UNIVERSITY MATHEMATICAL LABORATORY
CAMBRIDGE

Technical Memorandum 63/5

AN EXPERIMENT WITH A SELF-COMPILING COMPILER FOR A SIMPLE LIST-PROCESSING LANGUAGE (PART 2.)

by

M.V. Wilkes.

August 1963.

Herbert Schorr

University Mathematical Laboratory,
Corn Exchange Street,
Cambridge.

Technical Memorandum 63/5

AN EXPERIMENT WITH A SELF-COMPILING COMPILER FOR A SIMPLE LIST-PROCESSING LANGUAGE (PART 2.)

by

M.V. Wilkes.

Herbert Schorr

August 1963.

AN EXPERIMENT WITH A SELF-COMPILING COMPILER FOR A SIMPLE LIST-PROCESSING LANGUAGE (PART 2)

Part 1 of this paper, which appeared as Technical Memorandum 63/1, described how an elementary compiler for a simple list-processing language was written in the language itself and manually converted to machine code in such a way that the resulting compiler was capable of compiling itself. Once this had been done it was possible to improve both the compiler and the language by a series of bootstrapping operations, in each stage of which the compiler then existing was used to compile its successor. The language has become known as WISP.

Use of the final form of the language, which contained conditional expressions, was illustrated by a program for formal differentiation of algebraic expressions. In part 2 of the paper I shall give an annotated version of the compiler for this form of the language and describe how it was further developed with a view to the transfer of the system to other computers.

The Compiler TR 1

The development of the language in the way described on p.12 of Technical Memorandum 63/1 carried with it a corresponding substantial development of the compiler. The version finally arrived at is given below in full with annotations, and it will be observed that it is written in the language of TR 0.1, although that compiler must be provided with a few more standard forms and corresponding machine code translations than have been mentioned so far. It is hoped that the following remarks will enable the annotations to be understood.

Conditional statements are compiled as though they were written in terms of conditional jumps and labels. These labels are anonymous in the sense that they are introduced by the compiler without the programmer knowing anything about them. For example, the following conditional statement

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

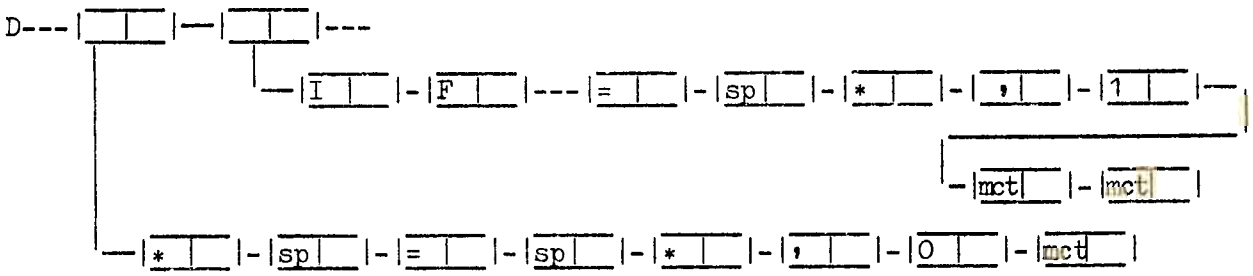
is compiled as though it were expressed in the following form.

```
IF CAR S ≠ :* GO TO 81
L = S
TO 80
81
UNSET LABEL 81
R = CDR R
IF CAR R ≠ :X GO TO 81
CAR L = :1
TO 80
81
UNSET LABEL 81
CAR L = :0
80
UNSET LABEL 80
```

A short machine code subroutine for issuing labels was written; this would issue labels in sequence starting, as will be observed, from the value 80. Since the EDSAC Assembly Routine provides a rather small number of different labels it is necessary that the labels should be unset when they have served their purpose in order that they may be used again. Re-issue of a label is made possible by providing the label issuing subroutine with a second entry point, entry at which causes the counter within the subroutine to be stepped back. The introduction of conditional statements means that compilation no longer consists of putting out a self-contained machine code translation for each statement on a one to one basis. For example, an opening square bracket - brackets are treated for the purpose of compilation as though they were statements - causes no orders to be compiled, but a label is put on the stack for use when the closing square bracket is encountered. An IF clause has two machine code translations associated with it. One is put out at once and a reference to the other is left on the stack for use after the statement controlled by the IF clause has been compiled. For example, the first IF clause in the above example leads to the immediate output of a machine code translation corresponding to IF CAR S ≠ :* GO TO 81 and, when the statement L = S has been compiled, a further machine code translation corresponding to the statement TO 80 is put out.

It will be seen that the action of the compiler is quite different when it is dealing respectively with ordinary statements, IF clauses, and brackets. Other types of statement also require their own special treatment and the compiler is, therefore, provided with a number of distinct subroutines. Control is sent to the appropriate subroutine by means of a switch and each standard form has a switch number stored along with it. Ordinary statements have the switch number 0, and IF clauses have the switch number 1. The following diagram shows how the

standard forms * = * and IF CAR * = * appear on the list of standard forms.



Note that the mct references are now stored in the CAR of a register and not in the CDR as was done in the case of earlier compilers.

It might be thought that the design of the subroutines required to deal with brackets and IF clauses would be quite complicated. This, however, is not so if the compiler is constructed from the beginning on sound recursive lines. In the present case the whole program is regarded as a compound statement, and must be enclosed in round brackets (later versions of the compiler automatically enclose the program in round brackets during input, thus saving the programmer the trouble of doing so himself). The heart of a compiler is a routine - which will be referred to as the Statement Routine - whose label is 08, and the function of this routine is to compile a single statement. The master routine of the compiler is very short and calls in the compile routine by the statement TO 08 AND BACK, with the object of compiling the compound statement that is the whole program. The first statement encountered is, naturally enough, an opening round bracket. This has switch number 2, and control is accordingly sent to subroutine 2 (label 52) which is as follows

```
52
TO 08 AND BACK
TO 52.
```

This subroutine has no RETURN statement and the statement routine is called in recursively and repeatedly to compile the statements within the compound statement. Control is, in fact, trapped in subroutine 2, and it can only be released when one of the statements turns out to be a closing round bracket. This sends control to subroutine 3 which is as follows

```
53
LEVEL DOWN
RETURN.
```

The effect of the LEVEL DOWN statement is that the succeeding RETURN statement sends control back, not to subroutine 2 (from which the statement routine was called in on this recursion), but to the place to which it would have been sent back by a RETURN instruction at the end of subroutine 2. If the closing bracket marks the end of the whole program being compiled, this means that control is returned to the master routine.

An IF clause is dealt with in a somewhat similar manner by subroutine 1, which uses the statement routine recursively to compile the dependent statement. This dependent statement may itself be a

compound statement containing perhaps conditional statements. It is surprising how cheaply facilities for the nesting of conditional and compound statements can be obtained once the statement routine has been designed recursively.

The compiler TR 1 is given in full below, except that the subroutine for dealing with COPY has been omitted.

TR 1

```
RESET STACK                 Reads in standard forms, using 26 recursively
G = F
F = CDR F
E = F
D = F
TO 26 AND BACK
CAR E = CAR L
F = L
WAIT
TO 93

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 = :.         The list of standard forms is terminated by a dot
F = CDR F

CAR F = INPUT               This statement reads the switch number for the
                            standard form just read
22
F = CDR F
CAR F = INPUT
TO 21 IF CAR F = :,
SUBROUTINE 79               Subroutine 79 was designed for the original
A = CDR G                   compiler, and puts the mct label into CDR G.
CAR F = A                   Note that there may be more than one mct label
TO 22                       corresponding to a given standard form, and
21                          that each one is placed in the CAR of a register
F = CDR F
CDR E = F
CAR E = L
E = CDR E
TO 26 AND BACK
24
RETURN
```

```
UNSET 21 - 25
93
P = F
29
S = Q
Q = F
F = CDR F
A = F
CAR A = INPUT
TO 21 IF CAR S = ::
TO 22 IF CAR A = :(
TO 22 IF CAR A = :)
TO 22 IF CAR A = :[
TO 21 IF CAR A ≠ :]
22
B = :,
CAR Q = B
Q = CDR Q
CAR Q = CAR A
Q = CDR Q
CAR Q = B
F = CDR Q
TO 23
21
CAR Q = CAR A

23

TO 28 IF CAR Q = :*
TO 29
28
TO 27 IF CAR S = :*
TO 29
27
WAIT

06
Q = P
TO 08 AND BACK


E = CAR D
E = CDR E
E = CDR E
E = CDR E
CAR E TO STACK
SUBROUTINE 77
WAIT
```

Reads in the program to be compiled

Puts commas round all brackets enclosing conditional or compound statements so that the brackets are treated by the compiler as statements

The program is terminated by two consecutive stars

The master routine of the compiler. 08 is the routine that translates a statement: the whole program is enclosed in round brackets and thus forms a statement. "TO 08 AND BACK" calls, therefore, for the whole program to be translated.

This is merely a rather roundabout way of compiling the symbols necessary to terminate the compiled program tape.

```
Q = CDR Q                          Prepares for the translation of a second program
CDR Q = F
F = P
TO 93                              End of master routine.

08                                 The statement routine

TO 15 IF CAR Q = : ,               The following section compares the program,
TO 15 IF CAR Q = :,                statement by statement, with the list of
TO 16                              standard forms.  It follows the general lines
15                                 of the corresponding section of earlier compilers.
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 ≠ :.
WAITS                              Waits if a match is not found
12
R = CDR R
Z = CDR Z
03
TO 14 IF CAR Z = :,
TO 05
13
CAR R TO STACK                     This section deals with two-character identifiers,
R = CDR R                          e.g. A5.  If α and β are the two upper
Z = CDR Z                          characters in the stack, subroutine 78 forms
TO 03 IF CAR R = : ,               10^5 α + β (mod 2^8) and leaves this quantity at
TO 03 IF CAR R = :,                the top of the stack
TO 13 IF CAR Z = :*
CAR R TO STACK
R = CDR R
SUBROUTINE 78
TO 03
14
TO 25 IF CAR R ≠ : ,
R = CDR R
25
TO 19 IF CAR R = :,
TO 07
```

```
19
Z = CDR Z
R = CDR R
SWITCH 49 - CAR Z
```

This point is reached when a match has been found. The switch number (N, say) is in CAR Z, and the switch statement compiles into a jump to a+N, where a is the address corresponding to label 49.

```
49
TO 50
TO 51
TO 52
TO 53
TO 54
TO 55
TO 56
```

Switch directory; each TO statement compiles into a single jump order

```
50
Z = CDR Z
CAR Z TO STACK
SUBROUTINE 77
48
RETURN
```

This is the subroutine for dealing with the ordinary type of statement. Subroutine 77 (COMPILE) uses the mct and any other necessary information from the stack

```
51
Z = CDR Z
SUBROUTINE 40
CAR Z TO STACK
Z = CDR Z
CAR Z TO STACK
SUBROUTINE 77
TO 08 AND BACK
SUBROUTINE 77
SUBROUTINE 41
RETURN
```

Subroutine for dealing with an IF clause. There are two mct's which both go into the stack; one is used at once by subroutine 77 (COMPILE), and the second is left on the stack for use after the statement controlled by the IF clause has been translated. Subroutine 40 issues a label to the stack, and subroutine 41 returns the label for re-issue later

```
52
TO 08 AND BACK
TO 52
```

Subroutine for dealing with "(" ; no mct

```
53
LEVEL DOWN
RETURN
```

Subroutine for dealing with ")" ; no mct

```
54
SUBROUTINE 40
60
TO 08 AND BACK
TO 60
```

Subroutine for dealing with "[" ; no mct. Note that a label is put into the stack (by subroutine 40) for later compilation when "]" is encountered

```
55                                  Subroutine for dealing with "]". There is
LEVEL DOWN                          one mct, which uses the label left over in
Z = CDR Z                           the stack from "[".
CAR Z TO STACK
SUBROUTINE 77
SUBROUTINE 41
RETURN

56                                  Subroutine for dealing with "COMMENT"
R = CDR R
TO 56 IF CAR R ≠ :,
Q = CDR R
RETURN
**
```

A modification later made to the compiler was intended to facilitate program error diagnosis. This consisted in making the compiler copy on to the output tape any statements in the program which could not be matched with one of the standard forms. Experience had shown that without this facility it was sometimes difficult to locate an illegal statement.

Although the conditional statements available with TR 1 are of great assistance to the programmer, they do not go as far as could be desired, since they do not allow for Boolean connectives in the IF clauses. One would like, for example, to be able to write IF CAR A = :+ or IF CAR A = :-. This is a feature which will, no doubt, be introduced into future versions of WISP, but to do so within the present framework is not entirely straightforward.

GARBAGE COLLECTION

The early versions of WISP were founded on the assumption that it was the programmer's responsibility to return to the free list any registers no longer required. In fact, the programmer did not need to bother much about this since the scale of the programs then being handled was such that 16,000 words of storage could be regarded as infinite. When one came to consider the application of the system to larger problems, however, the advantage of automatic garbage collection (as used in LISP) became clear. The first of these is obviously that it saves the programmer trouble. Apart from being troublesome, however, when complicated list structures with sublists exist, it may be exceedingly difficult, if not impossible, for the programmer to know what is garbage and what is not.

The garbage collector now incorporated in the system resembles McCarthy's in that it proceeds in two phases. In the first, all named lists are traced and all registers connected with them are marked by having the sign digit made into a one. In the second phase, all registers not so marked are collected and attached to the free list, the sign digits of marked registers being at the same time restored to 0.

For this type of garbage collector to work it is necessary that all lists and sublists should be terminated in some standard manner, and that the programmer should avoid the use of circular lists. The convention was adopted that lists and sublists should be terminated by having the CDR of the last register set equal to 0.

It was decided to introduce, along with the garbage collector, another facility, namely, the automatic extension of lists by the attachment of a register taken from the free list. Whenever a statement of the type A = CDR B is encountered, a test is made to ascertain whether CDR B = 0. If it does not, the operation proceeds. Otherwise a register is detached from the free list and linked to CDR A. The operation can then proceed and 0 is left in CDR A. If the free list is found to be exhausted, the garbage collector is called into action. If the garbage collector does not succeed in collecting any garbage then the machine reports the fact and comes to a halt. The garbage collector is also called into action if necessary by the statements PUSH DOWN * and CAR * → *. It should be noted that no automatic extension or garbage collecting operation can be initiated by the statement CDR * = CDR *, which is interpreted as performing a straightforward copying operation. The free list is now attached to a base register not accessible to the programmer. This, however, is of no significance on account of the automatic feature just described and the programmer can, in effect, use any list as the free list.

There is no doubt about the advantages of having an automatic garbage collecting system. Experience has also confirmed the utility of the automatic extension facility, since this saves the programmer much trouble in the setting up of lists, and tends to shorten programs written in symbolic language. It can, however, lead to a slight increase in the length of the compiled program and in the running time. As a further aid to the programmer, it has been arranged that the routine which sets up the free list in the first place shall attach one register to each identifier. This is certainly a convenient adjunct to the automatic extension facility, but it does use up a number of storage registers and it introduces a new programming pitfall in that the programmer must be on his

guard when he re-uses identifiers that have been used for other purposes; in particular he may need to re-attach a register after a statement of the type * =:*. This standard form will probably be dropped from future versions of the language; good programming practice would in any case appear to demand that symbols should not be put into a base register, but that the CAR of a register should be used to hold them.

A point in connection with the design of the garbage collector may be mentioned. It is quite easy with some small extensions to write the garbage collector in WISP, and it would be natural to make use of recursive facilities in doing so. If this is done, however, a special free list, unusable by other programs, must be provided for the use of the garbage collector, since it will operate when, by hypothesis, the ordinary free list is exhausted. Moreover, it is difficult to decide how large to make this private free list, since, in the worst conceivable case, half the store could be required; presumably some compromise, based on statistical considerations, would have to be made. The difficulty was avoided in the present instance by writing the garbage collector non-recursively, and by providing it with memory for one branch point only. The disadvantage of this is that the garbage collector must continually be starting again at the beginning of each list structure, and its operation is in consequence rather slow. No doubt an improvement could be made by providing memory for several branch points instead of only one, but this was not considered necessary at the present stage.

PROGRAMMING TECHNIQUE

What is good and what is bad programming in a given language only becomes apparent, as regards its finer points, as experience is accumulated. During the evolution of WISP new facilities were continually becoming available and some re-thinking of programming philosophy was necessary in order to make effective use of them. This accounts for the fact that the standard of programming in the examples given in this paper is uneven.

It goes without saying that the power of a programming language such as WISP or IPL resides largely in the recursive techniques that are available. The use of these does not come naturally at first and one has to force oneself to think recursively whenever possible. Later one must learn to be more discriminating and avoid recursive techniques where they are unnecessarily inefficient. An example of over-enthusiasm in the use of recursion is to be found in the section of TR.0.1 that reads in the standard forms, and an example of restraint in the use of recursion is to be found in the compile routine of TR 2. (See page 23.)

Some training is necessary also if one is to make efficient use of conditional statements, although this will come more naturally to those who have had experience of ALGOL or one of the other programming languages in which conditional statements may be used. I have noticed that newcomers to the later forms of WISP have tended to use conditional statements simply for conditional jumps, thereby losing one of the main advantages of conditional statements, which is to reduce the number of labels that occur in a program. It has already been remarked that the conditional statements available in WISP at the present time are by no means as powerful as could be desired.

It follows from the nature of lists that CAR and CDR are symmetric; either may contain a symbol or an address pointing to another item. When I first started on the WISP project I thought that the exploitation of the symmetry would be an important aspect of programming. The garbage collector, of course, imposes an assymetry since it looks for the symbol 0 in the CDR of an item as the indication of the end of a list or sublist. Even before the introduction of the garbage collector, however, I began to realize more and more that it is better to regard CAR and CDR in quite different lights. Symbols are always held in a CAR and, with the exception of a 0 to indicate the end of a list, a CDR always contains a link.

In a system such as WISP in which temporary names are freely introduced by the programmer, the provision of an automatic garbage collector does not wholly absolve the programmer from responsibility in this regard. He must be careful to see that temporary names are detached from lists which are likely to become garbage in order that the garbage collector may not be misled into thinking that they are still required.

MACHINE INDEPENDENCE AND TRANSFER TO OTHER MACHINES

So far the interest in this project had centred around the series of bootstrapping operations which led from a primitive language to a more highly developed one. At this stage it became appropriate to make a critical examination of the system to determine to what extent it was truly independent of the particular machine on which it had been developed and, after such improvement as seemed necessary, to subject it to the test of being transferred to another machine.

Complete machine independence can never be achieved, if only because some minimal basis of machine coding will be necessary on

transfer to a new machine in order to marry the system to the input and output facilities provided on that particular machine. A routine written in machine language will also be necessary for setting up the free list in the first instance.

Some of the editing performed by the input and output routines described at the beginning of the paper, could be performed by routines written in symbolic language and this would make a slight simplification of those routines possible. The gain would, however, not be very significant, and it was decided not to make the change, at any rate for the present. It was, however, clear that a very substantial improvement could be made by replacing the machine code compile routine, which, up to this time, formed an essential part of the system, by one wholly written in symbolic language. For this purpose it was necessary that the machine code translations should be stored in a list structure (with one character per word) as was already done for the standard forms themselves. Up to the present the machine code translations had been sorted in a block of consecutive registers.

This change necessitated a fairly extensive re-writing of the compiler and the opportunity was accordingly taken to express it in terms of the language accepted by TR 1. This new compiler was functionally equivalent to TR 1 and would compile the same assembly code.

One consequence of the re-arrangement was that it became a straightforward matter to provide facilities whereby the programmer using the system could, if he wished, define new standard forms to supplement those already provided. This he can now do by writing the statement NEW SF followed by the new standard form and its machine code translation. This new facility turned out to be very relevant to the matter considered in the next paragraph. A further facility provided was that of defining new standard forms in terms of already existing standard forms instead of in terms of machine code. There is clearly a potential utility for such a facility, but no practical experience of its use has yet been acquired.

There are various ways in which one may proceed when it is necessary to go outside the scope of the language proper. I am here primarily concerned with this problem as it arises in connection with the compiler, but a few additional remarks on it from a user's point of view may not be out of place.

Two approaches are possible. The first is to resort to assembly

language programming. This may, perhaps, seem a confession of defeat, and if the language is supposed to be complete enough to replace assembly language for all purposes, then indeed it is. There is, however, another way of looking at a system of automatic programming; it may be regarded as an aid to the writing of an assembly language program to be used as a labour-saving device by people who are familiar with assembly language. If one looks at it this way it is natural to slip into assembly language when the system fails. WISP includes, in fact, two ways of introducing assembly language into a program. Subroutines written in assembly language may easily be connected with WISP programs, since WISP labels are compiled directly into assembly language labels, and the WISP statements for calling in and returning from a non-recursive subroutine compile into instructions identical with those used in assembly language programming. The other way is to make use of the COPY statement. This is convenient when a few machine instructions only - or even single instructions - are required, and it enables full use to be made of the conditional statements and recursive facilities of WISP. Assembly language patching is a help in maintaining the efficiency of the compiled program, but on the other hand it reduces machine independence.

The other approach is to define new standard forms to give the additional facilities required. As explained above, with the latest version of the compiler this is very easy to do. Since the machine code translations have to be written in assembly language, the defining of new standard forms is perhaps only another way of patching the program. It does, however, have the advantage that, if the new standard forms are carefully phrased, the program retains transparency. Moreover, from the point of view of the evolution of the language, the experience gained by expanding it with new standard forms is most valuable.

It appeared that there were four operations in the WISP compiler that could best be dealt with by special standard forms defined for the purpose. Two of these respectively increase and decrease the number in the CAR of a specified register by 1 and are made use of by the label issuing subroutine. Another provides for decimal input which is needed to read in the base from which anonymous labels run, and the fourth is a decimal print statement required by the compile routine. These last two standard forms make use of standard input and output routines.

In addition, provision must be made for translating two-character identifiers, and it appeared that this could best be done by providing a short machine code subroutine (cf. subroutine 78 of TR 1).

The new compiler, TR 2, is given in full at the end of the memorandum.

Transference of the system to another machine

It follows from what has been said that the WISP system can be divided into three parts, (1) the environment, (2) the compiler, and (3) the tables of standard forms and machine code translations. It is convenient for the present purpose to consider the environment as including the machine code subroutine used to translate two-character identifiers, although this is something peculiar to the compiler and is not needed at run time by compiled programs. Apart from this the environment consists of routines for input and output, for setting up the free list at the beginning of a program, for the automatic extension of lists where required, and a garbage collector. In the EDSAC 2 realization of the system the environment contains about 240 instructions of which 80 are accounted for by the garbage collector. It is probably better to omit the garbage collector in the first instance when transferring the system to another machine, and to add it later when the system is working.

The environment must be re-written in machine language for each new machine to which the system is transferred. The compiler is, of course, written in machine independent language. The tables are also written in machine independent language as far as their form is concerned, although their content is naturally dependent on the machine for which the compiling is being done. The tables are read into the machine under the control of the compiler. Thus the input consists of the program to be translated, together with the tables giving standard forms with their machine code translations for the machine on which the object program is to be run. Note that this need not be the same as the machine being used for the translation.

This last observation points the way to the procedure that is used for transferring the system from one machine to another. This is illustrated in table 1. Line 1 shows the system in operation on machine A. It is shown taking its own compiler in symbolic form and translating it into an assembly language version. The arrow indicates that this program could be read back into the machine and would then be identical with the compiler previously there. Line 2 differs only in that new tables, giving machine code translations for machine B, have been used for input; the output is a version of the compiler in the assembly language appropriate to machine B. The arrow indicates that this is read into machine B, together with an environment which has been specially written in machine code. The system is now capable of operating on machine B, and in line 3 of the table it is shown translating its own compiler.

So far the system has been successfully transferred to an

Elliott 803 computer and its transference to an IBM 709 computer is in progress. The transference of a programming system to another computer is a major operation and it is not surprising if all does not go as smoothly as the foregoing discussion might suggest. The various subroutines used in the environment can, of course, be checked out by means of short programs written for the purpose, but errors in the standard forms are more difficult to locate. It was soon realized when transferring the system to the 803 that it would be a good idea to write in symbolic language a short series of test programs, which would enable the standard forms to be checked systematically before an attempt was made to compile the whole compiler. Even so, one rather subtle error was not picked up by these tests. It will be realized that an error in the standard form affects the version of the compiler used in the new machine as well as object code produced by it. Such errors, therefore, involve some to-ing and fro-ing from one machine to another. It might be better to transfer a simple version of the system in the first instance and to make use of one or more stages of bootstrapping thereafter. It is hoped to report further on the problems involved in transferring the system from one computer to another when more experience has been obtained.

A minor problem in transferring a system arises on account of differing character sets, and naturally some decision must be taken as to what symbols to use when the symbols used in the original realization are not available. Means must then be provided for bridging the gap between the character sets and character codes on the two machines. Perhaps the simplest way to do this is to take a printed output from computer A, edit it where the characters differ, and key punch it for computer B. In the case of the 803 computer, however, it was found convenient to use that computer itself for doing the necessary transliteration, a short program being written for the purpose. The tables of standard forms were edited and key punched separately for the two computers.

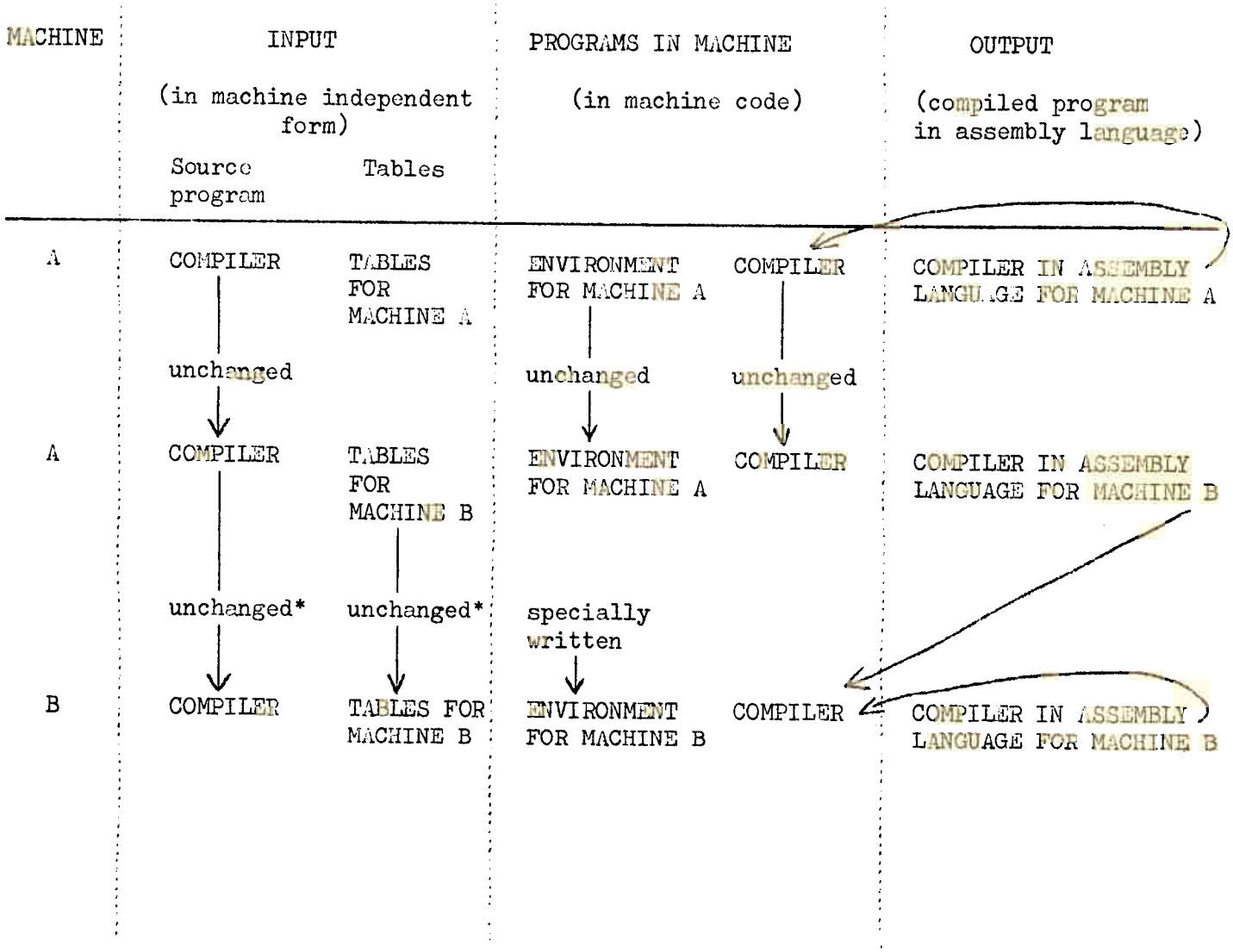
Since WISP compiles into an assembly language, the existence of a satisfactory assembly routine for machine B is a pre-requisite for successful transfer. In the case of the IBM 709, the FAP Assembler was found to be entirely satisfactory, but in the case of the Elliott 803, a modification to the Assembly Routine was found desirable. This was because in 803 assembly language labels are written after the instructions to which they refer, whereas the WISP compiler puts them in front, and to change this would not be a trivial matter. Even if it appears that changes to the compiler would enable a more satisfactory system to be established on the new machine, it is, nevertheless, desirable to effect the transfer in the first

instance with the compiler changed as little as possible. The proper time to make changes is after the transfer has been made and when the compiler is capable of compiling itself on the new computer.

Experience suggests that machine dependence is more likely to creep into a system such as WISP via features of the Assembly Language of the original machine, rather than as a result of features of the logical design of the machine. Perhaps one should say that the system is more likely to turn out to have assembly language dependent features than machine dependent features. An example is provided by the WISP statement UNSET ** - **. This appears in the language as a direct consequence of the existence of a similar feature in EDSAC 2 assembly language, which provides the programmer with a rather small number of distinct labels, and must, therefore, enable him to re-use them as many times as is necessary. In FAP the number of distinct labels that may be used is virtually unlimited, and there is no provision for unsetting. Consequently, the 709 version of WISP will not have the UNSET statement, and some WISP programs written for EDSAC 2 will need a small amount of editing before they will run on the 709.

TABLE I

TRANSFERENCE OF SYSTEM FROM MACHINE A TO MACHINE B



* except for substitutions made necessary by incompatible character sets

I give below representative extracts from the standard form tables for the three machines. These are all expressed using LDSAC keyboard characters. They are followed by an annotated version of the compiler TR 2.

In writing the machine code translations use is made of four symbols with special meanings. These are known as reserved symbols, and are defined by being placed at the head of the table of standard forms. The third reserved symbol indicates that the figure following - r, say - is not to be put out as part of the machine code translation, but that the identifier corresponding to the rth star in the standard form is to be taken instead. The other reserved symbols have special meanings only when they occur at the beginning of a line. The first is interpreted by the routine that reads in the machine code translations as meaning that the identifier following it is to be assigned as a name to the machine code translation about to be read. The second is addressed to the same routine and is also followed by an identifier. It indicates that the machine code translation to be used at this point is identical with one already read in and assigned that identifier as a name. The fourth reserved symbol is used to terminate the machine code translations in a way that will be sufficiently clear from the tables themselves.

The naming facility for machine code translations was introduced in order to give extra flexibility in possible future developments of the system. It is not used much in the TR 2 standard forms.

Following the reserved symbols is a decimal number giving the base from which anonymous labels run.

The statement X - this has no connection with the X used with TR 0.1 - was intended to enable a terminating symbol to be put out at the end of the program. It is, however, rarely used by the programmer, since the compiler is arranged to put out the terminating symbol automatically, making use of the name J2 for this purpose.

Figure 4 is part of the list of standard forms and shows how the sublists containing machine code translations are arranged. The section illustrated contains the two IF statements included in the extract from the table given below. These statements each have two mct's associated with them. Other types of statement may have one or none.

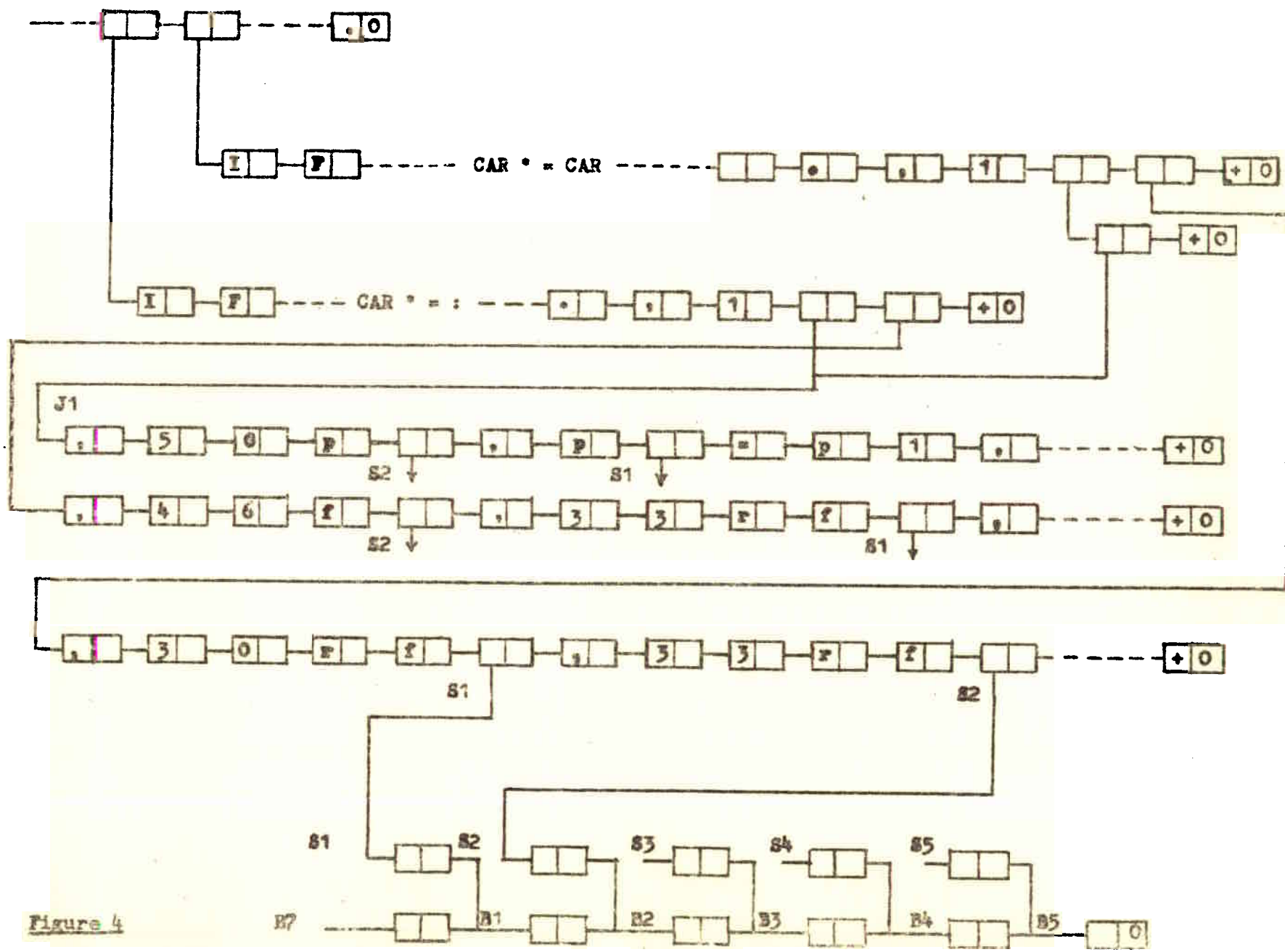


Figure 4

EXTRACTS FROM STANDARD FORMS TABLE FOR EDSAC 2, ELLIOTT 803, IBM 709 (or 7090) FAP

```
(*-+
80

X,0
(J2
 *
++

(,2
++

),3
++

[,4
++

],5
p-1=p1
=-1/109
++

COMMENT,6
++

COPY,7
++

* = CAR *,0
30rf-2
99f3
100f3
39rf-1
++

TO **,0
50p-1-2
++

CAR * = :*,0
46f-2
8rf-1
62f6
29rf-1
++
```

```
(*-+
80

X,0
(J2
)
++

(,2
++

),3
++

[,4
++

],5
r000-1)
06 0n40 207.3(
00 -1n00 99
++

COMMENT,6
++

COPY,7
++

* = CAR *,0
00 -2.2/30 0.2
51 20n20 -1.2
++

TO **,0
400n
r000-1-2(
++

CAR * = :*,0
73 3.1/30 1
00 -2n00 0
00 -1.2/10 0.2
03 0.1n
00 -1.2/24 0.2
++
```

```
(*=+
1

X,0
(J2
        END
++

(,2
++

),3
++

[,4
++

],5
A=1     EQU *
++

COMMENT,6
++

COPY,7
++

* =     CAR *,0
        CAL* -=2(2
        LGR  18
        STA -=1(1
++

TO      **,0
        TRA L=1=2
++

CAR *   = :*,0
        AXT =2(4
        PXD (4
        STD* -=1(2
++
```

```
IF CAR * = :*,1
(J1
50p-2
p-1=p1
=-1/109
+
46f-2
33rf-1
62f8
53p-3
++
```

```
IF CAR * = :*,1
(J1
40 On
r000 -2(
r000 -1)
060n40207.3(
00-1n0099
+
00 -1.2/30 0.2
73 3.1/07 1
00 -2n03 1.1
00 3.1/46 3
40 On
r000 -3(
++
```

```
IF CAR * = :*,1
(J1
        TRA  A=2
A=1     EQU *
+
        AXT  =2(4
        STZ  /TEMP
        SXD  /TEMP(4
        CAL* -=1(2
        ANA  /CAR
        SUB  TEMP
        TNZ  A=3
++
```

```
IF CAR * = CAR *,1
*J1
+
30rf-1
33rf-2
62f8
53f-3
++
.
.
.
.
.
+++
```

```
IF CAR * = CAR *,1
*J1
+
00 -1.2/30 0.2
03 1.1n
00 -2.2/07 0.2
03 1.1n73 3.1
00 3.1n46 2
40 On
r000-3(
++
.
.
+++
```

```
IF CAR * = CAR *,1
*J1
+
        CLA* -=2(2
        SUB* -=1
        ANA  /CAR
        TNZ  A=3
++
.
.
.
.
.
+++
```

Note that 6, 8 (EDSAC 2) 0.1, 1.1 (803) and CDR, CAR (709) contain masking constants for CDR and CAR respectively.

```
42
40
WAIT 0
CAR F = INPUT                                  Reads in the four reserved characters
[ IF CAR F ≠ :,, TO 40 ]
CAR W1 = INPUT, CAR W2 = INPUT
CAR W3 = INPUT, CAR W4 = INPUT

10
CAR F = INPUT                                  Reads in base number from which
[ IF CAR F ≠ :,, TO 10 ]                       anonymous labels run
CAR L1 = DEC INPUT

B1 = CDR B7, B2 = CDR B1                       Sets up the list structure that
B3 = CDR B2, B4 = CDR B3, B5 = CDR B4          takes the place of the stack used
CAR S1 = CAR W3, CDR S1 = B1                   in earlier compilers (see fig.4)
CAR S2 = CAR W3, CDR S2 = B2
CAR S3 = CAR W3, CDR S3 = B3
CAR S4 = CAR W3, CDR S4 = B4
CAR S5 = CAR W3, CDR S5 = B5

S1 = D
TO 30 AND BACK
TO 44

48
CAR S = CAR Z2, CAR R = CAR Z3
TO 47                                          47 is the machine code subroutine
                                               that deals with two-character
                                               identifiers (cf. subroutine 78 of TR 1)
COMMENT, READ STANDARD FORMS

30
Z = CDR R, Z = CDR Z
25
R1 = CDR R, R2 = CDR R1
CAR R2 = INPUT
[ IF CAR R = CAR W4
  [ IF CAR R1 = CAR W4
    [IF CAR R2 = CAR W4
      (R = F, F = CDR F, CDR R = :0, TO 26 )]]]
R = R1, TO 25
```

```
26
[ IF CAR Z = :, (Z = CDR Z, TO 26 )]     Sets up the list structure
CAR D1 = Z, D1 = CDR D1, CAR D1 = :.     illustrated in fig.4
21
[ IF CAR Z ≠ :, (Z = CDR Z, TO 21)
Z = CDR Z, Z1 = CDR Z
IF CAR Z1 ≠ :,
Z = CDR Z1, Z = CDR Z ]
PUSH DOWN Z, D2 = Z, Z = CDR Z, CDR D2 = :0
22
CAR D2 = Z

24
Z1 = CDR Z
[ IF CAR Z = :,
[Z2 = CDR Z1, Z3 = CDR Z2
  IF CAR Z1 = CAR W1
    ( TO 48 AND BACK
     A = CAR S, CDR A = Z
     Z4 = CDR Z3, CDR Z = CDR Z4, TO 24 )
  IF CAR Z1 = CAR W2
    ( TO 48 AND BACK
     B = CAR S, B = CDR B, CAR Z = B
     CDR Z = CDR Z3, TO 24 )
  IF CAR Z1 = CAR W4
   [ CDR Z1 = :0
  IF CAR Z2 = CAR W4
   [ IF CAR Z3 = CAR W4, RETURN
     Z = Z3, TO 26 ]
   D2 = CDR D2, Z = Z2, TO 22 ]]
  IF CAR Z = CAR W3
    ([ IF CAR Z1 = :1, CAR Z = S1
       IF CAR Z1 = :2, CAR Z = S2
       IF CAR Z1 = :3, CAR Z = S3
       IF CAR Z1 = :4, CAR Z = S4
       IF CAR Z1 = :5, CAR Z = S5 ]
     CDR Z = CDR Z1 ) ]
Z = CDR Z, TO 24

UNSET 40 - 41                            Label issuing routine
40
PUSH DOWN L
CAR L = CAR L1
INCR CAR L1
RETURN

41                                       Label cancelling routine
POP UP L
DECR CAR L1
RETURN
```

```
COMMENT, READ PROGRAM

UNSET 05 - 25
44
A = Q
CAR A = :(
WAIT 8
23
R1 = R
R = A
A = CDR A
CAR A = INPUT
[ IF CAR R1 ≠ :,, TO 23
  IF CAR R ≠ :*, TO 23
  IF CAR A ≠ :*, TO 23 ]
CAR R = :), CDR R = :0

TO 08 AND BACK                          Master routine of the compiler
CAR M → J2
TO 77 AND BACK
WAIT 3, TO 44

43                                      Inserts, if necessary, a comma at
R = Q                                   the end of each statement immediately
CAR S = :+                              before it is scanned. This is an
21                                      alternative to putting commas round
[ IF CAR R = :,, RETURN                 brackets enclosing statements as is
  IF CAR R = ::, R = CDR R              done in TR 1
  IF CAR R = :(, TO 22
  IF CAR R = :), TO 22
  IF CAR R = :[, TO 22
  IF CAR R = :]
( [ 22, IF CAR S = :+, R = CDR R ]
    PUSH DOWN R, CAR R = :,, TO 21 ) ]
    CAR S = :-, R = CDR R, TO 21

COMMENT, COMPILE

77
Z → CAR M                               Replaces the machine code compile
10                                      routine in TR 1
[ IF CDR Z = :0, ( )
  IF CAR Z = ATOM
    ([ IF CAR B7 = :+
      (PRINT DEC CAR Z
       CAR B7 = :-, RETURN )
       IF CAR Z = CAR W3, CAR B7 = :+
       PRINT CAR Z ]
       Z = CDR Z, TO 10 )
```

```
        A = CDR Z, CAR M → A
        A = CAR Z, CAR M → A
        TO 77 AND BACK
        TO 77 AND BACK ]
        RETURN

UNSET 10 - 10

08                                          Functionally equivalent to the
[ IF CAR Q = :,, TO 15                      corresponding routine in TR 1
IF CAR Q = :,, ( 15, Q = CDR Q, TO 08 )]
TO 43 AND BACK
E = D
09
B = B7
Z = CAR E
R = Q
05
[ IF CAR Z = :*, TO 13
  IF CAR R = CAR Z, TO 12
  07
  E = CDR E
  IF CAR E ≠ :., TO 09 ]
R = Q
PRINT : ,
10
[ IF CAR R ≠ :,, (PRINT CAR R, R = CDR R, TO 10 )]
Q = CDR R
PRINT : ,
WAIT 7
TO 08
12
R = CDR R, Z = CDR Z
03
[ IF CAR Z = :,, TO 14, TO 05 ]
13
B = CDR B, CAR B = CAR R
R = CDR R, Z = CDR Z
[ IF CAR R = :,, TO 03
  IF CAR R = : , TO 03
  IF CAR Z = :*, TO 13 ]
TO 47 AND BACK
R = CDR R
TO 03
14
[ IF CAR R = : , R = CDR R ]
Z = CDR Z, Q = CDR R
SWITCH 49 - CAR Z
49
TO 50
TO 51
TO 52
TO 53
```

```
TO 54
TO 55
TO 56
TO 57
TO 58
TO 59

50
Z = CDR Z, Z = CAR Z, CAR M → Z          Ordinary statements
TO 77 AND BACK
RETURN

51
Z = CDR Z, A = CAR Z, CAR M → A          IF statements
Z = CDR Z, A = CAR Z, CAR M → A
TO 40 AND BACK
CAR B3 = CAR L
TO 77 AND BACK
TO 08 AND BACK
CAR B1 = CAR L
TO 41 AND BACK
CAR B2 = CAR L
TO 77 AND BACK
RETURN

52
TO 08 AND BACK, TO 52                    (

53
LEVEL DOWN, RETURN                       )

54
TO 40 AND BACK                           [
16
TO 08 AND BACK, TO 16

55
LEVEL DOWN                               ]
Z = CDR Z, Z = CAR Z, CAR M → Z
CAR B1 = CAR L
TO 41 AND BACK
TO 77 AND BACK
RETURN
```

```
56
R = CDR R                                   Comment
[ IF CAR R ≠ :,, TO 56 ]
Q = CDR R
RETURN

57
[ IF CAR R = :+                             Copy
( PRINT : , Q = CDR R, RETURN )
PRINT CAR R
R = CDR R, TO 57 ]

58
Z = CDR R, TO 26 AND BACK, Q = CDR Z3       New SF;  calls in again the routine
RETURN                                      for reading standard forms

59
UNSET 49 - 58
Z = CDR Z                                   Provision has been made for two-digit
SWITCH 49 - CAR Z                           switch numbers beginning with a 9.
49                                          Only one has so far been used, namely,
TO 50                                       90, for standard forms defined in
                                            terms of other standard forms
50                                          instead of in terms of assembly
UNSET 20 - 24                               language.  Variable information is
Z = CDR Z, R = CAR Z                        indicated in these by using the
PUSH DOWN M, CAR M = :*                     appropriate reserved symbol
20                                          exactly as is done in an mot
[ IF CAR R = ATOM [ CDR R = :0, TO 21 ]
CAR M → R
S = CAR R, CAR M → S
S = CDR S, CAR R = CAR S ]
R = CDR R, TO 20
21
CAR M → Q
Q = CAR Z
TO 08 AND BACK
Q → CAR M
22
[ IF CAR M ≠ :*
( S → CAR M, R → CAR M
  CAR R = S, TO 22 ) ]
POP UP M
RETURN

START AT 42
```

Acknowledgements

I would like to express my gratitude to colleagues and students who have taken an interest in this project. Mr. J.S. Biggs and Mr. P. Grant wrote the garbage collector and added the automatic extension facility. Mr. N.E. Wiseman played an active part in the operation of transferring the system to the other machines, particularly to the Elliott 803. Dr. H. Schorr is concerned with the transfer to the IBM 709.

The experiment of transferring the system was made possible through the kindness of Mr. S.L.H. Clarke, Assistant General Manager, Elliott Brothers (London) Ltd., and of Dr. H. Lipps, Computing Laboratory, CERN, Geneva. I would like to thank them, and also Mr. I. Gould and Mr. B. Elliott of their respective staffs, for their enthusiastic cooperation.

Dr. Harry Huskey first interested me in bootstrapping techniques, and I am also indebted to him for a number of helpful discussions on the problem of transferring a system from one computer to another.

Finally, I would like to give my special thanks to Mrs V. Bayley for her help in running the system on the EDSAC and in developing it through its various stages.