UNIVERSITY MATHEMATICAL LABORATORY
CAMBRIDGE

T e c h n i c a l   m e m o r a n d u m

No.63/1

An experiment with a self-compiling compiler for a simple list-processing language.

by

M.V. Wilkes

February 1963.

Herbert Schorr

Technical memorandum

No.63/1

An experiment with a self-compiling compiler for a simple list-processing language.

by

M.V. Wilkes

Herbert Sch

February 1963.

AN EXPERIMENT WITH A SELF-COMPILING COMPILER FOR A SIMPLE LIST-PROCESSING LANGUAGE.

This is a report on what was essentially a student's exercise in list processing and machine-independent compiling. I was anxious to obtain some experience in the manipulation of lists and had found the machine code very ill-adapted for that purpose. My first object was to establish a language in which simple list processing was possible with reasonable convenience. At the same time, I wanted to see how soon it was possible to get "off the ground" a series of compilers, each being written in the language of the previous one, and to obtain, by actual experience, some understanding of the way in which boot strapping and other techniques can be applied to generate higher level compilers.

I decided initially that the series of compilers should cater for symbol manipulation only, and that the different problems that arise in dealing with arithmetic expressions should be left on one side. I also decided that the target language should be that accepted by the EDSAC 2 assembly routine, the particular advantage of doing this being that a system of labels was thereby made available. The object program was to be punched out on paper tape.

The language is not - so far at any rate - a fully developed list processing language as is, for example, IPL5. Rather it is a language - or the nucleus of a language - in which the basic subroutines which provide the programming facilities in IPL5 could be written. The peculiarities of EDSAC 2, particularly as regards the way in which the main store is addressed, were obviously very much in mind when the essential foundations of the system were laid.* Without transferring the system to other machines, it is hard to say whether this has made it seriously machine-dependent; I am inclined to believe that it has not, but this may be wishful thinking.

The system of compilation may briefly be described as follows. Successive lines of the program - that is, statements, - are scanned and compared successively with a series of standard forms, until a match is

*EDSAC 2 is a single address machine, which originally had a working store of 1024 registers. This was known as the free store and there was in addition a fixed store in which certain subroutines were permanently wired. A 16,000-word store was later added, and this is known as the main store. Since there were not nearly enough address digits for registers in the main store to be addressed individually, a system of indirect addressing was resorted to, the lower numbered registers in the main store being used to hold the indirect addresses. There is a special modifier register associated with the main store. Punched paper tape is used for input and output.

found. Variables are indicated in the standard form by a star, and when one of these is encountered during the scanning process, the corresponding symbol in the statement is placed in a stack. This stack plays a central part in the compilation process. Each standard form has associated with it a pro-forma list of machine orders, or machine code translation, which represents its translation into machine orders. The translation of a statement is performed by a machine code routine known as COMPILE which copies out the machine code translation with appropriate substitution of variable addresses taken from the stack.

Since the statements that the compiler can handle are defined by means of the standard forms and the machine code translation of machine orders, there is no particular need to keep their number down to the absolute minimum. The disadvantage of a very short list of possible statements is that it leads to lighly inefficient compiling, and, although efficiency is not a prime consideration in the first compiler of a series, it can never be lost sight of completely.

Machine code routines

In addition to COMPILE just mentioned, several special machine code routines are used. Some of these are so machine-dependent that they can almost be regarded as part of the machine. Where the interface occurs between what is regarded as the compiler proper, and written in symbolic language, and the part written in machine code, is a matter for deliberation in any particular case. For the purposes of the present work, the following basic machine code routines were prepared.

INPUT (76 half registers)

This routine reads single characters from the input tape and translates from the EDSAC 2 5-digit input code to a 6-digit code in which figures from 0 to 9 are represented by their binary equivalents. Blank tape, line feed symbols, and all spaces after the first of a sequence, are ignored. A comma is made to have the same representation in the 6-digit code as carriage return. This gives a certain freedom in layout by permitting, for example, the writing of two statements on the same line. (The printers in use in the Laboratory have no comma, and print a suffix 2 instead.)

FORM FREE LIST (11 half registers)

This routine, which is activated when the compiler is first read into the store, connects together all registers of the main store, from 512 onwards, to form a list which is given the name F.

COMPILE (101 half registers)

This routine has already been mentioned. It punches on to the output tape the sequence of orders copied from one of the machine code translations in the main store. Each order in the machine code translation occupies the whole word in the main store, the first half-word giving the "stem" of the order and the second half-word indicating how the address is to be constructed using information from the stack.

LABEL (11 half registers)

This short routine is necessary since the assembly routine does not provide labelling facilities for orders placed in the main store.

With some hindsight it is possible to see that some of these machine code routines were perhaps more elaborate than was strictly necessary; INPUT in particular could have been simplified by leaving some of the editing to be done by the compiler proper.

The Language

As an introduction to the language, I give below a description of certain selected statements with examples of their use. This is followed by a complete list of permitted statements.

Each register of the available part of the main store is divided into two sections known, as in LISP, as CAR and CDR. Normally, both CAR and CDR contain a main store address, but, on occasion, they may contain pure symbols. Lists and list structures are constructed by placing in the CAR and/or CDR of one register addresses which point to other registers whose CAR and CDR in turn contain either similarly pointing addresses or symbols.

The letters A to Z are used for the names of lists. Each of these letters corresponds to one of the early registers in the main store, the CDR of which points to the first element of the corresponding list. The CAR of the register is unused. In a given program, of course, some of the letters A to Z and the corresponding registers, may not be used. Use can also be made of the symbols on the figure shift of the teleprinter.

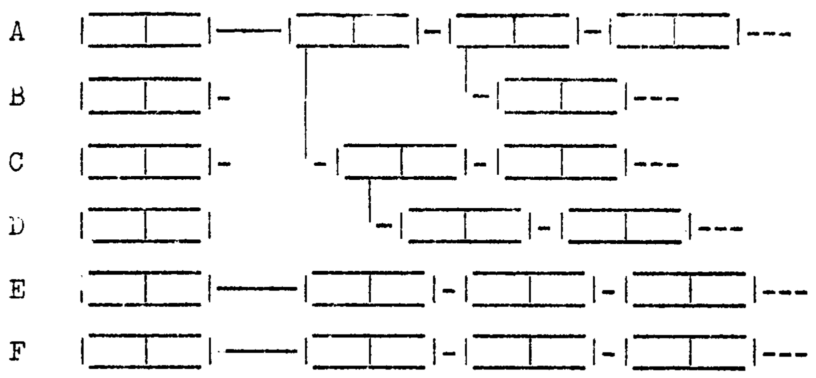


Fig. 1

In fig.1 the rectangles on the extreme left represent the registers just referred to; A is the name of a list structure, and E and F are the names of simple lists. F is initially the free list to which all available storage registers are attached. The statement B = A associates the name B with the list A, so that B becomes an alternative name for the list. Its programming effect is to copy the CDR of the register corresponding to A into the CDR of the register corresponding to B. Similarly, C = CDR A associates the name C with the list whose first member is the second member of list A. Fig. 2 shows the configuration of the relevant part of the store after the execution of the statements B = A, C = CDR A, D = CAR A.

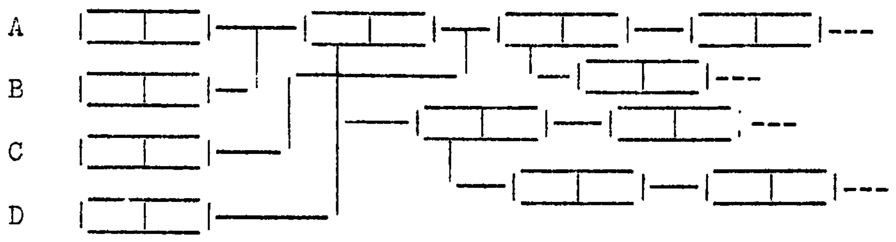


Fig. 2

Normally, in diagrams such as fig.1 and fig.2, it is not necessary to show the registers on the extreme left since these are the registers permanently assigned to the letters A to Z, and they form part of the machinery of the system and, as such, do not concern the user of lists.

Fig. 3 contains the same information as fig.2, and it will be seen that the names A, B, C, have been placed in the appropriate places on the list structure.

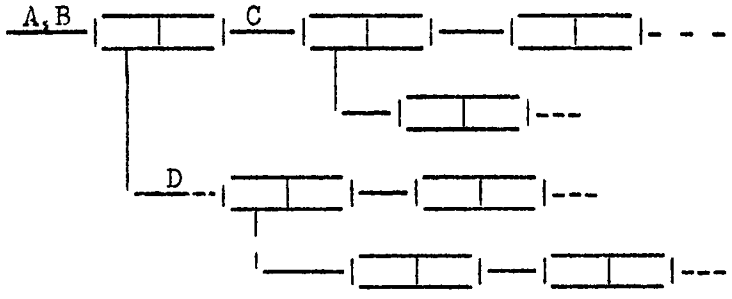


Fig. 3

A further example is provided by the following sequence of statements

```
D = F
F = CDR F
CDR D = B
B = D
```

which have the effect of taking a register from the beginning of the free list and attaching it to the beginning of the list B.

The stack has two pointers, 1 and 2, which initially both point to the bottom of the stack. The instruction CAR A TO STACK causes CAR A to be placed at the top of the stack and pointer 2 to be moved up. The statement RESET STACK sets pointer 1 equal to pointer 2. The statements LEVEL UP and LEVEL DOWN were provided to enable recursion to be handled. LEVEL UP causes current values of both pointers to be stored at the top of the stack and then sets both pointers to point to the next available register in the stack. LEVEL DOWN reinstates the old values of both pointers. As it turned out, these statements were not needed in the early part of the exercise since it was found more convenient to incorporate the level changing sequences in other statements.

The statement CAR E = INPUT causes a symbol to be read from the input tape and copied into CAR E. This statement is used, for example, in the section of the compiler that reads the program to be compiled into the machine.

Statements are normally terminated by a carriage return symbol, and appear one on a line. Alternatively, if, for layout reasons, it is desired to have more than one statement on the same line, a comma may be used for termination. This symbol has, in every way, the same effect as a carriage return symbol.

Two symbols (normally figures) are used for labels; these are treated quite independently, and there is no notion in the system of numerical ordering. A label is terminated in the same way as a statement and normally appears on a line by itself.

An example of an unconditional jump statement is TO 37, and of a conditional jump statement TO 37 IF CAR A = CAR B; the latter causes a jump to 37 if CAR A and CAR B contain identical quantities, whether these are intended to be interpreted as addresses or symbols. It is sometimes necessary to be able to test for the presence of a particular symbol regardless of any meaning that symbol may have as the name of a list. An example of a statement that may be used for such a purpose is TO 50 IF CAR A = :X. The : preceding the X is intended to indicate that the symbol following stands for itself.

The statement TO 50 AND BACK is used to enter a closed subroutine and provides for the storage of the link. The subroutine itself ends with the instruction RETURN, which sends control back to the main program. These statements do not allow a subroutine to be used recursively. A statement of the type TO 50 AND BACK can be used to call in a machine code subroutine.

A full list of the statements that the first compiler of the series capable of compiling itself, TR 0, would accept is given below; this list is in fact the list of standard forms referred to earlier and is set out exactly as it was punched on the input tape. The stars indicate places where symbols can occur. The first item X is a dummy statement intended to be punched for synchronizing purposes at the beginning of a program tape immediately after the blank tape "leader". It is hoped that the other statements will be self-explanatory in the light of the explanations given above.

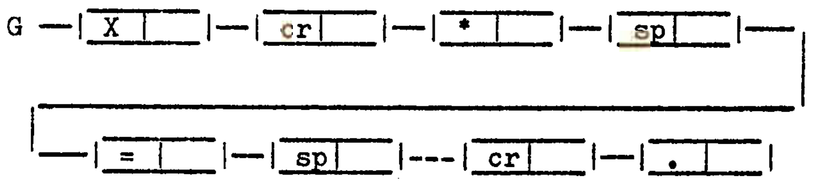
```
   X
   * = CDR *
   TO **
   TO ** IF CAR * = :*
   * = *
   **
→  RESET STACK
→  WAIT
   CAR * = INPUT
   CAR * = *
—  CAR * TO STACK
   * = CAR *
   TO ** AND BACK       ([illegible])
   TO ** IF CDR * = :*
   TO ** IF CAR * = CAR *
—  LEVEL UP
   LEVEL DOWN
—  CDR * TO STACK
   START AT **          print CAR*
   RETURN
   .
```

A simple compiler was written in the above language and translated manually into machine code. The translation was effected by the use of slips of paper on each of which was printed, on the right-hand side, one of the possible symbolic statements, and, on the left-hand side, the corresponding machine code translation, blanks being left on both sides in which variable information could be written. A sequence of these slips, with the variable information carefully filled in, was pasted up on a sheet of paper to constitute the program. It is admitted that this device was to some extent a toy; but, by partially mechanizing the process of compiling, it helped to systematize the work and to minimize errors. It had the advantage of keeping the programmer strictly to the operations covered by the list of available symbolic statements.

The machine code (left-hand) version of the compiler was now punched, the machine code routines added, and the whole program checked out. The result was a compiler capable of reading the symbolic (right-hand) version of itself and compiling a program identical with that already in the machine. The symbolic version of the compiler, TR 0, is given below with sufficient annotation to enable its action to be followed.

```
X
30
G = F
26
CAR F = INPUT
TO 25 IF CAR F = :.
F = CDR F
TO 26
```

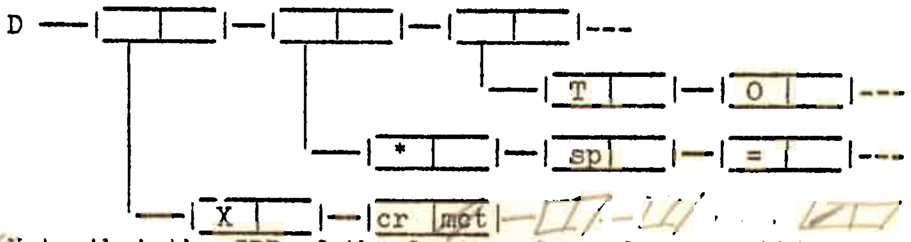
This section reads the standard forms from the input tape and stores them in the form of a list G, each character occupying the CAR of one storage register; thus



cr indicates carriage return and sp indicates space. The end of the list of standard forms is indicated by a dot following a carriage return.

```
25
F = CDR F
D = F
23
TO 24 IF CAR G = :.
CAR F = G
F = CDR F
21
TO 22 IF CAR G = :/
G = CDR G
TO 21
22
E = CDR G
TO 79 AND BACK
G = E
TO 23
24
WAIT
```

The list is now converted to a list structure as follows:



*Note that the CDR of the last member of each sublist contains the address of the beginning of the corresponding machine-code translation (mct). This is placed there by the LABEL routine called in by the statement TO 79 AND BACK, 79 being the label assigned to that routine. The LABEL routine was tailor-made for this use and puts the address directly into CDR G.

```
93
P = F
29
S = Q
Q = F
F = CDR F
CAR Q = INPUT
TO 28 IF CAR Q = :*
TO 29
28
TO 27 IF CAR S = :*
TO 29
27
WAIT
```

The program to be compiled is now read into the store and formed into a list P. The end of the program is indicated by two consecutive stars.

```
06
Q = F
08
E = D
09
RESET STACK
Z = CAR E
R = Q
05
TO 13 IF CAR Z = :*
TO 12 IF CAR R = CAR Z
07
E = CDR E
TO 09
12
R = CDR R
Z = CDR Z
TO 14 IF CAR Z = :,
TO 05
13
CAR R TO STACK
TO 12
14
TO 10 IF CAR R = :,
TO 07
10
CDR Z TO STACK
TO 77 AND BACK
Q = CDR R
TO 11 IF CAR Q = :*
TO 08
11
WAIT
**
```

The first line of the program is scanned and compared character by character with the first standard form, that is, with the first sublist of D. Whenever a star is encountered in the standard form, the corresponding symbol in the program is placed in the stack. If complete agreement (including the terminating carriage returns) is not found the stack is reset and the next sublist tried. When agreement is found, the address of the appropriate machine code translation is taken from the end of the sublist containing the standard form and placed in the stack. The machine code subroutine COMPILE (label 77) is then entered to effect the compilation. The process is repeated for successive lines of the program.

Although the compiler just described was capable of compiling itself, and I came very near to making it do this, I did not actually do so, since it was clear at that stage that certain modifications to the system were desirable; I therefore used the program to compile a new version - TR 0.1 - of the compiler in which these modifications were incorporated. Thus the first lesson was learned that when a bootstrapping operation is in progress, each compiler generates its successor, and only when stability is reached (if ever) does a compiler actually compile itself.

In passing to TR 0.1 some modifications and additions were made to the standard forms and corresponding machine code translations. For example, the GO TO ** AND BACK and RETURN statements were modified to allow for recursion, and a new statement SUBROUTINE ** was introduced for calling in machine code subroutines.

At the same time the following new statements were introduced:

```
CAR * = CAR *
CDR * = CDR *
TO ** IF CAR * ǂ :*
TO ** IF CDR * ǂ :*
CDR * = *
```

TR 0.1 was originally written in the language of TR 0 and compiled by TR 0. It was later re-written in its own language and used to compile itself. This latter version is given in full below with annotations to indicate how it differs from TR 0. The section which reads in the standard forms and constructs a list structure from them was modified so as to make use of the recursive facilities now available; this carried no advantage in itself but enabled the recursive facilities to be tested. Additional features of a minor character were introduced into later sections; these were the ignoring of spaces and extra carriage returns at the beginning of lines, and arranging that the machine should come to a WAIT statement instead of going into a loop if presented with a statement that could not be matched from the table of standard forms.

TR 0.1

```
RESET STACK
E = F
D = F
TO 26 AND BACK
CAR E = CAR L
F = L
WAIT
TO 93
```

After preliminary setting, the subroutine 26 is called in to read the standard forms from the input tape and to form the list structure D (see TR 0). The final dot is left in CAR L and is transferred to the end of the list D

```
26
F = CDR F
L = F
23
CAR F = INPUT
TO 25 IF CAR F = :,
F = CDR F
TO 23
25
TO 24 IF CAR L = :.
G = F
F = CDR F
SUBROUTINE 79
CDR E = F
CAR E = L
E = CDR E
TO 26 AND BACK
24
RETURN
```

This subroutine, which uses itself recursively, sets as follows:

> Read characters from input tape until a cr is encountered and form them into a list. Jump out if last character was a dot following a cr; otherwise attach list to E so that it forms a new sublist of D and re-enter the subroutine

```
93                        Reads the program to be compiled and constructs the
P = F                     list P.   Identical with the corresponding section
29                        in TR 0
S = Q
Q = F
F = CDR F
CAR Q = INPUT
TO 28 IF CAR Q = :*
TO 29
28
TO 27 IF CAR S = :*
TO 29
27
WAIT

06                        This section is identical with TR 0 except where
Q = P                     noted below
08
TO 15 IF CAR Q = :,       Ignore spaces and extra cr's in front of statement
TO 15 IF CAR Q = :,
TO 16
15
Q = CDR Q
TO 08
16
E = D
09
RESET STACK

Z = CAR E
R = Q
05
TO 13 IF CAR Z = :*
TO 12 IF CAR R = CAR Z
07
E = CDR E
TO 09 IF CAR E ≠ :.       Wait if all sublists of D have been tried without
WAIT                      finding agreement with the line of the program
12                        being scanned
R = CDR R
Z = CDR Z
TO 14 IF CAR Z = :,
TO 05
13
CAR R TO STACK
TO 12
14
TO 19 IF CAR R = :
TO 07
19
CDR Z TO STACK
SUBROUTINE 77
Q = CDR R
TO 11 IF CAR Q = :*
TO 08
11
WAIT
**
```

Present status of the language

It is not necessary to describe in detail the subsequent development of the language. Several distinct stages were passed through, new statements being introduced. and old ones omitted at each stage.

The most important innovation was the abandonment of the conditional jump statements in favour of conditional statements resembling those of McCarthy. Conditional statements are enclosed in square brackets and are composed of a series of ordinary statements. Some of these are preceded by an IF clause (for example IF CAR A = CAR B) and are executed only if the condition specified in the clause is satisfied. Control passes to the statements in the order in which they are written and, if none of the IF conditions are satisfied, finally reaches the closing square bracket, and passes to the next statement in the program. If,however,one of the conditions is satisfied, the statement controlled by that IF clause is executed, and control then skips the remaining statements and jumps to the end of the conditional expression.

The statement controlled by an IF clause may be compound, that is it may be composed of several ordinary statements. This is indicated by enclosing the statements in round brackets. One or more of the statements in a compound statement may itself be a conditional statement.

An example of a conditional statement is

```
[IF CAR S = :*, L = S
 R = CDR R
 IF CAR R = :X, CAR L = :1
 CAR L = :0]
```

If CAR S = :* this is equivalent to

```
L = S;
```

otherwise, it is equal to

```
R = CDR R, CAR L = :1
```

if CAR R = :X, and to

```
R = CDR R, CAR L = :0
```

if CAR R ≠ :X .

The number of identifiers available for use by the programmer has been increased. In addition to the letter A, use may now be made of A1, A2, ..., A7, and similarly for other letters of the alphabet. These symbols when they stand for themselves, are known as atoms, and an IF clause of the form IF CAR A = ATOM, may be used to test whether a given symbol is atomic or not.

Instructions of the type PUSH DOWN A and POP UP A have been provided; the first inserts a new register (taken from the free list) at the beginning of list A, while the second removes a register from list A, and returns it to the free list. Thus the programmer may use any list as a push down list. In order to simplify as far as possible the use of a push down list for communicating with a subroutine, statements,of which the following are examples, are provided.

CAR A → B	equivalent to	PUSH DOWN A, CAR A = B
B → CAR A	equivalent to	B = CAR A, POP UP A.

The stack used in earlier versions (which was implemented as a consecutive list) has been suppressed and an ordinary push down list is now used for subroutine linkage; the list is not, however, available to the programmer. If when a compiled program is run the capacity of the store is exceeded and no more registers are available on the free list, it has been arranged that the machine shall report. This can happen either because the problem is too big for the available storage, or because the programmer has failed to return to the free list registers no longer required.

A full list of available statements is given below, and will, it is hoped, be sufficiently intelligible. WAIT and OPTIONAL STOP carry numerical identifiers and correspond to machine code facilities of the EDSAC. UNSET enables labels that have been used to be unset ready for re-use. COMMENT causes everything to be ignored up to the next comma or carriage return. COPY causes whatever is on the input tape to be copied into the compiled program, and its most important use is to enable machine code sections, or subroutines, to be incorporated into a program.

```
* = *                    IF CAR * = :*
* = CAR *                IF CAR * ≠ :*
* = CDR *                IF CDR * = :*
CAR * = *                IF CDR * ≠ *
CDR * = *                IF CAR * = CAR *
CAR * = CAR *            IF CAR * = ATOM
CDR * = CDR *            IF CAR * ≠ ATOM
CAR * = INPUT
* = :*
CAR * = :*
CDR * = :*
```

(continued overleaf)

```
**                          PRINT CAR *
UNSET ** - **               PRINT :*
TO **                       PUSH DOWN *
TO ** and back              POP UP *
RETURN                      CAR * → *
START AT **                 * → CAR *

SUBROUTINE **               WAIT *
SUBROUTINE END              OPTIONAL STOP *
                            COMMENT
                            COPY
```

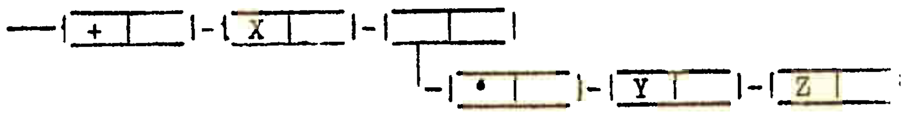
<u>Example</u>

As an example of a program written in the language just described, I give a program for formal differentiation of algebraic expressions such as

$$X + (X.Y)/(X + Y)$$

Differentiation is with respect to X. Note that products must be enclosed in brackets.

The program consists of a master routine (label 55) which reads in the expression to be differentiated (terminated by two stars) and then makes use of four subroutines 50 - 53, each of which uses itself recursively. Subroutine 50 re-arranges the expressions into a <u>Polish list</u>, which is a form of Polish notation. For example,

X + (Y.Z) becomes 

Subroutine 51 prints a Polish list as a conventional mathematical expression. Subroutine 52 performs the formal differentiation. Finally, subroutine 53 performs a certain amount of simplification; for example, it removes certain zero terms and contracts products in which one of the factors is unity. No attempt has been made, however, to do more than rather obvious pieces of editing, and the results printed are not necessarily in their simplest form, nor in a form pleasing to a mathematician.

Examples of differentiated expressions are given below. The first line in each pair is the original expression, and the second line is the derivative.

```
(X+(X.Y))
(1+Y)

(X+(X/Y))
(1+(1/Y))

(X+(X.(X+Y)))
(1+(X+(X+Y)))

(X+((X+Y).(X+Z)))
(1+((X+Y)+(X+Z)))
```

The program is given on the following pages. Note that PRINT :, causes a carriage return and line feed to be punched on the output tape, and that ( ) following an IF clause is a dummy statement for which no orders are compiled.

When an expression has been differentiated, the program resets itself ready to accept another expression. Reconstituting the free list was particularly easy in the case of the program since it had not been necessary to disturb the CDR of any register taken from the free list. There was thus a "thread" running through all registers used which enabled the registers to be returned to the free list in one operation.

I would like to thank Mrs V. Bayley for all the help that she has given me in checking out this system.

```
COMMENT, DIFFERENTIATE

55
PRINT :,
WAIT 5
P = F
CAR F = :(
F = CDR F
CAR F = INPUT
03
S = F
F = CDR F
04
CAR F = INPUT
[IF CAR F = :,, TO 04
IF CAR F = :  , TO 04]
[IF CAR F = :* [IF CAR S ≠ :*, TO 03] TO 03]
CAR S = :)

Z = S
Q = P
P1 = F, F = CDR F
UNSET 03 - 04
CAR M → P1
TO 50 AND BACK
CDR Z = F
F = Q
CAR M → P1
TO 51 AND BACK
P2 = F, F = CDR F
CAR M → P2
CAR M → P1
TO 52 AND BACK
CAR M → P2
TO 53 AND BACK
CAR M → P2
TO 51 AND BACK
OPTIONAL STOP 1

F = P1
OPTIONAL STOP 2
TO 55
```

```
COMMENT, FORM POLISH LIST

50
A → CAR M
S = CDR P
[IF CAR P = :(, P = CDR P
CAR A = CAR P
P = CDR P
RETURN]
CAR A = F
L1 = CAR A, L2 = CDR L1, L3 = CDR L2
F = CDR L3
CAR M → L3
CAR M → L1
CAR M → L2
TO 50 AND BACK
L1 → CAR M
CAR L1 = CAR P
P = CDR P
TO 50 AND BACK
P = CDR P
[IF CAR P = :), P = CDR P]
RETURN

COMMENT, PRINT POLISH LIST

51
PRINT :,
60
OPTIONAL STOP 4
A → CAR M
[IF CAR A = :0 ( )
IF CAR A = ATOM, PRINT CAR A
L1 = CAR A, L2 = CDR L1, L3 = CDR L2
PRINT :(
CAR M → L3
CAR M → L1
CAR M → L2
TO 60 AND BACK
L1 → CAR M
PRINT CAR L1
TO 60 AND BACK
PRINT :)]
RETURN
```

```
COMMENT, DIFFERENTIATE POLISH LIST

52
WAIT 6
A → CAR M
D → CAR M
[IF CAR A = ATOM
([IF CAR A = :X, CAR D = :1, CAR D = :0] RETURN)]
B1 = CAR A, L2 = CDR B1, B3 = CDR B2
CAR D = F
L1 = CAR D, L2 = CDR L1, L3 = CDR L2
F = CDR L3
[IF CAR B1 = :+, TO 05
IF CAR B1 = :-, TO 05
TO 06]

05
CAR L1 = CAR B1
CAR M → L2
CAR M → B2
CAR M → L3
CAR M → B3
TO 52 AND BACK
TO 52 AND BACK
RETURN

06
CAR L2 = F
R1 = CAR L2, R2 = CDR R1, R3 = CDR R2
F = CDR R3
CAR L3 = F
S1 = CAR L3, S2 = CDR S1, S3 = CDR S2
F = CDR S3
[IF CAR B1 = :., TO 07
IF CAR B1 = :/, TO 08
WAIT 7]

07
CAR L1 = :+
CAR R1 = :., CAR S1 = :.
CAR R2 = CAR B2
CAR S3 = CAR B3
CAR M → S2
CAR M → B2
CAR M → R3
CAR M → B3
TO 52 AND BACK
TO 52 AND BACK
RETURN
```

```
08
CAR S2 = F
U1 = CAR S2, U2 = CDR U1, U3 = CDR U2
F = CDR U3
CAR S3 = F
V1 = CAR S3, V2 = CDR V1, V3 = CDR V2
F = CDR V3
CAR L1 = :-
CAR R1 = :/, CAR S1 = :/
CAR U1 = :., CAR V1 = :.
CAR V2 = CAR B3, CAR V3 = CAR B3
CAR U2 = CAR B2, CAR R3 = CAR B3
CAR M → R2
CAR M → B2
CAR M → U3
CAR M → B3
TO 52 AND BACK
TO 52 AND BACK
RETURN

COMMENT, EDIT

53
A → CAR M
[IF CAR A = ATOM, RETURN]
L1 = CAR A, L2 = CDR L1, L3 = CDR L2
CAR M → A
CAR M → L2
CAR M → L3
TO 53 AND BACK
TO 53 AND BACK
A → CAR M
L1 = CAR A, L2 = CDR L1, L3 = CDR L2
[IF CAR L1 = :.
[IF CAR L2 = :0, CAR A = CAR L2
IF CAR L3 = :0, CAR A = CAR L3
IF CAR L2 = :1, CAR A = CAR L3
IF CAR L3 = :1, CAR A = CAR L2]
IF CAR L1 = :-
[IF CAR L3 = :0, CAR A = CAR L2
IF CAR L3 = CAR L2, CAR A = :0]
IF CAR L1 = :/
[IF CAR L2 = :0, CAR A = CAR L2
IF CAR L3 = :1, CAR A = CAR L2]
IF CAR L1 = :+
[IF CAR L2 = :0, CAR A = CAR L3
IF CAR L3 = :0, CAR A = CAR L2
IF CAR L2 = CAR L3
(CAR L1 = :.
CAR L2 = :2)]]
RETURN
```

**