R Silver

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)
    ...
```

```
    TSX UNSAVE,4
        RS1+1,0,5
    LXD RS1,4
    TRA 1,4
RS5
RS4
RS3
RS2
RS1
```

The reason that IR4 is handled differently from the other quantities that have to be saved is that it is used to enter SAVE and UNSAVE and therefore must be saved before SAVE is entered and restored after the exit from UNSAVE.

This procedure for handling protected temporary storage has another virtue which shows up in the routine for differentiation described later. In the multiplication case the program arising from the first calls its argument J and the program arising from the second must use this quantity J to compare with K and must also have V which is one of the arguments of the original diff. If the argument J of the first is stored in a protected temporary storage register, it will be there when the routine compiled from the second wants it in spite of the fact that the earlier values of K on the second maplist will have used diff. The general property of the present system of handling protected temporary storage is that the values stored therein are accessible to all explicitly written lower level routines but are not changed either by them or by the use of the original routine as a subroutine. (As an aside, it may be remarked that if an attempt were made in the diff routine to expand out one of the references to diff in it by substituting the definition

of diff there would occur a collision of bound variables (unless the names were changed) and that this collision would reflect itself programwise in a collision of temporary storage. The relation between bound variables and protected temporary storage is close and might be worth a systematic investigation.)

2. A Revised Version of "Maplist"

2.1 Problems posed by the old maplist.

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 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 second 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 specifying the function f using the Church notation for functional abstraction if necessary. In addition to the above mentioned defects the 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).

The program for maplist can be written.

maplist (L,f) = (L - 0→0,1→cons (f(L), maplist (cdr(L),f)))

A SAP version of this maplist is given in the appendix to this memorandum. It uses the programming convention that the arguments are given to maplist in the ac and the mq that the return is by TRA 1,4, and that the function f is given in the form of an instruction TXL F where F is the address of a routine for computing f which expects its argument in the AC, returns its result in the AC, and returns by TRA 1,4.

2.2 Functional abstraction - The Church λ's.

In order to be able to use forms in the definition of the functions which are the arguments of maplist it is necessary to use functional abstraction as developed by Church. We digress from the subject of programming into mathematical logic in order to explain the idea of functional abstraction.

We shall call an expression f a functional expression when we have given rules for assigning a value to expressions such as f(3) or f(3,4) according to the number of arguments f is supposed to take. An expression such as $x^2 + y$ does not meet this requirement because, should we write $(x^2 + y)(3,4)$, which no-one does, it would be ambiguous whether we meant 3 to replace x or y. Other difficulties also arise. To clarify this matter Church invented his λ-operator. $(\lambda x).\mathcal{E}(x)$, where $\mathcal{E}(x)$ is some expression in the symbol x, denotes the function whose value for a given argument is to be obtained by substituting that argument for x in the expression $\mathcal{E}(x)$ and evaluating that expression. Thus $(\mathcal{E}x).x^2$ denotes the function which squares its argument. $(\lambda xy).(x^2 + y)$ is the function whose value is obtained by substituting the first argument for x and the second argument for y. Thus $(\lambda xy).(x^2 + y)(3,4) = 13$ and $(\lambda xy).(x^2 + y)(4,3) = 19$. The letter λ serves as a quantifier in that the variable on which it operates becomes bound and can be replaced by another variable provided this is done everywhere it occurs in the expression on which λ operates and provided the new variable does not occur in this expression. Thus we can write $(\lambda xy).(x^2 + y) = (\lambda xu).(x^2 + u)$. An expression $x^2 + y$ without λ's is called a form and thus we may say that the λ-operator has the effect of transforming forms into functions.

The use of the λ-operator and its properties are fully described in Church's tract, The Calculi of λ-Conversion, which appeared as an Annals of Mathematics Study.

For now at least, we only need the simplest properties of the λ-operator. We shall write $\lambda(J,\mathcal{E})$ for the function whose value is obtained by substituting the argument for J in the expression $\mathcal{E}$ and use a similar notation for functions of several variables. Thus $\lambda(J,J*J+1)$ denotes the function whose value is obtained by squaring the argument and adding 1. The function which we have written above $(\lambda xy).(x^2+y)$ will be written $\lambda(x,y,x*x+y)$ in our programming system.

2.3 An example

As an example of the use of the new maplist and of the use of functional abstraction we shall rewrite the routine for differentiating simple algebraic expressions. In addition to the changes occasioned by the new maplist we shall change the notation we used for constants and the identity function and make the routine one which can perform partial differentiation with respect to a given variable. We need the following conventions for the representation of expressions by list structures:

1. As before a sum or product is represented by a list whose first word has plus or times in its address part and whose subsequent elements are words containing the expressions to be added or multiplied.
2. A constant or variable is represented by the location of its property list.

The routine for differentiating the expression represented by the list structure starting in location L with respect to the variable V is:

diff(L,V) = (L = V⟶C1,car(L) = 0⟶C0,car(L) = plus ⟶
cons(plus,maplist(cdr(L),λ(J,diff(car(J),V)))),car(L) = times ⟶
cons(plus,maplist(cdr(L),λ(J,cons(times,maplist(cdr(L),λ(K,
(J≠K⟶ copy(car(K)),1⟶diff(car(K),V)))))))),1⟶error)

In this formula C0 and C1 represent the constants 0 and 1 respectively.

Tentatively, we expect that an expression maplist (L,λ(J,$\mathcal{E}$)) will be compiled as follows. As mentioned earlier maplist must be supplied its functional argument in the form of an instruction TXL for a suitable location F. The program in F will begin STO J and from then on will simply be a program for evaluating the expression $\mathcal{E}$. Thus we see that λ has a quite simple effect on the object program.

2.4. The second version of maplist.

The recursive definition of maplist given in the previous section is maplist(L,f) = (L=0 0,1 cons(f(L), maplist (cdr(L),f))). A straightforward compilation from this definition leads to a program that when L is not zero computes the arguments of the cons before computing the value of the cons. Because of this the routine proceeds to the end of the list L before it takes any words from free storage; that is, it goes to the end of the list and works backwards. In order for the routine to be able to find its way back it has to store three words in the PPDL for each element of the list. The time required by maplist comes to about 1.7 milliseconds per list element exclusive of the time required to compute the f(J)'s. It is possible to rewrite maplist so that it works forward in the list. The program then goes as follows:

```
   function(maplist(L,f))
/  L = 0→return(0)
   maplist = cons(f(L),0)
   M = maplist
a1 L = cdr(L)
   cdr(M ) = cons(f(L),0)
   cdr(L) = 0→return(maplist)
   M = cdr(M)
\  go(a1)
```

This program takes about .4 milliseconds per element of L exclusive of the time required to compute the f(J)'s.

One is very reluctant to say that routines like maplist should be described by programs like the above which is certainly much less clear than the previous description. On the other hand it is hard to see how to make the compiler take a description like the recursive definition of maplist and produce a program like the second version. Tentatively, we expect to recode a very few routines like maplist which are much used for high speed and accept the speed penalty in exchange for ease of statement for other routines. The speed disadvantage is probably greater in maplist than in almost any other routine. It should be noted that the reason the second procedure can be made to work is that the value of maplist, namely the location of the constructed list, can

3. Some Examples of SAP Language Programs

We give some examples of the way in which some of the programs described earlier in LISP1 (List Processer) have been coded by hand in SAP. These examples are given so that the reader can confirm his understanding of the meaning of the conventions of LISP and also so that he can consider the problem of designing a compiler which will produce results not too much worse than the hand coded examples. Moreover, it should help users of the system with their hand coding before the compiler becomes available. All the routines described here are debugged.

1. cons(a,d)

The first function we shall describe is cons(a,d) which puts its two arguments in a word taken from the free storage list and returns with the location of the word taken as the value of the function. We are using the same conventions as Fortran I to the effect that the first two arguments of a function are given to it in the AC and the MQ respectively and that the return is by TRA 1,4 with the answer in the AC.

```
CONS    STQ T1              DEC
        ARS 18              ADD
        ADD T1              MAKE WORD
CONS1   SXD T1,4
CONS2   LXD FR E,4
        TXH *+4,4,0         OUT OF FREE STORAGE
        SXD FROUT,4         NO FREE STORAGE
        TSX FROUT+1,4       XX
        LXD FROUT,4
        LDQ 0,4             CONSTRUCT WORD
        STQ FREE
        STO 0,4
        PXD 0,4
        LXD T1,4
        TRA 1,4
T1
```

2. copy(L)

This routine copies a whole list structure into free storage and returns with the location of the new copy. Its program in LISP is

copy(L)=(L≠0→0,car(L)=0→L,1→cons(copy(car(L)),copy(cdr(L))))

This has been translated into the SAP program:

```
COPY    TZE 1,4             L=0
        SXD CS1,4
        PDX 0,4             L
        SXD CT1,4           L
        CLA 0,4             CWR(L)
        PAX 0,4             CAR(L)
        TXH C1,4,0          CAR(L)=0
        CLA CT1             L
        LXD CS1,4
        TRA 1,4
C1      TSX SAVE,4
            CS1+1,,2
        LXD CT1,4           L
        CLA 0,4             CWR(L)
        STO CS2
        ANA DECM            CDR(L)
        TSX COPY,4          COPY(CDR(L))
        LXA CS2,4           CAR(L)
        STO CS2             COPY(CDR(L))
        PXD 0,4
        TSX COPY,4          COPY(CAR(L))
        LDQ CS2
        TSX CONS,4          CONS(COPY(CAR(L)),COPY(CDR(L)))
        TSX UNSAVE,4
            CS1+1,,2
        LXD CS1,4
        TRA 1,4
CS2
CS1
CT1
DECM        ,,-1
```

3. maplist(L,f)

We shall give the SAP versions of both the slow but easily described maplist and the fast but more complicated one which is actually used. It would be desirable that the compiler would compile the description of the slow one into the program of the fast one.

The slow maplist is defined by

maplist(L,f)=(L=0→0,1→cons(f(L),maplist(cdr(L),f)))

Its SAP program is

```
MAPLIS TZE 1,4              L = 0   0
       SXD MS1,4
       TSX SAVE,4           SAVE 3 REGISTERS AND IR4
           MS1+1,,4
       STO L
       STQ F
       TSX F,4              F NOW CONTAINS A TXL TO A SUBROUTINE
       STO F(L)             WE HAVE F(L)
       LXD L,4
       CLA 0,4
       ANA DECM             MASK TO KEEP DECREMENT ONLY, WE HAVE CDR(L)
       LDQ F
       TSX MAPLIS,4         THE RECURSIVE STEP. MAPLIST(CDR(L),F).
       STO T1
       LDQ T1
       CLA F(L)
       TSX CONS,4           CONS(F(L),MAPLIST(CDR(L),F))
       TSX UNSAVE,4
           MS1+1,,4
       LXD MS1,4
       TRA 1,4
DECM       ,,-1
T1
F(L)                        PROTECTED TEMPORARY STORAGE
L                           XX
F                           XX
MS1                         XX
```

The second version of maplist is

function(maplist(L,f))

/ L = 0→return(0)

```
        maplist = cons(f(L),0)
        M = maplist
a1      L = cdr(L)
        cdr(M) = cons(f(L),0)
        cdr(L) = 0→return(maplist)
        M = cdr(M)
       \go(a1)
```

which gives the SAP program

```
MAPLIS TZE 1,4
       SXD MS1,4
       TSX SAVE,4
           MS1+1,,5
       STO MS2              M=L
       STQ MS3              F
       TSX MS3,4            F(M)
       LDQ MZERO
       TSX CONS,4           CONS(F(M),0)
       STO MS4              N
       STO MS5              Q
MLOP1  LXD MS2,4            M
       CLA 0,4
       PDX 0,4              CDR(M)
       TXH MPRG1,4,0        CDR(M)=0
       CLA MS4
       TSX UNSAVE,4
           MS1+1,,5
       LXD MS1,4
       TRA 1,4
 MPRGI SXD MS2,4            M=CDR(M)
       CLA MS2
       TSX MS3,4            F(M)
       LDQ MZERO
       TSX CONS,4           CONS(F(M),0)
       LXD MS5,4            Q
       STD 0,4              CDR(Q)=P
       STO MS5              Q=P
       TRA MLOP1
```

```
MS5                     Q
MS4                     N
MS3                     F
MS2                     M
MS1
MZERO  PZE
```

4. diff(L,V)

As a final example we give the SAP version of the diff program described earlier. It shows how expressions including λ's may be compiled. The reader should note carefully how the quantity J is made available to the lower order function definition (the section called PI in the example)

The LISP form of diff is

```
diff(L,V) = (L=V→C1,car(L) = 0→C0,car(L) = plus→
cons(plus,maplist(cdr(L),λ(J,diff(car(J),V)))),car(L)=times→
cons(plus,maplist(cdr(L),λ(J,cons(times,maplist(cdr(L),λ(K,
(J≠K→copy(car(K)),1→diff(car(K),V)))))))),1→error)
```

This compiles into

```
DIFF   STO L
       STQ T1
       SUB T1
       TNZ D1
       CLA 0§C1D
       TRA 1,4
D1     SXD DS1,4
       LXD L,4              CAR(L)=0→C0
       CLA 0,4
       PAX 0,4
       TXH D2,4,0
       LXD DSL,4
       CLA 0§C0D
       TRA 1,4
D2     TSX 0§SAVE,4
           DS1+1,,3
       STQ V
       LXD L,4
       CLA 0,4
```

```
        PAX 0,4
        STO CWRL
        PXD 0,4
        CAS O$PLUSD
        TRA *+2
        TRA DP
        CAS TIMESD
        TRA *+2
        TRA DM
        PMR LITES,FPR
        TRA O$ERROR
DP      LDQ ETA1
        TRA D3
DM      LDQ NU1
D3      LXD CWRL,4
        PXD 0,4
        TSX MAPLIS,4
        TSX UNSAVE,4
            DS1+1,,3
        STO T1
        LDQ T1
        CLA O$PLUSD
        LXD DS1,4
        TRA O$CONS
ETA1    TXL ETA
ETA     SXD ETS1,4
        PDX 0,4
        CLA 0,4
        PAX 0,4
        PXD 0,4
        LDQ V
        LXD ETS1,4
        TRA DIFF
ETS1
NU1     TXL NU
NU      SXD NS1,4
        TSX O$SAVE,4
            NS1+1,0,2
        STO J
```

```
        LXD CWRL,4
        PXD 0,4    CDR(L)
        LDQ PI1
        TSX MAPLIS,4
        TSX UNSAVE,4
            NS1+1,0,2
        STO T1
        LDQ T1
        CLA TIMESD
        LXD NS1,4
        TRA 0$CONS
PI1     TXL PI
PI      SXD PIS1,4
        PDX 0,4
        SUB J
        TZE PD
        CLA 0,4
        PAX 0,4
        PXD 0,4
        LXD PIS1,4
        TRA 0$COPY
PD      CLA 0,4
        PAX 0,4
        PXD 0,4
        LDQ V
        LXD PIS1,4
        TRA DIFF
V
CWRL
DS1
T1
L
ADDM        -1
J
NS1
PIS1
```

5. Additional Functions and Subroutines

5.1 select $(a;v_1,e_1;\ \ldots\ ;v_n,e_n;e)$ is the same as the conditional expression $(a=v_1\rightarrow e_1,\ldots,a=v_n\rightarrow e_n,1\rightarrow e)$ in effect. However, it may be compiled in open form using CAS instructions and moreover provides a convenient abbreviation for one of the more common cases of conditional expression.

5.2 $\mathrm{list}(i_1,\ \ldots,\ i_n)$

This function has as value a list constructed from free storage containing the items $i_1,\ldots,i_n$ in the address fields of the successive words. We may describe it recursively for ourselves by writing:

$\mathrm{list}(i)=\mathrm{cons}(i,0)$

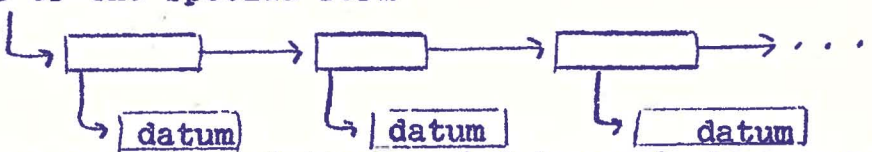
$\mathrm{list}(i_1,\ldots,i_n)=\mathrm{cons}(i_1,\mathrm{list}(i_2,\ldots,i_n))$

However, it should be noted that this is not a definition in the language and cannot easily be made into one. An extension of the language would be required to allow the definition of functions of variable numbers of arguments by recursion on the number of arguments. This is not the same as defining a function of a list of arguments by recursion which is allowed.

Although we cannot define list within the language in terms of cons, there will be no difficulty in defining the program to compile list in the compiler. This is because the compiler will have the arguments in the form of a list of expressions. The resulting program might either be an open subroutine of variable length containing a number of references to cons or, though this is more unlikely, a reference to a closed subroutine which can determine for itself in some way the number of arguments.

5.3 eql(L1,L2)

The value of eql(L1,L2) is 1 or 0 according to whether two lists of the special form



have the same number of items and data words in corresponding places are equal. Such lists will be called one-level lists. They are presently used to store the external names of objects

on their property lists. There will also be other uses. The program for eql(L1,L2) is

eql(L1,L2)=(L1=L2→1,L1=0∨L2=0→0,1→atom(car(L1))=cur(car(L2))∧eql(cdr(L1),cdr(L2)))

5.4. cpl(L)

It is also necessary to be able to copy one-level lists. We have

cpl(L)=(L=0→0,1→cons(consw(cwr(car(L)))),cpl(cdr(L))))

which does it.

5.5. search (L,p,f,u)

This routine searches the list L for an element satisfying the condition p and if it finds one exits with f of that element; if the search is unsuccessful search exits with the value of the expression u. We have

search(L,p,f,u)=(L=0→u,p(L)→f(L),1→search(cdr(L),p,f,u))

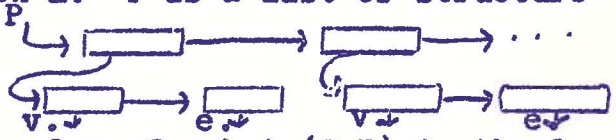
5.6 subst(L,V,M) (substitute L for V in M)

Substitution in expressions can take many forms and we have only begun to explore the possibilities. The present routine is

subst(L,V,M)=(M=0→0,equal(M,V)→copy(L),car(M)=0→M,1→ cons(subst(L,V,car(M)),subst(L,V,cdr(M))))

5.7. sublis(P,E)

sublis(P,E) makes the list of substitution P in the expression E. P is a list of structure



The value of subst (P,E) is the location of a newly formed list in which each occurrence of an object v_i in the expression E is replaced by the location of a copy of the corresponding e_i

sublis(P,E)=maplist(E,λ(J,search(P,λ(K,equal(car(J),car(car(K)))),λ(K, copy(car(cdr(car(K))))),(car(car(J))=0→car(J),1→subst(P,car(J))))))

Actually, the way the program is written the v's may be expressions and not merely objects.

5.8 error

In many program <u>error</u> is listed as the value of an expression under certain conditions which should not occur. What this means is that if the conditions do occur the routine will go to the error routine. Under certain conditions it may be possible to provide for the error routine to do something that will make it possible for the program to continue, but until we understand programming better this will not usually be the case and all that will be possible is to print some sort of diagnostic and terminate the run.

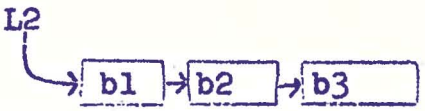
In general, the error will occur in a subroutine deep in the hierarchy of subroutines. What we would like is to know exactly where in the program the error occurred which really means knowing where we were in each of the routines of the hierarchy. Our recursive subroutines will present certain problems in this respect which we shall have to solve, but for the present we will discuss only the non-recursive case. The error routine can trace its way back up the hierarchy provided each routine includes the following information in the calling sequence whenever it uses a TSX.

1. Where it has stored the following information
   1.1 Its name
   1.2 Where it saved ir4
   1.3 In the recursive case where it has used SAVE
   1.4 A list of quantities to be printed out.

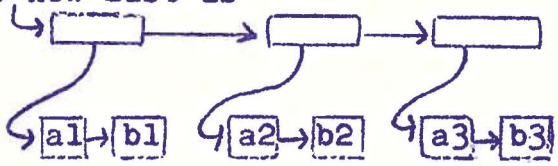
2. Whether it has made any provision for action to be taken in case of an error in this use of the subroutine and if so, what action. It should be noted that while making these provisions for error analysis complicates the task of writing the program, most of these complications can be put on the compiler, and that while the extra information requires space in the program it does not take time unless an error actually occurs.

5.9 pair(L1,L2)

This function constructs a list whose elements are pairs consisting of corresponding elements of the list L1 and L2. If L1 and L2 are not the same length an error interrupt occurs. For example if the original lists are



the new list is

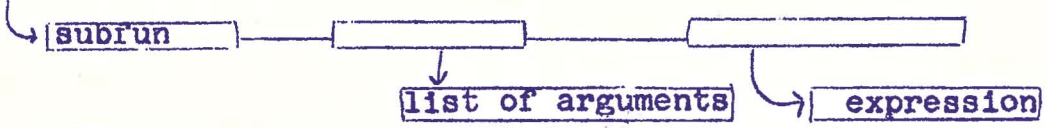


The program for pair is:

pair(L1,L2)=(L1=0∧L2=0→0,L1=0≠L2=0→error,1→cons(cons(copy(car(L1)),cons(copy(car(L2)),0)),pair(cdr(L1),cdr(L2)))

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.



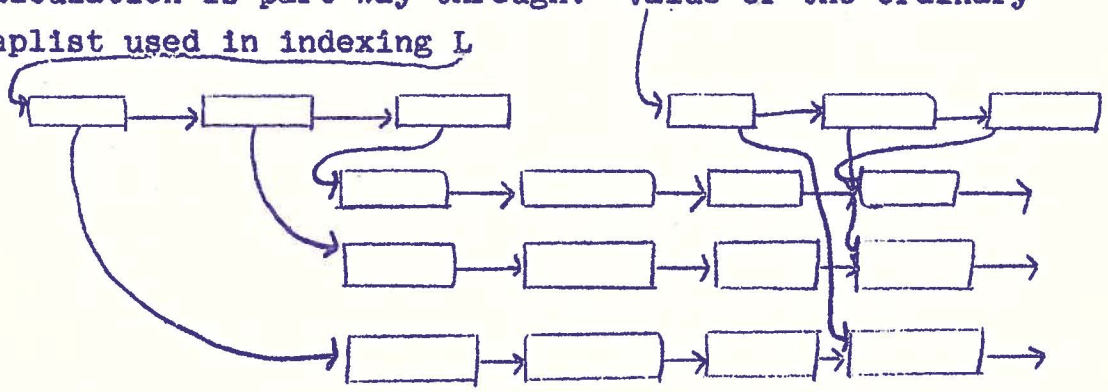
There is a routine apply(L,f) whose value is the result of applying a function to a list of arguments. This routine expects the function f itself to be described by an expression. The kinds of expressions for functions which apply will interpret has not been determined and for the present we shall only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

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 pair 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 maplist.

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. Value of the ordinary maplist used in indexing L



The calculation required to accomplish the above is described in the recursive definition:

dblmaplist(L,f)=search(L,λx(J,car(L)=0≠car(J)=0), λ(J,error),car(L)=0→0,1→cons(f(L),dblmaplist(maplist(L,λ (K,cdr(car(K)),F))))

The inner ordinary maplist in this definition would be a space thief unless the compiler arrange for the list to be erased once it was used.

Still higher order maplists are possible but have not yet been considered. It is worth looking to see whether a maplist of infinite order can be defined in the system.

It is worth while to compare the above with a simple simultaneous maplist of two lists defined by:

maplist2(L1,L2,f)=(L1=0∧L2=0→0,L1=0≠L2≠0→error,
1→cons(f(L1,L2),maplistd(cdr(L1),cdr(L2),f)))

5.12 diff(L,V)⟶

This routine differentiates the expression in the list structure in location L with respect to the object V. In the present version the expressions which may be differentiated are combinations of constants and variables using the connectives plus and times. The routine is

diff(L,V)=(L=V→C1,car(L)=0→C0,car(L)=plus⟶
cons(plus,maplist(cdr(L),λ(J,diff(car(J),V)))),car(L)=times⟶
cons(plus,maplist(cdr(L),λ(J,cons(times,maplist(cdr(L),λ(K,
J≠K→copy(car(K)),1→diff(car(K),V)))))))),1→error)

This version of the differentiation routine which illustrates most of the principles involved in compiling recursively defined functions was the first one to be hand-compiled and demonstrated. A more elaborate differentiation routine can differentiate expressions of fixed numbers of variables by looking up the formulas for their gradients on the property lists of the functions. Such differentiation routine is:

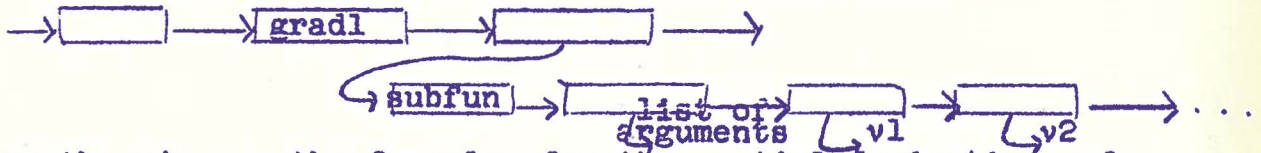
diff(L,V)=(L=V→C1,car(L)=0→C0, car(L)=plus⟶
cons(plus,maplist(cdr(L),λ(J,diff(car(J),V)))),car(L)=times⟶
cons(plus,maplist(cdr(L),λ(J,cons(times,maplist(cdr(L),λ(K,
(J≠K→copy(car(K)),1→diff(car(K),V))))))),1→cons(plus,
maplist2(apply(grad(car(L)),cdr(L)),cdr(L),λ(J,K,list(times,
copy(car(J)),diff(car(K),V))))))

The last term of the conditional expression involves the definitions of several auxiliary things.

1. maplist2, apply, and list are discussed elsewhere in this section. grad is defined by

grad(M)=search(cdr(M),λ(J,car(J)=gradl,λ(J,car(J))), error)

2. The gradient of a function is assumed to be stored on its property list as shown in the following diagram.



Here the v's are the formulas for the partial derivatives of the function. The operation of the table look up differentiation may be briefly described by saying that the routine looks on the property list of car(L) for the word gradl, then substitutes the expression in the list cdr(L) for the arguments in the gradient, and then forms the sum of products of the partial derivatives listed in the gradient list with the derivatives of the arguments of the function in the expression being differentiated.

It should be noted that we could not have used this table look up method to differentiate sums and products because according to our conventions the number of terms in a sum or product is not fixed. A still more elaborate routine would be required to read a formula for differentiating a function of a variable number of terms. The partial derivatives would presumably have some sort of recursive definition.