Artificial Intelligence Project---RLE and MIT Computation Center

Symbol Manipulation Language---Memo 5

by N. Rochester 10. 11. 58

Table of Contents

Introduction

This memorandum presents a function for differentiating an expression with respect to one variable by incrementing the variable and taking the limit as the size of the increment approaches zero. Also presented are all the new tools needed to do it.

The purpose of this study was to study symbolic manipulation and this kind of differentiation seemed to present the right magnitude and character of problem.

The primary problems that arose were both in performing substitutions and in performing simplifications.

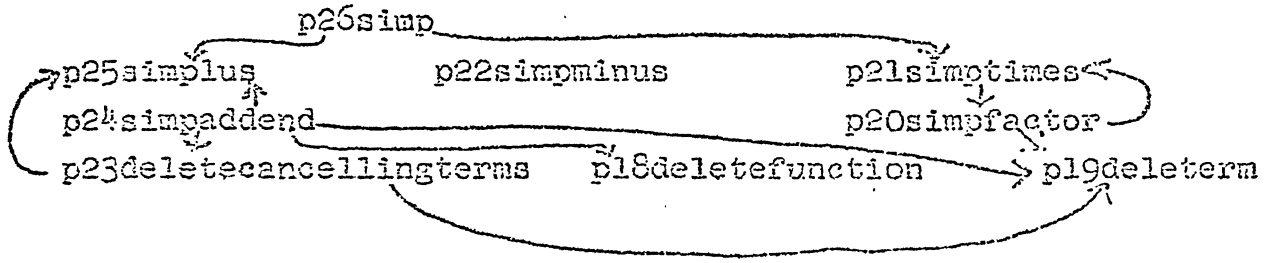
The first major problem in substitution was solved by inventing subfp (p.6). The most difficult substitution was a substitution of a new list structure for particular pairs of factors in products of unspecified length occurring in list structures of unspecified complexity. This was accomplished by locating the products and factoring them (p.16).

Algebraic simplification is self explanatory (p.26).

This kind of differentiation is an imperative statement to which the machine responds by producing a declarative statement (p.4)

The work extended over a long period of time. No obsolete notation remains but some of the latest and most concise modes of expression have not been put into the older programs. In particular, simp (p26), uses a lot of special functions which use other special functions which are defined in terms of functions which are of more general value. This would be more concise, although, perhaps less intelligible, if the technique of recursive definition of functional abstraction (p 15) were used. Also the format rules (p.12) make later definitions easier to read. Some details were not followed on earlier definitions. The date of each definition is given.

With minor exceptions noted in the text, the order of the presentation of instructions is such that each is defined in terms that have already been defined. This means that some of the instructions will not seem very useful at the time they are defined if the reader reads the pages in order. An arrow from one functional abstraction to a second indicates that the second is used in the definition of the first:



How to keep the results of a proof.

Suppose we have an n-step proof generated by

$$F_n(F_{n-1}(---F_2(F_1(L\qquad\qquad))---))$$

and we wish to save each line of the proof. They can be put on a list as follows

$\lambda(J,\mathrm{cons}(F_1(\mathrm{car}(J)),J))(\mathrm{cons}(L,0))$
$\lambda(J,\mathrm{cons}(F_2(\mathrm{car}(J)),J))(\mathrm{above})$

- - - - - - - - - - - - - - - - - - - -

$\lambda(J,\mathrm{cons}(F_{n-1}(\mathrm{car}(J)),J))(\mathrm{above})$
$\lambda(J,\mathrm{cons}(F_n\ (\mathrm{car}(J)),J))(\mathrm{above})$

This list is too simple to solve all problems in organizing the steps of a proof.

subst(L,M,N)

Means "substitute L for M in N".

In more detail it means copy the list structure at N replacing all instances of M with L. L and M can be list structures and L can be a function of M and N. The value of subst is the address at which the new list structure is stored.

subst(L,M,N)=

(N=0→0,equal(N,M)→copy(L),car(N)=0→N,

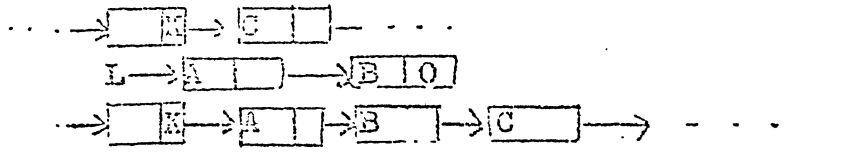
1→cons(subst(L,M,car(N)),subst(L,M,cdr(N)))

subfp(f,P,N)
substitutes f(J) for each address J
such that P(J)=1 that it finds in L.
subfp(f,P,N)=
(N=0→0,P(N)→f(N),car(N)=0→N,
1→cons(subfp(f,P,car(N)),subfp(f,P,cdr(N))))

1958OCT 28

insert(L,K)

insert list L at K in another list



insert(L,K)=cdr(L)=0→cons(copy(car(L)),K),

1→cons(copy(car(L)),insert(cdr(L),K))

canceladdend (a,J)

copies of the list structure at J eliminating the addend a from J. If J is a sum it eliminates the first instance of a. If the major connective of J is a predicate it canceladdends the argument (s). The value is the address of the new list.

```
canceladdend (a,J)
=(equal(car(J),a)→copy(cdr(J)),
  1→cons(copy(car(J)),canceladdend(a,cdr(J))))
```

cancelfactor(a,J)

cancelfactor copies the list structures at J, eliminating the factor a from J. If J is a products it removes the first instance of a. If J is a sum it removes the first instance of a as a factor in each of the terms of the sum. If J is predicate it cancelfactors the argument (s). The value is the address of the new list structure.

It will be able to factor A and A+B from A(A+B) but will not be able to factor B from A(BC) but it would be able to factor out (BC).

```
cancelfactor (a,J)=
(J=0→0,equal(a,car(J))→cons(cl,copy(cdr(J))),
pred(car(J))∨car(J)=plus→
      cons(car(J),maplist(cdr(J),λ(K,cancelfactor(a,K)))),
car(J)=times→cons(times, cancelfactor(a,cdr(J)),
1→cons(copy(car(J)),cancelfactor(cdr(J))))
```

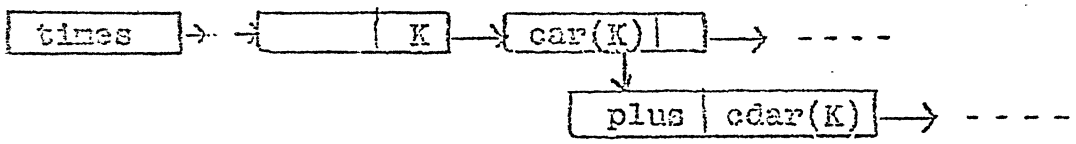
```
Multal(J)=
(J=0→0,
 car(J)=0→J,
 car(J)=times→λ(F(K),
    (K=0→cons(times,multal(cdr(J))),
     caar(K)=plus→multal(cons(plus,maplist
         (cdar(K),λ(L,cons(times,insert
         (cancelfactor(car(K),J),L))))),
     1→F(cdr(K))),
 1→cons(multal(car(J)),multal(cdr(J))))
```

Multal converts products of sums to sums of products.

x.(a+b+c).y=x.y.a+x.y.b+x.y.c

The expression must be cleared of confusion by simp if multal is to find all cases. Furthermore, multal creates confusion that can be removed by simp.

At the time when F discovers a sum under a product the list is like this:



L=M

= is a predicate that has as arguments two similar arguments (i.e. both are 1-bit, 15-bit, or 36-bit quantities). The value is 1 if L and M have the same value and 0 otherwise. There is no program for = in list processing language since this is part of the logical foundation and hence is describable only in metalanguage (English or SAP).

format rules:

1.

All of the separate conditions of a conditional begin in the same column and this is a different column from the immediately adjacent expressions:

eg. (A→B,B→(C→D,E→F),G→H) is written

```
A→B,
B→(C→D,
    E→F),
G→H)
```

2.

"above" and "below" are used freely so that

F(G(Z)) = G(Z) = F(below)
F(above) G(Z)

Replace(L,M)

Replace L with M has as arguments two similar arguments (i.e. both are either 1-bit, 15-bit, or 36-bit quantities) and the value of L is replaced by the value of M. This function has as its value the address of the word containing L. This function is used instead of = in the sense of replacement.

There is no program for replace in list processing language since it is part of the logical foundation and is therefore describable only in a metalanguage (e.g. English or SAP).

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.

var and recursive definition of functional abstractions

```
var(J)=
(car(J)≠0→0,
 1→λ(F(K),(K=0→0,
           car(K)= var→1,
           1→F(cdr(K))))(cdr(J))).
```

The predicate var(J) has the value 1 iff J is a variable. I have assumed that a variable has ---→[var]→--- somewhere in its property list. "var" is the address of the property list of the predicate "var".

This definition illustrates the use of a recursive definition of a functional abstraction. F is the name of the functional abstraction
λ(F(K),(K=0→0,car(K)=var→1,1→F(cdr(K)))).
However it would not be correct to write
F=λ(F(K),(K=0→0,car(K)=var→1,1→F(cdr(K))))
and expect it to have the value 1 as a result of the definition under the λ since F is a bound variable.

"pred" "const" and other predicates that indicate that an object is a member of a set are similar.

Factorpairs (P,Q,M)
factors from each product in a list structure M all pairs of factors such that P of the first and Q of the second. It does not look inside products to find further products to work on.

```
Factorpairs (P,Q,M)=
(M=0→0,
car(M)=0→M,
car(M)=times→cons(times,λ(lofofi(J,K),
   (K=0→cons(cons(times,copy(J)),0),
   P(K)→λ(lofose(L),
     (L=0→cons(cons(times,copy(J)),0),
     Q(L)→cons(list(times,copy(car(K)),copy(car(L))),
       λ(J,lofofi(N,N))(deleterm(L,deleterm(K,J)))),
     1→lofose(J,K,cdr(L))))(J,K,J),
   1→lofofi(J,cdr(K))))(cdr(M),cdr(M)),
1→cons(factorpairs(P,Q,car(M)),factorpairs(P,Q,cdr(M)))
```

If this definition seems hard to read, then first read deleterm (p.19) and var(p.15). Also note that

λ(N,lofofi(N,N))(Z)=lofofi(Z,Z).

This is just a trick to get two arguments from one. Note also that lofofi is mnemonic for "look for first" and "lofose" is mnemonic for "look for second".

Paradoxes etc.

The language is powerful enough so that it is capable of expressing paradoxes and infinite processes. The programmer is responsible for keeping such problems under control. Examples are:

1. increase=λ(F(J),subst((plus,X,X),X,F(J)))
2. F(below)
   G(above)
3. P=λ(J, P(J)=0)

It is possible to manipulate some of these more complicated statements by processes that can be described in list processing language even though the machine can no more evaluate them by brute force than a person can evaluate

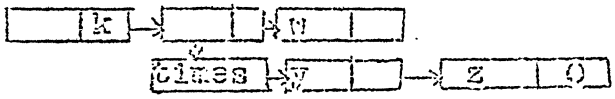
$$\sum_{i=0}^{\infty} 2^{-i}$$

by adding up all the terms.

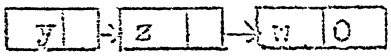
```
deletefunction(K,L)=
(cdr(L)=0->cons(copy(car(L)),copy(cdr(K))),
 1->cons(copy(car(L)),deletefunction(K,cdr(L))))
```

Deletefunction(K,L) deletes the function car(car(K)) and arguments up to L and replaces the function with the remaining arguments.

For example, if there is the following structure,



deletefunction(K,cdr(car(K))) has as its value the address of the following structure that it produces:



```
Deleterm(J,K)=
(K=0→0,
 K=J→copy(cdr(K)),
 car(K)=0→K,
 1→cons(deleterm(J,car(K)),deleterm(J,cdr(K))))
```

Deleterm (J,K) has as its value the address of a new list structure that it creates by copying K with the exception of term J which it deletes.

1958 NOV 5

```
simpfactor(K,J)=

(K=0→J,car(K)=C0→C0,
 car(K)=C1→simptimes(deleterm(K,J)),
 car(car(K))=minus→cons(minus,
     simptimes(subst(cdr(car(K)),car(K),J)),
 car(car(K))=times→simptimes(subst(deletefunction(K,cdr
     (car(K))),K,J)),
 1→simpfactor(cdr(K),J))
```

This goes through the factors of a product and replaces the product with zero if it finds a zero product, deletes a factor of unity, brings minus outside the product, and removes unnecessary times's.

```
simptimes(J)=

(J=0→0,
 car(J)=0→J,
 car(J)≠times→cons(simptimes(car(J)),simptimes(cdr(J))),
 1→(cdr(J)=0→C1,
     cdr(cdr(J))=0→simptimes(car(cdr(J))),
     1→simpfactor(cdr(J),J)))
```

This function has, as its value, the address of a list which has the same meaning as the list named by the argument but the new list contains the following syntactic simplifications:

1. A times with no arguments is replaced with C1
2. A times with one argument is replaced by its argument
3. A times which occurs as a factor is replaced by its factors
4. A one occurring as a factor is deleted
5. If a minus occurs as a factor it is replaced by its argument and the minus is moved outside the times.
6. A times with a C0 as an argument is replaced by C0.

1958 NOV 5

```
simpminus(J)=

(J=0→0,
 car(J)=minus→(
    car(cdr(J))=minus→simpminus(cdr(cdr(J))),
    car(cdr(J))=plus→simpminus(cons(plus,
       maplist(cdr(cdr(J)),λ(K,cons(minus,copy(car(K)))))))),
    1→cons(minus,simpminus(cdr(J)))),
 car(J)=0→J,
 1→cons(simpminus(car(J)),simpminus(cdr(J))))
```

Simpminus (J) changes minus minus a term to the term and changes minus a sum to the sum of minus the addends.

```
deletecancellingterms(J,K,L)=

(K=0→cons(plus,simplus(cdr(J))),
 L=0→deletecancellingterms(J,cdr(K),cdr(cdr(K))),
 (equal(car(K),car     (cdr(car(L))))∧eqr(car(L))=minus)∨
    (equal(car(cdr(car(K))),car(L))∧eqr(car(K))=minus)→
    simplus(deleteterm(K,deleteterm(L,J))),
 1→deletecancellingterms(J,K,cdr(L))
```

Deletecancellingterms is a detail in simplus that deletes two cancelling addends if it can find any. If it does find them it deletes and returns control to simplus. If it does not find them, the last step of simplifying this sum has been completed so it constructs a word of the simplified list structure and then returns control to simplus.

```
simpaddend(J,K)=
 (K=0→deletecancellingterm (J,cdr(J)),cdr(cdr(J)),
  car(K) ((0→simplus(deleterm(K,J)),
  car(car(K))=plus→simplus(subst(
      deletefunction(K,cdr(car(K))),J)),
  1→simpaddend(J,cdr(K)))
```

Simpaddend is a detail in simplus that deletes arg addends and replaces a sum occurring as an addend with its arguments. If it does any of these things it returns control to simplus. If it finds nothing to do it passes control on to deletecancellingterm.

```
Simplus(J)=
(J=0→0,
 car(J)=0→J,
 car(J)=plus→(
      cdr(J)=0→0,
      cdr(cdr(J))=0→simplus(cdr(J)),
      1→simpaddend(J,cdr(J))),
 1→cons(simplus(car(J)),simplus(cdr(J))))
```

Simplus causes a plus with no addends to be replaced by zero, a plus with one addend to be replaced by the addend, a plus which occurs as an addend to be replaced by its arguments, a zero occurring as an addend to be deleted, and two cancelling addends to be deleted.

simp(J)=simplus(simpminus(simptimes(J)))
simplifies an algebraic expression by doing the following things:

times 1.1) A times with no arguments is replaced by C1.
1.2) A times with one argument is replaced by the argument.
1.3) A times which occurs as a factor is replaced by its factors.
1.4) A one occurring as a factor is deleted.
1.5) If a minus occurs as a factor it is replaced by its argument and the product is then replaced by minus the product.
1.6) A times with a zero argument is replaced by the constant zero.

minus 2.1) A minus of a minus of a term is replaced by the term.
2.2) A minus of a plus of terms is replaced by the plus of the minus of the terms.

plus 3.1) A plus with no addends is replaced by zero.
3.2) A plus with one addend is replaced by the addend.
3.3) A plus which occurs as an addend is replaced by its arguments.
3.4) A zero occurring as an addend is deleted.
3.5) If a sum includes a term and minus that term as addends they are both deleted.

The definition of simp is in terms of functions which are recursively defined but are not very interesting in themselves. These are:

simptimes , simpfactor,
simpminus,
simplus, deleteaddend, deletecancellingterms

These definitions use two more recursively defined functions that may have other uses:
deleterm, deletefunction.

This is a guide to the differentiation program on p.28. It shows what the program would do to a representative expression. The guide is written in conventional algebraic form rather than in list form.

$y=uv$

1) $y+\Delta y=(u+\Delta u)(v+\Delta v)$

2) $(y+\Delta y)-y=(u+\Delta u)(v+\Delta v)-uv$

3) $\Delta y=u\Delta v+v\Delta u+\Delta u\Delta v$

4) $((\Delta y).(\frac{1}{\Delta x}))(\Delta x)=u((\Delta v).(\frac{1}{\Delta x}))\Delta x+v((\Delta u).(\frac{1}{\Delta x}))(\Delta x)$
$+((\Delta u).(\frac{1}{\Delta x}))(\Delta x)((\Delta u).(\frac{1}{\Delta x}))(\Delta x)$

5) $(\Delta y).(\frac{1}{\Delta x})=u.(\Delta v).(\frac{1}{\Delta x})+v.(\Delta u).(\frac{1}{\Delta x})+(\Delta u).(\frac{1}{\Delta x}).(\Delta v).(\frac{1}{\Delta x}).(\Delta x)$

6) $\left(\frac{\Delta y}{\Delta x}\right)=u.\left(\frac{\Delta u}{\Delta x}\right)+\left(\frac{\Delta u}{\Delta x}\right)+\left(\frac{\Delta u}{\Delta x}\right).\left(\frac{\Delta v}{\Delta x}\right).(\Delta x)$

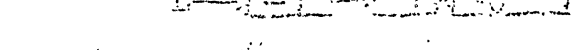
7) $\lim\left(\frac{\Delta y}{\Delta x}\right)=u.\lim\left(\frac{\Delta v}{\Delta x}\right)+v.\lim\left(\frac{\Delta u}{\Delta x}\right)$
$+\lim\left(\frac{\Delta u}{\Delta x}\right).\lim\left(\frac{\Delta v}{\Delta x}\right).\lim(\Delta x)$

8) $D(y)=u.D(v)+v.D(u)+D(u).D(v).0$

9) $D(y)=uD(v)+v.D(u)$

11:58 Nov. 18

Differentiation in list processing language of an algebraic expression, E, constructed of =,), (, plus, times, minus, constants, and variables. The expression is differentiated with respect to x, but by the use of λ, x could be replaced by any other variable. It is assumed that the expression to be differentiated is an equality at L.



1) subfp(λ(J,(plus,J,(Δ,J))),var,L)
2) λ(J,K,list(=,(plus,copy(cadr(J)),(minus,copy(cadr(K)))),
(plus,copy(caddr(J)),(minus,copy(cacdr(K))))))(above,L)
3) simp(multal(simp(above)))
4) subfp(λ(J,(times,(Δ,cadr(J)),(exp,(Δ,x),(minus,c1)),(Δ,x))),
λ(J,car(J)=Δ∧cadr(J)≠x),above)
5) cancelfactor((Δ,x), simp(above))
6) factorpulve(λ(L,caar(L)=Δ∧cadar(L)=0),
λ(L,equal(car(L),(exp,(Δ,x),(minus,c1)))),above)
7) subfp(λ(K,cons(lim,copy(K))),λ(J,equal(J,(Δ,x))),
equal(J,(times,(Δ,caddar(J)),(exp,(Δ,x),(minus,c1))))),above)
8) subfp(λ(J,(caddr(J)=n∧n≠0,1=(diff,cadaddr(J)))),
λ(J,car(J)=lim),above)
9) simp(above)

The notation (exp,(Δ,x)),(minus,c1)) for $1/\Delta x$ is quite arbitrary and is used as a name. Later when division is introduced to the system this detail may be changed. Notice the guide on p.27.