AN ALGEBRAIC LANGUAGE FOR THE MANIPULATION OF SYMBOLIC EXPRESSIONS

by John McCarthy

Abstract: This memorandum is an outline of the specification of an incomplete algebraic language for manipulating symbolic expressions. The incompleteness lies in the fact that while I am confident that the language so far developed and described here is adequate and even more convenient than any previous language for describing symbolic manipulations, certain details of the process have to be explicitly mentioned in some cases and can be left to the program in others. This memorandum is only an outline and is sketchy on some important points.

I. Introduction

First we shall describe the uses to which the language can be put and the general features that distinguish it from other languages used for these purposes.

1.1. Applications of the language

1.1.1. Manipulating sentences in formal languages is necessary for programs that prove theorems and also for the advice taker project.

1.1.2. The formal processes of mathematics such as algebraic simplification, formal differentiation and

integration, etc. are conveniently programmed in this language.

1.1.3. A compiler can be conveniently written in this language except for input and output section. We shall illustrate this in discussing the compiler for this language.

1.1.4. Heuristic programs, i.e. programs involving tentative procedures are best written in this sort of language. One reason is that trees of alternative actions are conveniently represented.

1.1.5. In general this language is best suited for representing expressions whose number and length may change in ways which it is not convenient for the programmer to predict and which have sub-expressions with the same characteristics. It is not so convenient for representing lists of fixed length where one frequently wants the nth element where n is computed rather than obtained by adding 1 to n-1.

1.2. Features of the language

1.2.1. Expressions are represented by lists each element of which occupies a computer word. Each computer word of a list in addition to containing a datum also contains the address of the word containing the next element of the list. 0 for the address of the next element indicates the last element. If one element of an expression is a subexpression the word corresponding to this element contains

the address of the word containing the first element of the subexpression. In the IBM 704 or 709 whose 36 bit word is divided (for the convenient use of certain machine instructions) into two 15 bit parts (address and decrement) and two 3 bit parts (prefix and tag) lists are represented by storing in the decrement part of a word the address (in our system actually the complement of the address) of the next word or thelist. The datum is contained mainly in the address part of the word with the prefix and tag used either for data or indicators of the kind of list structure.

The use of this kind of computer representation of list structure was first developed by Newell, Simon and Shaw for heuristic programming.

1.2.2. The convenience of algebraic notation for the description of procedures derives in large part from the fact that frequently the output of one procedure serves the input for another and the functional notation enables us to avoid giving the intermediate result a name. Also the order in which a functional formula is written permits us to start mentally from the desired result and build it up successively from its component computations.

The other main advantage of the algebraic notation for list structure processing was first noticed by Gelernter. If we make routines which form lists functions whose output is the address of the list formed, complex structure can be formed by single expression compounding the list forming functions.

Algebraic notation for list processing is not used by Newell, Simon and Shaw, perhaps beaause to do so is most convenient when a compiler is available, but is used by Gelernter in the geometry program. This was accomplished (on the advice of the present author) by using the Fortran compiler together with a set of machine language coded functions for handling the primitive list processes that go from one element of a list to the next.

1.2.3. It is frequently convenient to define certain processes by recursion. This means that the program defining a subroutine may use the subroutine itself. This presents the programming difficulty that the intermediate results formed by a subroutine must be protected from alteration when the routine is used as a subroutine of itself. This is best accomplished by storing these results in list structures. Newell, Simon and Shaw use this feature extensively and provide what they call push-down lists for saving intermediate results. The geometry program has not used recursive programs so far, and the list structure storage of temporary results is difficult in Fortran because of the comparative inaccessibility of some of these intermediate results (eg. index registers, the register in which subscripts are saved, and the temporary storages used within formulas).

In the present system the saving of temporary storage is handled automatically without specific attention by the programmer when a subroutine is potentially capable of calling on itself.

1.2.4. Conditional expressions. One of the weakest features of present programming languages is in the means they provide for treating conditionals, that is the calculation of quantities where the operations used depends on the result of certain prior tests. The use of propositional quantities, predicates, and conditional expressions, essentially solves this problem and eliminates most of the small scale branching from source programs. In combination with the feature of recursive definition it permits certain subroutines to be defined by single formulas in this language that are quite involved programs in other language.

1.2.5. The use of functions and predicates as parameters of subroutines makes possible some very powerful routines for searching, transforming, and manipulating list structures.

2. Kinds of Quantity and Forms of Statement

2.1. Kinds of quantity

There are several kinds of quantity used in this language. The reader will notice the omission of the list or list structure itself as a kind of quantity. This is because while a number of interesting and useful operations on whole lists have been defined, most of the calculations we actually perform cannot as yet be described in terms of these operations. It still seems to be necessary to compute with the addresses of the elements of the lists.

2.1.1. Integer quantities. As in Fortran these serve as indexes or addresses. In a compiler written for the

IBM 704 or 709 these will also occupy the decrement parts of words when referred to by single symbols. We do not use typographical conventions to distinguish the integers from other kinds of quantity, but rely on either context or specific statement. Arithmetic with these quantities will be as in Fortran.

2.1.2. Whole words. Whether we will make the arithmetic symbols stand for floating point operations is as yet unsettled.

2.1.3. Propositional quantities. A propositional quantity is represented as a single bit (i.e., has value 0 or 1). How these shall be stored is not yet determined. In the IBM 709 the SI register might seem to be their natural home. Propositional expressions are recursively formed by the Boolean operations $\wedge$, $\vee$, and $\sim$ etc. from propositional quantities. They are also formed by predicates (functions taking values on the set of two elements consisting of 1 for truth and 0 for falsehood). Examples of such predicate are $(a=b)$ and $(a < b)$ where a and b range over arithmetic quantities. Other predicates may be constructed by the procedures for defining functions.

Quantities of other kinds may be constructed from propositional quantities by means of <u>conditional expressions</u>. If $p_1,\ldots,p_k$ are propositional expressions and $c_1,\ldots,c_k$ are expressions of any one kind then $(p_1 \rightarrow c_1,\ldots,p_k \rightarrow c_k)$ is an expression is the c corresponding to the first of the p's which is true. If none of the p's is true the whole statement

involving the conditional expression is not to be executed.

2.1.4. Locational quantities. A point in the program may be labelled and the address of such a point (to which control may be transferred) is called a locational quantity. The computations with these quantities is limited.

2.1.5. Functional quantities. These will certainly be allowed as parameters of subroutines, but their full possibilities might not be exploited in an early system.

2.2 Kinds of Statement

This list is again incomplete.

2.2.1. The arithmetic (Fortran term) or replacement statement is the most important kind. It has the form a=b where a and b have the following forms:

a has one of the following forms:

1. The name of a variable (we shall not go into the typographical rules for names at this point.)

2. A(i) where a is the name of a variable which has been designated as subscriptable and i is an integer expression. (Arrays of more than one dimension may not be included in the first system.)

3. cwr(i), cpr (i), ctr (i), car (i), csr (i) cir (i), chitr (i,n) or csegr (i,n,m).

In all the above i represents an integer expression designating a register in the machine and the expression represents the contents of a certain part of that register. For example, statement beginning car (i) = causes a quantity to be computed and stored in the address part of register leaving the rest of the register unchanged.

The b in a statement a=b is an arbitrary expression whose value is compatible with the space allotted for it. The recursive rules for the formation of expressions are similar to those of Fortran or the proposed international algebraic language.

2.2.2. Control is transferred by the "go" statement. go(e) causes control to be transferred to the location given by evaluating the locational expression e, (If e is a conditional expression then transfer of control will be conditional).

2.2.3. The flexibility of the go statement is increased by the "set" statement set (A; q_1,..., qm) causes an array A of size to be established whose contents are the quantities q_1,..., q_n. In particular the q's may be locational expressions and then the expression A(i) where i is an integer expression denotes the ith of the locational expressions mentioned.

2.2.4. Subroutines are called to be executed simply by writing them and their argument as statements. (i.e., as in Fortran but without the word CALL.)

2.2.5. Declarative sentences. These have the form I declare (...) where the dots represent a sequence of assertions of one of the following forms:

1. (a; p_1,...,p_n)

This causes the expressions p_1,...,p_n to be entered in the property list associated with the symbol a. Each symbol in the program has such a property

list which is used and added to by the compiler and is also made available to the object program.

2. $(a_1,\ldots,a_n,p)$

This causes the expression p to be put on the property list of all the a's.

2.2.6. Compound statements. A sequence of statements can be enclosed in "parentheses" and given a name. The symbols to be used for these "vertical parentheses" are not yet determined.

2.2.7. Iteration statements. The exact forms of "do" (Fortran notation) to be provided are not determined but they should include the Fortran kind, and a do over an explicitly given list or a referred to list structure.

2.2.8. Subroutine definition. This will resemble the Fortran system except that

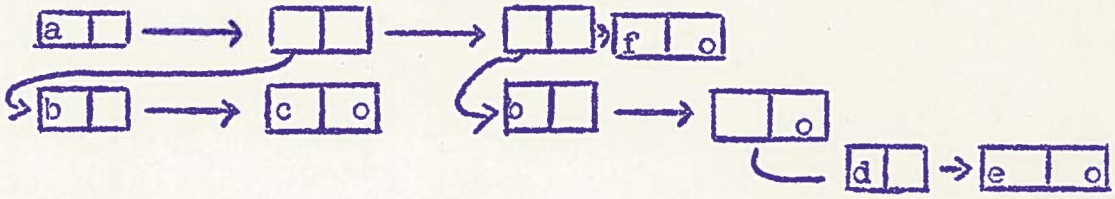
1. Parameters can include functions and locational expressions.
2. Subroutines can either be compiled within a program or separately.
3. Symbolic reference to variables as well as to other subroutines will be provided.

2.2.9. Input-output will be handled by subroutines which can be made quite flexible through the use of property lists.

3. Representation of Formal Expressions by List Structures and the Basic List Structure Operations

3.1. Representation of Expression.

The use of list structures for storing data corresponds to the mathematical system which uses the sequences as the basic expression. Thus, the sequence (a, (b,c), (b, (d,e)), f) corresponds to the list structure



However, some additional distinctions are required; here we follow Gelernter. A word of a list L_1 may refer to a list L_2 but we may not wish to regard the list L_2 as a part of the list L_1 in the sense that when the list L_1 is erased, the list L_2 should also be erased. Also, for certain kinds of data it is frequently convenient to use a whole word.

Therefore, we shall regard bits 1 and 2 of a word as the indicator part of the word and adopt the following convention as to its meaning:

00 the address part of the word is a datum. (It may be the address of a list, but if so this new list is not to be regarded as a sublist of the old one.)

01 the address part of the word refers to a datum word which is to be erased when this list is erased.

10 the address part of the word refers to a sublist of

the given list.

The case of a full datum word is only a matter of convenience to the program, but the distinction between a sublist of a given list and a reference to another list as an element of a list is a basic distinction; it is the distinction between an element of a sequence being another sequence or a name of a sequence.

While a sequence is the basic object represented by a list structure certain other expressions are also so represented by considering them as sequences.

For example, a functional expression $f(e_1,\ldots,e_h)$ is represented by the sequence $(f,e_1,\ldots,e_h)$ and hence by the corresponding list structure. Expressions involving operations like + are represented by sequences; namely a+b+c is represented by the sequence (+,a,b,c). This corresponds to the "Polish notation" of mathematical logic, but the operations can have a variable number of arguments since the end of the sequence of arguments is explicitly displayed.

3.2 The Basic Single Word Functions.

Programs for manipulating list structures are described mainly in terms of replacement statements (i.e., of the form a=b). The right sides of these statements are compositions of certain "functions". These functions, however, are not all functions in the mathematical sense for two reasons:

1. The value of a "function" may depend on the state of storage of the machine. For example cwr (n) (contents of the whole of register n) is the 36 bit quantity which is stored in register n.

2. The execution of the program corresponding to the "function" may change some of the arguments and certain quantities referred to by them.

Before describing these "functions" we will mention the free storage list. This is a list of all the registers not currently in use by the program. A certain fixed register has the address of the first word on the list and each word on the list has the address of the next word in its decrement part (the rest of the word being zero). The last word has zero in its decrement; this being the usual signal for the end of a list. The routines that create lists take registers from the beginning of the free storage list and those that erase lists put the storage back on the free storage list.

The functions for list manipulation are divided into those concerned with single words or at any rate with a fixed number and those routines dealing with lists as a whole. The latter of course are compounded from the former and naturally we shall discuss the former first.

The basic functions are divided into a number of classes.

3.2.1. First we have those that extract parts of a 704 word and form a word from parts. We shall distinguish the following parts of a word and indicate each of them by a characteristic letter.

w the whole word

p the prefix (bits s, 1, 2)

i the indicator (bits 1 and 2)

s the sign bit

d the decrement (bits 3-17)

t the tag (bits 18-20)

a the address (bits 21-35)

Corresponding to these we have the functions pre, ind, sgn, dec, tag and add which extract the corresponding parts of the argument word. The result is regarded as an integer and hence is put in the decrement part of the word.

In addition to the above we can get the nth bit of a word w with the function bit (w,n) and the segment of bits from m to n with the function seg(w,m,n). (Needless to say the others are all special cases of seg.) For putting a word together out of parts we have the functions

1. comb 4(p, d, t, a) which forms a word out of the four parts indicated by the arguments.

2. comb 5(s, i, d, t, a) which forms a word from a still more detailed prescription).

3. choice (c, a_0, a_1,) This forms a word whose nth bit is the nth bit of a_0 if the nth bit of c is o and is the nth bit of a_1 if the nth bit of c is 1.

3.2.2. Next we have the reference functions which extract a part of the word in the register whose number is the argument. These functions are cwr, cpr, csr, cir, cdr, ctr, and car. For example, car (3) is the 15 bit quantity found in the address part of register 3. In addition we have cbr (n,m) which extracts the mth bit of register n and csgr (n,m1,m2) which extracts the segment of bits from m1 to

m2 of the word in register number n.

Needless to say, these functions are all combinations of the extraction functions and cwr. For example, car (n) = add (cwr (n)).

3.2.3. The storage functions. In this system storage in a register can be accomplished in two ways. The simplest is by writing statements of one of the forms

cwr ( ) =

cpr ( ) =

csr ( ) =

cir ( ) =

cdr ( ) =

ctr ( ) =

car ( ) =

cbr (,) =

csgr (,,) =

The second is by using one of the functions stwr, stpr, stsr, stir, stdr, sttr, and star. Each of these has two arguments, the number of the register into which the datum is to be stored and the datum itself. The rest of the word referred to is unchanged and the value of the function is the old contents of the field referred to. It is this facility for getting the old contents to serve as an argument of a further process that gives this second method of storage some advantages. There are two additional storage functions stbr and stsgr of 3 and 4 argument respectively which store a single bit and a segment.

3.2.4. The construction functions. These construct elements of list structures by taking words from free storage consw (w) puts the argument w into the first word on the free storage list, shortens the free storage list, and has a s value the address of the word into which the datum is stored. The other functions in this class are compounds of this one with construction functions but occur so frequently that they need special names, consel (a,d) puts a in the address and d in the decrement part of a word from free storage. consfl (w,d) takes two words from free storage puts w in one of them, puts the address of that one in the address field of the other with 1 in the indicator field and d in the decrement field. consls (a,d) is like consel (a,d) except that the indicator field gets a 2.

3.2.5. The erase function erase (J) returns the word in location J to the free storage list. Its value is the old contents of this word.

3.2.6. Pointer moving functions. These operations move a pointer in a list structure, keeping a list for reversion purposes.

Point (J,K) creates a list 

and points J at it; mova (J) moves the pointer in the address direction; movd (J) moves the pointer in the decrement direction; movup (J) moves the pointer up in the structure i.e., deletes the first entry from the list belonging to the structure; kill (J) erases the list structure belonging to the pointer.

3.3. Basic operations on whole list structures.

This set, although incomplete, is adequate for the examples given in the next section.

3.3.1. eralis (J) erases the list structure to which J points and returns it to the free storage. It does not have a value. It can be described in terms of the elementary operations of the previous section by the following program. However, whether it will pay to use the facilities of the compiler or to write the routine in machine language as an elementary routine will depend on the efficience of the compiler.

```
        subroutine eralis (J)
/       J = 0  ⟶  return
        go (a (cir(J)))
a(1)    jnk = erase (car (J))
a(0)    eralis (dec (erase(J)))
                return
a(2)    eralis (car (J))
\       go (a(o))
```

3.3.2. copy (J) This function copies the list structure to which J points into free storage. Its value is a pointer to the copied structure. A program for it is function copy (J)
/copy = (J = 0 ⟶ 0, cir (J) = 0 ⟶ consel (car(J), copy (cdr (J)), cir (J) = 1 ⟶ consfl (cwr (car (J)), copy (cdr (J))), cir (J) = 2 ⟶ consls (copy (car(J), copy (cdr (J))))
\ return

This program ignores tags and signs. A version which does not is

function copy (J)

/copy = (J=0 → 0, 1 → consw (comb 4(cpr (J), copy (cdr(J)), ctr (J), (cir (J) = 0 → car (J), cir(J) = 1 → consw (cwr (car (J))), cir (J) = 2 → copy (car (J))))))

\ return

3.3.3. Search (L,J,p_1,p_2,p_3,M). The value of search (L,J,p_1,p_2,p_3,M) is the address J of the <u>next</u> element of the list structure L which satisfies the condition p_1. (p_1 is a propositional expression in J and J is a dummy variable.) p_1 and p_3 are expressions in J which define the list structure searched in that p_1 is the condition that the structure continues in the address direction and p_2 is the condition that it continues in the decrement direction.

M is a location in which is stored a structure which keeps track of where we are in the structure so that the search can be continued (by another statement or a return to the same one) from where it left off.

If L = 0 or if no element is found the value of search is 0.

3.3.4. Maplist (L,J,f(J)). The value of this function is the address of a list formed from the list L by mapping the element J into f(J).

3.3.5. list $(i_1,\ldots,i_n)$. The value of list $(i_1,\ldots i_n)$ is the address of a list whose items are $i_1,\ldots,i_n$. Appropriate

indications are included in the words if the items are full words or sublists.

The major form of input consists of symbolic expressions on IBM cards.

Such expressions are translated on input as follows:

1. Certain characters on a card are regarded as connectives and both serve as punctuation and to denote certain operations. With the present Fortran characters these include everything but the digits, the letters and the decimal point immediately following a digit. These characters or sequences of them except when in a quoted text separate the text into parts. Blanks are ignored except in column 1 of a card where they are regarded as a period and right after a dot where they indicate that it is to be regarded as a period rather than as a decimal point. A format may also designate certain column transitions as punctuation symbols.

The primary way expressions are written is as sequences of the form $(e_1,\ldots,e_m)$ where the e's are expressions. This is read in as a list, each e being put in the address part of a word with the decrement containing the address of the next word on the list. If an e is not a term the corresponding expression is read in to free storage and the address of the resulting list is the datum of the higher list. If the item is a term it is treated as follows:

1. An integer is converted to binary and stored in the address part of the word.

2. A symbol is looked up in the symbol table and its value stored in the address part.

3. A floating point number is stored in a data word and the address of this word is stored.

4. A text is stored character by character in a list structure.

3.3.6. Read (source, format, symbol table).

Read (source, format, symbol table) has as its value the address of the structure into which the data read is put. The first argument is the source of input data, the second is the format in which this data is stored, and the third is a symbol table relating literal symbols in the external medium to the addresses where the referents of these symbols are to be stored.

3.3.7. Equal (L1,L2). This predicate tests the equality of the list structures to which L1 and L2 point. It is defined by the formula:

$$\exists\, \text{equal}(L_1,L2) = (L_1 = L_2 \rightarrow 1,\ \text{cir}(L1) \neq \text{cir}(L_2) \rightarrow 0,\ \text{cir}(L1) = 0 \wedge \text{car}(L1) \neq \text{car}(L2) \rightarrow 0,\ \text{cir}(L1) = 1 \wedge \text{cwr}(\text{car}(L1)) \neq \text{cwr}(\text{car}(L2)) \rightarrow 0,\ \text{car}(L1) = 2 \wedge \sim \text{equal}(\text{car}(L1), \text{car}(L2)) \rightarrow 0,\ 1 \rightarrow \text{equal}(\text{cdr}(L1), \text{cdr}(L2))$$

APPENDIX I.

As an example of the use of the language we shall give a program for analytically differentiating a simple class of formulas. The formulas in question are in a single variable x and constants; these are combined by addition and multiplication.

A formula is represented by a list structure. A sum is a list whose first word contains a constant representing "plus" in its address part and the summands in the address parts of the successive elements. A product has "times" as its first element followed by the factors. The variable x has "x" in the address part and a constant has 1 in its tag part and the datum in the address part.

The function diff (J) where J points to the structure representing the formula to be differentiated produces a new formula, the derivative of the old, and its value is a pointer to this formula. The program is:

```
function diff(J)
diff = (ctr(J) = 1 ⟶ 0, car(J) = "x" ⟶ 1, car (J)
= "plus" ⟶ consel("plus", maplist(cdr(J),K,diff(K))), car
(J) = "times" ⟶ consel("plus", maplist {cdr(J),K, consel
("times", maplist{cdr(J),L,[L = K ⟶ diff(L), L ≠ K ⟶
copy (L)]}})))
return
```

consel(0,0) consel(1,0)