Artificial Intelligence Project---RLE and MIT Computation Center
Symbol Manipulating Language---Memo 3---Revisions of the Language
John McCarthy

This memo supersedes the earlier memoranda of the same title in almost all matters of detail, but some of the general remarks in the first memo are not repeated here and should be read for an explanation of the motivation for the development of the language.

1. Representation of Symbolic Expressions by List Structures

The kinds of expression the language is designed to manipulate include functional expressions as in elementary calculus, calculator programs either in machine language or in an algebraic language such as this one or Fortran, and the expressions for propositions as they occur in the propositional calculus, the functional calculi, and other formal languages of mathematical logic. It should be emphasized that we are presently concerned with a language of imperative statements for describing processes for manipulating such expressions and not with a declarative language for making assertions about the expressions. The problem of expressing assertions about expressions will be studied later in connection with the advice taker.

The expressions to be manipulated are represented in the machine in a special way which facilitates the description of their manipulation. The translation between the internal representation and more or less conventional ways of representing the expressions outside the machine is handled by the read and print programs. The preliminary version of these programs which is presently being debugged (Oct. 21, 1958) translates between the internal notation and a restricted specialized external notation. The direction in which the allowed external notation will be generalized in later versions will be described in connection with the descriptions of the read and print programs; at present it seems that very little compromise will be required with the conventional notations beyond that required by the need to write expressions linearly with a limited set of characters.

1.1 External form of expressions

We shall first describe the restricted external notation by the following recursive rules. First we define a symbol as one of the following:

1. A sequence of letters and digits containing at least one letter. The length of the expression is limited to 120 characters though if there should be any reason to do so there is no difficulty about extending this simply by increasing the length of an array in the read routine from its present length of 20 words.

2. A sequence of digits which may contain at most one decimal point in the interior. These symbols represent numbers and their length is not limited by the <u>read</u> routine or <u>print</u> routine, but will be limited by the kinds of number arithmetic included in the program and by the conversion routines which are not part of the <u>read</u> and <u>print</u> package.

We can now define the external expressions allowed:

1. A symbol is an expression.

2. If $e_1,e_2,\ldots,e_n$ are expressions, so is $(e_1,e_2,\ldots,e_n)$; that is, a sequence of expressions is an expression. The special case of a sequence of one element is allowed and the resulting expression is considered to be different from the element itself, i.e. we distinguish between e and (e).

As an example, we shall describe how elementary functional expressions are represented in this notation. The rule is simply that a functional form is represented by a sequence consisting first of the name of the function followed by the list of its arguments. Thus the expression that is represented in ordinary mathematical notation by

x(x+1)sin(y)

is represented in our notation by

(times,x,(plus,x,1),(sin,y)).

This resembles the Polish notation used in mathematical logic except that parentheses are explicitly included. This permits symbols of varying numbers of characters and functions of varying numbers of arguments.

(Note: this supersedes the notation given in the descriptions of either of the previous versions of the differentiation routine. In particular the symbols <u>const</u> and <u>var</u> of these notations are no longer needed or rather may be relegated to the property lists)

1.2 Internal form of expressions.

Expressions are represented internally by lists. A list is a sequence of 704 words arbitrarily ordered in memory except that register zero is excluded.*1 Each word contains in its 15 bit decrement part the location of the word containing the next element of the list. The decrement part of the last element of a list contains 0. The 15 bit address part of the word contains the datum of the element of the list.*2

There are two kinds of list element. Namely an element may either be a sublist or it may be a symbol. When the element is a sublist the address part of the word contains the location of the first word of the sublist. When the element is a symbol the address part of the word contains the location of the property list of the object the symbol represents. This property list whose meaning and format will be described in the next section has zero in the address part of its first word. Thus the routines which manipulate list structures can tell when they have reached the bottom of an expression, since the property list of the object represented by a symbol is not considered part of an

---

*1 The location is represented by the 2's complement of the address of the register containing the address of the next element. This use of the word location conflicts with the usual one in which the location of a word is the address of its register, but it does not seem desirable to choose another word. The 2's complement notation which is made convenient by the subtractive nature of indexing on the 704 need be considered only in connection with machine language programs. The user of the system need only consider that each word contains the location of the next word and need not worry about how this location is represented.

*2 The tag and prefix parts of the word are not used and are presumed to be zero. Thus the use of an indicator field as in the earlier versions of the system is abolished. This is done by removing type 1 words from list structures and relegating them to property lists. The distinction between what were formerly called type 0 and type 2 words is accomplished in a manner presently to be described.

expression in the sense that it is not erased when the expression is erased, it is not copied when the expression is copied, and it is not printed when the expression is printed.

We shall use the terms list and list structure in slightly different senses. When we say list structure we are referring to the entire expressions down to the object symbols composing it, while when we say list we are referring to the top level.

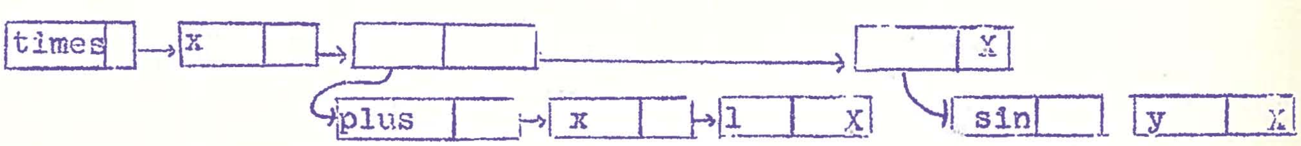
As an example, we shall describe the list structure corresponding to the functional expression

x(x+1)sin(y)

which was represented in our restricted external notation by

(times,x,(plus,x,1),(sin,y))

We use a pictorial notation in which a word is represented by a rectangular box divided into a left and right sub-box in which are put the address and decrement parts of the contents of the register represented by the box. (Note that the address occurs to the left of the decrement in this notation as in SAP which is the reverse of their positions in the 704 word.) An arrow from a sub-box to a box means that the corresponding field of the word contains the location of the word represented by the box to which the arrow points. When a box is left blank and no arrow issues from it the corresponding field contains zero. If the reader is puzzled by this description perhaps a picture will be worth 10,000 words. Here is the picture of the above expression.



The symbols times, plus, x, y, sin represent the locations of the property lists of the objects represented by these symbols. It is important to note in the case of the constant 1 in the expression that the number 1 is not in the list structure itself. The fact that a given symbol represents a constant which has the numerical value 1 will be found on the property list of the object associated with that symbol.

1.3 Objects and Their Property Lists.

In the paper on the advice taker an object was defined as an entity about which we wish to record something that cannot be deduced from the form in which it is represented or at least do not wish to deduce from this form. Although the system being described here is not as ambitious as the proposed advice taker system, it turns out that the concepts of object and property list are quite useful. The first use of the property list is to represent the correspondence between the symbol used for an object inside the computer and the symbol used in external media. In this respect it is a generalization of the symbol table of SAP with the added feature that it is designed to be used by the program at running time as well as during compilation. Conceptually, we should not identify the object either with the external symbol or with the location used to represent it in list structures. In fact, it may be worth while to consider an object which we refer to as sin or x as a "thing in itself" which is not identical with any representation of it. In the present system we shall include the following kinds of information about objects in their property lists whenever it is appropriate to do so.

1. The internal name of the object is the location of its property list.

2. The external name of the object (if it has one, and until routines are created which invent objects all objects will be introduced from the outside and therefore will have external names).

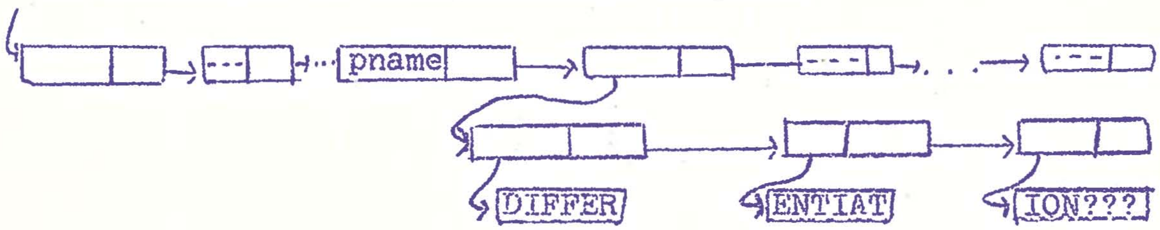
3. Whether the object represents a number, and if so whether the number is a constant or is changed by the program and also what the current value of the number is.

4. If the object is a function this fact will be noted and such facts as the location and calling sequence of program for evaluating the function will be given. If it is appropriate, formulas for differentiating or integrating the function

may be given.

5. Adjectives which are applicable to the object may be noted on its property list.

Except for the fact that the address field of its first word contains 0, the information on a property list is not stored in a fixed order. It is a list of items each of which is identified by an object symbol in the list itself. The order in which items will be represented has been determined only in the case of the external name. We shall give the representation of the external name of the term DIFFERENTIATION as an example of the convention adopted.

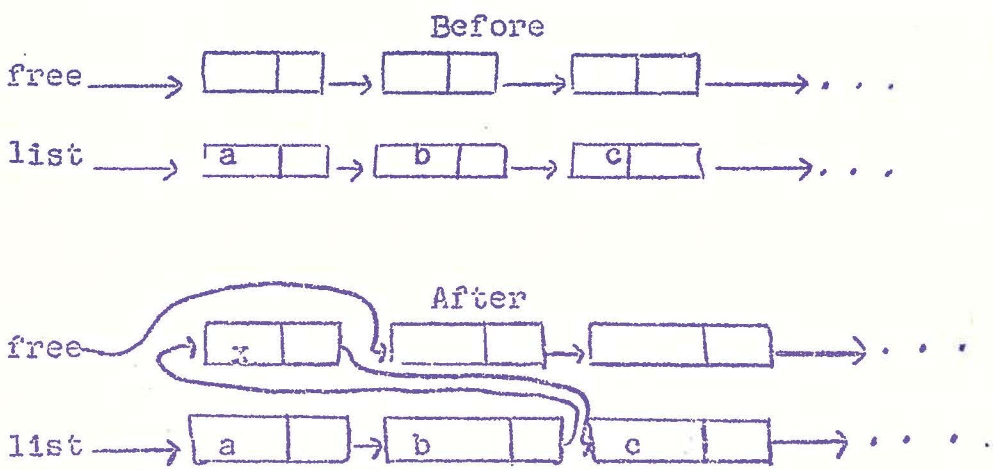


In the above diagram the address field of the first word on the property list is left blank indicating that this field contains zeroes, the fields with dashes may contain any locations, the symbol pname represents the location of the property list of the concept of external name, and the words containing capital letters contain 6 characters in standard 704 notation except that ? represents the illegal character whose octal form is 77. The print routine recognizes the illegal characters as terminating the word.

From the way external names are represented it should be clear that property lists do not meet all the conditions for lists prescribed in the previous section. This is inevitable since they must be able to refer to non-list quantities such as external names, numbers in integer or floating point form and also programs. This means that not all the routines to be described in subsequent sections of this report can be applied to property lists without disaster. However, because the conventions are preserved on the top line at least, some of these routines and in particular the search routine is applicable to property lists.

1.4 The Free Storage List

One of the main advantages of a system of representing expressions by list structures is that the structures can be extended or collapsed at any point. This is accomplished with the aid of a certain list called the free storage list which contains those registers which do not contain information at any given time. Initially, this list may have 20,000 registers and as list structures are extended they grow at the expense of the free storage list. When an expression is no longer needed the <u>erase</u> routine returns its registers to the free storage list. We shall illustrate the use of the free storage list by giving diagrams showing the situations before and after an item x is inserted in a list by putting it in a word taken from the free storage list.



After the basic routines have been defined which take words from the free storage list and put them back there, it will not be necessary to mention the free storage list explicitly any more. However, its existence is one of the main reasons for the flexibility of the system.

The use of list structures for representing symbolic expressions was first put to extensive use by Newell, Simon, and Shaw in their <u>Information Processing Languages</u>.

2. Changes in the Elementary Functions of the System.

This section refers to the first memorandum of this title. The revisions in the system described in the previous section and some experience in programming in the system and hand-compiling the resulting programs suggest some changes in the set of elementary functions.

1. The functions which refer to parts of the word other than the address and the decrement can be omitted.

2. The functions referring to whole words are retained but will be used only inside property lists.

3. The distinction between consel and cons$_{ls}$ is abolished so we will call the new function cons.

4. The storage and pointer functions have not been used so far and hence are tentatively dropped.

The functions which operate on whole structures all have had to be completely revised and are described in the following sections, along with the present versions of the elementary functions.

Descriptions of Subroutines

The following subroutines have been adopted for use in the system.

1. add (w),dec(w).

These extract the 15 bit address and decrement parts respectively of a 36 bit quantity. They are coded as open subroutines.

2. comb(a,d) combines two 15 bit quantities to make a 36 bit quantity. It is coded as an open subroutine.

3. cwr(n).

The value of cwr(n) is the 36 bit contents of the register in location n. (Remember that the location is the 2's complement of the address of the register). cwr is coded as an open subroutine.

4. car(n), cdr(n).

The values of car(n) and cdr(n) are the 15 bit contents of the address and decrement parts respectively of the register in location n. They are coded as open subroutines. They are related to previously defined routines by the formulas

car(n) = add(cwr(n))   and
cdr(n) = dec(cwr(n))

5. consw(w).

This function takes the first word in the free storage list, puts w in it and returns with the location of the word as the value of consw(w). The situations before and after the execution of a program step

A = consw(w)

are shown in the figure.

Before

After

consw is available as a debugged SAP language routine.

6. cons(a,d)

This puts <u>comb</u>(a,d) into a register taken from free storage and returns with the location of the register. We have the relation.

cons(a,d) - consw(comb(a,d))

<u>cons has</u> been debugged.

7. erase (L)

Execution of <u>erase</u> (L) returns the word in location L to the free storage list. Its value is the former contents of the erased word.

This concludes the list of functions dealing with single words. The remaining functions deal with whole lists and list structures

8. copy (L)

The list structure starting in L is copied into free storage and the value of copy (L) is the location of the lead word of the copied structure. The program for <u>copy</u> is $\text{copy}(L) = (L=0 \rightarrow 0, \text{car}(L) = 0 \rightarrow L, 1 \rightarrow \text{cons}(\text{copy}(\text{car}(L)), \text{copy}(\text{cdr}(L))))$

9. equal (L1,L2)

The list structures starting in L1 and L2 are compared and the result is 1 if the structures agree both as to form and as to the identities of the objects in corresponding places. The program is

$$\text{equal}(L1,L2) = (L1=L2 \rightarrow 1,\ \text{car}(L1) = 0 \vee \text{car}(L2) = 0 \rightarrow 0, 1 \rightarrow \text{equal}(\text{car}(L1), \text{car}(L2)) \wedge \text{equal}(\text{cdr}(L1), \text{cdr}(L2)))$$

10. eralis (L)

This routine erases the list structure starting in L. Its program is

```
    subroutine (eralis(L))
    L = 0∨ car(L) = 0 → return
    M = erase (L)
    eralis (add(M))
    eralis (dec(M))
    return
```

11. maplist (L,f)

<u>maplist</u> constructs a list in free storage whose elements are in 1-1 correspondence with the elements of the list L.

The element corresponding to the element of L in location J is f(J). maplist is described more fully elsewhere in the memorandum.

maplist is debugged (Oct. 29)

12 print (L)

print (L) prints the list structure L in the restricted external notation. 119 character lines are used. Location of output is controlled by the sense switches as in UASPH2. print is debugged (Oct. 29)

13 read

The value of read is the location of a list read from cards or off-line tape according to the sense switch controls of UACSH2. The list is written in the restricted external notation. If the external name read is not found on any property list a new object with that name is created.

The afore-mentioned routines are sufficient for the differentiation program. The descriptions of additional routines follow.