EARLY HISTORY OF LISP

(1956-1959)

Herbert Stoyan

University of Erlangen

Germany

1. Material
2. McC 1975
3. algebraic programming
4. Functions
5. Lists as datastructures

Unklar:
1) Programmiererfahrung McC
2) Rolle von Minsky

Remarks before ...

. Why history ?

Do we learn from history ?

Have fun !

. How reliable are personal recollections?

The problem of viewing things in the perspective of the more recent events

Should we prefer written sources to the writer's recollections?

If a program is described - does that mean it is implemented that way?

Contents

A PROPOSAL FOR THE DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE

J. McCarthy, Dartmouth College
M. L. Minsky, Harvard University
N. Rochester, I. B. M. Corporation
C. E. Shannon, Bell Telephone Laboratories

August 31, 1955

A Proposal for the

DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE

We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.

The following are some aspects of the artificial intelligence problem:

1) Automatic Computers

If a machine can do a job, then an automatic calculator can be programmed to simulate the machine. The speeds and memory capacities of present computers may be insufficient to simulate many of the higher functions of the human brain, but the major obstacle is not lack of machine capacity, but our inability to write programs taking full advantage of what we have.

2) How Can a Computer be Programmed to Use a Language

It may be speculated that a large part of human thought consists of manipulating words according to rules of reasoning

PROPOSAL FOR RESEARCH BY JOHN MCCARTHY

During next year and during the Summer Research Project on Artificial Intelligence, I propose to study the relation of language to intelligence. It seems clear that the direct application of trial and error methods to the relation between sensory data and motor activity will not lead to any very complicated behavior. Rather it is necessary for the trial and error methods to be applied at a higher level of abstraction. The human mind apparently uses language as its means of handling complicated phenomena. The trial and error processes at a higher level frequently take the form of formulating conjectures and testing them. The English language has a number of properties which every formal language ~~as typed~~ described so far lacks.

1. Arguments in English supplemented by informal mathematics can be concise.

2. English is universal in the sense that it can set up any other language within English and then use that language where it is appropriate.

3. The user of English can refer to himself in it and formulate statements regarding his progress in solving the problem he is working on.

4. In addition to rules of proof, English if completely formulated would have rules of conjecture.

The logical languages so far formulated have either been instruction lists to make computers carry out calculations specified in advance or else formalizations of parts of mathematics. The latter have been constructed so as:

1. to be easily described in informal mathematics

2. to allow translation of statements from informal mathematics into the language.

3. to make it easy to argue about whether proofs of certain classes of propositions exist.

1.A language for AI

FORTRAN

- algebraic notation for programming

IPL

- list processing

The geometry theorem prover project

a marriage of FORTRAN with list processing

Basicfunctions to access the 704 word:

p	decrement	t	address

Accessfunctions:

XCPRF
XCDRF
XCTRF
XCARF

Constructorfunctions:

XCOMB4F
XCOMB5F

The result:

FLPL

The marriage was a happy one...

But an important problem remained open:

How to understand things like XCARF as functions mapping integers onto integers?

The programming problem

~~The problem~~

Programming is the problem of describing procedures or algorithms to an electronic calculator. It is difficult for two reasons

1. At present there does not exist an adequate language for human beings to describe procedures to each other. To be adequate such a language must be

a. ~~Compl~~ Explicit. There must be no ambiguity about what procedure is meant.

b. Universal. Every procedure must be describable

c. Concise. If a verbal description of a procedure is unambiguous in practice (i.e. if ~~it~~ well trained humans do not err about what is meant), there should be a formal description of the procedure in the language which is not much ~~more wordy~~ longer.

In order to achieve

The driving example during 1957 :

chess

Proposal of conditional function

$\mathrm{IF}(p,a,b)$

Massachusetts Institute of Technology
Cambridge 39, Massachusetts

To: P. M. Morse

From: J. McCarthy

Date: December 13, 1957

SUBJECT: A PROPOSAL FOR A COMPILER

ABSTRACT

This memorandum contains the first version of the first two chapters of a proposal for a compiler. Comments on the points raised so far and complaints about ambiguities are earnestly solicited.

CHAPTER 1

1. Introduction

The purpose of an automatic coding system in scientific computing is to reduce the elapsed time between the decision to make a computation and getting the results. It can make feasible computations which, without it, would be too complicated to undertake.

This report describes a proposed new automatic coding system which I hope will be a sufficient advance over those now available or soon to be available to justify the effort of writing the required translation program. The specifications for the system are presented in sufficient detail for evaluation of its merits, but would be subject to modification in the course of writing the translation program. A number of the ideas to be presented have been suggested by the Fortran system for the IBM 704, the proposed Scat system for the IBM 709, and the Flowmatic system for the UNIVAC. The source language is mainly independent of the machine being used, except that the provisions for referring directly to machine registers and their parts, which we believe must be included in any powerful source language, have been worked out only for the IBM 704.

In what follows, underlined terms are defined by the sentences in which they occur.

1.1 What is an Automatic Coding System

An automatic coding system has two parts. These are

1. a source language in which procedures for solving

Characteristica of the proposed programming language

1. large set of data types
   (numbers, logical types, lists, tables, vectors etc.)
2. describing results in terms of inputs independent of control structure
3. conditional expressions
4. extensibility of the language
   (abbreviations,
   private typing conventions etc.)
5. functions with multiple values
6. implementation of interpreters
   or compilers
7. modification, construction,
   compilation and execution
   of statements at runtime

The first notation for conditional expressions

IF(P, exp1:Q, exp2: ...OTHERWISE, ex]

Proposal For A Programming Language

<u>General</u> This report gives the technical specifications of a programming language proposed by the Ad Hoc Committee on Languages of the Association for Computing Machinery. The membership of this committee is as follows:

J. W. Backus (I.B.M.)
P. H. Desilets (Remington Rand)
D. C. Evans (Bendix Aviation Corp.)
R. Goodman (Westinghouse)
H. Huskey (University of California)
C. Katz (Remington Rand)
J. McCarthy (M.I.T.)
A. Orden (Burroughs Corp.)
A. J. Perlis (Carnegie Institute of Technology)
R. Rich (Johns Hopkins University)
S. Rosen (Burroughs Corp.)
W. Turanski (Remington Rand)
J. Wegstein (U. S. Bureau of Standards)

The objectives of the Ad Hoc Committee in designing the language described herein were to provide a language suitable for:

(1) publication of computing procedures in a concise and widely-understood notation,

and

(2) accurate and convenient programming of computing procedures in a language mechanically translatable into machine programs for a variety of machines.

It is recognized that certain one-for-one substitutions of one character-sequence for another will often be required to put a program written in the proposed language into a form mechanically acceptable by the input equipment of a given machine.

Certain subsidiary properties were taken to be necessary or strongly desirable to satisfy the two main goals above:

(a) The set of rules required to specify the syntax of the language should be kept as brief and uncomplicated as possible.

GO TO Statements

A GO TO statement may specify some statement other than the one immediately following as the statement to be executed next. They have the form:

GO TO e

where e is a designational expression. A designational expression is defined as follows:

1. The name (a symbol) of an imperative statement in the program containing this designation.
2. An expression s(E) where s is a symbol and E is an integer expression. For each such symbol s there must be a corresponding declarative statement:

   SWITCH $s(e_1, e_2, \ldots, e_n)$

   where each e_i is a designational expression. The statement designated when E has the value k is that one (if any) designated by e_k. If $E \leq 0$ or $E > n$, no statement is designated.
3. An expression of the form

   $(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n)$

   where each p_i is a boolean expression and each e_i is a designational expression. The statement designated is that one (if any) designated by e_k where p_k is the first true expression of $p_1, p_2, \ldots, p_k$. If no p_i is true, no statement is designated.

When e designates no statement, the statement to be executed next is the one following "GO TO e" in the program.

VARY and LOOP Statements

A VARY statement causes a segment of program immediately following it to be executed several times, once for each of a number of values of a variable given in the VARY statement. The segment of program to be repeated is terminated by a matching LOOP statement: namely, the first subsequent LOOP statement which is not the mate of some other VARY statement. Thus VARY and LOOP act like left and right parentheses respectively in their role of designating segments of program. A VARY statement has the following form:

VARY v := r

where v is a variable and r is a list of values. A list of values may have one

This proposal contains

Conditional Expressions

in the form of:

arguments of GOTO

Conditional Statements

A conditional statement is one which has the effect of one of several given statements in accordance with certain conditions which exist when it is encountered. Let any of the following imperative statements be termed a module:

1. A replacement statement
2. A GO TO statement
3. A RETURN statement
4. A STOP statement
5. A procedure statement
6. A substitution statement

Then conditional statements are recursively defined as any statement of the following form:

$$P_1 \rightarrow S_1, P_2 \rightarrow S_2, \ldots, P_m \rightarrow S_m$$

where each P_i is a boolean expression and each S_i is a module or a conditional

statement enclosed in parentheses.

The effect of a conditional statement is that of the single statement S_i following the first true boolean expression, P_i, $i = 1, 2, \ldots, m$. More precisely, the effect of the conditional statement given above, where the name of the next statement is NEXT, is the same as that of the following sequence of statements:

```
            GO TO(P1→N1, P2→N2, ..., Pm→Nm)
            GO TO NEXT
    N1)     S1
            GO TO NEXT
    N2)     S2
            GO TO NEXT
              .
              .
              .
              .
    Nm)     Sm
    NEXT)
```

LABEL Statements

A LABEL statement is a declarative statement which associates a symbol (label) with an arbitrary sequence of statements occurring in the program containing the LABEL statement. The form of a LABEL statement is:

$$\text{LABEL } s(s_1, s_2)(s_3, s_4)\ldots(s_{2n-1}, s_{2n})$$

where each s_{2i-1} is the name of a statement and either each s_{2i} is the name of

The result of the Zürich Meeting:

Conditional expressions discarded

But McCarthy proposed even more futuristic things.

June 10, 1958

From John McCarthy

To: A. J. Perlis and W. Turanski

Subject: Some Proposals for the Volume 2 (V2) Language

I. General Remarks

1.1 The material that was cut out of Volume 1 and not subsequently restored does not amount to enough to justify a Volume 1 1/2. Therefore I think we should not try to produce an immediate report but should aim after long range goals.

1.2 Our major effort with respect to Volume 2 should be to make it possible to change the language within the language. This may mean having parts of the compiler under the control of the program ~~and~~ at object time.

1.3 The problem of compilation has two parts. These are translation and optimization. The translation rules determine from the given source program a number of possible translates which will perform the desired calculation. The optimazation rules selects the best of these according to certain criteria. I think that for now we should concentrate on the translation problem and leave optimazation for later. This means that we are interested in general ways of defining transformation of text that do not involve scans of alternative ways of doing the same thing but are of a more straightforward nature.

In order to tackle the translations problem in its barest form I propose that we consider translations of texts which have a certain of a preliminary translation into it which lengthens the translation process somewhat. I think that the advantages will outweigh this disadvantage. In order to make clear the expression notation we give an example of a program written both in the expression notation and a notation like that of Volume 1. but which has vertical parenthesis. The advantages of the Volume 1 notation from the human point of view are clear.

3. Functional Variables, Forms and Fluents

3.1 Functional Variables

In V 1 any functions which appear are constants, i.e. we never refer to a function f which is sometimes sine and sometimes cosine. We propose that functional quantities be admitted to Volume 2, that is our symbols can represent not merely integers or floating point numbers or ~~bullion~~ boolean quantities but also functions and the function which a symbol represents can be changed by appropiate statements just as the number represented by a symbol is changed by the execution of the statements in a Volume 1 program. For convenience we shall give examples in an operational notation rather than in the uniform expression notation of the previous section. We will describe the kinds of program we want to admit without stopping to propose a way of representing functions in the computer. Here is a sample program.

```
f = sin
g = f + cos
a = g (3)
g= g + 3
print (a)
```

What is finally printed by this program is sin (3) + cos (3). If functional quantities are admitted we shall want the following operations on functions:

1. Addition, subtraction, multiplication and division for numerical valued functions. In general we shall want any operations which were appropiate on the range of a set of functions.

2. Composition. $f \circ g$ defined by $(f \circ g)(x) = f(g(x))$ is appropiate whenever the domain of f and the range of g coincide.

3. Abstraction from forms. Elementary mathematics is plagued by ambiguity between functions and their values. Most mathematical texts depend upon context to tell the reader which is meant. In dealing with a computer we must avoid this ambiguity and therefore I have chosen to propose that we use the Church lambda notation. According to Church $x^2 + y^2$ is not a function but a form in x and y. We can make from it a function by writing f = lambda (x, y) $(x^2 + y^2)$. The lambda symbol is a quantifier and makes x and y into dummy variables. Thus we have lambda (x, y) $(x^2 + y^2)$ (3, 4,) = 25 and lambda (x, y) $(x^2 + y^2)$ $(x + 1, y) = (x+1)^2 + y^2$. This implies that we must also admit forms into our system and an appropiate collection of operations on them. We will not go into this right now.

4. Operations on functions such as differentation, other differential operators, and integration. These are defined only when the functions are represented in certain ways, i.e. one cannot differentiate a function represented only by a subroutine. Note that the operator D which takes functions into functions may again be regarded as a function whose domain and range are spaces of functions. We shall admit to the system variables whose values are higher order functions such as D but will not guarantee in the present system to provide an adequate set of quantities of this kind in a first version of the system.

This paper contains...

1. the first reference to Lambda-Calculus made by computer language people

2. the idea of making functions into first-class datatypes

3. proposed operations on functions:
   a) Basic-functions of value-set
   b) symbolmanipulation of expression-representation
   c) composition

4. compiler written in a rule-oriented manner

People connected with early LISP-development

John McCarthy , Ass. Prof. (*Dept.EE*)

Steven B.Russell , Programmer (*AI Lab.*)
Klim Maling , Programmer (*AI Lab.*)

Robert Brayton , Ph D-Student (*Math.Dept.*)
David C.Luckham , Ph D-Student (*Math.Dept.*)
David M.R.Park , Ph D-Student (*Math.Dept.*)

Nathaniel Rochester , visiting Prof. (*Dept.EE*)

Early Users

James R.Slagle , Ph D-Student (*Math.Dept.*)
Paul W.Abrahams , Ph D-Student (*Math.Dept.*)
Louis Hodes , Ph D-Student (*Math.Dept.*)

Daniel G.Edwards , undergraduate-Student (*Dept.EE*)
Seymor Z.Rubenstein,undergraduate Student (*Dept.EE*)
Solomon H.Goldberg , graduate Student (*Dept.EE*)

Interested discussants

Marvin L.Minsky , Ass. Prof. (*Math.Dept.*)
Dean Arden , Ass. Prof. (*Dept.EE*)
Claude Shannon , Professor (*Dept.EE*)
Hartley Rogers,Jr., Ass. Prof. (*Math.Dept.*)
Roland Silver (*Lincoln Labs.*)
Alan Tritter (*Lincol Labs.*)

further related people:

Dan Bobrow, Pat Fischer,
T.Kurtz, W.E.Hansalik, W.Lee, V.Yngve,
P.Fox, P.Bagley, W.D.Comfort,J.C.McPherso
M.Levy, L.Sutro, W.Carter, B.Chartres

N. Rochester

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 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), cbitr (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; $q_1, \ldots,$ qm) causes an array A of size to be established whose contents are the quantities $q_1, \ldots, q_n$. 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 arguments 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. $(a; p_1, \ldots, p_n)$

This causes the expressions $p_1, \ldots, p_n$ to be entered in the property list associated with the symbol a. Each symbol in the program has such a property 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_o, a_1,) This forms a word whose nth bit is the nth bit of a_o 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

The driving example during summer/fall of 1958:

differentiation

Importance of recursive functions

and functionals

maplist(list,
variable,
expression)

Highlights of the proposed language:

1. starting from FORTRAN
2. adding new statements:
   a)a "set"-statement (A; q1,...,qm)
   b)"declarative"-statements (a; p1,...,pn)
3. list processing
   a) Basic-functions: cwr,cpr,cdr,ctr,car
   b) extracting functions: pre,dec,tag,adr
   c) modification functions: stpr,ctdr,sttr,star
   d) modification by using basic-functions on the left side of assignment stmt.
   d) constructorfunctions: comb4, comb5, consw, consel, consls
   e) other functions: pointer movement, erasure etc.
4. no Lambda-notation
5. data types: integers (= addressses !), registers, truth values, locations, functions

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.

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

equal (L1,L2) = (L_1 = L_2 → 1, cir (L1) ≠ cir (L_2) → 0, cir (L1) = 0 ∧ car (L1) ≠ car (L2) → 0, cir (L1) = 1 ∧ cwr (car(L1)) ≠ cwr (car(L2)) → 0, car(L1) = 2 ∧ ~ equal (car(L1), car (L2)) → 0, 1 → equal (cdr(L1), cdr(L2))

```
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
```

After difficulties of D.C.Luckham to implement maplist:

Introduction of Lambda-notation

maplist(list,λ(var,body))

Artificial Intelligence Project---RLE and MIT Computation Center

Symbol Manipulating Language --Memo 2

A REVISED VERSION OF "MAPLIST"

by John McCarthy

The version of *maplist* in memo 1 was written "maplist(L,J,f(J))" where J is a dummy variable which ranges over the address parts of the words in the list L and xx f(J) was an expression in J. This version had two serious defects. First, the location of the word in which J was stored was frequently needed. The xxx second xx turned up when I tried to write the SAP program for *maplist*. The designation of J as the name of the indexing variable cannot conveniently be done in the calling sequence of *maplist*. Instead we do it in specifxying the function f using the Church λ notation for functional abstraction if necessary. In addition to the above mentioned defects the x old version was ambiguous in that it did not say how words of the three types should be treated.

The new *maplist* is written "maplist(L,f)". Its value is the location of a list formed from free storage whose elements correspond in a 1-1 way with the elements of L. The element of the new list which corresponds to the element of the old list in location J has address part f(J) and *always has indicator 2*. The new *maplist* thus always produces xxxxxxxxxxxxxx a list of lists. This lack of

```
diff(L,V) = (car(L)= const→ copy(CO),car(L)= var→(car
(cdr(L)) = V→ copy(C1),1→ copy(CO)),car(L)= plus →
consel(plus,maplist(cdr(L),λ(J,diff(car(J),V)))),car(L)= times→
consel(plus,maplist(cdr(L),λ(J,consel(times,maplist(cdr(L),
λ(K,(J ≠ K →copy(car(K)),1→ diff(car(K),V)))))))))
```

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 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 consls 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.

AI-Memo 3

1. first usage (10/21/58) of name

LISP

2. simplification of the design in the light of some experience:

   a) work only with address and decrement
      - new atom-symbol structure
      - consel and consls clash to cons

   b) storage and pointer functions dropped
      - dead of RPLACA/RPLACD -forerunner

Artificial Intelligence Project--RLE and MIT Computation Center
Symbol Manipulating Language--Memo 4--Revisions of the Language

John McCarthy

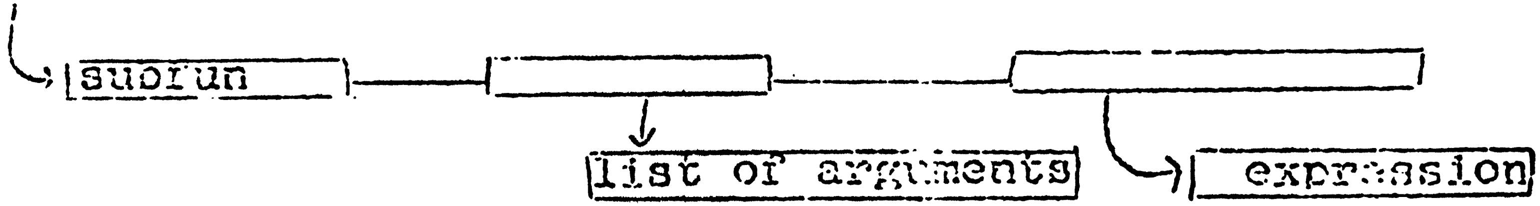
1. Protected temporary storage.

When a routine is defined recursively as are maplist and diff (that is, when the routine itself occurs in the program defining the routine, certain special problems with temporary storage arise. Specifically, the execution of the routine as a subroutine of itself makes use of the same temporary storage registers. There are a number of ways to avoid a conflict over temporary storage, and after much argument the following solution has been adopted. Those temporary storage registers which should be preserved when the routine uses a subroutine which may use the subroutine itself, form a single block of consecutive registers private to the routine which is called the block of protected temporary storage of this routine. The register in which IR4 is stored is also included in this block. Except for the register in which IR4 is stored the routine is required to be transparent to the registers of the block; that is the contents of this block must be the same when the routine exits as they were when it was entered. In order for the routine to be able to use the registers of the block it must save them before it uses them and restore them afterwards. The situation is then similar to the SHARE convention on IR1 and IR2. They are saved by a routine which puts them on what is called the public push down list or PPDL, and before the main routine exits they are restored from this list. The SAVE and UNSAVE routines are used as follows; a program using them might be

```
SXD RS1,4
TSX SAVE,4
    RS1+1,0,5
...
...     (program that uses RS2 to RS5)
```

5.10 Substitutional functions.

The value of a substitutional function applied to a list of arguments is the result of substitutions these arguments for the objects on an ordered list of arguments in a certain expression containing these arguments. A substitutional function is represented in the machine by a list structure as shown below.



There is a routine apply(L,f) whose value is the result of applying a function to a list of arguments. This routine expects the function f itself to be described by an expression. The kinds of expressions for functions which apply will interpret has not been determined and for the present we shall only consider the case where car(f)=subfun. Thus our initial version of apply is:

AI-Memo 4

1. **description of function calling conventions (SAVE, UNSAVE)**

2. **some new functions select, list, search, subst, pair**

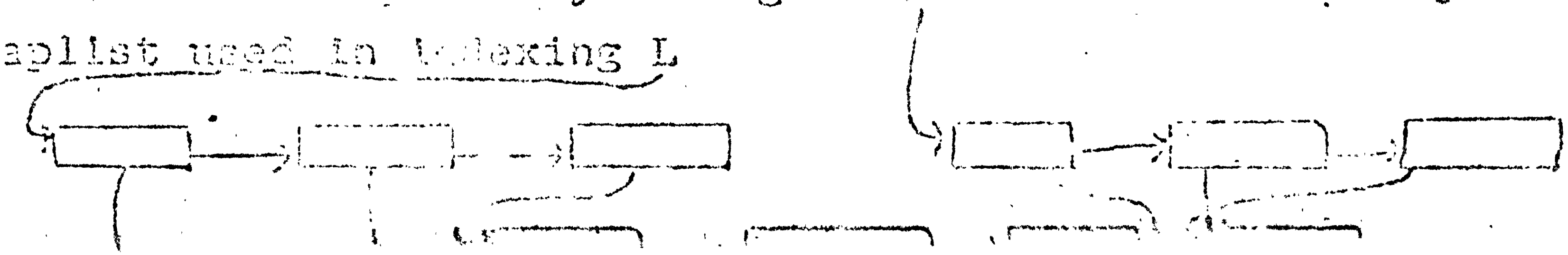
3. **first variant of apply for "substitutional" functions**

apply(L,f)=(car(f)=subfun→sublis(pair(car(cdr(f)), L),car(cdr(cdr(f)))),1→error)

This definition presents the problem that the list created by the <u>pair</u> has not further use after apply has been evaluated and is not attached to any named variable. Therefore unless the compiler is made to insert instructions to erase such auxiliary lists they will steal space permanently from the free storage list.

5.11 The second order <u>maplist</u>.

Consider a list of lists each of which has the same number of elements. It is desired to scan over these lists in parallel and to create a new list whose elements correspond to the elements of the listed list but whose value is a given function f of a list corresponding elements of the listed lists. The figure shows the situation when the calculation is part way through.



AI-Memo 5

(Nat Rochester)

1. first simplification program

2. proposals to simplify notation:

   a) for compositions of car and cdr
      write: c...r

   b) format rules
      - indentation of conditional expressions
      - writing compositions decomposed

3. proposal to name recursive Lambda-expressions
   (name (var), body)

4. revival of REPLACE

Artificial Intelligence Project---RLE and MIT Computation Center

Symbol Manipulation Language---Memo 5

by N. Rochester

18. 11. 58

Table of Contents

(pages 19–25: all of these are parts of simp)

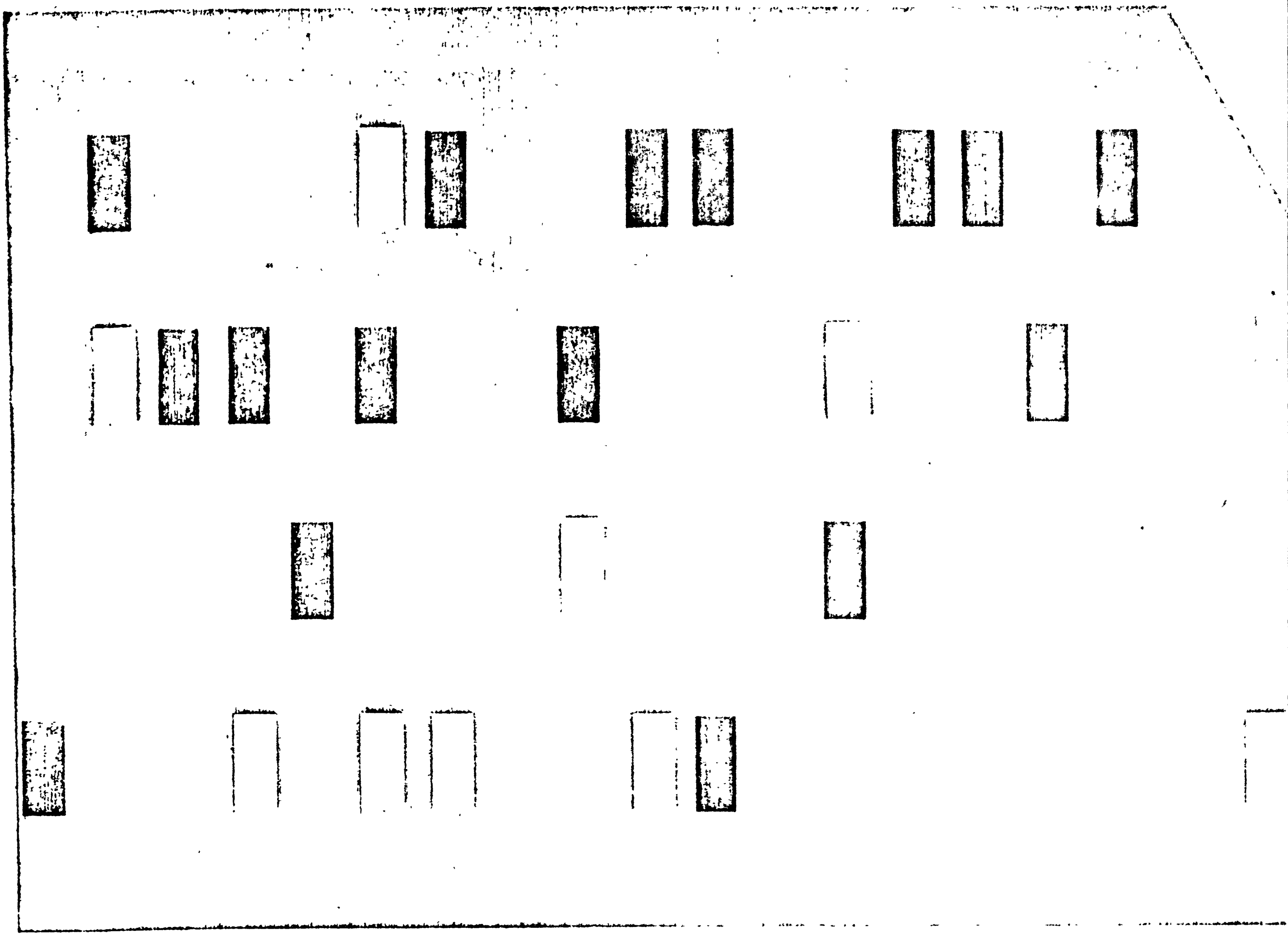
Nests of car and cdr

to express a nest of car's and cdr's it is necessary to write c and r only once

car(car(J)) = caar(J)
car(cdr(J)) = cadr(J)
car(cdr(car(cdr(car(cdr(J))))))=cadadadr(J) etc.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

COMPUTATION CENTER



Research and Educational Activities

SEMI-ANNUAL REPORT

NUMBER 4 DECEMBER, 1958

COOPERATING COLLEGES OF NEW ENGLAND

The semi-annual report
of the
MIT-computer center
December 1958

A routine for applying a function to an argument, where the function is described by a symbolic expression has been programmed but ot yet debugged. This routine will be the basis of an interpreter.

will be used for programming the <u>advice taker</u> system but which is also of more general use.

During the past 3 months, the project has developed a programming language (called LISP) for manipulating symbolic expressions, and has coded and debugged the major subroutines. The use of electronic computers for symbolic work, such as formal differentiation and integration, checking proofs and finding proofs in formal logical systems, and translating from a source programming language to machine language has not been developed as far as the programming of numerical calculation is concerned, partly because of the non-existence of standard ways of describing such computations.

Our programming language has been developed as a more or less machine-independent way of describing symbolic processing. The language to date has been described in internal memoranda of the Artificial Intelligence Project and in a forthcoming Research Laboratory of Electronics technical report. Its main features are:

1. Expressions are represented in the machine by list structures similar to those used by Newell, Simon, and Shaw in their Information Processing Languages.

2. Externally, expressions are represented by sequences written with parentheses and commas. (TIMES,X,(PLUS,X,1),(SIN,Y)) is a typical sequence, corresponding to the elementary form X(X+1)sin(Y).

3. Programs are written in an algebraic form resembling FORTRAN.

4. By the use of conditional expressions and recursive definitions, it is possible to describe complicated processes very briefly and in a way that is natural to use.

5. Functional abstraction as described by Church is used to convert forms into functions.

At present, routines written in LISP are hand-translated into SAP, but we expect to begin on a compiler soon. A routine for applying a function to an argument, where the function is described by a symbolic expression, has been programmed but not yet debugged. This routine will be the basis of an interpreter.

A routine for differentiating elementary functions analytically has been written and will be available for demonstrations as soon as suitable input-output facilities have been added.

The problem of the gap:

between November 1958 and March 1959
no saved record.

However:

After the Teddington-conference McCarthy started with writing the

universal LISP-function.

And when at all then in this time the famous event may have happened...

- **Steve Russell started handtranslating some version of the** apply **function.**
- **McCarthy drafted his paper "Recursive Functions...".**

3.1 Representation of S-functions as S-expressions.

The representation is determined by the following rules:

1. Constant S-expressions can occur as parts of the F-expressions representing S-functions. An S-expression $\mathcal{E}$ is represented by the S-expression. (QUOTE, $\mathcal{E}$)

2. Variables and function names which were represented by strings of lower case letters are represented by the corresponding strings of the corresponding upper case letters. Thus we have FIRST, REST and COMBINE, and we shall use X,Y etc. for variables.

3. A form is represented by an S-expression whose first term is the name of the main function and whose remaining terms are the arguments of the function. Thus combine[first[x]; rest[x]] is represented by (COMBINE,(FIRST,X),(REST,X))

4. The null S-expression Λ is named NIL.
5. The truth values 1 and 0 are denoted by T and F. The conditional expression

   write$[p_1 \rightarrow e_1;\ p_2 \rightarrow e_2;\ \ldots,\ p_k \rightarrow e_k]$

is represented by

   (COND.$(p_1,e_1),(p_2,e_2),\ldots,(p_k,e_k)$)

6. $\lambda[[x;\ldots;s];\mathcal{E}]$ is represented by (LAMBDA,(X,...,S); $\mathcal{E}$)
7. label$[a;\mathcal{E}]$ is represented by (LABEL,a,$\mathcal{E}$)
8. x=y is represented by (EQ,X,Y)

3.1 Representation of S-functions as S-expressions.

The representation is determined by the following rules:

1. Constant S-expressions can occur as parts of the F-expressions representing S-functions. An S-expression $\mathcal{E}$ is represented by the S-expression. (QUOTE, $\mathcal{E}$)

2. Variables and function names which were represented by strings of lower case letters are represented by the corresponding strings of the corresponding upper case letters. Thus we have FIRST, REST and COMBINE, and we shall use X,Y etc. for variables.

3. A form is represented by an S-expression whose first term is the name of the main function and whose remaining terms are the arguments of the function. Thus combine[first[x]; rest[x]] is represented by (COMBINE,(FIRST,X),(REST,X))

4. The null S-expression Λ is named NIL.

5. The truth values 1 and 0 are denoted by T and F. The conditional expression

write[$p_1 \rightarrow e_1$; $p_2 \rightarrow e_2$; ..., $p_k \rightarrow e_k$]

is represented by

(COND,(p_1,e_1),(p_2,e_2),...,(p_k,e_k))

6. λ[[x;...;s]; $\mathcal{E}$] is represented by (LAMBDA,(X,...,S); $\mathcal{E}$)

7. label[a; $\mathcal{E}$] is represented by (LABEL,a,$\mathcal{E}$)

8. x=y is represented by (EQ,X,Y)

AI-Memo 8

First draft of "Recursive Functions..."

1. consequent usage of
   FIRST, REST and COMBINE
   instead of CAR, CDR and CONS.
2. first proposal of QUOTE.
3. M-Language still called "F-language".
4. S-Language uses commata.
5. Truth-values in F-language: 1 and 0.
6. first design of translation rules.
   between F- and S-expressions.
7. first LISP-interpreter variant.

```
apply[f,args] = eval[combine[f;args]]
eval[e] = [first[e] = NULL →[null[eval[first[rest[e]]]]→ T;
                            1→ F];
           first[e] = ATOM →[atom[eval[first[rest[e]]]]→ T;
                            1 → F ;
           first[e] = EQ →  [eval[first[rest[e]]] = eval[first[rest[rest[e]]]]→ T;
                            1→ F];
           first[e] = QUOTE→ first[rest[e]];
           first[e] = FIRST→ first[eval[first[rest[e]]]];
           first[e] = REST→ rest[eval[first[rest[e]]]];
           first[e] = COMBINE →combine[eval[first[rest[e]]];
                                        eval[first[rest[rest[e]]]]];
           first[e] = COND →evcon[rest[e]];
           first[first[e]] = LAMBDA→evlam[first[rest[first e]]];
                                          first[rest[rest[first[e]]]];
                                          rest[e]];
           first[first[e]] = LABEL→eval[combine[subst[first[e];
                                                       first[rest[first[e]]];
                                                       first[rest[rest[first[e]]]]
                                                 rest[e]]]]
evcon[c] = [eval[first[first[c]]] = 1→eval[first[rest[first[e]]]];
            1 →evcon[rest[c]]]
evlam[vars;exp;args] = [null[vars]→eval[exp];
                        1→ evlam[rest[vars];
                                 subst[first[args];first[vars];exp];
                                 rest[args]]]
```

L I S P

Programmer's Manual

MIT Artificial Intelligence Project

MODIFICATIONS

1.	cons(a,d)	3/3/59
2.	consw(w)	3/3/59
3.	copy(L)	3/3/59
4.	equal(L1,L2)	3/3/59
5.	eralis(L)	3/3/59
6.	erase(L)	3/3/59
7.	maplist(L,f)	3/3/59
8.	Open Subroutines	3/3/59
9.	search(L,p,f,u)	3/3/59
10.	apply	3/3/59

The first apply-eval

1. substituting call-by-name interpreter.
2. evaluates terms only.
   No variables etc.
3. eval not necessary.
4. Errors:.
   a) truth-values represented by
      T and F.
   b) substitution function substitutes
      everywhere.
   c) clause for Lambda-expressions
      forgotten.
5. McCarthy corrected a)
   and first case of b).

The first known interpreter

1. deep-binding call-by-value interpreter.
2. evaluates terms and variables.
3. eval contributes heavily.
4. clauses of eval:
   a) variables
   b) CONST - constants (S-Exprs).
   c) VARC - variables evaluated.
   d) VARE - variables to be evaluated.
   e) LABEL - evaluate function body.
   f) SUB - substituting the A-list.
   g) INTV - for truth-values.
   h) normal term.
   i) COND - forgotten.
5. CAR, CDR, CONS separate.
   Other: SUBR or EXPR on P-List.

Lisp program for single statement interpreter

```
APPLY(F,L,A)=select(car(F);
          -1,app2(F,L,A);
          lambda,eval(caddr(F),append(pair(cadr(F),L),A));
          label,apply(caddr(F),L,append(pair(cadr(F),caddr
                              (F)),A));
          apply(eval(F,A),L,A))
EVAL(E,A)=select(car(E);
          -1,search(A,λ(J,caar(J)=E),λ(J,cadar(J)),error);
          intv ,search(cadr(E),λ(J,car(J)=int),λ(J,cdadr(J)),
                              error);
          sub,sublis(A,eval(cadr(E),A));
          const,cadr(E);
          label,eval(caddr(E),append(pair(cadr(E),caddr(E)),
                              A));
          varc,search(A,λ(J,cadar(J)=cadr(E)),λ(J,cadar(J)),
                              error);
          care,search(A,λ(J,caar(J)=cadr(E)),λ(J,eval(cadar(J),
                              cdr(J)),error);
          apply(car(E),maplist(cdr(E),λ(J,eval(car(J),A))),A))
APP2(F,L,A)=select(F;car,caar(L);cdr,cdar(L);cons,cons(car(L),cadr(L));
          list,L;null,car(L)=0;atom,caar(L)=-1;
                              search(F,λ(J,car(J)=subr∨expr),
                                   λ(J,(car(J)=subr→app3(F,L,
                                   1→apply(cadr(J),L,A))),
          search(A,λ(J,caar(J)=F),λ(J,apply(cadar(J),L,A)),
                                   error))
evcon(E,A) = (E=0→error,eval(caar(E),A)→eval(cadar(E),A),1→evcon
                                   (cdr(E),A))
```

March 3, 1959

Modification number 10

Author: S. Russell

The 2. theoretical apply-eval

1. deep-binding call-by-name interpreter.
2. evaluates terms and variables.
3. eval contributes heavily.
4. clauses of eval:
   a) variables
   b) basic-functions
   c) LAMBDA-expression
   d) LABEL-expression
   e) evaluation of arguments for functions on A-list
5. APPLY quotes its arguments.

Theory behind practice.

XIII. ARTIFICIAL INTELLIGENCE*

Prof. J. McCarthy
Prof. M. L. Minsky
Prof. N. Rochester†
Prof. C. E. Shannon
P. W. Abrahams
D. G. Bobrow
R. K. Brayton
L. Hodes
L. Kleinrock
D. C. Luckham
K. Maling
D. M. R. Park
S. R. Russell
J. R. Slagle

A. THE LISP PROGRAMMING SYSTEM

The purpose of this programming system, called LISP (for LISt Processor), is to facilitate programming manipulations of symbolic expressions.

The present status of the system may be summarized as follows:

(a) The source language has been developed and is described in several memoranda from the Artificial Intelligence group.

(b) Twenty useful subroutines have been programmed in LISP, hand-translated into SAP (symbolic machine language for the IBM 704 computer) and checked out on the IBM 704. These include routines for reading and printing list structures.

(c) A routine for differentiating elementary functions has been written. A simple version has been checked out, and a more complicated version that can differentiate any function when given a formula for its gradient is almost checked out.

(d) A universal function apply has been written in LISP, hand-translated, and checked out. Given a symbolic expression for a LISP function and a list of arguments apply computes the result of applying the function to the arguments. It can serve as an interpreter for the system and is being used to check out programs in the LISP language before translating them to machine language.

(e) Work on a compiler has been started. A draft version has been written in LISP, and is being discussed before it is translated to machine language or checked out with apply.

(f) The LISP programming system will be shown in this report to be based mathematically on a way of generating the general recursive functions of symbolic expressions.

The mathematical LISP system is described in more detail in Section XIII-D.

B. ENGINEERING CALCULATIONS IN LISP

The application of the List Processing Language to the calculation of properties of linear passive networks is being studied by N. Rochester, S. Goldberg, C. S. Rubenstein, D. J. Edwards, and P. Markstein. A series of programs in List Processing Language is being written. These will enable the IBM 704 computer to accept a description of a

*This work is supported jointly by Research Laboratory of Electronics and the Computation Center, M.I.T.

†Visiting Professor of Electrical Engineering, M.I.T.

```
    apply[f;args]=eval[cons[f;appq[args]];NIL]
where
    appq[m]=[null[m]→NIL;T→cons[list[QUOTE;car[m]];appq[cdr[m]]]]
and

        eval[e;a]=[
atom[e]→eval[assoc[e;a];a];
atom[car[e]]→[
            car[e]=QUOTE→cadr[e];
            car[e]=ATOM→atom[eval[cadr[e];a]];
            car[e]=EQ→[eval[cadr[e];a]=eval[caddr[e];a]];
            car[e]=COND→evcon[cdr[e];a];
            car[e]=CAR→car[eval[cadr[e];a]];
            car[e]=CDR→cdr[eval[cadr[e];a]];
            car[e]=CONS→cons[eval[cadr[e];a];eval[caddr[e];a]];
            T→eval[cons[assoc[car[e];a];evlis[cdr[e];a]];a]];
            caar[e]=LABEL→eval[cons[caddar[e];cdr[e]];cons[list[cadar[e];car[e];a]];
            caar[e]=LAMBDA→eval[caddar[e];append[pair[cadar[e];cdr[e]];a]]]

            and
                evcon[c;a]=[eval[caar[c];a]→eval[cadar[c];a];T→evcon[cdr[c];a]]
            and
                evlis[m;a]=[null[m]→NIL;T→cons[list[QUOTE;eval[car[m];a]];
            evlis[cdr[m];a]]
```

of the words with character information means that the association lists do not themselves represent S-expressions, and that only some of the functions for dealing with S-expressions make sense within an association list.

c. Free-Storage List

At any given time only a part of the memory reserved for list structures will actually be in use for storing S-expressions. The remaining registers (in our system the number, initially, is approximately 15,000) are arranged in a single list called the _free-storage list_. A certain register, FREE, in the program contains the location of the first register in this list. When a word is required to form some additional list structure, the first word on the _free-storage list_ is taken and the number in register FREE is changed to become the location of the second word on the free-storage list. No provision need be made for the user to program the return of registers to the free-storage list.

This return takes place automatically, approximately as follows (it is necessary to give a simplified description of this process in this report): There is a fixed set of base registers in the program which contains the locations of list structures that are accessible to the program. Of course, because list structures branch, an arbitrary number of registers may be involved. Each register that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of _car_ and _cdr_ operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a _car_-_cdr_ chain from any base register. Such a register may be considered abandoned by the program because its contents can no longer be found by any possible program; hence its contents are no longer of interest, and so we would like to have it back on the free-storage list. This comes about in the following way.

Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free-storage list, a reclamation cycle starts. First, the program finds all registers accessible from the base registers and makes their signs negative. This is accomplished by starting from each of the base registers and changing the sign of every register that can be reached from it by a _car_-_cdr_ chain. If the program encounters a register in this process which already has a negative sign, it assumes that this register has already been reached.

After all of the accessible registers have had their signs changed, the program goes through the area of memory reserved for the storage of list structures and puts all the registers whose signs were not changed in the previous step back on the free-storage list, and makes the signs of the accessible registers positive again.

This process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists. Its

```
N=NETWORK

(LAMBDA,(N),(CONS,(CONST,NODE),(NODAL,(CDR,N),(INTV,0))))

(LABEL,NODAL,(LAMBDA,(J,K),(COND,((NULL,J),K),
((INTV,1),(SEARCH,K,(CONST,(LAMBDA,(X),(EQUAL,(CAR,X),
(CAR,(CDR,(CAR,J))))))),
(CONST,(LAMBDA,(X),(NDLIS,J,K))),
(CONST,(LAMBDA,(X),(NDLIS,(CDR,J),(CONS,(CAR,(CDR,(CAR,J))),
K)))))))))

(LABEL,NDLIS,(LAMBDA,(J,K),(SEARCH,K,(CONST,(LAMBDA,(X),
(EQUAL,(CAR,X),(CAR,(CDR,(CDR,(CAR,J))))))),
(CONST,(LAMBDA,(X),(NODAL,(CDR,J),K))),
(CONST,(LAMBDA,(X),(NODAL,(CDR,J),(CONS,(CAR,(CDR,(CDR,
(CAR,J)))),K)))))))
```