digit⟩
⟨alphanumeric field⟩ ::= ⟨*a*-element⟩ ⟨terminator⟩ | ⟨*a*-element⟩ $
⟨*h*-signal⟩ ::= (⟨digit⟩ | (⟨alpha⟩ | (⟨sign⟩ | (⟨terminator⟩ | (.
⟨*h*-element⟩ ::= ⟨*h*-signal⟩ | ⟨*h*-element⟩ ⟨digit⟩ | ⟨*h*-signal⟩ ⟨alpha⟩ | ⟨*h*-element⟩ ⟨sign⟩ | ⟨*h*-element⟩ ⟨terminator⟩ | ⟨*h*-element⟩ .
⟨hollerith field⟩ ::= ⟨*h*-element⟩)

Definition of Numeric Fields

⟨integer⟩ ::= ⟨digit⟩ | ⟨sign⟩ ⟨digit⟩ | ⟨integer⟩ ⟨digit⟩
⟨integral part⟩ ::= ⟨integer⟩ . | ⟨sign⟩ . ⟨digit⟩ | . ⟨digit⟩
⟨fixed point number⟩ ::= ⟨integral part⟩ | ⟨fixed point number⟩ ⟨digit⟩
⟨exponent⟩ ::= *E* ⟨sign⟩ ⟨digit⟩ | *E* ⟨digit⟩ | ⟨exponent⟩ ⟨digit⟩
⟨floating point number⟩ ::= ⟨fixed point number⟩ ⟨exponent⟩ | ⟨integer⟩ ⟨exponent⟩
⟨numeric element⟩ ::= ⟨integer⟩ | ⟨fixed point number⟩ | ⟨floating point number⟩
⟨numeric field⟩ ::= ⟨numeric element⟩ ⟨terminator⟩ | ⟨numeric element⟩ $

Definition of Logical Records and File

⟨subfield ::= ⟨*a*-element⟩ | ⟨numeric element⟩ | ⟨terminator⟩
⟨field⟩ ::= ⟨subfield⟩ ⟨terminator⟩ | ⟨hollerith field⟩ | ⟨field⟩ ⟨field⟩
⟨logical record⟩ ::= ⟨field⟩ $ | ⟨field⟩ ⟨subfield⟩ $ | ⟨subfield⟩ $
⟨file element⟩ ::= ⟨logical record⟩ | ⟨file element⟩ ⟨logical record⟩
⟨file⟩ ::= ⟨file element⟩ ⟨physical file indicator⟩

Defined below are a few sample data fields and a possible input string which might be formed from them.

Field	*Type*
AB345	alphanumeric
12.3E-4	floating point number
123	integer
(PAGE ﾉ̸NO.24)	hollerith field
12.3	fixed point number

Input String

AB345, 12.3E−4$ ﾉ̸123 ﾉ̸ ﾉ̸ (Page ﾉ̸No.24), 12.3 ﾉ̸$

Although not shown in the syntax, restrictions were placed on the magnitude of numeric fields (for implementation purposes). The syntax above does not provide for handling physical end-of-record marks, since it was felt their inclusion would add nothing but length to this paper. For coding purposes, the routine which supplies the scanner with an input character was programmed to ignore physical record marks.

Figure 1 shows the construction of the decision matrix employed for the scanning of input character strings. The columns and rows are related where possible to the syntactic element which they represent as having been formed from the input characters. The cells in the matrix can contain two types of information, separated for readability by commas. Integers indicate the next setting of the row index. Alphabetic names refer to blocks of coding to execute before looking at the next input character. These coding blocks perform the following functions:

M	—	move character to output
MN	—	accumulate as binary integer
SS	—	set sign of number
SD	—	set decimal point position
TAH	—	terminate alphanumeric or hollerith field
ME	—	accumulate as binary integer for exponent
SE	—	set sign of exponent
TN	—	terminate integer field
TF	—	terminate and float binary number
ERR	—	print field error message
END	—	terminate scan

The method of processing is as follows: Initially set the current syntactic element to terminator, i.e., row index = 1. Get an input character and set the column index according to the syntactic category of the input character. Examine the contents of the matrix cell determined by the current values of the row and column indices. If the cell contains an integer, then set the row index for the next input character to this value. If the address of coding to be executed is given, then perform these instructions. In any event, always go back and get the next input character unless a signal was given for input scan termination.

Undefined input character constructions are treated as field errors. When an end-of-logical-record is recognized, the scan is terminated and the condition reported to a higher level calling routine. Physical end-of-file marks are handled by the I/O select routine which feeds the scanner an input character.

For illustration, consider a simple two-field logical record: AE ﾉ̸1.2$ The character "ﾉ̸" is a blank. Figure 2 shows the values assumed by the column and row indices, and the coding blocks executed during each step in processing this logical record.

Input Character	*Row Index*	*Column Index*	*New Row Index*	*Function*
—	1	—	—	Set row index = 1.
A	1	1	2	Move character to output.
E	2	5	2	Move character to output.
ﾉ̸	2	3	1	Terminate alphanumeric field; set row index = 1.
1	1	2	7	Accumulate as binary integer.
.	7	6	8	Set decimal point position.
2	8	2	8	Accumulate as binary integer.
$	8	9	—	Terminate and float number; return to calling routine.

FIG. 2. Illustration of processing a logical record

Summary and Conclusions

Employing the decision matrix technique for analyzing the construction of an input character string proved superior to an ad hoc approach in several ways. One, it provided relatively fast execution times. Secondly, once the syntax rules were decided upon and the matrix constructed, the programming itself became rather trivial, the reason being that the logic was concisely and clearly contained in the matrix itself. Thirdly, probably the most important advantage gained in our case was that the technique allowed simple modification of the syntax rules. By altering the contents of matrix cells, or at most by increasing the matrix size, the syntax and semantics of expected input strings can be drastically altered. This does not necessitate any recoding of the input scan, and at most requires the addition or replacement of blocks of coding to handle the new definitions.

It is suggested that this technique would prove useful for controlling the scan of a format statement specifying the organization of information for output data, or, conversely, for scanning a format statement describing the construction of input data, if this approach was used as opposed to our example in which the context of the input string determined the type of information.

One characteristic evident from the formation of our matrix is the number of empty cells, for example, or filled with ERR in this example. This can be generally true of applications of decision matrices to this type of process. This could result in a considerable amount of wasted space for more complex processes requiring larger matrices. This disadvantage can be overcome, however, by employing a densely packed matrix structure. For instance, see [3].

REFERENCES

1. IBM 709/7090 FORTRAN Reference Manual. Form 28-6054-2 (1961).
2. NAUR, P. (Ed.) Report on the Algorithmic Language ALGOL 60. *Comm. ACM 3* (May 1960).
3. HELLERMAN, H. Addressing multidimensional arrays. *Comm. ACM 5* (Apr. 1962), 207.