T H E   U N I V E R S I T Y   O F   M I C H I G A N

Technical Report

The Engineering Assistant:
DESIGN OF A SYMBOL MANIPULATION SYSTEM

Edgar H. Sibley

CONCOMP: Research in Conversational Use of Computers
ORA Project 07449
F.H. Westervelt, Director

supported by:

DEPARTMENT OF DEFENSE
ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION   ANN ARBOR

August 1967

ABSTRACT

This paper presents the design of a symbolic manipulation system from the viewpoint of a series of questions on user requirements, computer speed, generality, and extensibility. It then describes a working system designed for experimentation of computer aids in engineering algebras of various regimes, intended to be as general as possible in application.

CREDITS

Part of the work reported in this document was made possible through the support and sponsorship extended to the Massachusetts Institute of Technology, Electronic Systems Laboratory, by the Manufacturing Technology Laboratory, RTD, Wright-Patterson Air Force Base, under Contract No. AF-33(657)-10954, M.I.T. Project No. DSR 9442. Part of the work was performed in the Design Division of the Mechanical Engineering Department, M.I.T.

Work reported herein was also supported in part by Project MAC, an M.I.T. research program (Office of Naval Research Contract No. NR-4102(01)), and by the Concomp Project, a University of Michigan research project (Contract No. DA-49-083 OSA 3050). Both Project MAC and the Concomp Project are supported by the Advanced Research Projects Agency of the Department of Defense.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This article is intended primarily to describe the design of a presently working interactive symbol manipulation system, but the reasons for a given course of action and the implications of it are common to the design of other languages. The purpose of this introduction is therefore to formulate the questions that must be asked prior to finalizing the design of such a system, to discuss possible alternatives, and to suggest the implication and effect of the alternatives.

At the start of any computer project, decisions are made. These decisions are sometimes based on expediency, such as local availability of source languages, sometimes on knowledge of a specific softwear scheme, and sometimes on the organization of the particular machine being used. Whatever the reasons for them, their effect will be felt both by the designers (or implementers) and users of the system. Many people in the computing industry, however, tend to assume that certain basic practices are Universal Laws. The system designer should consider whether these sacred cows are necessary or reasonable. Examples will be given later, but typically we have the question of fixed syntax, using predefined precedence, excluding postfix operators, etc.

Thus some of the questions revolve around the syntax and semantics of the symbolic language being manipulated, and the system commands that effect manipulations, while others center on the utilization of space and computer time, and affect the way that the data and procedures are stored within the machine. The answers to these questions must devolve on the anticipated use and generality of the system, and it is obvious that there will always be a spectrum of symbolic manipulation systems, varying from the specialized, rapid, space-limited system to the slow, space-filling, general system. The first of these will be good for solving special one-shot problems, while the other may be useful for more general-purpose problem solving and as an experimental tool for designs of the former. The prime intent of the system described here falls in the latter category; it is not intended to be a machine-economical system.

We have now, therefore, described our first design question:

I. Is the system intended to be used for fast "production" type operations, or is it to be a general experimental tool?

Obviously the decision rests somewhere within a relatively large range, and the answer to this will affect some of the later questions.

Several considerations will affect the rigidity of the system. These have to do with the way of storing the data, the way of operating on the data, and the ability to add to the system. They are characterized by the type of data storage, the type of procedural language, the storage of the procedural language, and the syntax and semantics of both the command language and the symbolic language. To illustrate this, consider the LISP[1] language, which uses the same structure for both its data and procedures, which allows procedures to be used as data by other procedures. In such a system, where procedures are treated in the same way as data, the procedures are modifiable both by the programs as well as the user. They also provide a variable syntax and semantics of both command and symbolic language. LISP and similar languages are therefore extremely general, but they are relatively difficult to program by a novice. At the other end of the scale, there are languages like FORMAC[2] which work with a fixed syntax, are relatively fast, are not simply modified, but can easily be programmed by anyone with FORTRAN IV capabilities.

There are probably three basic ways to store data: in sequential storage in the form of tables; in threaded lists; and in hash-coded associations.* Because of problems of deletion and insertion, tables of sequential storage are seldom used for symbol manipulation. The TRAC[3] language is an exception to this rule. Probably the commonest method of storage of data for symbol manipulation is in the form of lists, e.g., LISP and SLIP[4]. The use of hash-coded addressing in storage of associations, as described by Feldman[5] and others, would be somewhat clumsy for the storage of the main symbol strings, but is very useful if it is necessary to describe some of the attributes of the various elements, e.g., of the operators. Thus the second question is:

II. What is to be the prime method of storing the input data?

With this is associated the question:

III. If user-specified procedures are to be added, how are they to be stored?

and also the question:

IV. If properties of various items, such as operators, are to be specified, how are they to be stored?

Obviously some of these questions will be redundant in a system dealing with fixed syntax.

In the Engineering Assistant, the prime decision of maximum generality requires a non-fixed syntax, with the

---

* These can be looked on as partially content-addressable storage.

ability to define new operators and modify the properties of old operators, where necessary, by the user. Thus the formula parser is intended to be written as a user-modifiable procedure, which could take count of both unary and binary operators, defined as either prefix or postfix, and in the case of binary also as infix. Further generality is also added by allowing the association rules of different operators to be either from the left or from the right. This allows the parsing of expression as either normal algebra, backward or forward Polish, or even mixed, such as the case of predicate calculus or integral calculus:

$$(\forall\ x[\exists\ y(x = y)])$$

$$[\int(\sin x)\ dx]$$

where ∀, ∃, and ∫ can be considered as prefix binary operators, and even:

$$(((2(H\downarrow 2)) + (0\downarrow 2)) = (2((H\downarrow 2)\ 0)))$$

where the implied operators associate to the right, etc. In order to achieve this generality, the properties of separate operators must be capable of user definition or modification and are, in fact, stored in list form, although in this case a hash-coding scheme may have been better.

At this point, another important decision must be made. This is the relation between the input string and its internal representation, and vice versa. In other words:

V. *Is an input statement to be retained in such a fashion that it can better be displayed to the user in a form identical to his input?*

Once again, this is a question with many possible answers. For a fast system, it may be necessary to modify the form in such a way that it is stored or manipulated economically. The degree of difference between the input and output of a sentence may, however, affect the user acceptance of the system. When the user inputs the string "2x" , he may not object to this being modified to "2 x" , whereas he may object to its being changed to "x*2" . The first of these examples merely illustrates the fact that the input program has recognized that the input string consists of two characters and has stored them accordingly, while the second example, having recognized this fact, has inserted a multiplication symbol (*) and transformed the expression into a canonical form where the alphabetic symbols are defined as preceding the numbers. Admittedly this example is contrived, but it is intended to suggest that a canonical ordering and implications may lead to awkward user-expressions. Another example of potential difficulty is found if the expression is changed into Polish form by the input routine. Unless special markers are used to delimit user-inserted parentheses, the expression will not be retained in the output form. For example, if the user's input is (a+b) + (a+b+c), which represents some specific groupings that the user wishers to retain, it would be unfortunate to reprint his expression as a+b+a+b+c or, a+a+b+b+c , because this removes some of the meaning from the original input expression.

For these and many other reasons, the Engineering Assistant is intended to retain the maximum compatibility between an input expression and its later output, with one major difference: blanks are always assumed as delimiters of atoms, and they are not retained internally. This means that the number of blanks in an output string is predicated only by the separation of the separate atoms. In future designs of the system, this lack of generality may be deemed sufficiently important to have the restriction removed.

The above paragraph has already shown two further problems in generality: the delimiting of the atomic parts of an expression, and the inplicit or explicit definition of variables. For although it is quite normal to consider the expression "2x" as consisting of two atoms, it is not necessary; nor is it necessary, or even likely, that the expression "x2" consists of two atoms. Now the atomic elements of most languages are formed on a series of rules of the form, "Numbers cannot appear at the beginning of atomic variables," or, "If a non-numeric and non-alphabetic character occurs in a word, it is assumed to be a delimiter." It would obviously be an advantage to the user to be allowed to define what his particular input rules were. Unfortunately this is extremely difficult, and in the first version of the Engineering Assistant, numbers were assumed to be separated from other characters. The problem of variable definition is discussed in many different ways. The arguments go from the statement: "All parts of the input string which are not numbers or special delimiters will be broken out as variables (sometimes with prescribed maximum length)," through "All single characters are considered variables, other strings must be predefined or are considered errors," to "All character strings which are to be considered as variables must be predefined."

The first of these statements is typical of LISP and the latest version of the Engineering Assistant, the last is used by those who feel that maximum redundancy allows for error checks, while FORTRAN and many other compilers fall somewhere in between. This therefore brings us to the questions:

VI. What are the delimiters, and how are the atomic elements formed?

and also

VII. Are variables to be defined explicitly?

The actual data representation can be either in the form of the input string or lists. Since most formula manipulation systems work faster on parsed expression, internal representation is normally substantially different from external representation. The reason for this is obvious, but may be illustrated by the following example. If we know that x = a+b , and we have the expression y = a+b/2 , can we make the substitution? With a simple string manipulator, it may not be obvious that this substitution is impossible. But if the expression is fully parsed, then we have (y = (a +(b/2))). There is now no match on the string (a+b) in the above expression, and therefore no syntactic substitution error can be made. The question that arises is:

VIII. How far should the internal representation differ from a fully parsed form?

There are probably three different ways of representing an expression in a parsed state: as a string with parentheses inserted (as in the above expression)*, as a series of lists and sublists; or in one of Lukasziewicz's forms. It should be noted that the normal LISP interpretation is highly redundant, being a fully parsed Lukasziewicz notation using a fixed syntax, i.e., predefined prefix operators in the form of lists and sublists, which suggests that the LISP language has certain disadvantages when the expressions are to be stored as normal lists.

---

* The use of "markers" in a string is merely a "parenthesizing" using different symbols.

In the Engineering Assistant, the internal storage is in the form of SLIP lists, with fully sublisted forms, and markers to show whether the parentheses were inserted by the user or by the machine during parsing. The ordering of the operators and operands is, however, retained completely throughout the manipulations, although the lists do not contain the actual operators and operands but pointers to these on special lists. These special lists are necessary in order to be able to determine rapidly whether elements were operators or operands, and in order to use ambiguous definitions of multiply defined operators and operands. This latter fact will be explained later in more detail.

The whole problem of the command language, and the manner in which procedures are written, affects the man-machine interface. The next question is therefore:

IX. How versatile should the command language be? Does it contain debugging aids? Are procedures and command language modifiable by the user?

The presentation of a good human-engineered system will affect its use, and hence the interface must be either extremely well designed or else user-modifiable, or both. Thus JOSS[6] is a relatively simple system, but extremely well human-engineered, which has led to its extensive use. The intent of the Engineering Assistant is to provide the tools necessary to modify the command language and hence to allow a good system to evolve; this will be described in more detail in the next section.

Finally, the design requirements of an engineering manipulation system have been described elsewhere,[7] but they may be summarized as follows:

1. Symbols for operators, functions, or variables do not have unique meanings, for the same symbol or group of symbols may be used for entirely different purposes in different (or even the same) context, e.g., the symbol + in the expression A + kB + C , where A, B , and C are matrices and k is a number.

2. Associated with each operator, function, or variable is a set of properties or values, etc. Thus an operator may be unary, have a precedence with respect to other operators, a numerical evaluation procedure, or a special way of writing its operands (e.g., in the case of exponentiation, the second or "right" argument is diminished in size and elevated above the line, and the operator symbol disappears from the expression). Similarly, a variable may have many properties such as its numerical value, a precision or tolerance, or a special procedure for printing it out (e.g., in the case of a matrix, the numbers are arranged in the box form rather than a long string).

These are derived from the operator-operand concept, and the following operations or procedures are needed by the system:

(i) Many properties of the operator (and its operand) can be best associated directly with that operator (or operand).*

These include such items as:

a. Descriptions of general properties (e.g., UNARY)

b. Descriptions of transformation rules, etc. (e.g., SIN↑2x = 1 - COS↑2x).

* Hence the potential use of hash coded association or relations.

c. Procedures, e.g., for their evaluation if operators or procedures, or their graphic display if an argument.

(ii) The external user does not generally fully parenthesize an expression, but it is not tractable in this condition and should therefore be parsed for internal use. However, this change should not be obvious to the user, unless he specifically requests it.

(iii) Although the user may understand his symbols, the computer may not have enough information to process the expression. This does not matter until user-interaction is needed. The computer should try to "resolve" the expression into components that have meaning, without infuriating the user with questions, and be satisfied with partial resolutions until either the information is forthcoming, or the user asks for aid, in which case questions must be asked to help in resolution.

(iv) Transformation rules should be both simple to input and to apply; they consist of the axioms or theorems of the symbolic system. Application of these rules may involve any or all of three types of conditional actions:

a. matching the rule with the expression or part of the expression;

b. checking the relationship between the parts to see whether the rule can be applied;

c. checking the arguments to see <u>how</u> they are replaced, i.e., there may be conditions associated with the replacement type, etc.*

---

* e.g., in predicate calculus, the axiom $\forall x\, A(x) \supset A(t)$ is applicable only if "A is a well-formed formula, and t is a term free of x in A." Thus the replacement condition is: Replace A(t) by A with all its free occurrences of x replaced by t, but naturally none of its bound occurrences. Then A(x) is replaced by A.

(v) The "evaluation" of an expression involves a general procedure but different methods of evaluation for the different operators and the various regimes or uses of these operators. The process must be recursive, and is as follows:

a. find the "top" operator;

b. find its variables;

c. if the variables are "evaluated," apply the procedure for the operator to the variables;

d. if not, apply the entire evaluation procedure to the sublist(s) until they are "evaluated" and then do the procedure of (c) .

(vi) Many complex operations (macro-operators) are made up of simple operations. These are procedures of symbol manipulation.

2. SYSTEM DESIGN

Before embarking on lengthy descriptions of special parts of the Engineering Assistant program, a short explanation of the system will be given. As described at the end of the last section, certain design decisions were made. The first requirement of the system is a method of definition of the operators, functions, and variables of the user's syntax. For this implementation, the information is arranged in the form of SLIP lists, and the heart of the system consists of the operator-, function-, and variable-definition program, which produces, as <u>data</u> of the system, the operator-, function-, and variable-lists. Figure 1 shows the overall system flow chart; the circles denote program, i.e., non-user-modifiable, parts of the system. The user, interacting with the CONTROL program, may define a new operator, putting

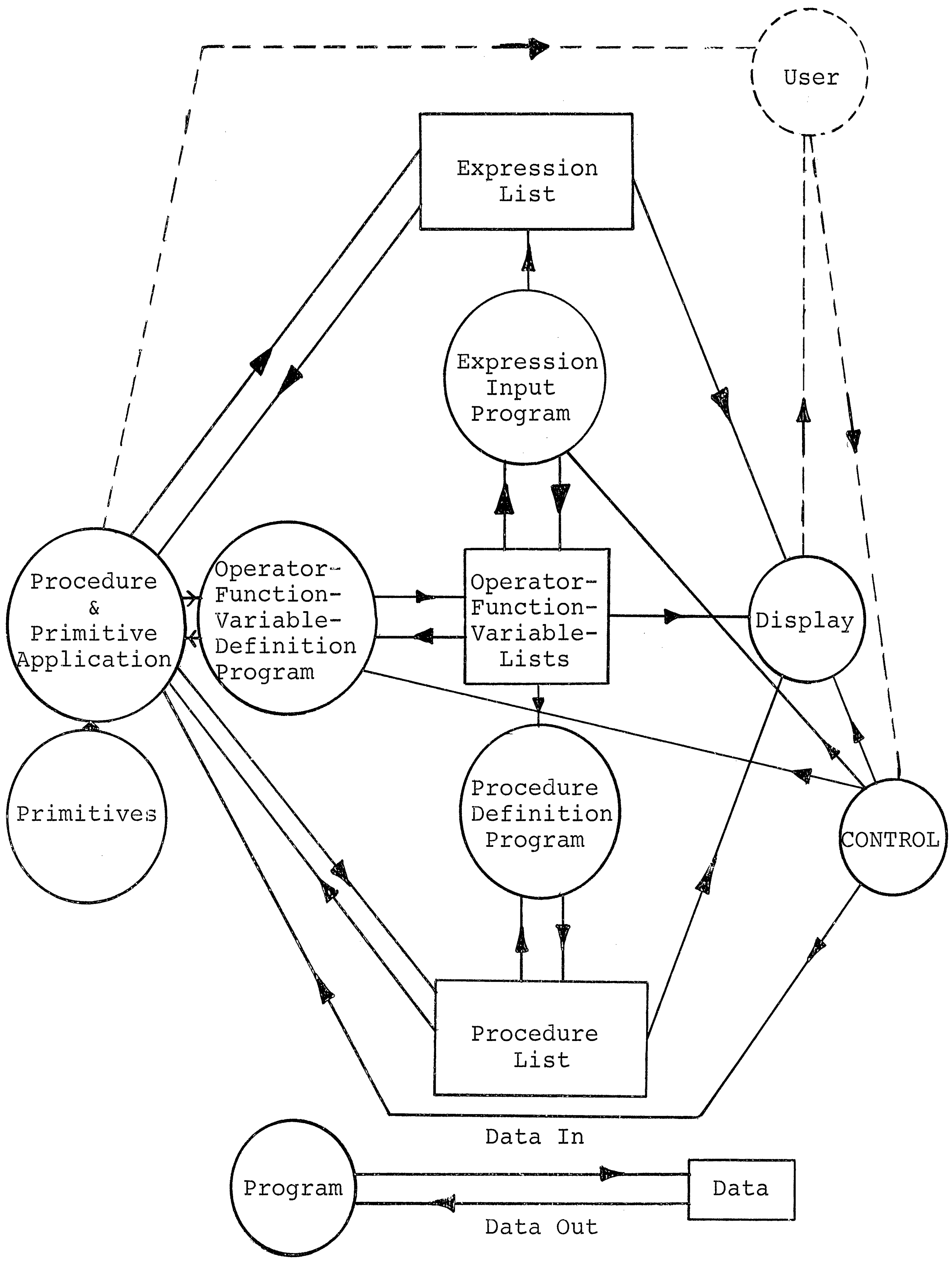

Figure 1: System Flow Chart.

new data into the operator list; in contradistinction, when defining new properties of a previously defined operator, the operator definition program must quiz the list to obtain its location prior to insertion of the new data. The flow of information between this program and its list is therefore in two directions.

The second requirement of the system is a means of inputting an expression. Obviously, unless some special delimiter (such as a blank) is used to break up the input expression into its component parts, the input program must be able to refer to the operator-, function-, and variable-lists to make comparisons, to recognize existing symbols, so as to reduce the expression to its atomic parts. The expression-input program therefore uses information in the operator-, function-, and variable-lists to break the expression into known groups of symbols; it then defines all unrecognized groups as new variables within the system.

Since this program is basically designed for the manipulation and evaluation of expressions, it requires a set of primitive programs for such operations. These primitives range from the extremely general to the highly specific. Since these cannot be changed by the user, they are shown as "program" in the system flow chart. However, because the primitives are not sufficient (in themselves) to do complex manipulations, a means is necessary for concatenating these primitives into a "procedure." Thus a procedure consists of a string of primitives that are executed in sequence except when a conditional statement or a transfer statement allows breaks or jumps in the program. Hence the procedures are the basic programs within the system. In order to input a procedure, the procedural-input program is used. This program "compiles" the procedure into the correct form, making error checks, and produces a procedure list.

Primitives or procedures are applied by another special (interpretive) part of the program that draws processing information from procedures and/or primitives and produces either immediate answers for the user, or generates new expressions or new information for the operator-, function-, or variable-list, or even a new procedure.

The user must also be able to interrogate any of the lists to determine the present state of the data (the user-defined "constants") of the system via a display program. These "constants" are the data in the operator-, function-, and variable-procedure and primitive lists.

The basic or meta-language of the system (the command language) will be discussed later.

The basic system is designed to allow:

1. definition of operators, functions, and variables;
2. input of expressions (which the computer proceeds to break into atomic parts);
3. input of procedures made up of primitives and/or other procedures, which are compiled into a special form;
4. application of primitives and procedures to produce new information by manipulation or evaluation;
5. display of all user-generated information presently available in the system.

To facilitate finding and correcting errors and violations of command language rules, some basic debugging tools have been built into most of the parts of the system.

2.1 System Design Constraints

The particular design constraints in this system involve the method of storing data, length of words, specially defined delimiters, and system "constants."

2.1.1 Data Structure

It is quite evident from the outset that the data of the system (defined originally to include operators, functions, and variables, expressions and procedures) will be constantly changing. New data must be inserted, and old data deleted for almost every operation. The necessity of using lists rather than blocks of storage is therefore almost self-evident. The system flow chart (Figure 1) shows three basic types of lists. They do not all have the same requirements, but they can all be programmed in a slightly modified version of SLIP, which was, indeed, the programming tool used.

2.1.2 Special Delimiters

Delimiters are often needed to determine the scope of an expression, and are taken for granted, or understood, by the user. The computer, however, requires special characters:

parentheses	( )
brackets	[ ]
period	.
comma	,
blank	b̸
carriage return	↩

Because a list is usually delimited by parentheses ( ) or brackets [ ] , these two symbols are given the status of special delimiters. Since they are paired, they give rise to one unfortunate problem: many users are not careful to see that each left parenthesis has its matching right parenthesis. The somewhat arbitrary choice of a zero level count (i.e., using +1 for left and -1 for right parenthesis) was adopted to delimit a listed expression, which means that an additional closing parenthesis in the middle of the expression may destroy information beyond the point where zero count occurs, thus invalidating the expression.

The period (decimal point or binary point) is commonly used to delimit whole from fractional numbers. Were the period used also as a character (such as multiply), it would be impossible to tell whether the string 2.3 stood for a decimal integer, two and three-tenths, or the product of two times three. For this reason, the period is used as a special number delimiter, and its use inside a non-numeric word is forbidden.

The comma is treated as a special character. It is made into a separate word; thus a comma cannot be used within a symbol or variable name.

Because the blank is often used in script as a delimiter, it also was given the status of a special break character, which does not appear in the internal representation.

The carriage return on the typewriter is a line delimiter, and is treated in three ways by the system:

a. if the next line contains further information, the carriage return is a simple delimiter;

b. if the next line is blank (i.e., there are two carriage returns in succession), the expression is assumed to have been closed, i.e., all necessary right parentheses, etc., will be added by the computer to finish off the lists;

c. if the expression has an extra right parenthesis, the carriage return is treated as an end to the expression.

2.1.3 System Constants and Identifiers

Although it may seem arbitrary to differentiate between a delimiter and a system constant, there is a basic difference in the way that these are treated. Thus a number, with or without a period, is treated as a special item in the system, and is "marked" accordingly. The main effect of this is apparent upon read-in of an equation or expression. To conserve space and time, the equation is completely scanned to determine its various atomic parts. Each operator, function, or variable is separated from the string of characters, and the characters are replaced by pointers to the particular list on which the element resides. However, all numbers are left in the list itself, appropriately identified as such. Thus in the expression A+2 , A is replaced by a pointer to A (on the variable list) and given an identification mark of 6; + is replaced by a pointer to + on the operator list, and given an identification mark of 4; and 2 is left on the expression list, but marked with a 1 for integer.

There is one further system constant: the "absence of information." It is always difficult to decide which character to use for this purpose, for the all-zero or all-blank can be valid information. We arbitrarily decided to use the octal number 5700570057 as the null of the system, although almost any other non-printable set would be acceptable.

2.2 Definition of Operators, Functions, and Variables

There are two principal uses of the operator-, function-, and variable-lists (henceforth abbreviated to OFV lists), which materially affect the form of the list. The first use is during input of expressions; the second is for storing the element properties.

2.3 Input of Expressions

The simplest way to input an expression is to read it character by character, and store it in exactly the same way. This retains the exact form of the input, and insures that the meaning will not be distorted. Such a string form, however, is extremely space- and time-consuming and is difficult to manipulate within the computer.

If we assume no previously defined OFVs, the input is constrained only by the system design constraints. Thus, given a string of characters, each symbol is scanned to determine whether it is one of the special delimiters or a number. When one of these is found,it is either broken out, or appropriate action taken (e.g., a sub-list is formed for a left parenthesis or ended for a right, etc.). However, user-defined operators and variables are broken out as separate words.

There is one further feature, illustrated in Figure 2, that is of importance. If there are two operators, e.g., SIN and SINH, the larger operator is broken out of the expression SINH Y, giving SINH Y instead of SIN HY. This feature is also important if the operator SIN↑ denotes exponentiation of the trigonometric function.

One further fact, which may not be obvious, is that new variables are defined by the read-in program for all "residues." These "residues" are interpreted temporarily as variables. Thus *C is a varaible if * and C have not previously been defined, and unless subsequent information and editing show otherwise.

```
DESCRIPTØR FØR ØPERATØR BINARY
```

These input statements define the operators and variables for Equation 1

```
DESCRIPTØR FØR VARIABLE TYPE

SET NEW ØPERATØR + TØ (BINARY, YES)

SET NEW ØPERATØR - TØ (BINARY, YES)

SET NEW ØPERATØR SIN TØ (BINARY, NØ)

SET NEW ØPERATØR SINH TØ (BINARY, NØ)

SET NEW VARIABLE A TO TYPE""""(TYPE,NUMBER)
```

" erases the previous character

```
SET NEW VARIABLE B TØ (TYPE, NUMBER)

EQUATIØN 1
```

Input of Equation 1

```
A+B*C-F(SINC+SINHY)

SHØW EQUATIØN 1
```

Computer representation of Equation 1

```
A + B * C - F ( SIN X + SINH Y )
```

NOTE: USER commands are distinguished by a solid line to their left.

FIGURE 2. Computer-Generated Output of Expressions.

2.4 Basic System Primitives

The intent of the primitives is to allow as many basic operations as possible on lists and data, and also to give as many useful subroutines as possible to the user. The primitives may be considered to fall into three main categories:

1. basic computer-type primitives, such as "addition of two numbers A and B";
2. basic list-type manipulations (including many of the primitives of SLIP), such as "return the top word of list A ";
3. operations on the various parts of the OFV lists, such as "give the description for the Nth item of operator A ."

One further programmed item, which falls somewhere between a primitive and the more elaborate compound primitives or "procedures" of the next section, is the conditional. It is of the form: IF(A, B, C, D, E, ... , Z) , where A, B, C, etc. are expressions, and the number of arguments must be odd. The conditional evaluates the first and subsequent odd arguments until it finds one that is true. When it has found one, it evaluates the next argument and returns this as its value. If there is no match at the last pair, the final argument is evaluated and returned as the value of the conditional. Thus we have: if A is true, give B ; otherwise if C is true, give D ; ... ; otherwise give Z .

2.5 Input of New Procedures

The definition of a new procedure is something akin to writing a program in a special language: that of the Engineering Assistant. This will often involve a knowledge of the basic system primitives, some previous procedures, and certain special control words. When a procedure has been written (either on-line or off-line), it is read in by a special procedural input program, which makes a substantial number of error checks, and produces a sort of compilation which takes up less space and is faster to manipulate than the original BCD strings. Figure 3 gives examples of a few extremely simple procedures. It should be noted that each procedure is a separate list, and that the life-span of its internal variables (the PROVARs) is therefore determined by the limits of the list.

Several primitives are used solely in procedures. They are:

a. GOTO — which has two arguments, the first of which must evaluate to a LABEL and the second of which must evaluate to a Boolean condition value.

b. RETURN — This is a special control word used as the LABEL part of the arguments of GOTO, representing the return from a procedure.

c. QUOTE — which takes a single argument. This argument is copied and passed on to the next statement. As an example, ADDONE (N) adds one to N and prints the result. Now we can construct a new procedure GIVEA using ADDONE and QUOTE as follows:

```
(GIVEA        INPUT ( )
 ADDONE      (QUOTE (3)) )
```

Then the procedure is to send the value of "3" as the argument of ADDONE, which then adds one, to get 4, and prints out the 4. Note, however, that no matter

```
DEFINE(NØT        INPUT(A)IF(A,F , T))

DEFINE
(ØR INPUT (A B ) IF ( A , T , B ) )
(AND INPUT(A B) IF(A,B,F))
(SUMDIF INPUT(Q W) PRØVAR(E R)
          SET(E,ADD(Q,W))
          SET(R,DIF(Q,W))
          PRINT(E) PRINT(R) DØ(T))
```

Note the lack of FORMAT for word position: blanks, "tabs," and carriage returns are the delimiters, but multiple blanks, etc. are treated as one.

```
PØØR INPUT
SUMDIF (  002 ) ( 002 ) 0+H 001 0&1 0↑1 0↑2 000 000 0+H ( 002 DIF ( Q , W)
) PRINT ( E ) PRINT ( R ) DØ ( T )
DØ YØU WISH TØ REPEAT INPUT
YES

TYPE DEFINITIØN
```

Computer detected error; there is no primitive or procedure called DIF (Subtraction is by use of SUB).

```
SUMDIF INPUT(Q W) PRØVAR(S D) SET(S,ADD(Q,W))
SET(D,SUB(Q W))
          PRINT(S)
          PRINT(D)
                  DØ(T))

ERRØR IN CØMMAND...
```

The user puts in an extra carriage return by mistake here. The computer cannot "understand" the carriage return.

```
DØ YØU WISH TØ REPEAT CØMMAND
NØ

APPLY AND(T)

ERRØR IN CØMMAND...
APPLY AND ( T )
```

Computer detected error; AND needs two arguments and only one was supplied.

```
DØ YØU WISH TØ REPEAT CØMMAND
YES

TYPE
APPLY AND(T,T)

ANSWER IS...
T
APPLY NØT(T)

ANSWER IS...
F
LET SUMDIF BE GIVE THE SUM AND DIFFERENCE ØF 1 AND 2

GIVE THE SUM AND DIFFERENCE ØF 453 AND 297
```

Use of LET to improve FORMAT to the user.

```

750
156
ANSWER IS...
T
GIVE THE SUM AND DIFFERENCE ØF 164 AND 593

757
-429
ANSWER IS...
T
```

Figure 3. Simple Procedural Examples and Their Application, with Errors.

how many times GIVEA is called, it always sends 3 to ADDONE, and no change is therefore made to its argument.

d. SPQUT — This is a special type of quote, which takes a single argument. This argument is passed on to the next statement. The action of the next or some succeeding statements may (and usually does) produce an updating of the original argument of the SPQUT. The command can be used in many ways to store information, e.g., it may be a method of adding new constraints to the system, or of counting the number of uses of a certain routine. As illustration of the difference between QUOTE and SPQUT, consider the routine:

GIVEB, which calls ADDONE in a similar fashion to GIVEA, but uses SPQUT in place of QUOTE:

```
(GIVEB      INPUT   ( )
 ADDONE     (SPQUT (3))   )
```

The first time, ADDONE prints out 4, but the argument of SPQUT is now 4, and hence the next time it will print 5, etc.

e. OPQ — This is intended to be an operator quote. It is a third category of a general class of quote procedures, designed to point to the BCD word used as either operators, functions, and variables, or descriptors and descriptions. Thus, it is used to compile the BCD word so that it can be referred to in a procedure.

f. DO which takes an arbitrary number of arguments. This causes a series of operations to be performed in sequence, and is used principally in a conditional, where the even arguments are themselves not a single procedure.

2.6 Primitive and Procedural Application

Any primitive or procedure may be applied to its arguments by means of the command APPLY followed by the name and a list of its arguments. The technique is essentially a special version of the well-known LISP interpreter, using a large number of primitives, and written in MAD-compiled machine code rather than in interpretive code itself.

2.7 The Command Language

Basically all information passes in and out of the program through the command language. Thus all of the command words must be considered sacrosanct. The reasons for this are as follows: Although a command word can also be used as the name of a procedure, there is a special command LET (to be described later) that can be used to introduce new command words.

Each command word will now be discussed; they will be grouped by type to conform with the description of the various parts of the system. Some examples of their use are shown in Figure 4.

2.7.1 Commands for OFV Definition

The following set of five commands deals with the addition or deletion of operators, functions, and variables, and their descriptions and descriptors. In general, in an entirely new system one or more descriptors must be defined, and then some new OFVs; later it may be necessary to add descriptors, and consequently new descriptions, to existing operators, etc. There are therefore two basic commands for addition of information to OFVs:

```
DESCRIPTØR FØR ØPERATØR STRANGE

VERIFY ØN

ØK
SET NEW ØPERATØR @ TØ (STRANGE,YES)

ØK
VERIFY ØFF

SET NEW ØPERATØR = TØ (STRANGE,NØ)

SHØW ØPERATØR

DESCRIPTIØNS FØR ELEMENT @
DESCRIPTØR     DESCRIPTIØN
STRANGE                          GENERIC @

DESCRIPTIØNS FØR ELEMENT @       SPECIFIC @
DESCRIPTØR     DESCRIPTIØN
STRANGE        YES

DESCRIPTIØNS FØR ELEMENT =       GENERIC =
DESCRIPTØR     DESCRIPTIØN
STRANGE

DESCRIPTIØNS FØR ELEMENT =       SPECIFIC =
DESCRIPTØR     DESCRIPTIØN
STRANGE        NØ

TRACE SET ADD SUB

GIVE THE SUM AND DIFFERENCE ØF 231 AND 4032

ENTERING... ADD
ARGUMENTS ARE... ( 231 ) ( 4032 )
LEAVING... ADD
VALUE IS... 4263
ENTERING... SET
ARGUMENT ARE... ( ) ( 4263 )
LEAVING... SET
VALUE IS... 4263
ENTERING... SUB
ARGUMENTS ARE... (231 ) ( 4032 )
LEAVING... SUB
VALUE IS... -3801
ENTERING... SET
ARGUMENTS ARE... ( ) ( -3801 )
LEAVING... SET
VALUE IS... -3801
4263
-3801
ANSWER IS...
T
```

If VERIFY ØN is used, ØK is given as an end response after execution of each command.

To see <u>all</u> present operators in the system, with <u>all</u> their descriptions.

This procedure is the example, with LET, of Figure 3. It is used here to illustrate TRACE, which will print the input arguments when one of the specified procedures or primitives is entered, and then gives the value on leaving. In this simple procedure, there is no "nesting," but for complicated tracing, the ENTERING and LEAVING are not necessarily in close proximity.

Figure 4: Simple Command Language Examples

A. DESCRIPTOR FOR ...

B. SET ... TO ...

This part of the command language operates on operator, function, and variable lists, and therefore one of the words in these five commands is either OPERATOR, FUNCTION, or VARIABLE.

DESCRIPTOR FOR OPERATOR BINARY, (PRECEDENCE, NUMSUB) sets two words on the descriptor dictionary, BINARY and PRECEDENCE. The pointers to these dictionary entries are put on the descriptor list that is strung from the header of the OPERATOR list, and if any PRECEDENCES are to be read-in as descriptions, the procedure NUMSUB will be used for their input (or output), since precedences are ordered numerically here.

As an example of definition of a new operator: SET NEW OPERATOR + TO (BINARY, YES) (PRECEDENCE, 100) would define a new operator + , and give it the two description YES and 100. Note that the 100 is read in by the special procedure NUMSUB, which was defined by the descriptor as shown in the first example of definition of a descriptor.

There are obviously three types of deletion: the deletion of a descriptor, deletion of a particular description, and a deletion of an entire element. These three deletions are of the form

C. DELETE DESCRIPTOR ... FOR ...

D. DELETE DESCRIPTION OF DESCRIPTOR ... FOR ...

E. DELETE ...

When a descriptor is deleted, all instances of its descriptions must also be deleted or the description lists become incorrectly ordered, and the system therefore automatically performs these deletions.

As examples:

```
DELETE DESCRIPTOR TYPE FOR VARIABLE
DELETE DESCRIPTOR NUMBEROFARGS FOR FUNCTION
DELETE FUNCTION BESSEL
DELETE VARIABLE Zn (TYPE, CHEMICAL)
```

It is thus possible to add or remove any part of the main operator, function, and variable lists.

2.7.2 Commands for Expressions

Apart from editing, evaluation, and algebraic manipulations of equations, which are dealt with by the procedures, the most important command is for read-in of an expression, which is done by the command

```
F.      EQUATION ...    ...
```

The expression or equation may be given a number or the number postion may be blank. In the latter case, the equation will be numbered sequentially with the latest number one more than the highest to date. Two carriage returns are necessary between the initial command line and the actual equation read in. This is to obviate the use of a special character symbol to change the mode of reading of the list from command to formula-input with replacement of operators, etc. Further equations may be given as input by adding two carriage returns after each equation subsequent to the new equation. These will be numbered sequentially following the first, and the input will be considered as equations until such time as three carriage returns are used to terminate the command. Examples are:

```
EQUATION 12 ↵ ↵  B + D = 5 + x ↵ ↵ ↵
EQUATION ↵ ↵   A       = 3 ↵ ↵  5 = B ↵ ↵ ↵
```

2.7.3 Commands for Procedural Definition

Two commands are specifically associated with the definition and redefinition of procedures:

G. DEFINE(...)
H. REDEFINE(...)

These two commands take a single list containing a procedure or a set of lists, compile the procedures, and put them as new definitions on the procedural name list.

2.7.4 Commands for Primitive and Procedural Application

Three commands are associated with the application of procedures and primitives; two of them are built-in debugging aids. The commands are:

I. APPLY...
J. TRACE...
K. UNTRACE...

APPLY causes the application of a procedure or primitive to its arguments, and an evaluation. It is invoked by calling a particular primitive or procedure name followed by its argument in the list form.

TRACE takes a set of arguments that must be primitive or procedure names, for example, TRACE AND, OR, and watches for their occurrence in a procedure. Whenever these procedures (or primitives) are called during execution, a print-out will be given.

Thus by judicious use of TRACE it is possible to debug an otherwise quite complicated program. In order to switch-off the trace feature for AND, the user initiates the command:

UNTRACE AND

2.7.5 Commands for Display of User Data

Although the form is different for the OFV, expression, and procedure list, the basic command for display of the information is of the form

L. SHOW...

When dealing with one of the OFVs, substantial amounts of information may be required, or only parts of it may be needed. Thus the entire information of one of the OFVs may be obtained by the statement SHOW O/F/V. However, to be more selective, all the + operator's information will be printed out by the command SHOW OPERATOR + , while only one operator's descriptor/descriptions will be printed out if the unique identifier pair is added, e.g., SHOW OPERATOR + (TYPE, CHEMICAL).

To display an expression, the command requires the word EQUATION followed by numbers separated by commas. For example, SHOW EQUATION 5,17 will cause output of equations 5 and 17.

2.7.6 Special System Commands

Three special system commands may be used to aid the user in preparing and saving data:

M. READ...
N. VERIFY...
O. SAVE...

The first of these allows off-line preparation of data. Thus it may be advisable for either a large volume of numerical data, or for definition of long and complicated programs, to have the ability to TYPESET and EDIT these large files before read-in to the main program, or after having discovered some previous bug in the data or program.

Thus, any time a new procedure is to be added, it is advisable to save the system. This is done by initiating the command SAVE NAME, which creates a file in the user's directory called NAME SAVED. This is a copy of the system, and will not be destroyed if some disasterous bug occurs in the next file to be read. The file which has been prepared off-line must be in 12-bit mode and must be stored in disk storage under some double name such as ADMIT FILE. To initiate read-in of this file, the command READ ADMIT FILE will cause opening and subsequent read-in of the file. Unless the file has some means of indicating "end of message" (for example with a special PRINT statement), some alternative mode may be desirable to show when a procedure has been completed. This is by means of the command VERIFY. On being given the command VERIFY ON, the command language will reply, OK, and give an OK after every subsequent completion of a command. At times the talkative computer can be very tedious, and this feature may be switched-off by the command VERIFY OFF.

2.7.7 The LET Command

The LET command allows the user the maximum ability to define his own syntax. This feature marks the beginning of a better user-oriented language.

The form of this command is

P. LET...BE...

and it is intended to allow definition of new commands by the user. As an example, if he would rather use the sentence, CHECK EQUATION 5 FOR ATOM BALANCE, instead of APPLY CHMEVL (5), he uses the command LET CHMEVL BE CHECK EQUATION 1 FOR ATOM BALANCE, where the order of the arguments is the "numbers" in the LET command.

3. Conclusion

The system thus described has been programmed on the Project MAC CTSS facility; details of its procedures and versatility are discussed fully in Reference 7. A further system is being designed, and will be implemented, probably using TRAC[3] as an intermediate language, at the Computing Center at The University of Michigan.

REFERENCES

1. McCarthy, J., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Massachusetts, 1962.

2. Tobey, R.G., Babrow, R.J., and Ziller, S.N., "Automatic Simplification in FORMAC," Proceedings, Fall Joint Computer Conference, 1965, pp. 37-53.

3. Mooers, C.N., "TRAC, A Procedure-describing Language for the Reactive Typewriter," Communications of the ACM, March 1966, pp. 215-219.

4. Weizenbaum, J., "Symmetrical List Processor," Communications of the ACM, September 1963, pp. 524-536.

5. Feldman, J.A., Aspects of Associative Processing, Technical Note 1965-13, M.I.T. Lincoln Laboratories, April 1965.

6. Shaw, J.C., JOSS: Experience with an Experimental Computing Service for Users at Remote Typewriter Consoles, Report No. P-3149, The Rand Corporation, Santa Monica, California.

7. Sibley, Edgar H., The Computer as Symbol Manipulator and Evaluator for Engineering, Doctoral Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, February 1967.