A user oriented programming language*

By Melvin Klerer and Jack May‡

An initial version of an operational software–hardware system for the purpose of facilitating the programming and analysis of well-formulated problems is summarized. A modified Flexowriter is used to generate computer-acceptable input when equations or computable requests are typed in much the same manner as they would appear in conventional mathematical texts. The purpose of this paper is to discuss and illustrate the programming language that has been developed to be used for this system.

There exists a numerically large class of users—particularly found in the scientific research establishment or the aerospace industry—whose problems may be characterized as well-formulated. If their problems indeed consist of sets of determinate equations and essentially computable requests, and if adequate methods of solutions are known, then there is no reason in principle why the intermediate steps of programming and analysis should not be done automatically. As has been pointed out in previous publications (see references), existing input devices, such as card punches or console typewriters, are quite inadequate for the direct input of mathematical equations as they are customarily represented. Most present input devices can handle only sequential information, and therefore dictate the essentially linear form which is found in such languages as FORTRAN or ALGOL. However, conventional representation of equations as found in handbooks or scientific texts demands a two-dimensional form. This representation is not only more convenient but essentially leads to less ambiguity.

Our input device is (similar to the MADCAP device, (Wells, 1961)) a Friden Flexowriter that has been modified so that subscript and superscript positioning can be done automatically under keyboard, paper tape reader, or direct computer control. In addition, the 88 available typable symbols have been chosen to give flexibility both in the typing of equations and the construction of mathematical operators of arbitrary size. These may be composed manually by typing stroke by stroke or by pushing an appropriate key in an optional console keyboard.

Our primary interest is the study of the entire interaction of the user, the computer (hardware and software), the problem, and its solution from an experimental point of view. Our fundamental philosophical orientation has been to explore those software–hardware methods that hold promise for furthering the goal of increased automation in the problem-solving process. More specifically, this system has been designed not only as a tool for the study of problem solving but also of self-teaching systems. By this we mean that it teaches the user how to operate itself and, from studying the user response, it teaches us how to modify the system to a more productive and successful use. For example, the first lesson is given by running a prepared paper tape off-line through the Flexowriter. Subsequent lessons are designed to occur as part of the experience in the using of this system. In part this is done by having the system print out the way it interprets the user input. Thus, in a certain sense we are asking the user "to play the game" without first telling him all the rules of the game. He learns whatever rules he needs, depending on the type of game he plays, that is, for the type of problem he presents. For example, if the system has interpreted a sequence of characters as a string of variables with implied multiplication it becomes apparent to the user that he has forgotten to predefine the set of characters in the manner which will be later indicated. The basic idea is the attempt to fashion an approach so that the user learns how to program the system in a purely informal fashion. At the same time the occurrence of dubious input forms gives us an empirical basis for modification of the system to decrease the frequency of output which must be corrected or redefined. In common with this attempt to design a self-teaching system is also the attempt to determine how concise one can make a user reference manual without impairing its practical utility. Our present very limited experience suggests that one $8\frac{1}{2}$- by 11-inch sheet printed on its two sides can be quite adequate. Thus, this one-sheet manual includes the vocabulary list, examples of how certain mathematical operators can be typed using the available symbols, some simple rules of presentation in typing out problems, and some examples of recommended forms of presenting certain types of problems. More details are given in Klerer and May (1964) but we can note that our actual pedagogic experience is at the moment entirely too limited to reach even a preliminary conclusion as to the success of this approach, either in its utility for education of the user to computer usage or as a research method for the system designer. Our further experience in this area will be recorded in future reports.

* Hudson Laboratories of Columbia University Contribution No. 201.
This work was supported by the Office of Naval Research under Contract Nonr-266 (84).

‡ *Columbia University, Hudson Laboratories, Dobbs Ferry, New York.*

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024

The programming language

Our criteria for designing a programming language have been as follows:

(1) There should be less human effort involved in communicating with a computer:
  (*a*) less instructions and therefore less errors;
  (*b*) less total time spent in coding a problem and debugging the problem;
  (*c*) less total time spent in setting up a system type of approach for a solution of the problem;
  (*d*) less higher level thinking necessary to solve the problem.

(2) The system should be easy to learn and therefore be subject to universal use.

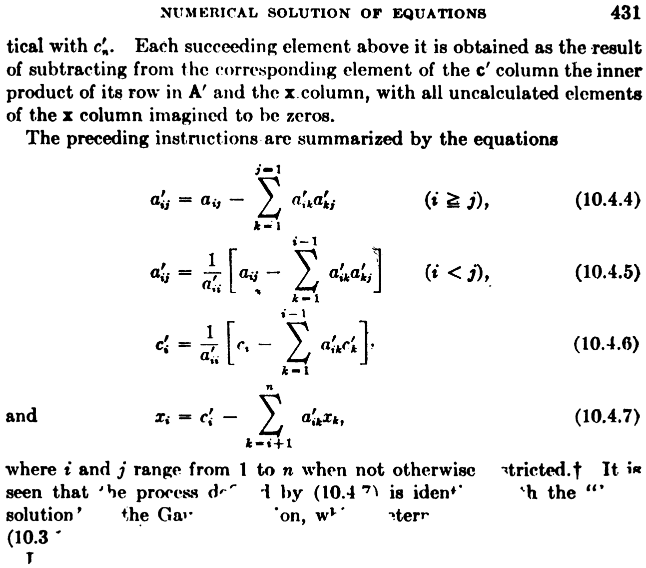
(3) It should be adaptable to a wide range of problems and applications.

(4) The automatic final product should be better than the hand product, that is, not only cheaper to produce but of better or equal quality than the average hand-made version. In this case it means that the machine-code programs should run at least as fast as those produced by the average programmer using present compiler systems or even machine-language coding. In fact this has been the stumbling block of most compilers. In general they are relatively inefficient. While our compilation time lies in the middle range between the slowest and the fastest alternate compiler systems offered by the manufacturer of our General Electric 225 computer, efficient object programs are produced. Typical calculations that have negligible data input run anywhere from two to four times faster than equivalent programs produced by the manufacturer's compilers.

However, our fundamental orientation in designing the language is not only that it should be easy to use but also should permit addition of alternate optional forms as a product of experience with the system. Our experience so far has been that it is possible to realize a quite flexible language even given the limitations of present machines. While some of our elementary forms are similar to the MADCAP language developed by Mark Wells (1961, 1963), we believe we have gone further, not only in the flexibility of representing all sorts of mathematical operators but also in the elimination of stylized restrictions for setting up multi-indexed and compound loops. In addition, our orientation toward language design has taken a somewhat different road, being intimately tied to the concept of a self-teaching system which can be used as a tool for research into the areas of automatic problem solving.

Illustrations

Fig. 1 is a photograph of a page from Hildebrand (1956) illustrating his prescription for the Crout method of solving linear equations. It is quite typical of the kind of thing that the unsophisticated user tries to



NUMERICAL SOLUTION OF EQUATIONS 431

tical with c'_n. Each succeeding element above it is obtained as the result of subtracting from the corresponding element of the **c**′ column the inner product of its row in **A**′ and the **x** column, with all uncalculated elements of the **x** column imagined to be zeros.

The preceding instructions are summarized by the equations

$$a'_{ij} = a_{ij} - \sum_{k=1}^{j-1} a'_{ik}a'_{kj} \qquad (i \geq j), \qquad (10.4.4)$$

$$a'_{ij} = \frac{1}{a'_{ii}}\left[a_{ij} - \sum_{k=1}^{i-1} a'_{ik}a'_{kj}\right] \qquad (i < j), \qquad (10.4.5)$$

$$c'_i = \frac{1}{a'_{ii}}\left[c_i - \sum_{k=1}^{i-1} a'_{ik}c'_k\right], \qquad (10.4.6)$$

and

$$x_i = c'_i - \sum_{k=i+1}^{n} a'_{ik}x_k, \qquad (10.4.7)$$

where i and j range from 1 to n when not otherwise ...tricted.† It is seen that ...he process d... ...d by (10.4 ...) is iden... ...h the "... solution' the Ga... ...on, w... ...terr (10.3 ...

Fig. 1.—A typical textbook formulation of the Crout method

program straight out of a text or handbook. As long as the equations are fairly well behaved and the set is not too large he will not get into trouble. **Fig. 2** is a version of Hildebrand's prescription as typed on our Flexowriter. It can be noted that the body of the typescript is a fairly reasonable transliteration of the text indicating double subscripting, large and small summation signs, implicit multiplication and a non-linear division format. One might also note that the case for $i = j = 1$ results in an ambiguous form which the system interprets properly.

Fig. 3 is an illustration of the fact that the operator symbols need not be symmetric or well composed, since the system will correctly recognize the conventional forms even when they are not symmetric or not centred with respect to the rest of the equation. Mistakes in making these symbols are very easily rectified since it is done in much the same manner as on an ordinary typewriter; that is, one back-spaces to the error and simply types over it. As an optional procedure it is also possible to back-space to the error and press a special button called "Erase" which is properly interpreted by the system. However, none of these operator symbols exist on the keyboard; they are composed from elementary strokes, diagonals, bars and special braces. They can be formed directly from the typewriter keyboard or they can be formed automatically by use of the auxiliary keyboard console which contains a set of 127 keys to control the Flexowriter remotely. At present about 80 of these console keys carry photographs of common mathematical symbols or canonical phrases. For example, there are small, medium-size and large-size summation signs and also alternate variations for the different types of upper and lower limits. There are various size square brackets and braces in addition to various size square roots and other

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024

DIMENSION A=(20,20), C=20, γ=20, X=20, α=(20,20).

READ n.

READ A_{ij} FROM j=1 TO n AND i=1 TO n.

READ C_i FROM i=1 TO n.

FROM j=1 TO n AND i=1 TO n IF i>j THEN $\alpha_{ij}= A_{ij}-\sum_{k=1}^{j-1} \alpha_{ik}\alpha_{kj}$ OTHERWISE $\alpha_{ij}= \dfrac{A_{ij}-\sum_{k=1}^{i-1} \alpha_{ik}\alpha_{kj}}{\alpha_{ii}}$.

FROM i=1 TO n COMPUTE $\gamma_i= \dfrac{C_i-\sum_{k=1}^{i-1} \alpha_{ik}\gamma_k}{\alpha_{ii}}$.

FROM i=n BY -1 UNTIL i<1 COMPUTE $X_i= \gamma_i - \sum_{k=i+1}^{n} \alpha_{ik}X_k$.

PRINT i(2), X_i FOR i=1, 2, ..., n. FINISH.

Fig. 2.—The Crout method as typed on the modified Flexowriter

convenient operators such as integral symbols and product operators. Production of the equivalent symbol on the typewriter is effected by simply selecting and pushing the proper key on the auxiliary console. The remaining keys on the console are used for initiating the first lesson on the Flexowriter and for demonstration purposes, while others are reserved for future implementations and extensions of the system into the area of direct user-machine communication.

Fig. 4 is an example of three equivalent programs. Each results in non-trivially different machine codes, but, of course, yields the same numerical answer. The first program is certainly the easiest form. The second program illustrates how one can use quite primitive commands if the special nature of the problem so warrants. The third program illustrates how "DO" type statements are handled. The command "DO FORMULA 3 FROM X = O TO W" will cause all statements up to but not including formula 3 to be executed for each value of X = O to X = W in unit steps. A more general "FROM" form is permitted as will be indicated later. The point here is that the system has been designed not only for the novice but also for the expert programmer and also for those situations where, for reasons of efficiency, it is necessary to use primitive commands.

Our explicit vocabulary list consists of only a little more than a hundred function names such as COSINE, command verbs such as DO, connectors such as OR and verbalization of arithmetic operators such as PLUS. But one can consider the vocabulary list to include implicitly all the recognizable composed symbols such as the various size sum operators, product operators, square roots, and all the various types of brackets that are recognizable. Of course, the normal arithmetic symbols, the Greek symbols, the upper-case alphabet, a partial lower-case alphabet, subscripted and superscripted quantities, and the corresponding different alphabet when typed in red is admissible vocabulary to the language. (A red B is interpreted as a different variable from a black B.) Certain words are inherently ambiguous and must be interpreted within the context of their use. For example, blanks are usually ignored or are simply interpreted as word delimiters. However, in certain cases when implied by the context, the absence of connectives such as the comma or "AND", that is the presence of a blank, will be properly interpreted. In this sense, there is a grammatical analysis of the individual statement and missing logical connectors are filled in as the context permits it. Of course, this type of grammatical structuring has only a remote connection with producing object program efficiency. We do it

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024

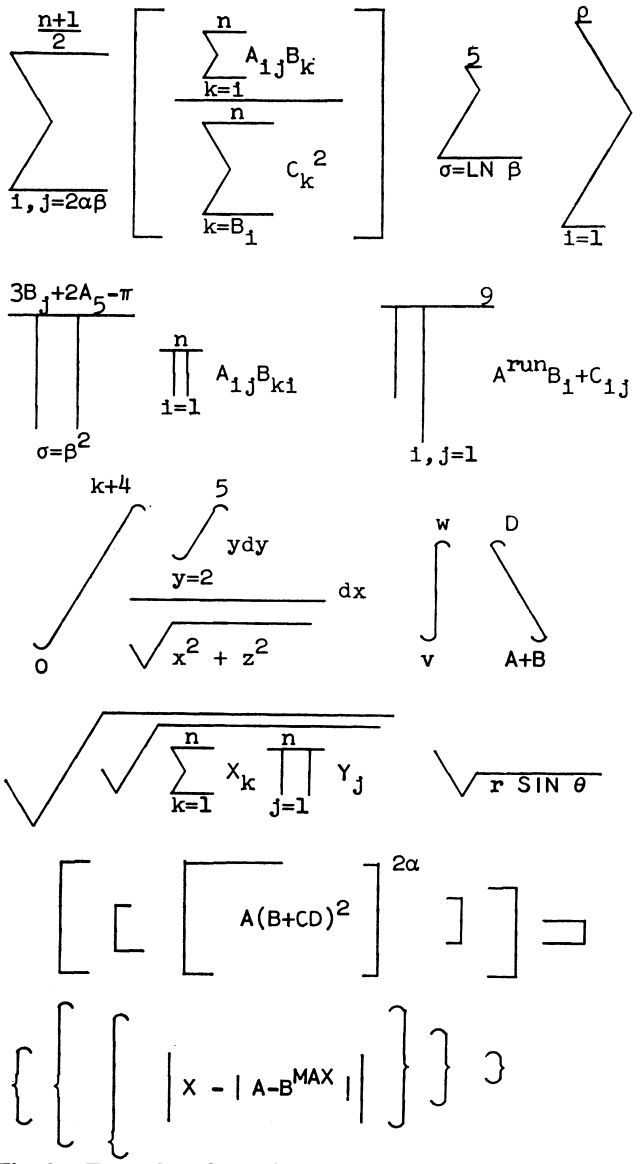


Fig. 3.—Examples of well formed and badly formed composed symbols which are recognized correctly

simply for the purpose of allowing a more flexible language, and also to produce a tighter compiler by producing canonical forms at some intermediate stage in the compiling aspect.

Any system of mechanical discourse, no matter how flexible, must have some rules explicit or implicit, and our system is no exception to the fact that programming language rules are, in general, a function of machine structure. Our rules are not those of natural language. They are simply a reflection of the finite and we think quite small memory of available machines. Therefore, it is for us a simple matter of convenience to require that the end of a statement be signalled by a period and the end of a program by the word FINISH or the phrase END OF PROGRAM. We also require that the names of variables with more than one character should be defined by a "special variables" statement. This is not too stringent a requirement as the normal upper case alphabet, a partial lower case alphabet and a partial set of Greek symbols may essentially be doubled by simply typing in red. If a variable has not been predefined then the system assumes that its initial value is zero. Superscripts are distinguished from exponents by simply typing the superscript expression in red. Comments are introduced by simply enclosing them in between a set of special braces { } at any convenient place in the text. These are not necessary restrictions and we could have avoided them if we had wished, but at the expense of a more elaborate translator than we wanted to construct on our first try. For example, while a dimensioning statement is not necessary for singly or doubly subscripted variables, as long as their range is somehow explicitly indicated in a program, dimensioning is necessary for those variables whose dimension is a function of the input data. (This restriction will be eliminated in the future versions of the system.) Nonetheless we feel that the present system has been fairly successful in limiting the number of explicit rules. Of course the system, by design, has inherent in it a certain other number of implicit rules, and these become plain to the user only,

```
MAXIMUM n=20.
READ n.
READ A(1), B(1) FROM 1=0 TO n.
X= SUM(i=0,n) { A(1) PRODUCT(j=1,n) B(j)A(j) }.
PRINT X.  FINISH.
```

```
DIMENSION x=20, y=20.
α=0, READ ω.
FORMULA 1.  READ x(α), y(α).
α=α+1. IF α≤ω GO TO FORMULA 1.
S=α=0. STATEMENT 1. β=α, P=1.
STATEMENT 2. P=Px(β)y(β), β=β+1.
IF β≤ω THEN GO TO STATEMENT 2.
S=S+Px(α) AND α=α+1.
IF ω≥α GO TO STATEMENT 1.
PRINT S.  END OF PROGRAM.
```

```
MAXIMUM W=20.
READ W.  ρ=0.
FROM X=0 TO W READ u(X), v(X).
DO FORMULA 3 FROM X=0 TO W.
σ=1.
FROM Y=X TO W COMPUTE σ=σu(Y)v(Y).
ρ=ρ+u(X)σ.
FORMULA 3.  PRINT ρ.
END OF PROGRAM.
```

Fig. 4.—Three alternate formulations of the same program

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024

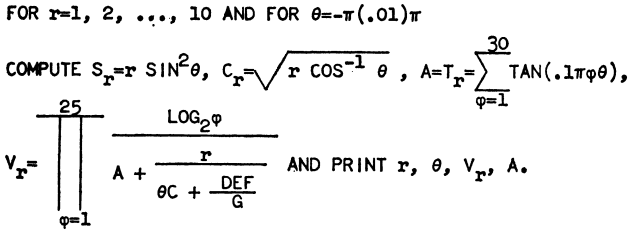


Fig. 5.—An example of an implicit loop

in fact, when he programs those problems where it is necessary for the system to point out that he has not used a format which is an acceptable mathematical or language form, or which has some inherent ambiguity. Our point of view here is to eliminate a good deal of formal programming instruction. As long as these cases are fairly uncommon, then the system can be considered to be successful in anticipating usage which is not explicitly covered in the one-sheet system manual.

In subsequent examples, we use the letters E, F, or G to denote arithmetic expressions, that is, E may denote the expression A + 2B + i; otherwise a single variable is meant. Braces denote a choice of forms. Square brackets denote those forms that are optional, i.e., need not be present. As an example to indicate the range of indexing, there are several possible variants of FROM or FOR forms. The general FROM phrase can be written as:

$$\text{FROM } i = E \text{ [BY F]} \left\{ \begin{matrix} \text{TO} \\ \text{UNTIL} \end{matrix} \right\} \left[j \left\{ \begin{matrix} = \\ < \\ \text{etc.} \end{matrix} \right\} \right] G$$

e.g.: FROM i = E TO G (Unit steps assumed)
FROM i = N BY 2·34 UNTIL i > A + B
FROM A = B + 5 BY 2 UNTIL Q = 20

or in its more restricted form as

FOR P = N, N + ΔN, . . ., M
FOR Q = N(ΔN)M

e.g.: FOR i = 1, 3, . . ., 49
FOR j = 0, 0·5, . . ., 7·5
FOR j = 0(0·5)7·5

Notice that our motivation here has been to use those forms which are conventional for ordinary mathematical discourse. Naturally the user is free to use either symbolic, fixed or floating-point numbers, as appropriate, and any upper- or lower-case letter may be used for symbols as long as its value has been prespecified in the program. Grammatical use of these FROM or FOR forms is likewise non-restrictive; for instance, they can be used either to begin or end a statement. For example, one can type COMPUTE $X_j = A + B_j$ FROM j = 1 TO 10 or, equivalently, FROM j = 1 TO 10 COMPUTE $X_j = A + B_j$. In fact, if it is permitted by the context, it can be used at any suitable place within a statement as will be shown in later examples.

The general structure of the IF statement can be represented as:

$$\text{IF F} \left\{ \begin{matrix} \leqslant \\ < \\ = \\ > \\ \geqslant \end{matrix} \right\} \text{G THEN} \left\{ \begin{matrix} \text{Y} = \ldots \\ \text{READ} \ldots \\ \text{COMPUTE} \ldots \\ \text{GO TO} \ldots \\ \text{CONTINUE} \end{matrix} \right\}$$

$$\left[\left\{ \begin{matrix} \text{ELSE} \\ \text{OTHERWISE} \end{matrix} \right\} \left\{ \begin{matrix} \text{Y} = \ldots \\ \text{READ} \ldots \\ \text{COMPUTE} \ldots \\ \text{GO TO} \ldots \\ \text{[CONTINUE]} \end{matrix} \right\} \right]$$

where both F and G, as previously noted, can be arithmetic expressions. An alternate form to handle more complex logical conditions can be typed as:

IF τ = G OR (E < F AND Q = r SIN² θ) . . .
THEN . . .
[OTHERWISE . . .]

There is no restriction on the number of "AND's" and "OR's" and parentheses are only necessary to resolve ambiguous forms.

In order to relieve the user of the burden of setting up "DO" or "BEGIN-END" type loops in a stylized form for many problems, we permit what we term an implicit loop, for example:

FOR i = 1(1)50 AND j = 0 BY 2 UNTIL Y > 2000
READ X_{ij}, Y{i, j} = $2X_{i,j}$ AND PRINT Y.

IF A < B PRINT A, C = D + E AND GO TO STATEMENT 1 ELSE GO TO FORMULA 2.

In our opinion this is an extremely powerful and non-trivial form. Aside from object-program memory size there is no intrinsic limit on the number of controlling indices or number of sub-statements within the body of the loop.

Fig. 5 indicates a loop and two indices (r and θ) controlling five sub-statements. π is interpreted in its numerical sense, that is, as 3·141 . . .; secondly, use is made of implied multiplication in the form for S_r together with the conventional SIN² form, T_r contains a convenient summation form and V_r the product notation. Sums and products may be several and intermixed. Commas are used interchangeably with "AND" (or blank spaces) to separate sub-statements within the implied loop. One can also write a statement such as:

READ A_i,

$$\text{COMPUTE Y} = \frac{A_i}{A^{MAX}} \text{ AND PLOT Y, i, } -1, 1$$

FROM i = 1 UNTIL Y ⩾ 1.

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024

This is an example of a simple loop controlling three separate statements controlled by one index i, but whose termination value is a function of a value generated by one of the sub-statements in the loop and not known before entering the loop. To illustrate this type of form within an "IF" statement one can type:

IF $\alpha > k$ COMPUTE $x = \sqrt{(\alpha - k)\Delta}$, $Y = B_{ij}x + C_0T$ AND PRINT Y, α, T, k OTHERWISE COMPUTE $x = 2\alpha k$, $Y = B_{ij}x + C_0T\Delta$ AND PRINT Y, α, T, k FROM $\alpha = 1$ TO n WITHIN $T = 2$ BY 0·01 UNTIL 3 AND FOR $k = 0(5)90$.

or

IF ($X \leqslant Y$ AND $\gamma < 0$) OR $|42 - \gamma/\epsilon| > (X - Y)^2$ THEN COMPUTE $T_{XY} = \gamma\epsilon^2 - \frac{X}{Y}$ AND $W = (YT_{XY})^{\gamma\epsilon}$ AND PRINT W, T_{XY}, X FROM $\gamma = 2k + 3$ BY $0{\cdot}01\tau$ UNTIL $W > 5800$ AND FROM $X = 1$ TO 100 OTHERWISE GO TO STATEMENT 2.

The first example indicates an IF type statement with only one Boolean variable resulting in three separate computations if true and also three separate computations if false, controlled by three indices α, T, and k. The word "WITHIN", as used above, is synonymous with AND FROM or FROM. The second example indicates how easily Boolean forms may be handled. The vertical bars have the conventional mathematical meaning as the absolute-value function.

At the moment our translator occupies so much of the memory that we must limit a typed line in its most general form to not more than 192 characters lengthwise by 40 half characters vertically, which is equivalent to a typed two-dimensional display of 19 inches long by not more than $3\frac{1}{3}$ inches deep. Statements may be continued on to as many lines as desired (as long as the compiled object program does not exceed the available memory). The above examples are also instructive for the perhaps unique way in which a comment form is used; that is, Y{i, j} is used to indicate the functional dependence of Y on i and j even though, literally, any expression between such simple braces is treated as a comment and does not generate code.

Of course, data can be typed on the Flexowriter, but there is no escaping the present general use of cards for the input of data. In an attempt to avoid undue restrictions, our card format is free field; that is, the number of data points may vary from card to card and may be either in fixed or floating point form,* or mixed without the need of any pre-defining information. For example, there may be one datum on the first card, five data on the second card, three on the third and so on. The system does the necessary housekeeping. Of course, not only can one read and punch cards but also one can control a high-speed printer in the usual way by such commands as PRINT Y_i FROM $i = 1$ TO n. Magnetic-tape operations are also provided for but they are essentially machine dependent. For reasons of efficiency we have avoided a general store or retrieve command where the user would have no control over the storage medium. To make a general store and retrieve command that relieves the user of the need to specify the actual storage medium such as magnetic core, discs, tape, etc., and yet is efficient requires some complex decision structures which are not present in the initial version.

Subroutines and procedures are defined by using the words SUBROUTINE (Name) or PROCEDURE (Name) before the named procedure and the words END (Name) at the proper terminal point. Entry into the procedure is by the use of the words CALL (Name), and return to the main program at any desired point is affected by use of the word RETURN.

As part of the input–output command system one also provides a simple plotting command such as PLOT Y, X, A, B. (A and B specify the minimum and maximum of Y, respectively, as a function of X.) Column labels and printer or typewriter messages are provided for by simple commands, such as PRINT MESSAGE . . . or TYPE . . ., PRINT LABEL . . ., PRINT HEADING . . ., etc.

In summary we might state that this is an operational, practical and effective system not uneconomic to implement. The present operational version has been designed to radically reduce programming effort for both the novice and the expert programmer, with no penalty paid in object program efficiency. But this has been accomplished by internal coding which is very machine dependent. We do not think we have exhausted techniques for improving efficiency, and more general methods for future implementation are being explored.

Our system efforts are just beginning in areas of automatic numerical analysis, decision problems and the more sophisticated areas which we lump into the general heading of automatic problem solving. Our perspective in this regard is more adequately covered in Klerer and May (1964).

* Typical card forms for floating point would be 2·3456, 23·456T-1, 0·23456T1, 234·56E-2, or 0·23456E1.

Acknowledgement

We gratefully acknowledge the engineering assistance given us by M. Epstein and the encouragement of R. A. Frosch and A. Berman. Fig. 1 is reproduced by permission of the McGraw-Hill Book Company.

References

BALKE, K. G., and CARTER, G. (1962). "The COLASL Automatic Coding System," Dig. Tech. Papers, ACM Nat. Conf., pp. 44–45.

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024

GAWLIK, H. J. (1963). "MIRFAC: A Compiler based on Standard Notation and Plain English," *Comm. ACM*, Vol. 6, pp. 545–7.
GREMS, M., and POST, M. O. (1959). "A Symbol Coder for Automatic Documenting," *Comp. News*, Vol. 147, pp. 9–18, and Vol. 148, pp. 15–19.
HILDEBRAND, F. B. (1956). *Introduction to Numerical Analysis*, New York: McGraw-Hill Book Co.
KLERER, M., and MAY, J. (1963). "Algorithms for Analysis and Translation of a Special Set of Computable Mathematical Forms," Columbia U., Hudson Labs. Tech. Rept. No. 113.
KLERER, M., and MAY, J. (1964). "An Experiment in a User Oriented Computer System," *Comm. ACM*, Vol. 7, pp. 290–4.
LOS ALAMOS SCIENTIFIC LABORATORIES (1958). "Maniac II," *Comm. ACM*, Vol. 1, p. 26.
VANDERBURGH, A. (1958). "The Lincoln Keyboard—a Typewriter Keyboard Designed for Computers Input Flexibility," *Comm. ACM*, Vol. 1, p. 4.
WELLS, MARK B. (1961). "MADCAP: A Scientific Compiler for a Displayed Formula Textbook Language," *Comm. ACM*, Vol. 4, pp. 31–36.
WELLS, MARK B. (1963). "Recent Improvements in MADCAP," *Comm. ACM*, Vol. 6, pp. 674–8.

A special-purpose compiler

By C. B. Jones*

This paper describes how a particular computer application was tackled using a "Compiler" approach, and how the work is to be extended.

Computer applications give rise to many differing forms of files. These files are usually organized so that the basic record contains details of options within the item type. Thus in the case of an order file, a customer order record contains an item number and a set of options within that item. This paper will consider motor vehicle order files in which the *options coding* is both complicated and extensive: a complete vehicle, with its options, can be coded on to an 80-column card which will be referred to as a V.O.C. (*vehicle order card*).

On the V.O.C. each vehicle option is allocated a field (frequently only one column). The value in the field indicates which of a number of possibilities is required for the particular vehicle. The size limitation of an 80-column card necessitates that a number of conditions cannot be coded directly, and are, therefore, implied. Thus, in some cases, it is necessary to examine the contents of several fields before an option is found to be required.

The requirements for a particular vehicle are coded into the V.O.C. manually. The information contained on the cards is interpreted by a number of computer programs for a variety of reasons, e.g. data vetting, parts scheduling and model costing. One of the programs is a translation program which will convert the format and coding of these cards.

The translation program

The existence of more than one format for the V.O.C. posed a communication problem. These formats differed, not only in the position of the fields, but also in the way in which the basic information was coded. Information for a field in one format may have been inferred from a number of fields in another format.

Example: the following trivial translation problem will be used as an example throughout the paper.

Base coding Col. 21 is type of vehicle:

Codes
1 Car A
2 Estate A
3 Car B
4 Estate B

Col. 22 is number of doors:

Codes
2 Two doors
4 Four doors

Required coding Col.31 is body type:

Codes
1 Car
2 Estate

Col. 32 is car type with number of doors:

Codes
1 Car A, two doors
2 Car B, two doors
3 Car A, four doors
4 Car B, four doors.

The complete translation problem from the base to the required coding could be specified in a number of statements, which assign a value to a field of the new card subject to a set of tests on certain fields in the old card. Thus, a typical statement would be "put a 1 in column 31

* *Ford Motor Company Ltd., Central Office, Warley, Brentwood, Essex.*

Downloaded from https://academic.oup.com/comjnl/article/8/2/103/390405 by guest on 20 April 2024