DEPARTMENT OF ELECTRICAL ENGINEERING
UNIVERSITY OF COLORADO

WISP

A Self Compiling List Processing Language

by

R. J. Orgass
Yale University

H. Schorr
IBM Systems Development Division

W. M. Waite
University of Colorado

M. V. Wilkes
University of Cambridge

February, 1967

PREFACE

WISP was originally conceived by M.V. Wilkes as a student exercise, and was then used to implement an EDSAC 2 Autocode compiler. Further development of WISP was carried on using the IBM 7094 at Columbia University, where the language was used extensively by students in an advanced programming course. Extensions of the language described in this manual are currently being designed at the University of Sydney and at Yale University.

The authors gratefully acknowledge the support, both in man-hours and computing time, of a number of organizations: Basser Computing Department, University of Sydney; Columbia University Computing Center; Electronics Research Laboratory, Columbia University; IBM Research Center, Yorktown Heights; National Science Foundation; The University Mathematical Laboratory, Cambridge; Yale Computer Center.

Finally, we wish to thank the staff and students of the Basser Computing Department for their thoughtful appraisal of several drafts of this edition of the manual. The typing and reproduction was in the capable hands of Miss Jane Blanton.

PREFACE

Wisp was originally conceived by M. V. Wilkes as an exercise in compiler construction, and was used to implement a translator for EDSAC2 Autocode. Further development was carried on using the IBM 7094 at Columbia University, where the language was used by students in an advanced programming course. Wisp has been successfully transferred to a variety of machines including the English Electric KDF9, CDC 6400, GE 235 and IBM System/360. The present version has evolved into a system which can be quickly and easily moved to a new machine, even by a person who is not an expert in either the language or the target machine.

The authors gratefully acknowledge the support, both in man-hours and computing time, of a number of organizations: Basser Computing Department, University of Sydney; Columbia University Computing Center; Electronics Research Laboratories, Columbia University; Graduate School Computing Center, University of Colorado; IBM Research Center, Yorktown Heights; National Science Foundation; the University Mathematical Laboratory, University of Cambridge; Yale Computer Center.

Finally we wish to thank the staff and students of both the Basser Computing Department of the University of Sydney and the Electrical Engineering Department of the University of Colorado for their thoughtful appraisal of several drafts of this edition of the manual.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

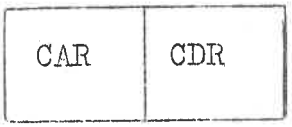
I. INTRODUCTION

This manual is intended as a complete description of the basic version of the Wisp list processing system. Basic Wisp may be thought of as a rudimentary symbolic assembly language for a list processing machine [1]. It has been found to be an ideal language with which to introduce the subject of list processing, as well as a flexible tool for use in non-numeric applications. This language has been used successfully for analytic differentiation, automatic printed circuit layout, solution of various path-tracing problems in graph theory, theorem proving and language translation. It has also provided a means of rapidly constructing very simple special editing programs for setting up tables, rearranging data, removing unwanted characters from masses of text, and so forth.

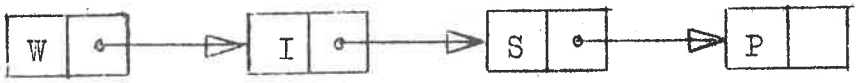
Data storage in Wisp is accomplished by means of a list structure, which is made up of list elements. Each list element (see Figure 1(a)) may be thought of as a box with two compartments called the CAR and the CDR, either of which may contain an atom or a pointer. An atom is a basic, indivisible unit of information such as a single character, and a pointer is the address of a list element. A linear list is a set of list elements in which the CDR of the first contains a pointer to the second, the CDR of the second points to the third, etc. (Figure 1(b)). The CAR of every element of a linear list must contain an atom. A branched list is a list in which the CAR of some element contains a pointer, as in Figure 1(c).

Statements in Basic Wisp are translated into machine code by a compiler, rather than run interpretively. The compiler is written in Basic Wisp to enable the language to be transferred to a new machine with ease (experience indicates that approximately two man-months are ample to obtain a working system). In order to ensure compatibility over a wide range of machines, the character set used for Basic Wisp has been kept as simple as possible; the language itself is completely independent of any particular host machine.

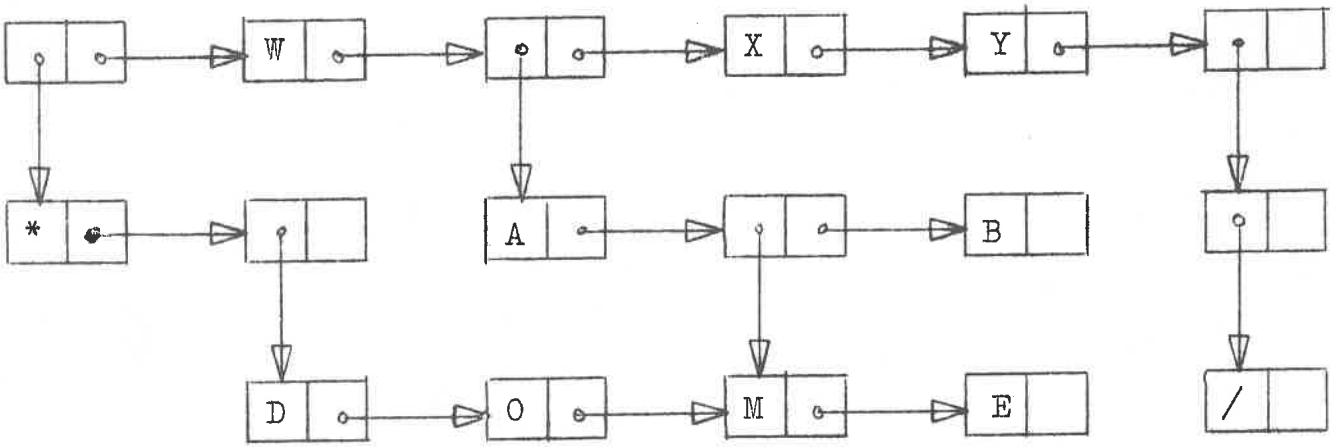
Chapter II of this manual describes the Basic Wisp language in detail, explaining the function of each statement and giving numerous programming examples. Chapter III covers the bootstrapping process by which Basic Wisp may be implemented on a new machine, and Chapter IV deals with an extension to handle more general data structures.



(a) A LIST ELEMENT



(b) A LINEAR LIST



(c) A BRANCHED LIST

[*[DOME]]W[A[ME]B]XY[[/]]

FIGURE 1

EXAMPLES OF LIST STRUCTURES

II. THE BASIC WISP LANGUAGE

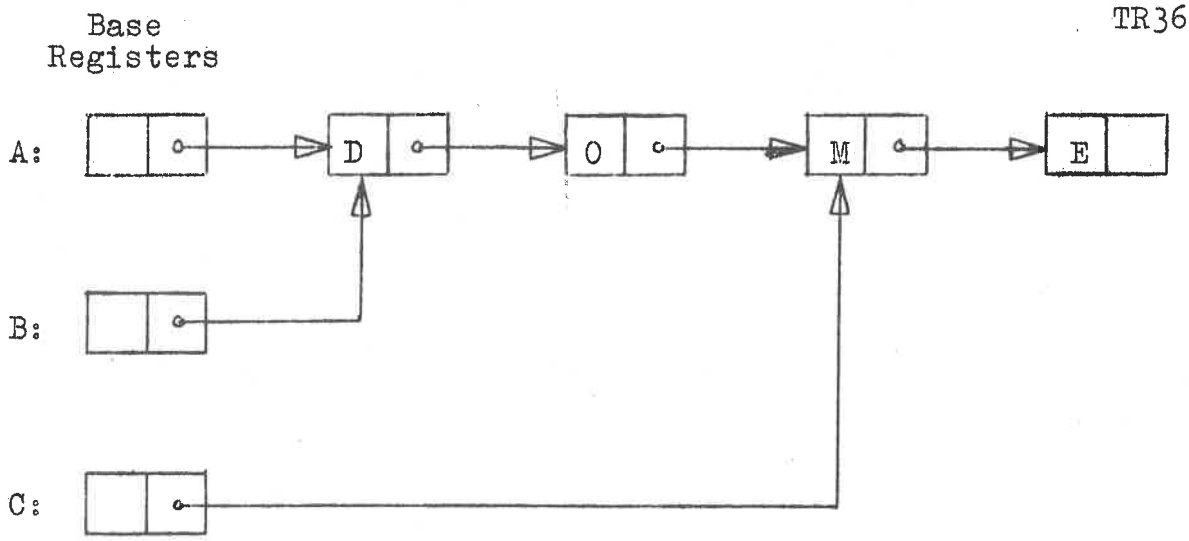
A. Atoms, List Names and Punctuation

Each atom corresponds to a special list element, called a base register, which occupies a fixed memory location. Each atom may be used as a list name, and the CDR of the atom's base register points to the first element of the list named. In Figure 2, for example, lists A and B are the same (i.e. base registers A and B point to the same element) and contain the characters "D O M E", while list C contains the characters "M E". Base registers are not generally shown when the list is diagrammed, but the list name is given to indicate the element to which the base register is pointing (see Figure 2(b)). A special atom, NIL, is used to indicate an empty CAR or CDR: In Figure 2, the CDR of the rightmost list element contains NIL.

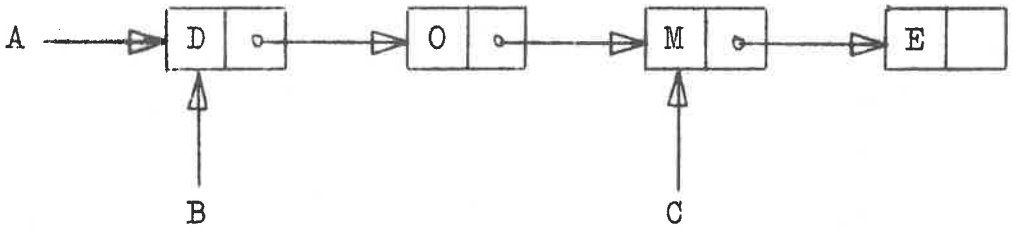
Internally, each atom is represented by the address of the corresponding base register. The programmer can make use of this fact to manipulate lists which are not directly accessible due to constraints imposed by the format of a Wisp statement. (We shall deal with this point at length in the section on Indirect Referencing.) The external representation of an atom is completely machine-dependent, and need bear no relation to the address of the corresponding base register. For this reason, the CAR of the base register is sometimes used to store the external representation of the atom.

In the text of a Wisp program, we use a quote (') to distinguish between a character standing for itself (i.e. an atom) and a character standing for the corresponding list. Thus 'A denotes the atom A, while A denotes the list A. (Note: The particular set of special characters available on a given machine may not contain all of those used in Basic Wisp. Appendix I gives the character sets used on each machine for which Wisp has been implemented, and their correspondence to the symbols used in this manual.)

A Wisp program is made up of a series of lines (on card-oriented machines, each card is a line; on paper tape a line is bounded by carriage return characters). Each line contains a series of statements separated by delimiters. Basic Wisp uses only two delimiters: "," and ".". The comma is used to mark the end of a statement, while the period not only ends a statement, but also causes the remainder of the current input line to be ignored. That portion of a line following the first period may therefore be used for comments, serialization, etc.



(a)



(b)

FIGURE 2

USE OF LIST NAMES

Any string of successive blanks is treated by the Wisp compiler as a single blank, unless it immediately follows a quote. In this case the first blank is considered to be an ordinary atom and the remainder of the string is treated as a single blank. Any string of blanks followed by a delimiter is treated as the delimiter alone. Blanks are significant within a Wisp statement--the elements of the statement must be separated by blanks. For example, "A = B." is a correct Wisp statement, but "A=B." is not. A complete list of Wisp statements is given in Appendix II, showing the spacing required. Any leading blanks appearing in a statement are suppressed, and any blank lines are ignored. Thus the statements:

```
CAR A = 'B.
CAR A = 'B   .
CAR      A = 'B   .
CAR   A =  'B     .
      CAR A =  'B .
```

are all equivalent. The statements:

```
CAR A = ' B.
C AR A = 'B.
```

are both incorrect.

A Wisp program is terminated by a void line (i.e. one which begins with a period). This line must be physically the last, and signals the compiler that the entire program has been processed.

B. Assignment Statements

An assignment statement is a statement of the form:

(Operand 1) = (Operand 2).

This statement should read "(Operand 1) is to be replaced by (Operand 2); (Operand 2) is to remain unchanged". The convention for designating operands is:

A - The contents of the CDR of the base register corresponding to A.

'A - The atom A (i.e. the address of the base register corresponding to A.)

CAR A - The contents of the CAR of the element pointed to by base register A

CDR A - The contents of the CDR of the element pointed to by base register A

A table of all assignment statements available in Basic Wisp is given in Appendix II (1), where an asterisk is used as a generic term for any list name. The effect of each assignment statement is shown pictorially in Appendix III.

Initially each base register points to a single, empty element (i.e. one whose CAR and CDR both contain NIL). The remaining elements are built into a linear list (called the free list) which is not directly accessible to the programmer. Additional elements can be obtained from the free list by the use of either of the two assignment statements:

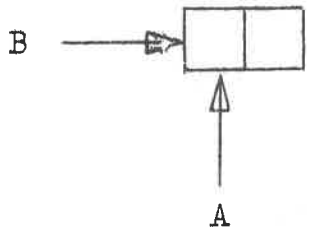
```
* = CDR *.
CAR * = CDR *.
```

If either of these is executed when the referenced CDR contains NIL, a new element will be taken from the free list and its address placed in the CDR before the contents of the CDR is copied (see Appendix III). If the referenced CDR does not contain NIL, no register is added.

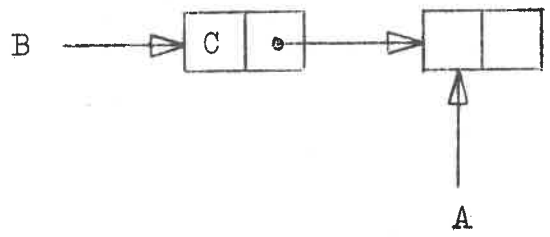
As an example of the use of assignment statements, consider the basic operations of building and altering a list. We can create a list, B, which contains the word "COW" by:

```
A = B.
CAR A = 'C, A = CDR A.
CAR A = 'O, A = CDR A.
CAR A = 'W.
```

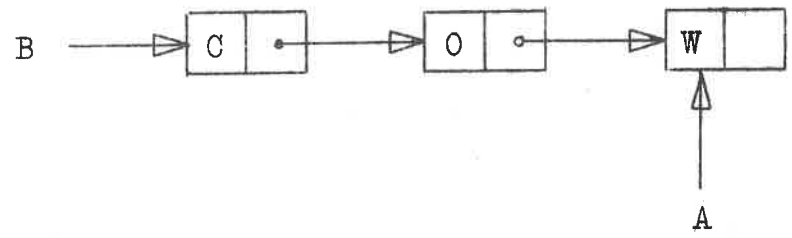
(See Figure 3 for diagrams.) This simple sequence illustrates two important concepts: (a) The use of a "travelling pointer", and (b) obtaining elements from the free list. The word "COW" was to be stored on list B. Since the base register B must always point to the first element of the list, we must use a second base register to point to elements further down the list. In this program, A was chosen to be the "travelling pointer". If we assume that list B was originally in its initial state (i.e. with one empty element attached to the base register), then successive characters required the addition of elements from the free list. This was accomplished by the use of "A = CDR A", as described earlier in this section.



(a) AFTER "A = B."



(b) AFTER THE SECOND LINE



(c) THE COMPLETE LIST

FIGURE 3

CONSTRUCTING A LINEAR LIST

Suppose now that we wish to alter the list shown in Figure 3(c) so that list B contains the word "CROW". We may insert a new element by:

CDR C = CDR B, CDR B = C.

(See Figure 4.) This element is then filled by:

CAR C = 'R.

The above sequence assumes that the first element of C was not involved in any important function elsewhere. Even if this assumption is unjustified the element will be inserted in list B and its CAR set equal to R; the effect on the list structure as a whole will not, however, be that intended. Suppose, for example, that we had used A instead of C. Recall that A is still occupied, pointing to the last element of B (Figure 3(c)). The result is shown in Figure 5.

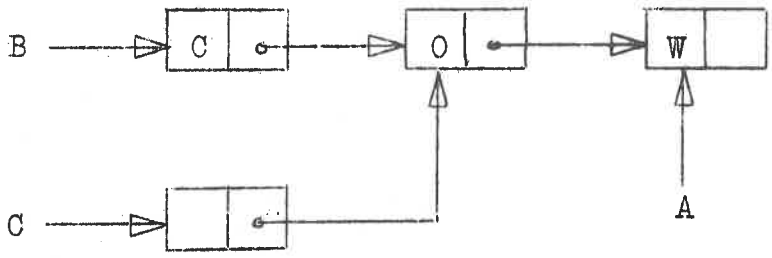
Figure 5 illustrates an important type of list - the circular list. Such constructions serve their purpose, but must be used with care. The reason is that they have no end, and hence may cause many procedures (such as copying a list) to enter infinite loops. The programmer must be aware of the hazards involved, and take suitable precautions when using these structures.

As a final example, suppose that we wish to transform the list "CROW" into "ROW". This is done simply by:

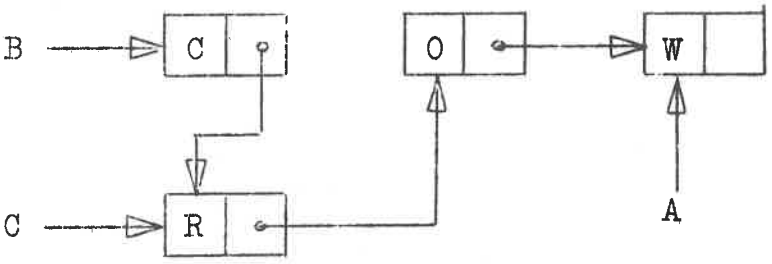
B = CDR B.

Note that no element is obtained from the free list, because CDR B is not empty. The result of this operation is shown in Figure 6.

It is perhaps worthwhile to point out that the original first element of B (i.e. that which contains the character "C") is now completely inaccessible to the programmer. There is no way of "backing up" along a list. It is for this reason that we resorted to the use of the travelling pointer A in the original construction. This point - the most frequent source of errors by novices - cannot be overstressed: any element which cannot be reached from some base register by means of successive applications of * = CAR * and/or * = CDR * cannot be reached at all. Such elements, which do not belong to the free list and cannot be reached by the programmer, are aptly named "garbage". In the course of execution there is an inexorable migration of list elements from the free list to the garbage heap, and if the program goes on long enough this can result in complete exhaustion of the free list. When such exhaustion occurs, a system procedure known as "garbage collection" is invoked. This routine, as its name implies, forms all of the garbage elements into a new free list, and the program can then



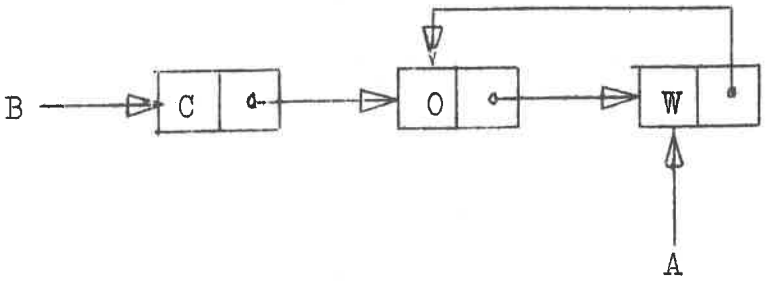
(a) AFTER THE FIRST STATEMENT



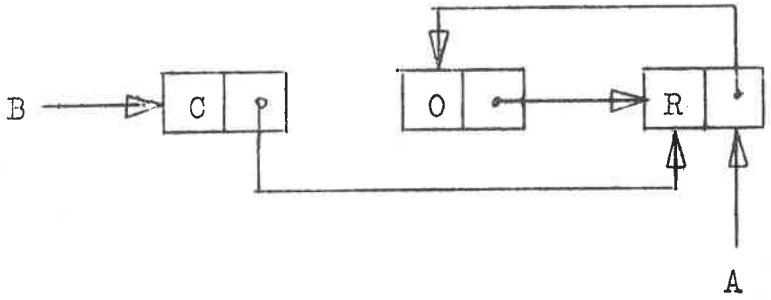
(b) COMPLETED

FIGURE 4

ADDING AN ELEMENT TO A LINEAR LIST



(a) AFTER THE FIRST STATEMENT



(b) COMPLETED

FIGURE 5

RESULTS OF USING "A" INSTEAD OF "C"

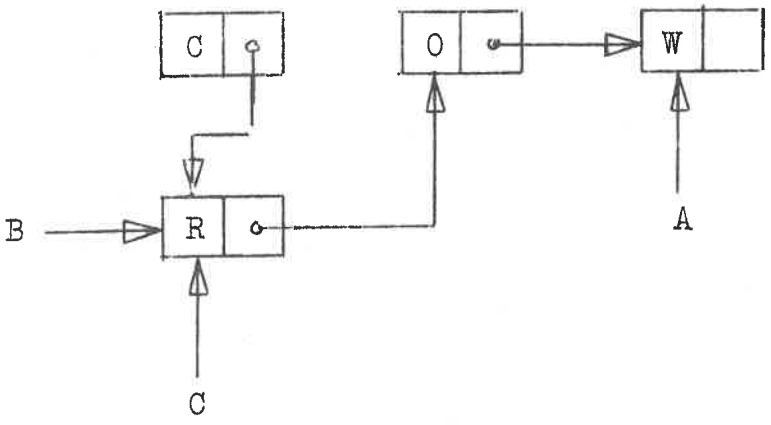


FIGURE 6

DELETING AN ELEMENT

continue. The details[1] are beyond the scope of this section [2].

C. Control Statements

Any Wisp statement may be given a name consisting of any string of alphabetic or numeric characters followed by a delimiter. Control may be transferred to a named statement by the use of the unconditional transfer "TO **.", where ** is a generic term for any name.

Wisp also provides conditional transfers, of the form:

TO ** IF (condition).

A transfer of control to the indicated statement is made when the condition is true, otherwise control continues in sequence. (A list of all available conditions is given in Appendix II (2).) Most conditions resemble assignment statements, and should be self-explanatory. The condition "CAR * = ATOM" ("CDR * = ATOM") is true when the referenced CAR (CDR) either contains a normal atom or is empty (i.e. contains NIL).

The statement "STOP." is used to terminate the execution of a Wisp program. Any pending I/O transfers will be completed.

To illustrate the use of these operations, consider a routine to change all of the commas on a linear list B to periods:

```
START, A = B.
LOOP, TO END IF CAR A ≠ ',,  CAR A = '..
END.
TO OUT IF CDR A = NIL, A = CDR A, TO LOOP.
```

Here the name "START" refers to the statement "A = B.", while "LOOP" refers to "TO END IF CAR A ≠ ',," and "END" refers to "TO OUT IF END LIST A,". The routine is entered by transferring to "START", and goes to "OUT" (not shown) when the entire list B has been scanned. Again A has been used as a travelling pointer. Note that the test for the end of the list must be performed before moving A, because otherwise a new empty element would be added to the end of the list.

FOOTNOTE:

[1] For those familiar with the implementation of other list processors, we note that the garbage collection procedure used by Wisp is not disturbed by the presence of circular lists.

D. Stack Operations

A stack is a list in which only the first element is of current interest. Stacks are used mainly for temporary storage, in connection with recursive subroutines. Any list may be used as a stack, and two operations are provided to manipulate stacks. The statement "PUSH DOWN *." causes an element with an empty CAR to become the first element of list A. The element which was first before the operation becomes the new second element, the old second element becomes the new third, and so forth. In order to preserve all pointers to the top of the stack, PUSH DOWN operates as follows: A new element is taken from the free list, and the CAR and CDR of the current top element of the stack are copied into it. NIL is then placed in the CAR of the top element, and its CDR is set to point to the new element. Thus the address of the top element of the stack remains constant, and all pointers are preserved.

The statement "POP UP *." has the opposite effect: The element which was second before the operation becomes the new first element, and so forth. As in the case of PUSH DOWN, all pointers to the top of the stack are preserved by actually returning the second element to the free list (after copying the contents of its CAR and CDR into the first element).

One caution should be observed with respect to the use of stacks: Normally only the first element of a stack is of interest, and the system described above assumes that all pointers to the stack point to the first element. Since the stack is a list, however, there are many ways in which the programmer can create pointers to its other elements. This should be done with great care, as PUSH DOWN and POP UP can cause these pointers to be pointed to unexpected places.

Often it is necessary to create an empty list. The statement "START LIST *." obtains an empty element from the free list and places its address in the CDR of the indicated base register. Any elements previously pointed to by the base register are left undisturbed. Recall that every list initially contains a single, empty element, and thus it is not necessary to start each list before using it for the first time.

One simple use of a stack is to reverse the order of the elements of a linear list. Suppose that we wish to create a list, B, which is a copy of list A, but in the reverse order. Suppose further that list B has been used elsewhere as a travelling pointer, and hence will not be in its initial state. The following routine would meet these specifications:

```
T = A,  START LIST B.
COPY, CAR B = CAR T.
TO FINISHED IF CDR T = NIL.
PUSH DOWN B.  T = CDR T,  TO COPY.
```

A more important example of the use of a stack is a routine which scans a branched list. There are two ways of scanning a list - "sequence left" and "sequence right". The difference lies in the direction taken at a branch point (i.e. an element whose CAR is not an atom). When sequencing left, the CDR is remembered on a stack and the scan is started on the sublist; when sequencing right, the CAR is remembered and the scan continues along the same list. Suppose that we wish to replace all commas on a branched list, B, with periods:

```
T = B.
CHECK, TO SAVE IF CAR T ≠ ATOM.
TO LOOP IF CAR T ≠ ',,  CAR T = '..
LOOP, TO NEXT IF CDR T = NIL.
     T = CDR T, TO CHECK.
SAVE, CAR S = CAR T.
     PUSH DOWN S, TO LOOP.
NEXT, TO END IF CDR S = NIL.
     POP UP S, T = CAR S, TO CHECK.
```

This particular routine proceeds by sequence right. If a sublist is detected, its pointer is placed on stack S by SAVE. When the current branch is exhausted, the stack is tested by NEXT, and the pointer set to the most recently-encountered sublist. When all sublists have been processed, as evidenced by the stack only having one element, execution ceases by a transfer to END. (Note that this routine assumes that no CDR contains a comma, and that stack S initially contains a single, empty element.)

E. Subroutines

Subroutine linkage is maintained automatically; any subroutine may call any other subroutine, including itself, at any time. The statement "TO ** AND BACK." causes control to be transferred to the indicated statement, after placing the return address on a private stack. Any addresses currently on the stack are pushed down by this operation. A return from the subroutine will be made to the statement immediately following "TO ** AND BACK.".

The statement "RETURN." causes a return to the most recent caller, and the return stack is popped up. If a return must be skipped, the stack may be popped up by the statement "LEVEL DOWN."; control continues in sequence. The state of the return stack may be sensed by the use of the conditional transfers "TO ** IF RETURN." and "TO ** IF NO RETURN."; neither of these operations alters the stack in any way.

F. Input/Output

The standard input to a Wisp program is a series of lines (one line per card on card-oriented machines). The input routine treats the lines as a single, long string of characters, with the first character of the second line immediately following the last character of the first, and so forth. The instruction "CAR * = INPUT." removes the next character from the input string and places it in the CAR of the first element of the indicated list. The divisions between lines are ignored by this operation. Thus if the sequence:

CAR A = INPUT, CAR B = INPUT.

is executed, and the last character of line i is placed in CAR A, then the first character of line i + 1 is placed in CAR B. The programmer can skip to the beginning of a new line by means of the statement "NEXT LINE ON INPUT.".

If a program attempts to read more data than the user has provided, the job will be terminated as though a "STOP." instruction had been executed. This allows the program to read data without explicitly checking for termination, but some care must be exercised. The input must be in the form of a series of logical records, each of which may be completely processed before the next is read. If this is not the case, the last information read will not be processed before termination occurs.

Wisp output, like input, is treated as a single, long string. There are two forms - printed and punched. There is no essential difference in usage between them, so that the two types of statements will be discussed together.

The statement "PRINT CAR *." ("PUNCH CAR *.") places the atom from the CAR of the first element of the indicated list onto the output string. Neither the base register of the indicated list, nor the CAR of the element is altered by this operation. If the contents of CAR * is not an atom, or is equal to NIL, the output request is ignored completely. The contents of a CDR cannot be printed (punched) directly, but must first be transferred to some available CAR. (The reason for this is that the CDR of a list

element is not normally used to store an atom.) A single Wisp atom may be placed in the output string by the statement "PRINT '*." ("PUNCH '*."), where * stands for any atom except NIL.

As soon as one output line overflows, the next is begun (see Chapter III for the length of an output line). There are situations in which the programmer wishes to start a new line before overflow, and this can be accomplished by the statement "FINISH LINE ON PRINT (PUNCH).". This statement causes the current line to be written out immediately, and the next character which is transmitted to the output will appear at the beginning of the next line. Successive executions may be used to skip lines of output.

Finally, the statement "NEW PAGE." causes the printer to be spaced up to the beginning of the next page. There is no counterpart of this operation for the punch.

G. Indirect Referencing

In Section A it was noted that certain lists could not be directly referenced, due to restrictions on the format of Wisp statements. For example, the list name "." can never occur in a Wisp statement, because it would be recognized by the compiler as a delimiter. Thus "A = CAR .." would be interpreted as "A = CAR.", an incomplete statement. Similar restrictions apply to comma, space and quote. These lists may, however, be referenced indirectly.

To understand the mechanism involved, we must first recall that each base register is really a list element. The CAR of the base register normally contains information of interest to the system, and should not be distrubed by the programmer. The CDR contains the pointer, and is the storage location meant when the list name appears in an instruction. Thus, "A = CAR B." places the contents of the CAR of the element pointed to by base register B into the CDR of the base register A. Moreover, an atom is represented internally by the address of its base register. This means that the statement "A = 'B." points the base register A to the base register B. We may therefore perform the impossible operation ". = ," by:

```
A = '., B = ',.
CDR A = CDR B.
```

By means of "A = '.," we have made "CDR A" an alias for ".", and similarly (by "B = ',.") "CDR B" an alias for ",".

The indirect referencing technique is most useful for selecting a list on the basis of some input data. For example,

suppose that we wish to read a list of words (separated by commas and terminated by a period) and place each word on the list corresponding to its first letter. This is easily done by:

```
NEXT, START LIST 1.
READ, CAR 1 = INPUT.  Read a character
TO READ IF CAR 1 = '␣.  Ignore leading blanks
2 = CAR 1.  Point 2 to the base register corresponding to the
            first letter of the word.
2 = CDR 2.  Point it to the first element of the letter's list.
PUSH DOWN 2, CAR 2 = 1.  Add the new word
LOOP, 1 = CDR 1, CAR 1 = INPUT.  Read the next character
TO END IF CAR 1 = '..  Get out if a period is found
TO LOOP IF CAR 1 ≠ ',.  Continue reading to comma
TO NEXT.  Process the next word
END.  Program continues here
```

Any of the operations listed in Appendix II may be used when a base register is the first element of the list except PUSH DOWN and POP UP. It is strongly recommended, however, that the CAR of the base registers be left undisturbed.

H. Integer Arithmetic

Although Wisp was designed as a symbol manipulation language, it is convenient to be able to perform some arithmetic. Therefore, a rudimentary arithmetic capability has been included. There are ten "number stores" (N1, N2, ..., N1Ø) available, each of which may contain a signed integer in the range of the machine (See Chapter III for details).

In order to describe the Wisp Arithmetic instructions, we will let "n" refer to any of the integers between 1 and 1Ø, "i" will refer to any signed integer in the range of the machine, and we will use "op" to represent any of the arithmetic operators addition, subtraction, multiplication, and division. The number stores may be manipulated by means of the Wisp instructions:

```
Nn = i.
Nn = Nn op i.
Nn = i op Nn.
Nn = Nn op Nn.
```

Wisp also contains conditional transfers which enable the programmer to test the contents of number stores. In order to describe these statements, we will let "rel" refer to one of the following relations: equality, inequality, less than, greater than, less than or equal, greater than or equal. The test instructions are:

TO ** IF Nn rel i.

TO ** IF Nn rel Nn.

Integers may be transferred between number stores and lists by means of the instructions:

LIST * = Nn.

Nn = LIST *.

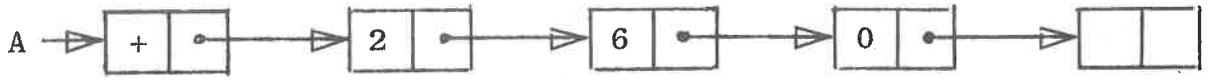
In both of these instructions, the referenced list must be a linear list. The first atom is the sign and the remaining atoms are the digits of the integer with leading zeros removed; the last element of the list contains the atom NIL. For example, suppose that we execute the instruction "LIST A = N1." where N1 contains +260. The resulting list is shown in Figure 7a. If N1 contained the number -124, then the list of Figure 7b would result. In all cases, the contents of Nn are not disturbed.

Lastly, an important warning: Only lists in the correct format will be correctly converted to integers by the statement "Nn = LIST *.". It is the responsibility of the programmer to be sure that the format is correct--the system makes no checks and will attempt to convert any list it is given.

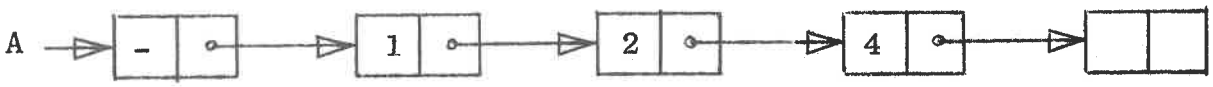
I. Programming Examples

The program shown in Figure 8 is typical of the simple, special purpose editing jobs for which a Wisp routine can be written by a novice in a few minutes. Its function is to compress text by deleting blanks and all characters enclosed in parentheses. In addition to removing these characters, the program ignores the original line divisions and packs the text into full lines. For example, the text shown in Figure 9(a) would appear as shown in Figure 9(b) after processing. It is assumed that parentheses must be balanced, although there is no limitation to the depth to which they may be extended. We do not require that the program check for parenthesis imbalance, and its behavior on encountering such an error is arbitrary.

Notice how this program keeps track of the depth of the parenthesis nest by the depth of recursion in the subroutine "DOWN": a left parenthesis causes recursion, while a right parenthesis



(a) The List Form of the Integer 260



(b) The List Form of the Integer -124

FIGURE 7

Examples of Integer Lists

```
READ, CAR I = INPUT.  READ THE NEXT CHARACTER.
TO LPAREN IF CAR I = '(.  ARE WE ENTERING A PAIR OF PARENTHESES, YES.
TO READ IF CAR I = ' .  NO, ELIMINATE A BLANK.
PUNCH CAR I, TO READ.  PUNCH OUT A NONBLANK CHARACTER.
LPAREN, TO DOWN AND BACK, TO READ.
DOWN, CAR I = INPUT.
TO BACK IF CAR I = ').  HAVE WE FOUND A MATCHING RIGHT PAREN, YES.
TO DOWN IF CAR I ≠ '(.  NO, ARE WE ENTERING A NEW PAIR OF PAREN-
                                                    THESES, NO.
TO DOWN AND BACK, TO DOWN.  YES, ELIMINATE ITS CHARACTERS.
BACK, RETURN.  MOVE BACK TO THE PREVIOUS LEVEL.
.       VOID LINE TO SIGNAL THE END OF THE PROGRAM.
```

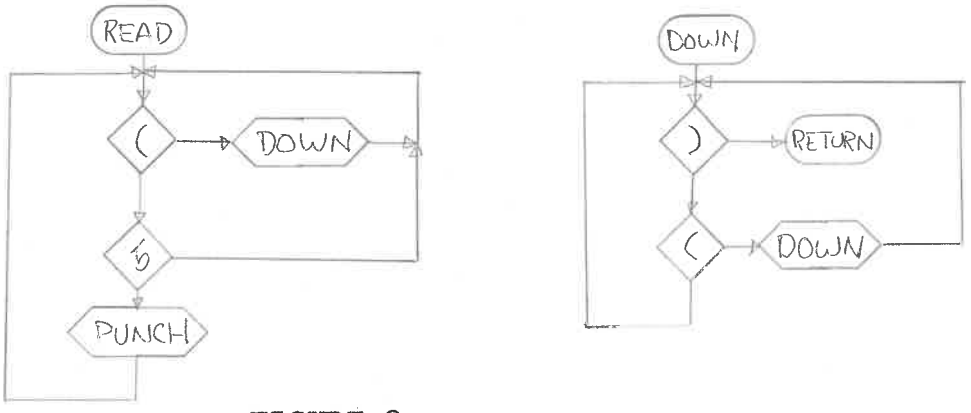


FIGURE 8

A SIMPLE WISP PROGRAM

```
TEXT    (JUNK).
GOOD    (BAD  (TO  (ANY  (DEPTH)  OF)  COURSE)).
```

(a) UNCOMPRESSED TEXT

```
TEXT. GOOD.
```

(b) COMPRESSED TEXT

FIGURE 9

ILLUSTRATION OF TEXT COMPRESSION

moves the routine up one level. Notice also that no character is considered a terminator. This means that the program will continue to read data until no more is available (See Section F).

Another way of remembering the depth of the parenthesis nest is by keeping a count of the number of unmatched left parentheses. When a left parenthesis is encountered, the count is incremented; when a right parenthesis occurs, it is decremented. The text editing program of Figure 10 uses this method to obtain the same results as the program of Figure 8.

One of the major applications of Wisp has been to the manipulation of expressions--analytic differentiation, algebra, symbolic logic, and so forth. For such problems it is convenient to store the expressions in prefix form: Each expression is represented by a list whose first element contains an atom defining the type of expression; subsequent elements contain pointers to the operands. (Figure 11 gives several examples of algebraic expressions stored in this way.) Normally, operators used in these expressions belong to a hierarchy which dictates the order in which they are to be performed. In algebra, for example, multiplications are performed before additions, exponentials before multiplications, and so forth. We shall say that operator A is stronger than operator B if their positions in the hierarchy are such that A's should be performed before B's.

Our second programming example (Figures 12 and 13) is a subroutine which reads a string representing an expression and stores it in prefix form. The variables and operators are single characters, and the expression is terminated by a period. The set of operators is defined by a linear list 0. This list contains the operators in order of decreasing strength, and the CAR of the last element contains a period. Only binary operators (i.e. operators which require exactly two operands) are allowed.

The routine is entered by "TO INPUT AND BACK." and returns to the following statement with CAR T pointing to the expression. If CAR S contains a period, the expression read was correct, otherwise an error was detected. In any case, the string form of the expression is stored on list I.

The algorithm used is the standard one for solving this problem: Two stacks, one for expressions (T) and one for operators (S) are used. The input string must contain operators and operands alternately, because of the restriction to binary operators. When an operand is read, it is placed on the top of T, all other entries moving down one place. When an operator is read, it is compared with the top entry of S to see if its strength is greater. If so, it is pushed onto S, and the scan continues. If not, the top two entries of T and the top entry of S are used to form an expression. This expression is then placed on T, and the comparison is made again.

```
N1 = 0.  Set the initial bracket count.
READ, CAR I = INPUT.  Read the next character.
TO LPAREN IF CAR I = '(.
TO READ IF CAR I = ' .  Eliminate a space.
PUNCH CAR I, TO READ.  Punch anything else.
LPAREN, N1 = N1 + 1.  Increment the bracket count.
READ2, CAR I = INPUT.  Read the next character.
TO LPAREN IF CAR I = '(.
TO READ2 IF CAR I ≠ ').
N1 = N1 - 1, TO READ2 IF N1 > 0.
TO READ.
```

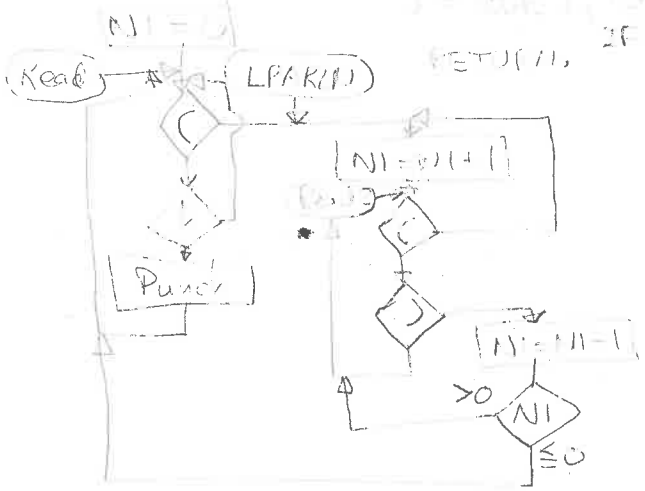
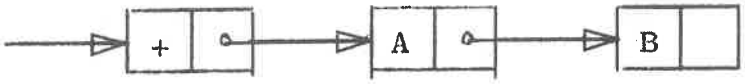
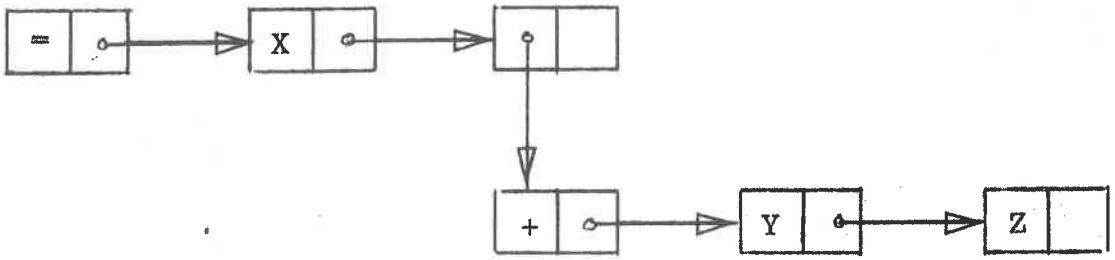


FIGURE 10

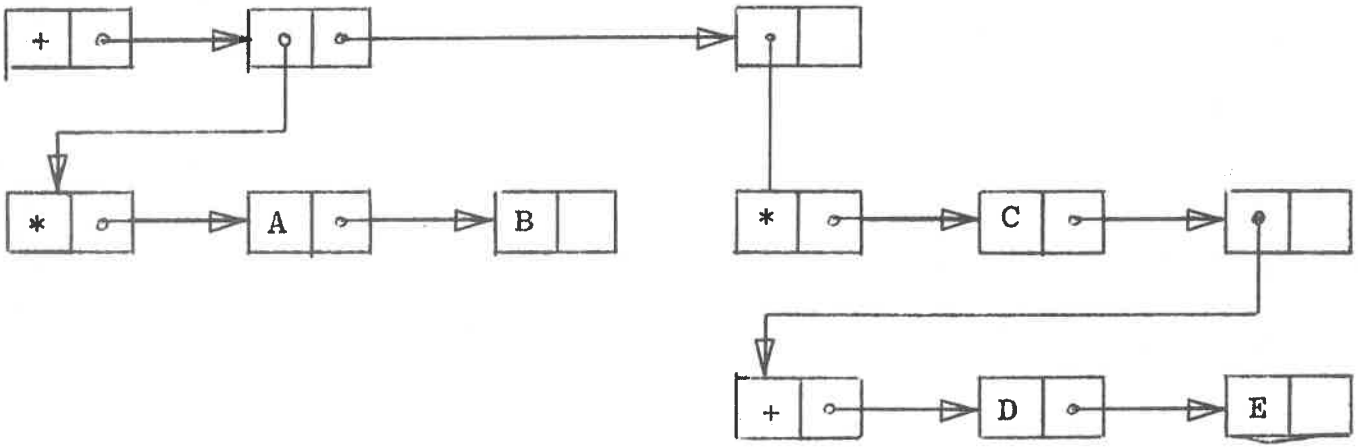
Illustrating the Use of Number Stores



(a) A + B



(b) X = Y + Z



(c) A * B + C * (D + E)

FIGURE 11

Examples of Algebraic Expressions

```
INPUT.  READ AN EXPRESSION, AND PUT IT IN PREFIX FORM.
   START LIST T.  START THE OPERAND STACK.
   START LIST S, CAR S = '..  START THE OPERATOR STACK.
   START LIST I, Z = 'I.  START THE INPUT LIST.

SUBEXP.  CONVERT A SUBEXPRESSION AND STORE IT IN CAR T.
   TO READ AND BACK.  GET THE NEXT NONBLANK CHARACTER INTO CAR Z.
   TO LPAREN IF CAR Z = '(.  IS THIS OPERAND AN EXPRESSION, YES.
   TO ERROR IF CAR Y = CAR Z.  NO, IS THIS CHARACTER AN OPERATOR, YES.
   PUSH DOWN T, CAR T = CAR Z.  NO, PLACE IT ON STACK T.

OPERATOR.  PLACE AN OPERATOR ON STACK S.
   TO READ AND BACK.
   TO RPAREN IF CAR Z = ').  IS THIS THE END OF A SUBEXPRESSION, YES.
   TO ERROR IF CAR Y ≠ CAR Z.  NO, IS THE CHARACTER AN OPERAND, YES.
   TO STACK IF CAR X = 'I.  NO, IS THE TOP ELEMENT OF S STRONGER, NO.

FIXEXP, TO UNSTACK AND BACK.  YES, CREATE AN EXPRESSION.
   TO CHECKOP AND BACK.  CHECK THE STRENGTH OF THE NEXT ELEMENT OF S.
   TO FIXEXP IF CAR X = 'S.  IS IT STRONGER, YES.

STACK, TO BACK IF CAR Z = '..  HAVE WE COME TO THE END, YES.
   PUSH DOWN S, CAR S = CAR Z.  NO, STACK THE OPERATOR.
   TO SUBEXP.  GET THE NEXT OPERAND.

READ.  POINT Z TO THE NEXT NONBLANK CHARACTER.
   Z = CDR Z, CAR Z = INPUT.
   TO READ IF CAR Z = ' .

CHECKOP.  CHECK CAR Z AGAINST THE LIST OF OPERATIONS.
   CAR X = 'I, Y = '0.

SCAN1, Y = CDR Y.  SCAN THE OPERATOR LIST.
   TO BACK IF CAR Y = CAR Z.  IS CAR Z STRONGER THAN CAR S, YES.
   TO STACKOP IF CAR Y = CAR S.  IS CAR S STRONGER THAN CAR Z, YES.
   TO SCAN1 IF CDR Y ≠ NIL, RETURN.
```

FIGURE 12

ROUTINE TO READ AN EXPRESSION

```
STACKOP, CAR X = 'S.  SET A SWITCH
   TO BACK IF CDR Y = NIL.

SCAN2, Y = CDR Y.  TRY TO FIND CAR Z ON THE OPERATOR LIST.
   TO BACK IF CAR Y = CAR Z.
   TO SCAN2 IF CDR Y ≠ NIL.

BACK, RETURN.

LPAREN.  THE OPERAND IS A SUBEXPRESSION.
   PUSH DOWN S, CAR S = '(.  SET A MARKER IN THE OPERAND STACK.
   TO SUBEXP.

RPAREN.  COMPLETE A SUBEXPRESSION.
   TO NOPAR IF CAR S ≠ '(.  HAVE WE USED ALL OF THE OPERATORS, NO.
   POP UP S, TO OPERATOR.  YES, GET RID OF THE MARKER.

NOPAR, TO ERROR IF CAR S = '..  WAS THERE A MATCHING LEFT PAREN, NO.
   TO UNSTACK AND BACK, TO RPAREN.

UNSTACK.  CREATE A SUBEXPRESSION.
   START LIST Y.
   CAR Y = CAR T, POP UP T.  SECOND OPERAND,
   PUSH DOWN Y, CAR Y = CAR T.  FIRST OPERAND,
   PUSH DOWN Y, CAR Y = CAR S.  OPERATOR.
   POP UP S, CAR T = Y.  PLACE THE COMPLETE EXPRESSION ON STACK T.
   RETURN.

ERROR, CAR S = 'E, RETURN.  SET THE ERROR FLAG AND GET OUT.
```

FIGURE 13

ROUTINE TO READ AN EXPRESSION
(CONTINUED)

Notice the use of the fact that '0 is the address of the base register 0 (see subroutine "CHECKOP"). It is desirable to have loop "SCANI" begin with the statement "Y = CDR Y.", in order to save a jump instruction. This means that if we initialize Y by "Y = 0.", the first character of 0 will not be checked (Why?). Initializing Y by "Y = '0." avoids this difficulty.

The following exercises are presented so that the reader may test his understanding of Wisp programming. They are arranged in increasing order of complexity, and a solution is given for the last one:

1. Write a program to encipher text by simple substitution of the cipher letter for the clear letter. You should begin by reading two lines which describe the cipher to be used. For example:

   ABCD ... Z

   ZYXW ... Z

   The first line defines the clear alphabet, the second the cipher alphabet.

   The clear text will follow the definition of the cipher alphabet, and will be terminated by a line whose first character is a slash. All nonalphabetic characters (e.g. space, comma, period, etc.) should be preserved by the program.

2. Alter your program to make the cipher more difficult to break as follows: Instead of preserving spaces, encipher them as though they were the letter Q, suppress all punctuation and output the cipher in five-letter groups. If the last group is not complete, encipher as many of the letters Q, Z, X, J (in order) as necessary to complete it.

3. Write a program to solve the puzzle known as the "Tower of Hanoi." This puzzle consists of three pins, set upright in a board. Initially the left-hand pin holds a number of flat discs of varying sizes, the largest being at the bottom and the smallest on top. These discs are to be moved to the right-hand pin, according to the following rules:

   (1) A move consists of removing the top disc from one pin and placing it on another pin.

   (2) Only one disc may be moved at a time.

   (3) No disc may be placed on a smaller disc.

The number of discs should be read as an input parameter, and will consist of a single digit. You should have some means of printing out the sequence of moves if desired.

4. Write a program to differentiate an algebraic expression which employs only the operators "+" and "*", and has been placed in prefix form by the routine "INPUT." of Figure 12.

5. Write a program which will print an expression which is given by a list T in prefix form (as created by "INPUT."). The resulting expression should contain only essential parentheses.

The solution of problem 5 shown in Figures 15 and 16 is due to R. Evans, a student in the Basser Computing Department, University of Sydney, Sydney, Australia. The atom "$" is used to represent exponentiation, "*" to represent multiplication. The operands of a prefix expression can be either atoms or subexpressions. If they are subexpressions, they must be parenthesized only in certain cases (see Figure 14 for examples). The strategy in any particular instance depends on the strength of the operators involved and whether the subexpression is the first or second operand of its main expression.

The decision on whether or not to insert parentheses is made by the routine "SUBXA" in the case of the first subexpression, and by "SUBXB" for the second. The program retains the operator of the main expression on a stack S, while the traveling pointer T addresses the operator of the subexpression. Q points to an atom giving the hierarchy of the operator addressed by T, as shown in Table I. CAR X contains the atom "I" if the operator of the subexpression has a higher strength than that of the main expression.

The information regarding relative operator strengths is obtained by the routine "OPRAT." Two lists, O and P are available which contain respectively the operators in decreasing order of strength and the hierarchy of each operator. The traveling pointers W and Q are used to address these lists, and CAR X is set to reflect the relative strengths of the main operator and that of the subexpression.

The remainder of the routine serves to sequence through the tree containing the prefix expression. It is assumed that this expression is initially addressed by CAR T, with CDR T equal to NIL. The stack Y is used as temporary storage during the sequence left scan of the tree.

Prefix form	Prefix form
* + ABC	(A + B) * C
+ * ABC	A * B + C
- + ABC	A + B - C
- C + AB	C - (A + B)
*/ABC	A/B * C
/*ABC	A * B/C
/A * BC	A/(B * C)
/*AB * CD	A * B/(C * D)

FIGURE 14

Expressions with only Essential Parentheses

Operator addressed by T	Atom addressed by Q
$	1
*	2
/	2
+	3
-	3

TABLE I

OPERATOR STRENGTHS

```
OUTPT,  TO ASSEM IF CAR T .NE. ATOM.      IS THE EXPRESSION A SINGLE OPERAND, NO.
        PRINT CAR T.                      YES, WRITE IT.
END,    PRINT '., RETURN.                 TERMINATE THE EXPRESSION.
ASSEM,  TO BRNCH AND BACK, TO END.        PROCESS A COMPLEX EXPRESSION.
BRNCH,  TO SAVE IF CDR T .NE. NIL.        IS THIS THE FIRST OF A PAIR, YES.
        TO SUBXB AND BACK.                NO, MOVE ONTO THE OPERAND EXPRESSION.
        T = CDR T.                        STEP ALONG TO ITS FIRST OPERAND.
        TO SKIPA IF CAR T = ATOM.         IS IT A SIMPLE VARIABLE, YES.
        TO BRNCH AND BACK.                NO, PROCESS IT RECURSIVELY.
LOOK,   TO BACK IF CAR Y = NIL.           ARE THERE MORE TO BE FINISHED, NO.
        T = CAR Y, POP UP Y.              YES, GO TO THE MOST RECENT.
        PRINT CAR S.                      PUT OUT THE DEFERRED OPERATOR.
        TO SKIPB IF CAR T = ATOM.         IS IT A SIMPLE VARIABLE, YES.
        TO BRNCH AND BACK.                NO, PROCESS IT RECURSIVELY.
        POP UP S, TO LOOK IF CAR S .NE.   ').
        PRINT CAR S, POP UP S, TO LOOK.   CLOSE OFF A SUBEXPRESSION.
SAVE,   PUSH DOWN Y, CAR Y = CDR T.       SAVE THE SECOND OPERAND FOR LATER.
        TO SUBXA AND BACK.                MOVE ONTO THE FIRST OPERAND.
        T = CDR T.                        STEP ALONG TO ITS FIRST OPERAND.
        TO SKIPA IF CAR T = ATOM.         IS IT A SIMPLE VARIABLE, YES.
        TO BRNCH AND BACK, TO LOOK.       NO, PROCESS IT RECURSIVELY.
 KIPA,  PRINT CAR T, PRINT CAR S.         PUT OUT THE VARIABLE AND OPERATOR.
?K      T = CDR T.                        MOVE TO THE SECOND OPERAND.
        TO SKIPB IF CAR T = ATOM.         IS IT A SIMPLE VARIABLE, YES.
        TO BRNCH AND BACK.                NO, PROCESS IT RECURSIVELY.
        POP UP S, TO LOOK IF CAR S .NE.   ').
        PRINT CAR S, POP UP S, TO LOOK.   CLOSE OFF A SUBEXPRESSION.
SKIPB,  PRINT CAR T.                      PUT OUT THE VARIABLE.
        POP UP S, TO BACK IF CAR S .NE.   ').
        PRINT CAR S, POP UP S, RETURN.    PUT OUT A ).
```

FIGURE 15

ROUTINE TO PRINT ONLY ESSENTIAL PARENTHESES

```
SUBXA,  T = CAR T, TO OPRAT AND BACK.    DECIDE ON PAREN INSERTION.
        TO NOBR IF CAR X = 'I.                (FIRST SUBEXPRESSION)
        TO BACK IF CAR S = 'S.
        TO NORR IF CAR Q = '2.
        TO BRAK IF CAR S = '/.
        TO BRAK IF CAR S = '*.
        TO NOBR.
SUBXB,  T = CAR T, TO OPRAT AND BACK.    DECIDE ON PAREN INSERTION.
        TO NOBR IF CAR S = NIL.               (SECOND SUBEXPRESSION)
        TO BRAK IF CAR S = 'S.
        TO NORR IF CAR S = '+.
        TO BRAK IF CAR Q = '3.
        TO NORR IF CAR S .NE. '/.
        TO NORR IF CAR Q = '1.
BRAK,   PRINT '(, PUSH DOWN S, CAR S = ').    INSERT '(' AND STACK ')'.
NOBR,   PUSH DOWN S, CAR S = CAR T, RETURN.   STACK THE OPERATOR.
OPRAT,  W = 'O, Q = 'P, CAR X = 'I.      DECODE THE RELATIVE OPERATOR STRENGTH.
 ANA,   W = CDR W, Q = CDR Q, TO BACK IF CAR W = CAR T.
SC      TO SWTCH IF CAR W = CAR S.
        TO SCANA IF CDR W .NE. NIL, RETURN.
SWTCH,  CAR X = 'S, TO BACK IF CDR W = NIL.
SCANB,  W = CDR W, Q = CDR Q, TO BACK IF CAR W = CAR T.
        TO SCANB IF CDR W .NE. NIL, RETURN.
```

FIGURE 16

ROUTINE TO PRINT ONLY ESSENTIAL PARENTHESES
(CONTINUED)

III. IMPLEMENTING BASIC WISP

The most promising approach to a programming system which can be rapidly and easily transferred from one machine to another seems to be the technique known as "bootstrapping": As much of the programming system as possible is written in a machine-independent problem-oriented language suited to the tasks for which the system is designed. The only program which must be rewritten is then the compiler for this problem-oriented language.

The difficulty with the bootstrapping approach is in finding a problem-oriented language with sufficient power and generality to describe the entire programming system, and yet which is simple enough to be compiled by a naive compiler. The twin goals of naive compiler and powerful language appear to be incompatible, and hence the bootstrapping process must be carried out in several stages. The first stage will be designed to make the compiler as simple as possible. This very simple compiler will compile a more complex program which in turn will compile a still more complex one and so on.

Basic Wisp is amenable to a two-step bootstrap process: A very naive compiler known as SIMCMP is the basis of the system, and it compiles the Basic Wisp compiler BASCMP. Both of these programs operate without the full dynamic storage allocation present in Basic Wisp. BASCMP may be used to compile a full dynamic storage allocation package, which is itself written in Basic Wisp (with some additional statement types).

The remaining sections of this chapter describe the bootstrapping process in detail, and the coding of the two compilers is given in the Appendices.

A. SIMCMP

The design goal of SIMCMP is to provide a compiler of minimum length and complexity which is still sufficiently powerful to support the next step in the bootstrap process. Experience indicates that the best overall approach to this problem is that of a simple macro processor. The macro processor accepts definitions of source language statements in terms of machine code translations and stores them in a table. It then reads program text and compares each line with all of the entries in the table. When a match occurs, the machine code translation of the macro is punched with suitable parameter substitution. SIMCMP is able to translate any source language which can be expressed as a series of simple substitution macro calls with single character parameters.

In a conventional macro processor, a macro definition has the form:

MACRO NAME(P_1,P_2, ...,P_n)
Code Body
END

The words MACRO and END serve to delimit the macro definition, NAME is the name of the macro, and P_1,P_2, ...,P_n are formal parameters. The macro is called by a line which has the form "NAME(A_1,A_2, ...,A_m)" ($m \leq n$). When this call is encountered in the program text, the code body of the macro "NAME" is evaluated with the values of $A_1 \ldots A_m$ substituted for $P_1 \ldots P_n$. If m is less then n, then values for actual parameters $A_{m+1} \ldots A_n$ are created by the processor in some regular way.

The line which introduces a SIMCMP macro definition may be any arbitrary character string, with parameters indicated by a special symbol called a parameter flag. We refer to this line as the template. Use of a template essentially allows the name part of a traditional macro to be replaced by a "distributed name" consisting of all portions of the line except the parameter flags. Each character of the template other than parameter flags must match the corresponding character of the input line exactly. A parameter flag, however, may match any single character. The template line can therefore be thought of as a mask consisting of literal characters interspersed with "holes" which may correspond to arbitrary characters in the input string being matched. The character matched by a given parameter flag automatically becomes the value of the actual parameter corresponding to the given formal parameter. This method of determining the actual parameters of a macro call is less direct than that of listing the actual parameters, but is the natural generalization of conventional actual parameter specification to the case where a macro has a distributed name and parameters are specified without delimiters.

By restricting the free format of the statements in Basic Wisp, it is possible to translate them using SIMCMP. The restrictions which are needed are:

1) Only one statement may be written on a line.
2) The statement must begin at the first character of the line, and must terminate with a period.
3) No superfluous spaces are allowed--the statements must have exactly the form shown in Appendix II.
4) Labels are restricted to two-digit integers.
5) The construction '. is not allowed.

The standard form test program of Appendix V and the Basic Wisp Compiler (Appendix V (3)) are examples of programs intended to be

translated by SIMCMP. The operations listed in Appendix II, Sections 1 through 5, are specified by their SIMCMP templates. The parameter flag is "*".

The code body of a SIMCMP macro is made up of lines in the target language. Each line consists of a series of literal characters and calls on actual parameters. A call on an actual parameter is signaled by a special character called a machine language parameter flag. This flag is followed by two digits, the first of which specifies the number of the formal parameter in the template, counting from the left. (A maximum of nine formal parameters, numbered 1-9, may be specified in a single template.) The second digit of the parameter call defines the type of conversion to be used in the particular substitution instance of the formal parameter. Figure 17 gives examples of complete SIMCMP macros for two machines on which Wisp has been implemented.

When an input has been matched to a given template, the lines forming the associated code body are punched out. Literal characters are punched exactly as they appear in the code body, and calls on formal parameters are replaced by the values specified by the conversion digit and the actual parameter value. Two conversion types are available, indicated by the digits 0 and 1. Type 0 conversion is a direct copy of the actual parameter into the output string; type 1 conversion replaces the actual parameter by a decimal integer. The details of type 1 conversion will be discussed later in connection with the implementation of the Wisp list structure.

With some target languages it is useful for SIMCMP to be able to generate arbitrary, unique symbols. This requirement is satisfied by the use of parameter zero. Up to ten unique symbols may be generated in any single macro by referring to parameter zero with conversions 0-9. These unique values are three-digit integers, the lowest of which is 100. The integers are supplied by a "location counter" which is maintained by SIMCMP. This location counter is initially set to the value 100 and, subsequent to the execution of a macro which contains references to parameter 0, its value is incremented by one more than the highest conversion digit used. This procedure ensures that the location counter will produce a unique set of symbols for each macro call. The ability to produce unique symbols is not necessary for every programming language. Therefore, instructions for deleting this feature have been included in the version of SIMCMP which appears in Appendix IV. An example of the use of parameter 0 is given in Figure 18, which shows a macro whose target language is Fortran II. The problem is that three labels must be specified for the IF statement. When the condition is not satisfied, we wish control to pass to the next statement, which must be given a label. SIMCMP produces this label automatically by means of parameter 0.

```
* = CAR *.
        CLA* LISTS+(21)
        ARS  18)
        STA  LISTS+(11)
)
TØ **.
        TRA  Z(10(20)
)
```

a) IBM 7040/44/90/94

MCT parameter flag = "("
MCT end-of-line flag = ")"

```
* = CAR *.
V:21P993;=M15;V:11P993;MOM15;.
SHL-16;SHLD-16;ERASE;SHC+16;.
=V:11P993;.
.
TØ **.
J:10:20;.
.
```

b) English Electric KDF9

MCT parameter flag = ":"
MCT end-of-line flag = "."

FIGURE 17

EXAMPLES OF SIMCMP MACROS

```
TØ ** IF CAR * = CDR *.
      I=CDR('31).
      J=CDR('41).
      IF(CAR(I)-CDR(J))'00,'10'20,'00.
 '00  CØNTINUE.
```

```
MCT parameter flag = "'"
MCT end-of-line flag = "."
```

FIGURE 18

EXAMPLE OF THE USE OF PARAMETER ZERO

The structure of SIMCMP places restrictions on both the source text and the target language with which the program is used. Restrictions on the source text are not important, since SIMCMP's only mission is to compile the next compiler in the bootstrap sequence. BASCMP is written in the restricted form of Basic Wisp described above, for which SIMCMP is adequate.

Restrictions on the target language are more serious, and here every effort was made to generalize the program. The target language will usually be a one-for-one assembly code, and SIMCMP requires that this assembly code be capable of assigning values to symbolic addresses which it generates. These symbolic addresses will be integers with or without alphabetic characters preceding or following them. The assembler must also have facilities for reserving at least one block of storage if SIMCMP is to be used to bootstrap a more complex programming system. Finally, it is desirable for the assembler to provide a symbol-defining pseudo operation which results in no code generation. This feature is not absolutely necessary and can be omitted if an instruction which has no effect (a "no-op") is available. The reason for this latter requirement is that BASCMP considers the definition of a label as a complete statement.

The program given in Appendix IV is written in Fortran. This approach was taken for two reasons: 1) Fortran is a generally available language, and hence any programming system based on SIMCMP is available wherever Fortran is available; 2) Fortran programs (at least of this simplicity) have a structure which is similar to the structure of a machine-language program on most contemporary machines. The Fortran version of SIMCMP is therefore easily translated by hand into machine language for a machine on which no Fortran compiler exists.

Because Fortran has no facilities for character manipulation, we map characters into non-negative integers for processing. The only assumption made about the mapping is that the digits 0 through 9 are converted into consecutive integers. This means that if we subtract the integer corresponding to the character zero from the integer corresponding to any digit, the result will be an integer equal to the digit. The mapping requires three "environment routines", one of which reads a character stream and supplies the corresponding integers one at a time. The other two accept integers and convert them into characters, one routing its output to the print stream and the other to the punch.

The data storage for SIMCMP consists of a single integer array and six temporary storage locations. The array must be large enough to hold all of the macro definitions (stored one character per word) and a single input line. 15 locations in the array plus two words per macro definition are used to hold control information.

The SIMCMP translator has two main phases: macro definition and macro expansion. During the macro definition phase, user-defined macros are read and stored in the array. The first line of the set of macros is called a flag line and contains five control characters. The first is the source language end-of-line flag, a character which marks the end of translatable information on an input line. The second character is the source language parameter flag which is used in the template to indicate a formal parameter. The third and fourth characters of the flag line are the MCT end-of-line and the MCT parameter flags respectively. The first marks the end of useful information on a code body line; the second indicates that a converted actual parameter should be inserted in the machine code output. The fifth character on the flag line must be the character zero. (For translating BASCMP, the source language end-of-line flag is the period, and the source language parameter flag is the asterisk. The MCT flags depend upon the target language.) Following the flag line is a series of macro definitions. Each macro definition is terminated by a line whose first character is the MCT end-of-line flag. SIMCMP expects the next line to be a new template. If a particular macro is the last, then the first two characters of its terminating line are machine code end-of-line flags. (Figure 19 gives an example of typical data for SIMCMP.)

After reading all of the macro definitions, SIMCMP enters the expansion phase and reads a single source statement into its array. Reading is terminated by recognition of a source language end-of-line flag. The statement is then translated by successively matching it against each of the defined templates. First, however, a check is made to ensure that the source line is not void (a void line terminates the program). During the matching process, a character which is matched against a source language parameter flag in the template is placed in the corresponding parameter storage word. If all characters are successfully matched, the input line is accepted as a call on the macro whose template is currently being scanned. If SIMCMP fails to find a match for some character of the input line, it attempts to match this input line to the next template. If there are no more templates, the line is assumed to be an error, and the transfer occurs to an error routine which prints the entire line. Such an error results in no code being transmitted to the output but does not stop the expansion phase. After printing out the error, control is transferred to the reading routine to get the next input line.

Since the templates are scanned in order, SIMCMP will assume that an input line which matches more than one is a call on the one which appears first. It is therefore the responsibility of the programmer to ensure that the ordering of macro definitions will not produce strange results.

```
.*)(0
```

Flags:
- Source language end-of-line = "."
- Source language parameter = "*"
- MCT end-of-line = ")"
- MCT parameter = "("

```
* = '*.
  "
  "
  "
))
```

Macro definitions

```
CAR A = INPUT.      BEGIN PROGRAM
      "
      "
      "
.
```

Program

FIGURE 19

DATA FOR SIMCMP

B. The Wisp List Structure

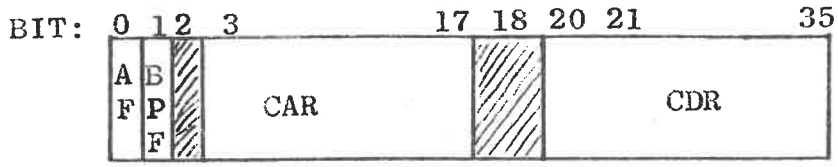
Recall from Chapter I that the basic storage unit of Wisp is the list element. Each list element contains pointers to two other list elements (each of which could be a base register). In any realization of Wisp, each list element will be referred to by its address, a unique integer which specifies the particular element. A pointer to an element is merely its address; when we say that each list element contains two pointers, we mean that it contains the addresses of two other list elements. In a conventional machine, therefore, enough storage to contain two addresses must be allocated to each list element. In addition, there must be room for two flags: the atom flag (AF), used to mark a base register, and the branch point flag (BPF), used during garbage collection.

There are many ways to define such a structure. Perhaps the simplest is to use two integer arrays, CAR(I) and CDR(I). This method is employed in a Fortran II implementation, and has also been used successfully with other list processing languages [7]. The address of an element is simply its index in the array, the atom flag is the sign of CAR(I), and the branch point flag is the sign of CDR(I).

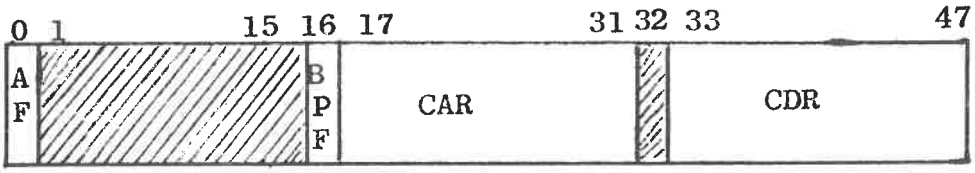
In general, Wisp is implemented in the assembly language of the target machine, rather than a higher level language such as Fortran. No array structure is available in assembly language, and the list elements must be specified in terms of machine words. Two such specifications are shown in Figure 20. Here the address of an element is its machine address, and the flags are specific bits which are set to 0 or 1. Implementation on a decimal machine would be similar, except that the flags might require a full digit.

Base registers should occupy specific memory positions, since the Wisp program must refer to all elements indirectly via the base registers. Double indirection (which would be required if the base register locations were unknown at compile time) wastes time and generally requires more coding than single indirection. Because of the fact that the base register positions should be fixed at compile time, they normally occupy the first locations of the list structure.

If the list space is defined as an array, with addressing being accomplished by an index to the array, then the base registers must be a part of the array. If, on the other hand, machine addresses are used to identify the list elements, then the base registers need not be part of the contiguous block of storage assigned to the regular elements. The reason is that a pointer can be any machine address, thus effectively including the entire memory in the list space. This property of the pointers is used to make the largest possible space available to the list structure by deferring storage allocation until after the program is loaded [5]. Then all of the memory which



(a) IBM 7040/7090



(b) ENGLISH ELECTRIC KDF9

AF = ATOM FLAG

BPF = BRANCH POINT FLAG

HATCHED AREAS ARE NOT USED BY BASIC WISP

FIGURE 20

EXAMPLES OF LIST ELEMENT LAYOUT

is not occupied by the program can be allocated to the list structure without requiring the programmer to know a priori how much memory the program will require. The base registers will be a part of the program space so that they may be referenced directly.

Base registers have exactly the same layout as ordinary list elements, except that the atom flag of each one which corresponds to a valid Wisp atom must be set. This flag allows for the implementation of the atom test (TØ**IF CAR* = ATØM.), as well as indicating the lists which must be traced prior to garbage collection. Base registers of private lists should not have their atom flags set. The CDR field of each base register points to the list addressed by the register, and the CAR field points to NIL.

There must be one base register corresponding to each character which is acceptable to the machine, and at least three others --one for NIL, one for the free list and one for the return stack. It will be convenient to define other private lists for various system functions, and these will also require base registers. The ordering of the base registers is normally determined by I/O considerations. Recall that the environment routines must convert the normal computer representation of a character into the address of its base register. This is done by converting it to a non-negative integer (see the description of the routine INPUT(i), in the previous section) and adding that integer to the address of the first base register. Naturally this conversion should be done as rapidly as possible, and usually some particular ordering can be converted faster than any other. For example, on the IBM 7040/7090 series machines the characters are represented internally by groups of six bits as shown in Table II. The obvious ordering of the base registers on these machines is also shown in Table II. With this ordering, conversion consists merely of adding the address of the symbol LISTS to the representation of the character.

Notice in Table II that the sequence of base registers has gaps. Base registers in these positions do not correspond to characters and cannot be used in any way by the program. We may therefore use them as the base registers of the private lists (such as the free list and the return stack). It is usual to put the NIL base register at one end of the base register group to facilitate testing for valid output requests. (Recall that a request to output NIL is ignored, as is a request to output a non-atom.)

In the initial state, there is one element of the list space attached to each base register corresponding to a character. NIL points to itself, the return stack points to NIL, and all of the remaining list elements are formed into a linear list which is attached to the free list base register. This initial state is produced by a routine which is a part of the Wisp system, and will be discussed in detail in Section III(D).

Character	6 Bit Representation	Base Register Address
0	000000	LISTS
1	000001	LISTS+1
2	000010	LISTS+2
3	000011	LISTS+3
4	000100	LISTS+4
5	000101	LISTS+5
6	000110	LISTS+6
7	000111	LISTS+7
8	001000	LISTS+8
9	001001	LISTS+9
=	001011	LISTS+11
'	001100	LISTS+12
+	010000	LISTS+16
A	010001	LISTS+17
B	010010	LISTS+18
C	010011	LISTS+19
D	010100	LISTS+20
E	010101	LISTS+21
F	010110	LISTS+22
G	010111	LISTS+23
H	011000	LISTS+24
I	011001	LISTS+25
.	011011	LISTS+27
)	011100	LISTS+28

Table II

Partial List of Character Representations
and Base Registers for the IBM 7040/7090

C. BASCMP

BASCMP (BASic CoMPiler) compiles the full range of Basic Wisp statements. It uses a subset of the language consisting of the statements listed in Appendix V(1). A test program for these statements is given in Appendix V(2), and a complete description and listing of the compiler is in Appendix V(3). In addition to the type of translation provided by SIMCMP, BASCMP translates user labels (each consisting of any string of alphabetic and numeric characters) into integers. At the end of the compilation, a reference table is printed giving the correspondence between the compiler-generated labels and the symbols used by the programmer, and showing which (if any) are multiply defined or undefined.

Any statement which the compiler considers to be in error is printed out; a flag indicating the type of error appears in the left margin. Optionally, correct statements may also be printed. This printing may be switched on and off by the program being compiled, and will be discussed further below.

BASCMP is similar to SIMCMP in many respects, but its template matching algorithm is considerably more sophisticated. As each statement is read from the input stream, it is put into a standard form by removing all leading spaces, replacing all strings of spaces by a single space, and removing all spaces immediately preceding the statement delimiter (the comma or period). The character immediately following a quote is not altered by the editing process. A description of the allowed input format is given in the latter part of Section II(A).

The statement resulting from the editing process is matched against all of the templates, as before. One difference is that a parameter flag will now match any substring of the input string which does not contain any spaces, commas or periods in positions other than the first. This feature allows labels and list names to be of indeterminate length, and creates ambiguity problems which are considerably thornier than those of SIMCMP.

Matching a single template against an input line is a special case of the following general problem [8]: "Given a pattern $\{e_1 e_2 \ldots e_n\}$ and a string S, locate consecutive substrings s_i in S such that each s_i is an acceptable value of the pattern element e_i." Pattern elements in BASCMP templates are either fixed strings or parameter flags. A fixed string must, of course, match some substring of the input statement exactly. A parameter flag, on the other hand, may match any single character or any string which contains no period, comma or space in any position other than the first. The substring matching a parameter flag will contain at least one character, and will be extended to include all subsequent characters up to the first space, period or comma.

The space, period or comma which terminates the extension is not included in the parameter.

Notice that the behavior of the parameter flag is considerably different from that found in SIMCMP. This difference requires an alteration in the BASCMP templates which involve labels: Wherever two consecutive parameter flags appear, they must be replaced by a single one. This alters the code body, since all of the subsequent actual parameter numbers are reduced. An example of label generation is given in Figure 21 (compare Figure 17).

Ambiguity in the templates is handled differently in BASCMP than in SIMCMP. In the latter, you will recall that the ordering of the macro definitions was important. This is not true in BASCMP. Here the templates are formed into a tree as shown in Figure 22, and the tree is scanned from the root. When the comparison reaches a node in the tree, a check is made to determine whether a parameter is possible at that point. If so, the state of the scan is saved, and the routine attempts to match some non-parametric branch. If at any stage a match is impossible, the scan is restored to the most recent possible parameter. This scan algorithm insures that standard forms with no parameters will be matched, even though a standard form which contains a parameter could match the statement also. For example, the Wisp statement "RETURN." will be correctly recognized, even though it would also match "*.".

The code body of a BASCMP macro allows somewhat more flexibility than that of a SIMCMP macro. Two additional conversions are defined: type 2 to indicate a label reference, and type 3 to indicate a label definition. BASCMP maintains a symbol table of user-defined labels which is altered by these conversions. A type 2 conversion enters a copy of the user's symbol with its associated (compiler generated) integer and a "U" flag. The flag means that this particular symbol has been used but not yet defined. Subsequent type 2 conversions do not alter this entry. The integer associated with the symbol always appears in the punch output stream in place of the conversion request.

Type 3 conversion sets the symbol table flag for the symbol to "space", indicating that the symbol has been defined. If the symbol is not in the table, it is inserted, along with an integer and the "space" flag. If the symbol is already flagged by "space", the flag is changed to "M" (meaning that the symbol is multiply defined). The integer associated with the symbol always appears in the punch output stream in place of the conversion request. Examples of the use of these conversions are given in Figure 21.

It is possible to specify to BASCMP that the integers used as labels should all be of a given length. This is important

```
TØ *.
        TRA Z(12)
)
*.
Z(13 NULL)
)
```

a) IBM 7040/44/90/94

MCT parameter flag = "("
MCT end-of-line flag = ")"

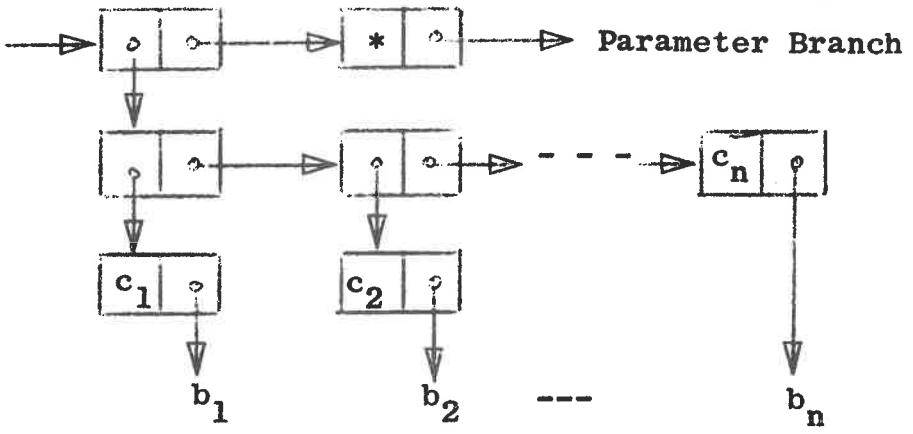
```
TØ *.
J:12;.
.
*.
:13;.
.
```

b) English Electric KDF9

MCT parameter flag = ":"
MCT end-of-line flag = "."

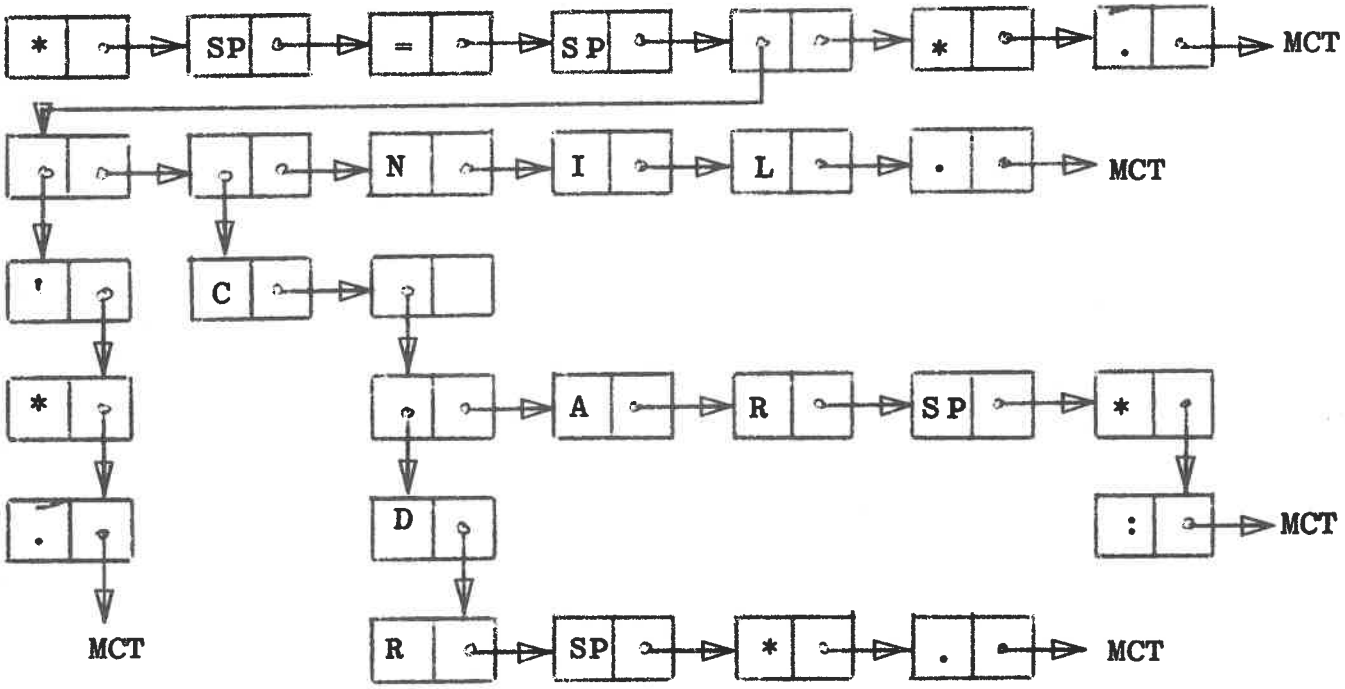
FIGURE 21

Label Generation in BASCMP



Non-parametric branches

(a) The structure of a node



(b) An example of a part of the tree
(assignment statements beginning with "* = ")

b=b[['.~][C[[CR'*.~]AR'*.~]]NIL.~]*.

FIGURE 22

The Macro Tree

in some cases (such as the 7040/7090, where operation codes must begin in column 8), but not others (such as the KDF9, where the positioning of a statement is immaterial). This specification is made by adding to the flag definition line a digit which gives the length of the integers; 0 is taken to mean that the length is variable.

The behavior of BASCMP may be altered during a compilation by means of "compiler switches" contained in the code bodies of the macros. A compiler switch is a call for conversion of parameter S, the type of conversion being the value of the switch. The values available are:

0 - Define new macros
1 - Copy machine code
2 - Start printing correct statements
3 - Stop printing correct statements
4 - Terminate the compilation
5 - Define new list names

Type 0 causes the compiler to skip to the next input line and begin to read macro definitions. These are in the same format as the original definitions, and are terminated by a line beginning with two MCT end-of-line characters. The macros may be completely new, or they may be redefinitions of operations already existing in the system. This feature allows the user to tailor Wisp to his particular requirements, which may vary from program to program.

A type 1 switch causes the compiler to skip to the next input line and begin to copy machine code onto the punch output medium. Each line of machine code must be terminated by an MCT end-of-line symbol, and the entire copying process is stopped by a line whose first character is an MCT end-of-line symbol. Copying is useful when a large block of machine code is to be inserted once into the translated Wisp program.

The type 2 switch sets up the print routine to print every statement. The original line divisions of the input are preserved, except that a new line will always be started when an incorrect statement is detected. A type 3 switch resets the print routine so that only incorrect statements are written.

The type 4 switch causes a transfer to the compiler's termination routine. This routine wraps up the compilation and prints the reference table.

The type 5 switch causes the compiler to skip to the next input line and read definitions of new list names. Each of these names must be longer than a single character, and must be followed by its definition. The definition will normally be a number, and

be separated from the name by a comma. Any number of names and definitions may be given on one line, and the line should be terminated by a period. Any number of lines may be present, and the definitions are closed by a void line.

Any compiler switch expect types 0 and 4 can appear anywhere in a code body; the scan of the code body resumes when the action specified by the compiler switch is completed. In general, each compiler switch will correcpond to a single Wisp statement. For example, in Figure 23, the Wisp statements "CONNECT LP." and "DISCONNECT LP." are used to start and stop the printing of correct statements. (A point to bear in mind when assigning new templates: Whenever possible, make the templates at least two words long to avoid confusion with labels.) Note that a macro must be included for the void line. This macro may cause information to be punched before termination.

The macro definitions of Figure 23 are those presently incorporated in the set of definitions for the IBM 7040/7090 system, and are certainly not the only ones possible. Any set of macro definitions must contain one which uses S0 if the programmer is to be allowed to define new operations. The other switches may be omitted, because the programmer could put them into macros of his own construction if he had access to S0. Alternatively, S0 could be omitted but other switches included. This would prevent a novice from accidentally destroying the normal definitions. Our feeling, however, is that the programmer should be given access to all switches.

When type 1 conversion is applied to a multi-character parameter which has been defined as a list name, the result is the string which was given as its definition. For example, suppose that the sequence of lines shown in Figure 24(a) were presented to BASCMP. The resulting output would be that shown in Figure 24(b). This feature is primarily useful in the compilation of the dynamic storage allocation package described in Section III(D), and a more detailed discussion will be presented at that time.

The fact that BASCMP attempts to match non-parametric branches means that it must have some way of returning to a parameter branch if the match fails at a later stage. The number of nodes at which both parametric and non-parametric possibilities exist is a priori unknown, so that an indeterminate number of branch points must be remembered. Moreover, the routine should return to the most recently-encountered branch point and proceed from there. These requirements suggest the use of a pushdown stack as temporary storage for remembering branch points. Use of a pushdown stack creates problems, however, because of the nature of the bootstrapping process. Recall that SIMCMP does not have any provision for dynamic storage allocation, and is not powerful

```
NEW SF.
(S0)
)
CØPY MC.
(S1)
)
CØNNECT LP.
(S2)
)
DISCØNNECT LP.
(S3)
)
.
        END(S4)
)
DEFINE LISTS.
(S5)
)
```

MCT parameter flag = "("
MCT end-of-line flag = ")"

FIGURE 23

Macros which Activate Compiler Switches

```
DEFINE LISTS.
SAM, 15, JØHN, SØME STRING.
BILL, ANØTHER STRING.
.
NEW SF.
TEST * * *.
LIST (10 IS (11)
LIST (20 IS (21)
LIST (30 IS (31)
))
TEST SAM JØHN BILL.
                                    MCT parameter flag = "("
                                    MCT end-of-line flag = ")"
```

(a) Input for "DEFINE" example.

```
LIST SAM IS 15
LIST JØHN IS SØME STRING
LIST BILL IS ANØTHER STRING
```

(b) Output corresponding to test input.

FIGURE 24

Illustrating the Behavior of List Definition

enough to compile a dynamic storage allocation routine. We further asserted that BASCMP required no dynamic allocation, but here we require a pushdown stack of indeterminate length.

The solution to the problem is to note that BASCMP has a block structure which is analogous to that of Algol, and similar techniques may be used for creation of the pushdown stack. We allocate a large block of storage as list space, and enter macros into it from the bottom up. Our pushdown stack propagates from the top down and is used for remembering both branch points and subroutine return points. The stack is manipulated by means of the two operations "STACK *." and "RECALL *.". The former places the contents of the addressed base register on the stack, pushing down what was already there. The latter pops up the stack, restoring the addressed base register with the most recently stacked information. The stack is thus completely outside of the list structure and is not handled as a normal Basic Wisp stack. It cannot be addressed by the program in the usual way and can be manipulated only by the two instructions given above.

The storage arrangement described in the previous paragraph is somewhat different from that used by programs written in Basic Wisp. There is no free list as such, and no way of reclaiming elements which are discarded by the program. Supervision of the storage area is thus reduced to a minimum--a distinct advantage when any supervision routines must be hand coded. Of course, the simplicity of the storage arrangement places limitations on the programmer, but this is not serious because BASCMP is the only program which must use this scheme. BASCMP is so constructed that list elements are never discarded, and garbage collection would therefore be superfluous even if we provided it.

Storage maintenance requires five operations:

1) Claim a new element from the bottom of the free space and increment the pointer to the free space.

2) Stack a base register, adjusting the stack pointer.

3) Recall a base register from the stack, adjusting the stack pointer.

4) Save a return address on the stack, adjusting the stack pointer.

5) Recall a return address, adjust the stack pointer, and transfer to the indicated address.

The first three operations can be described easily in Fortran or Algol, but the last two cannot. Most high level languages do not

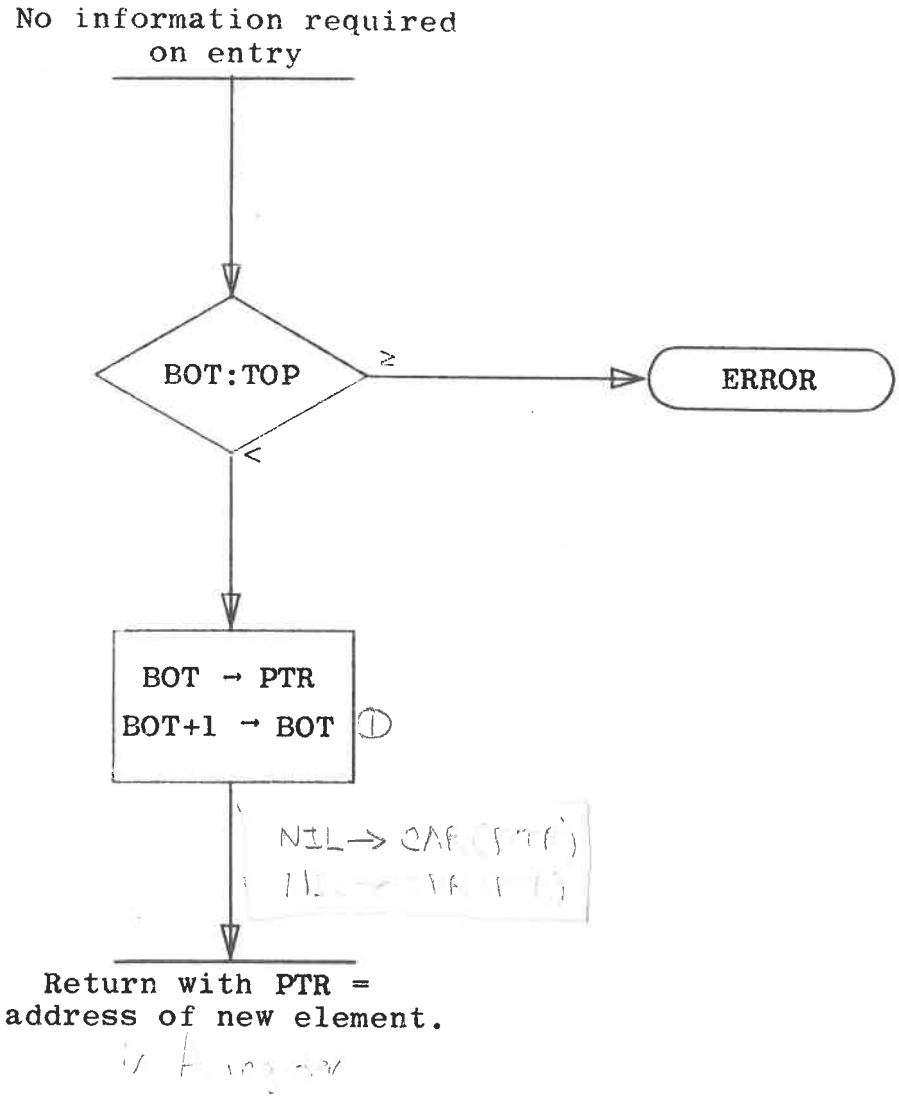


FIGURE 25

Claim a New Element

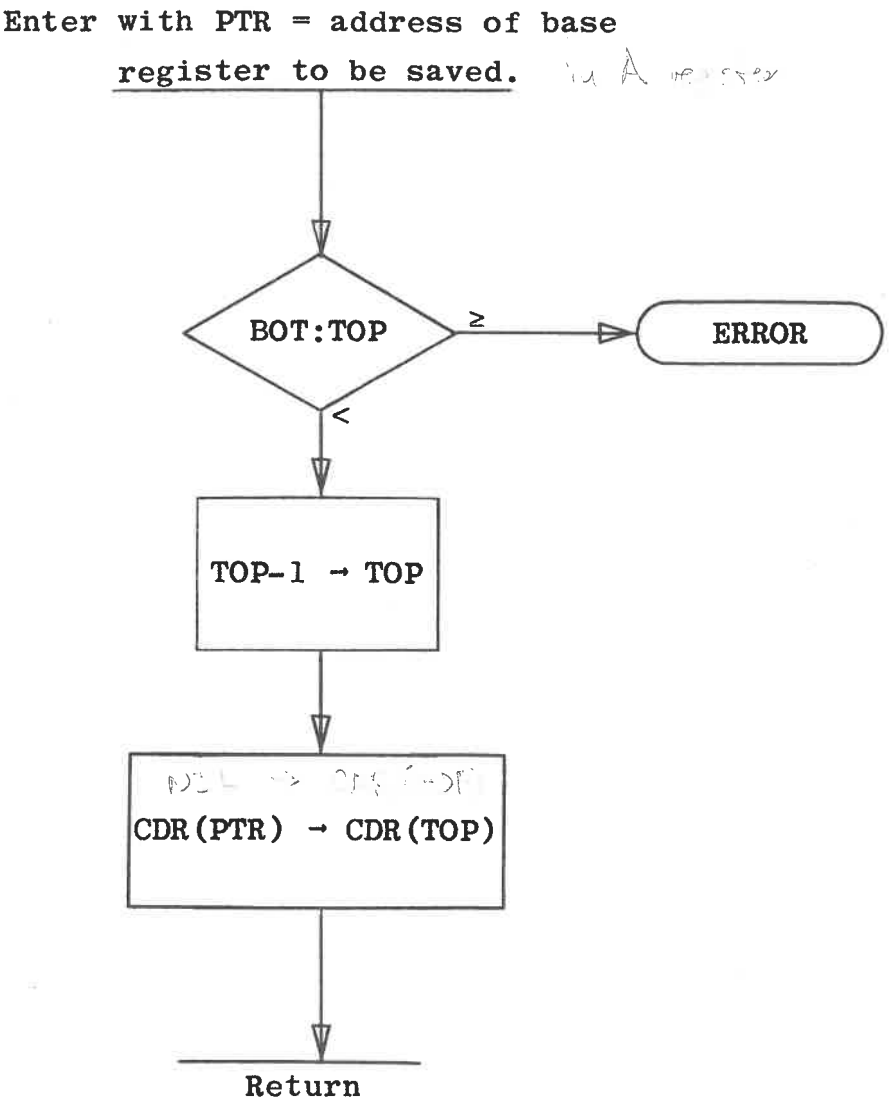


FIGURE 26

Save the Contents of a Base Register

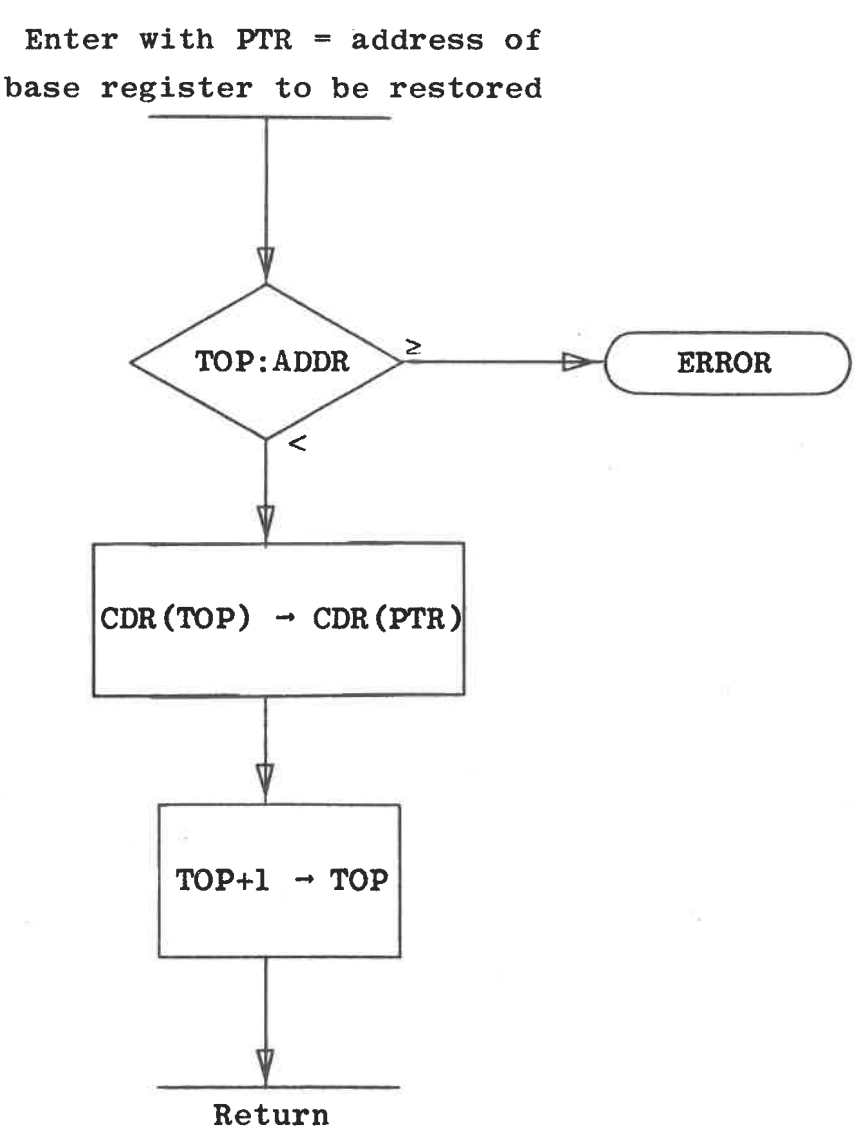


FIGURE 27

Restore a Saved Base Register

T9BACK

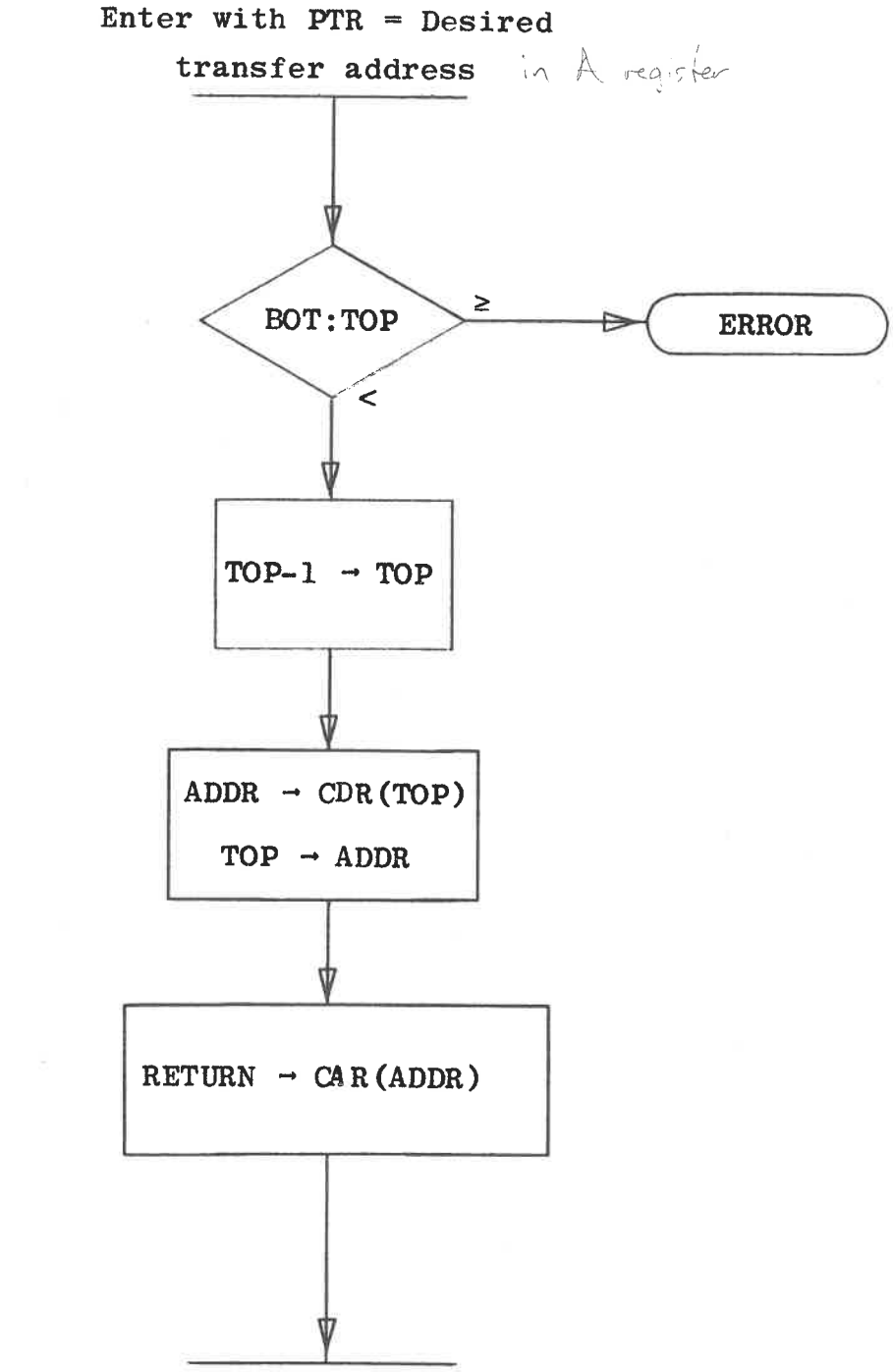


FIGURE 28

Mechanization of "TØ ** AND BACK."

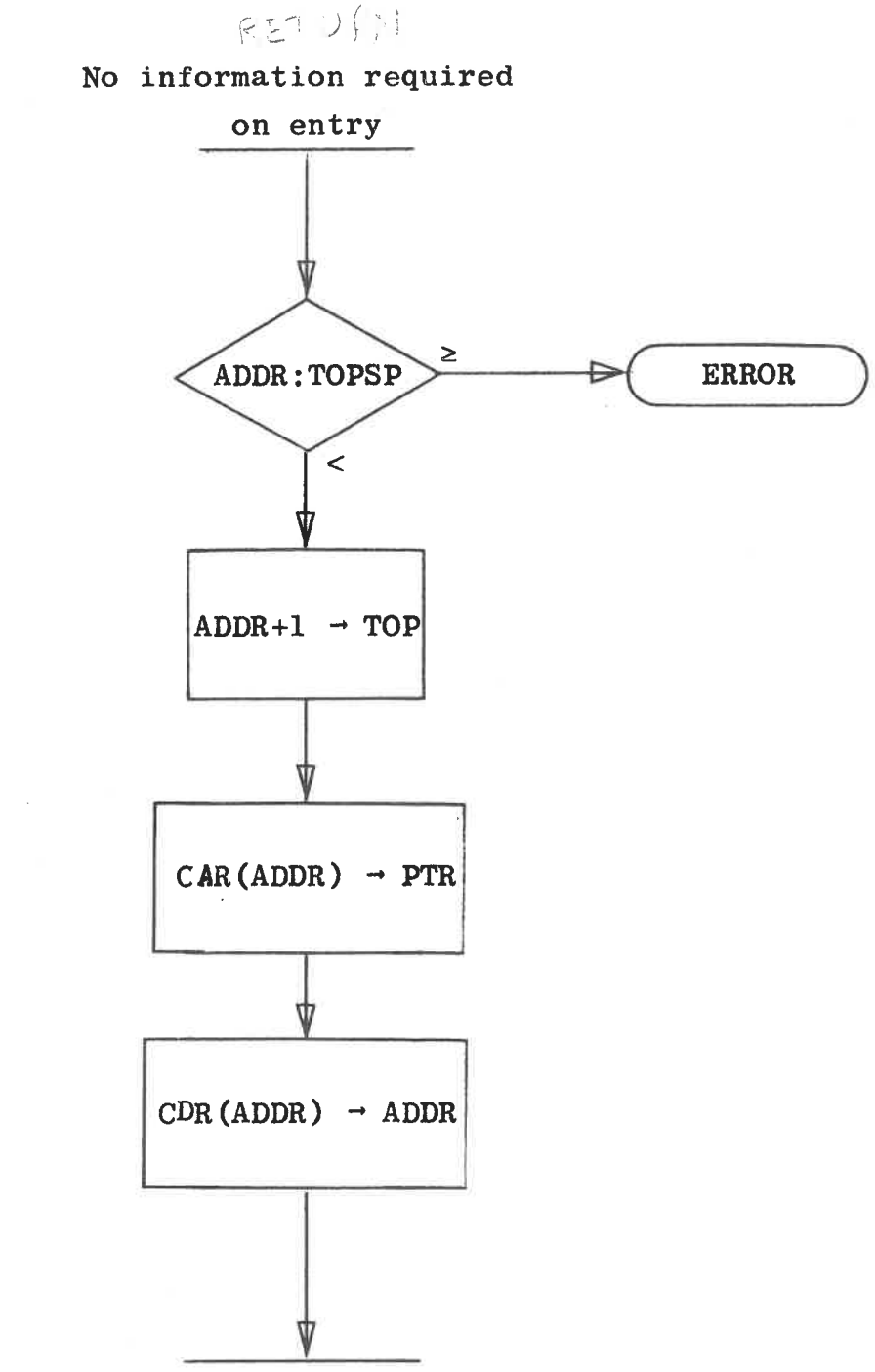


FIGURE 29

Mechanization of "RETURN."

permit the user to access the location counter or to transfer to a variable destination. There are usually ways of circumventing these restrictions, but they depend on local compiler idiosyncrasies and are therefore not suited to extensive discussion here. Implementation of operations 4 and 5 in assembly code presents no problem.

The five operations described above use four pointers to the free storage area:

BOT - The address of the first word not being used by the list structure.

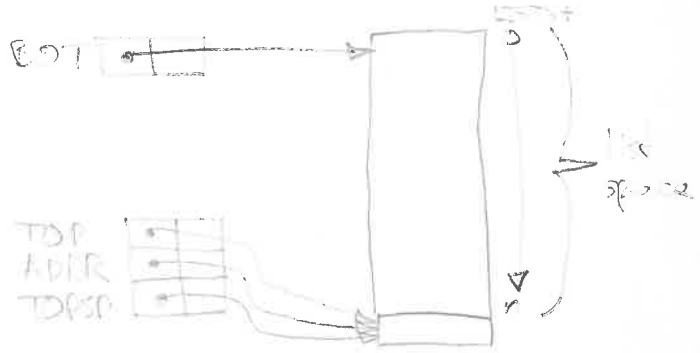
TOP - The address of the first word in use by the stack.

ADDR - The address of the word holding the most recent return address.

TOPSP - The address of the first word above the free space.

We shall assume that each word is capable of holding two machine addresses. If this is not the case, the only changes required will be in the incrementing and decrementing operations. Initially BOT points to the first word of the free space, and TOP = ADDR = TOPSP. Flow charts of each operation are given in Figure 25 through 29, and should be self explanatory. Note that stacked information is only available during the execution of the subroutine which saved it. A call to another subroutine insulates all previously stacked information, while a return merges the current stack with the free space. This behavior is appropriate in a system using recursive subroutines. Information is protected during multiple calls of the same subroutine, but is not retained when it is no longer needed.

The implementation of BASCMP, given that SIMCMP is available, should proceed as follows: (1) Code and check out the environment routines flowcharted in Figures 25 through 29. (2) Code the macros required for BASCMP (see Appendix V(1)). (3) Use SIMCMP to compile the test program of Appendix V(2), and run it with the environment. Experience has shown that the correct coding of the macros is the most difficult part of the procedure, so that this step should not be omitted. (4) When the test program executes satisfactorily, compile BASCMP.



D. DYNALC

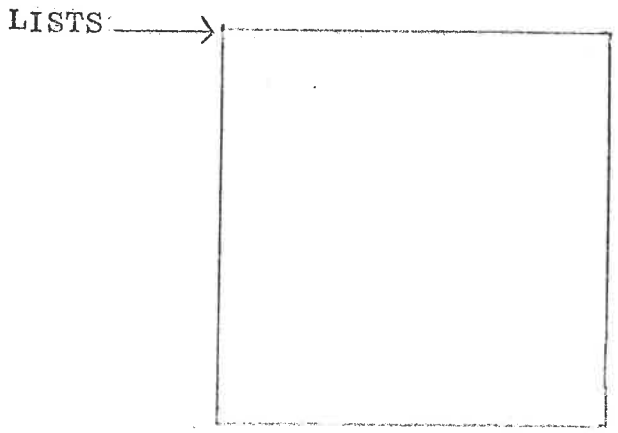
The dynamic storage allocator is a collection of routines which manages the Wisp list storage space. These routines are written in Basic Wisp, augmented by several additional operations. In order to perform its various duties, the allocator needs to have several "private" base registers. These are not accessible to the programmer, and the allocator does not alter any which are accessible.

The allocator also assumes a particular storage organization which is not a part of the definition of Wisp. Figure 1 shows this layout. Notice that the list space and base register area are disjoint. This is always true, even if they form one contiguous block of store - the list space is defined as the area between BOT and TOP.

Base registers which are accessible to the user are all linked together by their CAR fields. This "CAR chain" enables the garbage collection routine and initialization procedures to sequence through all of the accessible base registers, without making the assumption that they are contiguous. As a simple example of this linking, consider a machine with only two atoms, 0 and 1. This machine would have 3 accessible base registers: 0, 1 and NIL. They would be linked together by a CAR chain as shown in Figure 2.

Figure 2 depicts the so-called "initial state" of the Wisp machine. Only the CAR chain must be provided by some agency outside the Wisp system. The free list is set up by the routine INITAL, which also attaches a list element to each base register. INITAL is a part of the dynamic storage allocator, and must be entered before execution of the Wisp program begins.

How are base register locations determined in this system? How are list elements addressed? The answers to these questions depend to some extent on the available instructions of the target



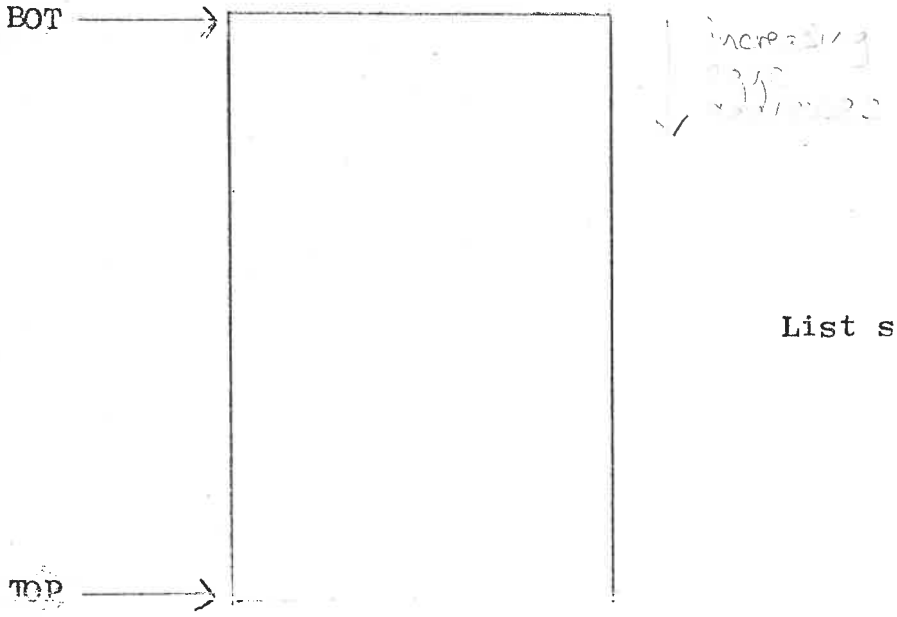


Figure 1

Wisp Storage Layout

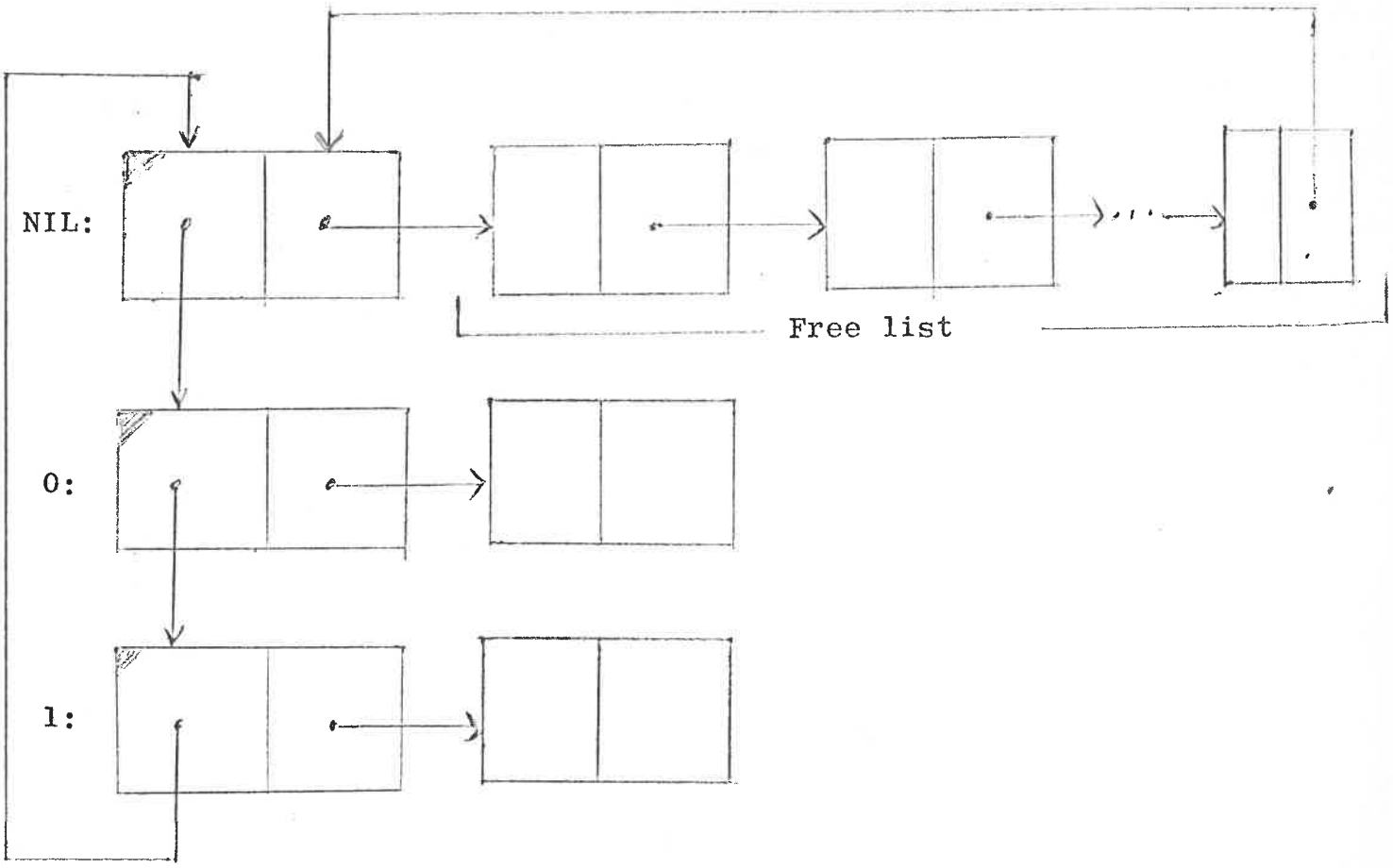


Figure 2
Initial Linkages

machine. There are really two ways of addressing the list space: by absolute machine address, or by address relative to the location LISTS (see Figure 1). If the target machine has sufficiently flexible indexing, I would recommend the latter. In the subsequent discussion I will assume that this scheme is, in fact, used. If this is not the case, the alterations should be obvious.

On most machines there is a natural mapping from the character set into some set of non-negative integers. These integers are the "internal representations" of the characters. By isolating the internal representation of a single character, we have the address of that character's base register (relative to LISTS). Since an atom is represented by the address of its base register, the atom is equivalent to the internal code for the character. This property makes code conversion trivial, and generally eases implementation difficulties.

The internal representations of the character sets of a number of machines do not form contiguous groups of integers. Often there are "gaps". That is to say, the character "space" might be represented by 0 and the character "/" by 15, with no valid character being represented by any integer between them. Thus the integers 1 through 14 would not address accessible base registers, and the locations LISTS+1 through LISTS+14 would normally be unused. These words would not be part of the CAR chain, and hence none of the allocator routines would mistake them for valid base registers. It would therefore be possible to use the words as private base registers or number stores.

If the sequence of internal representations has no gaps, then private base registers or number stores would have to be assigned additional words. This is totally irrelevant to the operation - in the presence of gaps, the CAR chain allows us to take advantage of

those words without endangering the initialization or garbage collection procedures; if the gaps do not exist, nothing has been lost.

Let us turn now to a more detailed consideration of the initialization procedure and differentiate the jobs of the external environment (assembler, compiler or operating system) and the dynamic storage allocator. The routine INITAL is the first routine entered. Its task is to set up the initial state of the Wisp machine as shown in Figure 2. Study of the routine's listing in Appendix I will show that the following information is required upon entry:

1) A CAR chain must be formed [illegible]

2) The CDR of the NIL base register must contain the address of the NIL base register.

3) The private base register LOLIM must contain the address BOT (see Figure 1).

4) The private base register UPLIM must contain the address TOP.

All requirements except (2) serve to describe the list space to the allocator. Requirement (2) is essentially a switch to prevent re-initialization. In many implementations this switch is not needed. In some cases, however, it is possible to compile a number of subroutines separately in such a way that any of them could play the role of a main program. The actual "main program" is nominated at run time, and hence the first operation of each subroutine must be a call on INITAL. Only the first such call should result in any initialization, however.

In order to understand the workings of INITAL, we must describe some of the operations added to Wisp in order to make storage management possible. (These are summarized in Appendix III.)

"ENTRY *." defines a symbol which is used as an entry point. This symbol will be global, and not just local to the subroutine in which it occurs. The operation also provides a mechanism for saving a return address. This mechanism is redundant when a recursive call is made on the entry point (by TØ * AND BACK.), but is necessary for a non-recursive call. All calls on the storage allocator are non-recursive. One other property of ENTRY is that it can be used to define the beginning of a segment on machines where this is important. The storage allocator is written so that there are no cross-references out of the "range" of an ENTRY. (The range of an ENTRY contains all statements up to the next ENTRY.)

The reciprocal operation is "EXIT *." . This causes a return to the place from which the corresponding ENTRY was called. EXIT will be in the range of some entry. If the symbol of that entry is SYM, then the exit operation must be EXIT SYM.

List elements have an order imposed on them by the memory structure of the target machine. STEP *. moves a base register from one list element to the next, according to this order. On a machine where each list element is a single word, STEP *. consists of adding 1 to base register *. If each list element is made up of 8 address units (e.g. System/360 bytes), STEP *. would be implemented by adding 8 to base register *.

One feature of the Basic Wisp compiler is the ability to define additional list names. Each name is associated with an integer, and if the name is used in a position where a list name is expected, the integer is output. This feature allows us to give the private base registers names, as summarized in Table I. The only name whose purpose may be unclear is FREE. This is needed because of the anomalous position of NIL. NIL is an atom, not the

Name	Function
FREE	An alias for the NIL base register
LOLIM	The base register containing BOT (see Figure 1)
UPLIM	The base register containing TOP
LPRIME	The base register for the return stack
P1	Base register for communication with program
P2	Scratch Register
P3	Scratch Register

Table I
Private Base Registers

name of a base register. There is no other atom for which this statement holds - all other atoms are also the names of base registers, and it is for this reason that we need to differentiate between the operands A and 'A. NIL is associated with a base register, however, even though there is really no way to reference it. Thus FREE must be defined to give us a name for the base register associated with NIL. The following equivalences hold:

A = 'FREE. $\Longleftrightarrow$ A = NIL.

A = FREE. $\Longleftrightarrow$ A = NIL, A = CDR A.

The operation of INITAL is quite straightforward. P1 is initially pointed to the bottom of the list space, and P2 sequenced through the accessible base registers by means of the CAR chain. At each base register an element is attached and cleared, and P1 advanced to the next element. The result is to attach a single, empty element to each base register, as shown in Figure 2.

When P2 finally points to the NIL base register, the free list is constructed. The element currently pointed to by P1 is attached to the CDR of the element pointed to by P2 (at first this element is just the NIL base register). P2 is then advanced to point to the new element, and P1 is stepped to the next element of the store. The process stops when P1 reaches the top. The top element of the store is attached to LPRIME.

INITAL and the pre-initialization work of setting up the store limits and CAR chain perform the setup of the list space. The remainder of the dynamic storage allocator is responsible for free list maintenance. Basically, two processes are involved: satisfying requests for list elements, and returning unused list elements to the free list. A peripheral activity is handling the return stack. Returning an element to the free list is almost a trivial task.

The current free list is attached to the CDR of the element being returned, and the free list base register is then pointed to this element. In effect, the free list is being pushed down.

Satisfying a request for a new element is usually just as trivial. If the free list happens to have only a single element left, however, a garbage collection is forced. Garbage collection involves tracing the entire accessible list structure, and marking each element. A sequential scan of memory is then made, and all unmarked elements are formed into a new free list. All marks are also erased during this process.

Let us examine the garbage collection process a bit more closely. All lists, including the return stack, must be marked. The return stack is basically different from all other lists in that the CARs of its elements do not contain pointers. Rather, they contain addresses in the program. The normal list trace routine assumes that both fields of all list elements contain pointers - a correct assumption in the case of any list except the return stack. Thus the return stack must be traced specially, and this is the purpose of the loop at +STKTR.

After the return stack has been traced, P1 is moved over the accessible base registers by means of the CAR chain. The operation "FLAG *." is used to set the branch point flag of the base register after its list has been traced. The purpose of this is to allow a check for a corrupted CAR chain. There are many ways to corrupt the CAR chain, most of them quite easy to accomplish. If the chain is intact, we must arrive back at NIL without encountering a flagged register.

At each base register, the routine TRACER is invoked to trace the list. TRACER expects P1 to be pointing to the base register of the list to be traced, and uses P2 and P3 for scratch storage. The trace algorithm will handle lists of any complexity, including circular lists. It is described in detail in a paper by Schorr and Waite (Comm. ACM, August, 1967). The new operations involved in the trace are * = ATOM., which sets the atom flag of the element pointed to by base register * and UNFLAG *., which resets the branch point flag of the element pointed to by base register *.

When all lists have been traced, the sequential scan of memory takes place. The code is straightforward; DEATOM *. resets the atom flag of an element. Following the garbage collection, all base registers must be reset. That is to say, the branch point flags must be cleared and the atom flags reset. Why the latter? Normally, base registers would not have their atom flags reset during garbage collection because they are not in the same area as the list elements. It is often useful, however, to define additional pseudo base registers during the course of executing a program. This feature was used, for example, in the implementation of the list processor FLIP. The pseudo base registers are ordinary list elements which have been inserted in the CAR chain and have had their atom flags set. As such they are in the regular list space, and their atom flags will be reset during garbage collection. It is for the benefit of these that we must set all atom flags after garbage collection.

E. The List Dump Routine

Wisp programs of any complexity are subject to obscure failures during development. Most of these failures can be diagnosed easily by dumping one or more lists at various times as the program progresses. The basic language provides no way of taking such dumps, since it is not capable of printing non-atoms. What is really needed is a routine which will print both fields of all elements of a list.

This snapshot feature is provided by a dump routine, in conjunction with the trace routine of the dynamic storage allocator. The dump routine prints all base registers and any list elements which have their atom flags set. (The atom flags are reset in the process.) Usually, a set of three new operations are defined:

SNAP *. Marks and prints the indicated list.

TRACE *. Marks the indicated list, but does not print it.

SNAP RETURN. Prints the return stack.

The procedure used to snap several lists is to trace all but one, and then snap the last.

In order to trace a list, the address of its base register is placed in private register P1 and TRACER is entered nonrecursively. The snapshot is then printed by entering SNAPL2 nonrecursively. If the return stack is to be printed, no trace is needed. A nonrecursive entry to SNAPR2 is sufficient. Note that "SNAP *." must cause both tracing and printing.

The snapshot routine itself is quite straightforward. It uses the normal Wisp operations, plus some of the special ones described in Section IV. The routine does require two machine coded procedures: DUMPER and TABOUT. These routines are used to

convert pointer values to strings of digits, and to form the output into lines. Private base registers P1-P3 are used for communication between the list dump routine and these procedures.

Before describing the machine code procedures, let us examine Figure 3, a section of a list dump from the CDC6400. The dump is organized into lines, with the base registers at the top. The base register section of the dump is headed by "*BR". The number at the extreme left of the line is the address of the first element of the line. Successive elements of the line are the elements immediately following this element. A sequence break causes a new line to be started.

Each element has two parts, which have different meanings for base registers and list elements. The left side of a base register is its name. Thus, on the 6400, base register A is at location 1. Base register B is at location 2, C at 3, and so forth. The right side of a base register is the pointer of the base register. In Figure 3, base register A points to the list element at 4421 and base register B points to the one at 4422.

The list elements are headed by "*EL", and are printed as CAR and CDR. In this particular implementation, if either of the fields contains an atom it is printed as an atom rather than the address of the atom's base register. The structure of the lines is the same as that of the base register lines, with the leftmost number being the address of the first element of the line. Notice the prevalence of sequence breaks in the dump of Figure 3. The element at 4426 had the atom "$" in its CAR, and its CDR pointed to location 4520. 4520 was the next element dumped, and thus it appeared on a new line. Neither the CAR nor the CDR of this element contained an atom, and hence both were printed as numbers. The element at

```
*BR
000001:  A 004421   B 004422   C 004423   D 004424   E 004744   F 005216
000011:  I 004431   J 004432   K 004433   L 004434   M 004435   N 004436
000021:  Q 004441   R 004460   S 004443   T 004426   U 004445   V 004446
000031:  Y 004451   7 004522   0 004453   1 004454   2 004455   3 004456
000041:  6 004461   7 004462   8 004463   9 004464   + 004465   - 004466
000051:  ( 004471   ) 004472   $ 004473   = 004474     004475   , 004476
000061:  [ 004523   ] 004426   : 004534   ≠ 004522   ↱ 004524   ∨ 005037
000071:  ↓ 005045   < 005045   > 005135   ≤ 005213   ≥ 004610   ¬ 004516

*EL
004426:  $ 004520
004520:  004536 004542  004426 NIL
004534:  $ 004536
004536:  004550 004554  004700 004704  L 004541  004534 NIL  T 004521
004546:  $ 004550
004550:  004607 004614  004563 004566  R 004553  004546 NIL  N 004537
004560:  $ 004562
004562:  C 004563       004575 004601  R 004565  004560 NIL  A 004552
004572:  $ 004574
004574:  C 004575       004763 004767  N 004577  004652 004654  004572 NIL  D 004564
004605:  $ 004607
004607:  004622 004625  005135 005130  O 004612  M 004613  004605 NIL  C 004551
004620:  $ 004622
004622:  004633 004641  0 004624  004620 NIL  A 004610
004631:  $ 004633
```

Figure 3

An Example of a List Dump

4521 was a member of the list being traced, and hence it appeared on the same line as 4520. The next was 4534 - a sequence break which forced a new line.

With this brief view of the dump output, let us consider the action of TABOUT. Notice in Figure 3 that each element begins in a specified position of the line, independent of the length of the preceding text. The function of TABOUT is to establish these positions. Perusal of Appendix II will show that TABOUT is entered at the completion of each element. It must output enough "space" characters to bring the line pointer to the next element position. If the line is exhausted, it should perform operations equivalent to P3 = NIL, FINISH LINE ON PRINT. A flowchart of TABOUT is given in Figure 4.

DUMPER is responsible for converting pointers and printing the resulting digit strings or atoms. It is always entered with P1 containing the address of the element, and P2 the field (CAR or CDR) to be printed. P3 is NIL if a new line is to be started. A flowchart is given in Figure 5. (The subroutine CONVERT merely converts a pointer to a digit string.)

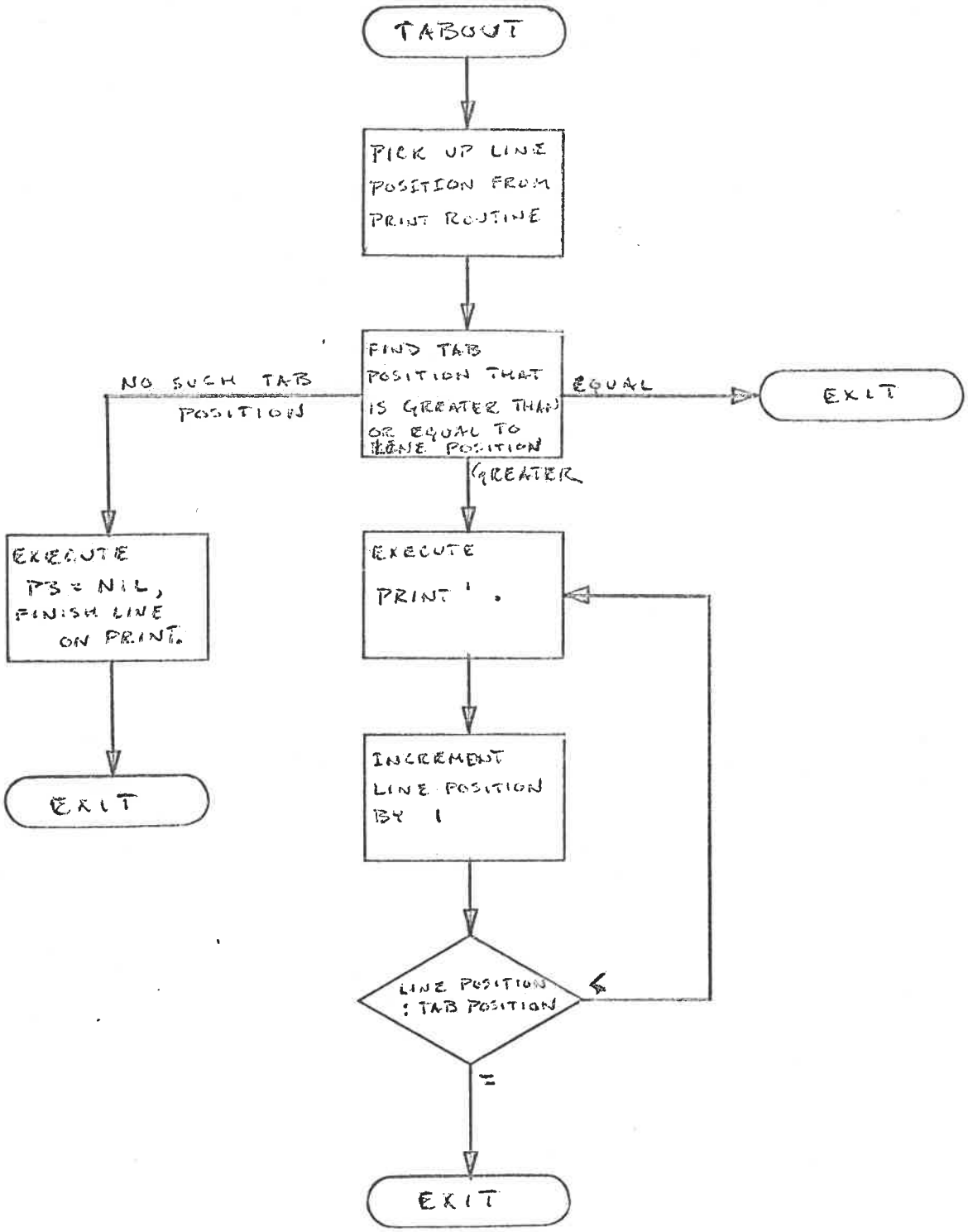


Figure 4

TABOUT Routine

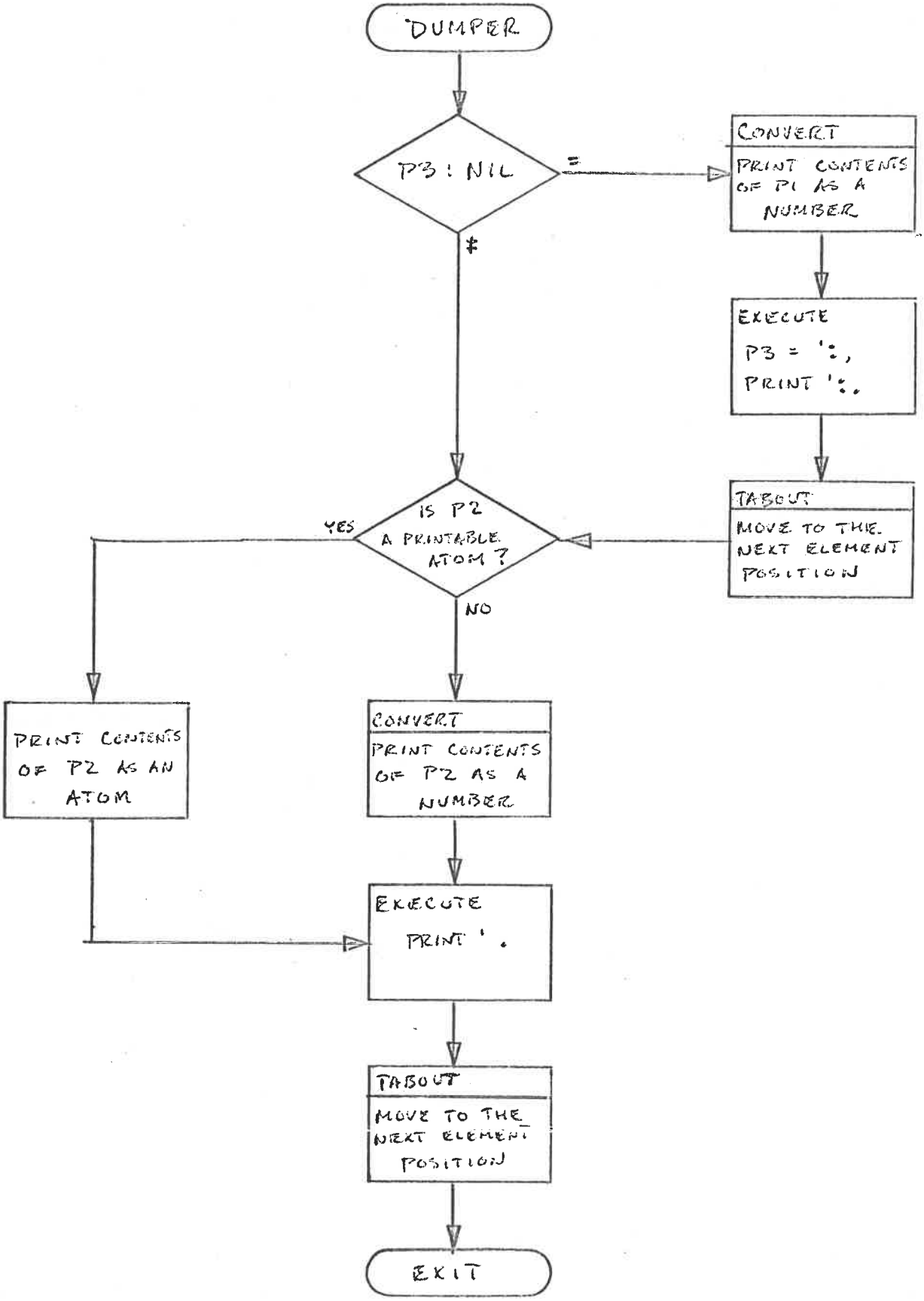


Figure 5
DUMPER Routine

IV. REFERENCES

1. Shaw, J.C., Newell, A., Simon, H.A., Ellis, T.O. "A Command Structure for Complex Information Processing", Proc. WJCC (1958), Institute of Radio Engineers, New York, 1959, P. 119.

2. Schorr, H., Waite. W.M. An Efficient, Machine Independent Procedure for Garbage Collection in Various List Structures, IBM Research Paper RC-1450, Sept. 29, 1965.

3. Wilkes, M.V. "Lists and Why They are Useful", Proc. 19th National Conference of the Association for Computing Machinery, (August 25-27, 1964), p. F1-1.

4. McIlroy, M.D. "Macro Instruction Extensions of Compiler Languages", Comm. ACM, 3 (April, 1960), p. 214.

5. Waite, W.M., Schorr, H. "A Note on the Formation of a Free List", Comm. ACM, 8 (August, 1964), p. 478.

An extensive bibliography on list processors and applications of list processing may be found in:

6. Newell, A. et.al., Information Processing Language-V Manual, 2nd. ed., Prentice-Hall, Englewood Cliffs, N.J. (1964).

7. Barnes, J.G.P. A KDF9 Algol List Processing Scheme. Computer J., 8 (July, 1965) 113.

8. Griswold, R.E., Polonsky, I.P. String Pattern Matching in the Programming Language SNOBOL. Bell Laboratories, July 1, 1963 (Unpublished).

APPENDIX I

CHARACTER SETS

(1) Basic Wisp Atoms

MANUAL	EDSAC 2 ATLAS 2	ELLIOT 803	IBM 7040/7090	ENGLISH ELECTRIC KDF9
A - Z	A - Z	A - Z	A - Z	A - Z
0 - 9	0 - 9	0 - 9	0 - 9	0 - 9
SPACE	SPACE	SPACE	SPACE	SPACE
.	.	.	.	.
,	$_2$(suffix 2)	,	,	,
*	*	*	*	*
=	=	=	=	=
'(quote)	$_{10}$(suffix 10)	'	'	$_{10}$(suffix 10)
≠	≠	/=	.NE.	≠

(2) Additional Atoms

EDSAC 2, ATLAS 2: > [ ] / ( ) → : + -

ELLIOT 803: $ % ? @ £ / ( ) : + -

IBM 70--: $ / ( ) + - .LT. .LE. .GE. .GT.

ENGLISH ELECTRIC KDF9: a - z ↑ < > ≤ ≥ x(multiplication)

÷ ; £ [ ] / ( ) : + -

APPENDIX II

OPERATIONS AVAILABLE IN BASIC WISP

(1) Assignment Statements (Section II (B))

```
* = *.
* = '*.
* = NIL.
* = CAR *.
* = CDR *.

CAR * = *.          CDR * = *.
CAR * = '*.         CDR * = '*.
CAR * = NIL.        CDR * = NIL.
CAR * = CAR *.      CDR * = CAR *.
CAR * = CDR *.      CDR * = CDR *.
```

(2) Control Statements (Section II (C))

```
**.
TO **.
STOP.

TO ** IF * = *.          TO ** IF * ≠ *.
TO ** IF * = '*.         TO ** IF * ≠ '*.
TO ** IF * = NIL.        TO ** IF * ≠ NIL.
TO ** IF * = ATOM.       TO ** IF * ≠ ATOM.

TO ** IF CAR * = *.      TO ** IF CAR * ≠ *.
TO ** IF CAR * = '*.     TO ** IF CAR * ≠ '*.
TO ** IF CAR * = NIL.    TO ** IF CAR * ≠ NIL.
TO ** IF CAR * = ATOM.   TO ** IF CAR * ≠ ATOM.
TO ** IF CAR * = CAR *.  TO ** IF CAR * ≠ CAR *.
TO ** IF CDR * = CDR *.  TO ** IF CDR * ≠ CDR *.

TO ** IF CDR * = *.      TO ** IF CDR * ≠ *.
TO ** IF CDR * = '*.     TO ** IF CDR * ≠ '*.
TO ** IF CDR * = NIL.    TO ** IF CDR * ≠ NIL.
TO ** IF CDR * = ATOM.   TO ** IF CDR * ≠ ATOM.
TO ** IF CDR * = CDR *.  TO ** IF CDR * ≠ CDR *.
```

APPENDIX II (Continued)

(3) Stack Operations (Section II (D))

PUSH DOWN *.

POP UP *.

START LIST *.

(4) Subroutines (Section II (E))

TO ** AND BACK.

RETURN.

LEVEL DOWN.

TO ** IF RETURN.

TO ** IF NO RETURN.

(5) Input/Output Statements (Section II (F))

CAR * = INPUT.

NEXT LINE ON INPUT.

PRINT '*.

PRINT CAR *.

FINISH LINE ON PRINT.

NEW PAGE.

PUNCH '*.

PUNCH CAR *.

FINISH LINE ON PUNCH.

PUNCH DECIMAL CAR *.

(6) Arithmetic Statements (Section II (M))

Nn = i.

Nn = Nn.

Nn = i op Nn.

Nn = Nn op i.

Nn = Nn op Nn.

TØ ** IF Nn rel i.

TØ ** IF Nn rel Nn.

LIST * = Nn.

Nn = LIST *.

APPENDIX III

OPERATION OF THE ASSIGNMENT STATEMENTS

The following diagrams illustrate the operation of each of the assignment statements listed in Appendix I (1), with the exception of those involving the atom NIL. Each diagram is in the form of "before" and "after" pictures. The left hand side shows the state of two lists before the execution of the statement, and the right hand side depicts the same lists just after execution. The square boxes represent the base registers of the indicated lists. Small letters w, x, y and z are used to stand for any atom or pointer, including NIL. A blank is used to denote NIL in those cases where it is known to be present.

Assigument statements using NIL behave in exactly the same way as the corresponding statement using an ordinary atom (i.e. * = NIL. works in the same way as * = '*.). The only difference lies in the base register address used to represent the atom.

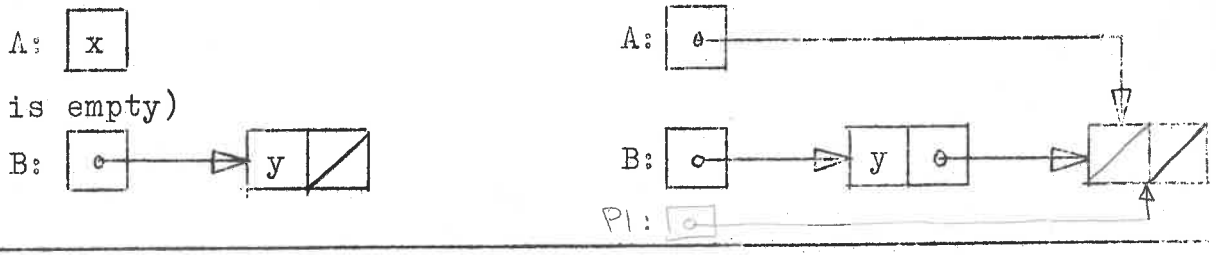
A = B.

A = 'B.

A = CAR B.

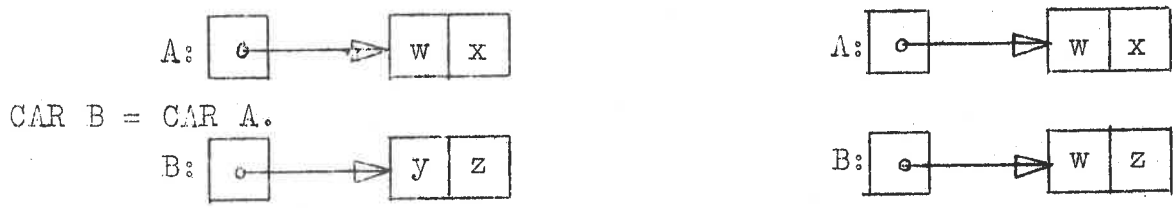
A = CDR B.

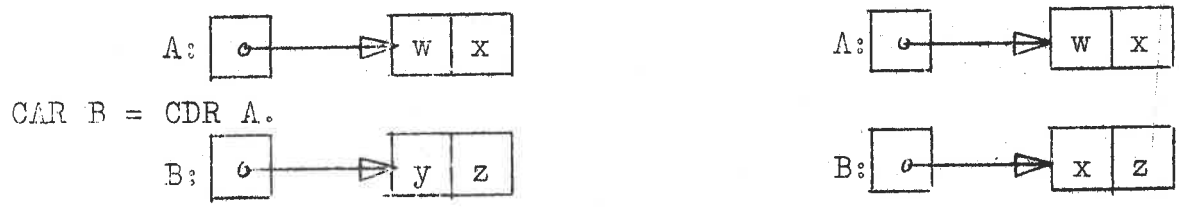
(If CDR B is empty)

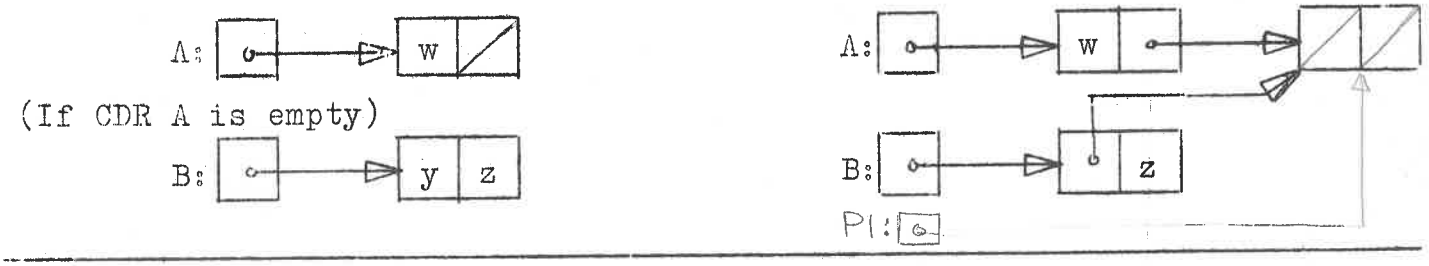


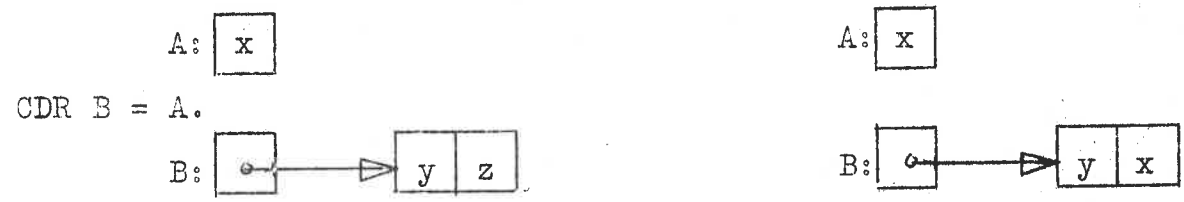
CAR B = A.

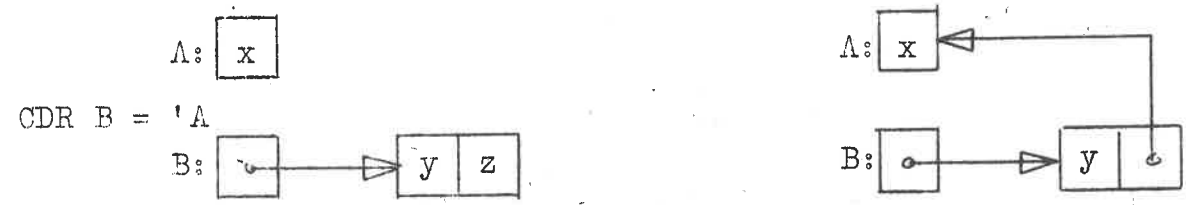

A: x
CAR B = 'A.
B: y z
A: x
B: z

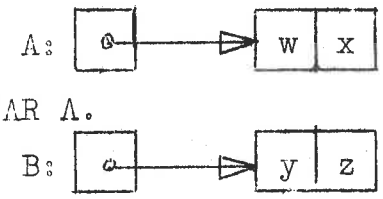

A: w x
CAR B = CAR A.
B: y z
A: w x
B: w z


A: w x
CAR B = CDR A.
B: y z
A: w x
B: x z

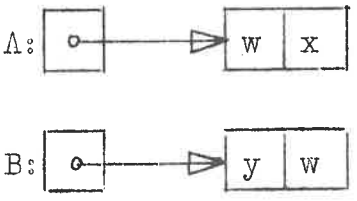

A: w
(If CDR A is empty)
B: y z
A: w
B: z
P1:

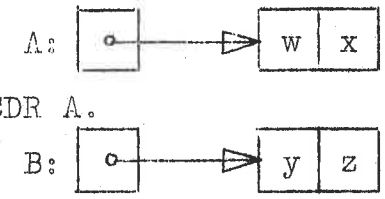

A: x
CDR B = A.
B: y z
A: x
B: y x


A: x
CDR B = 'A
B: y z
A: x
B: y

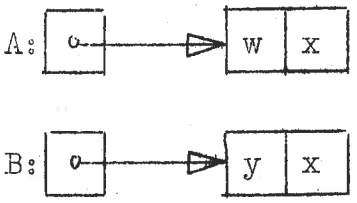


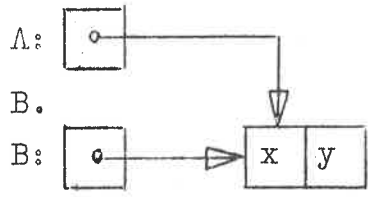
CDR B = CAR A.



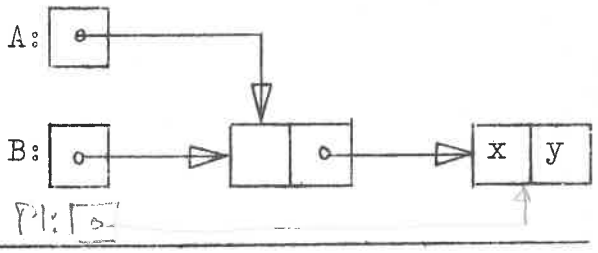


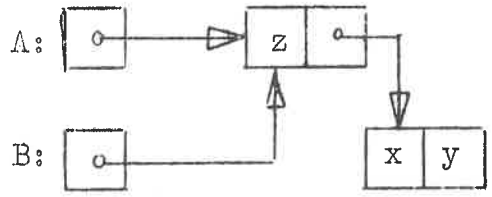
CDR B = CDR A.



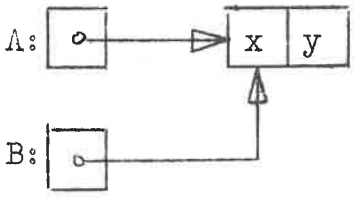


PUSH DOWN B.





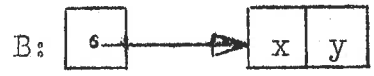
POP UP A.



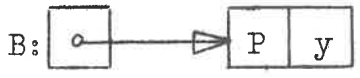
START LIST A.

Input string: PQ ...

CAR B = INPUT.



Input string: Q ...



APPENDIX IV

A Description of the Fortran Version of SIMCMP

The Fortran version of SIMCMP uses six temporary storage locations and one array. The layout of the array during the expansion phase is shown in Figure IV-1. The first 15 words are used for storage of the control characters and actual parameter values. The macro definitions begin in LIST(16) and continue through LIST(K)-1. Each macro definition has a two-word header, the first word giving the location of the next macro definition and the second containing an integer equal to the maximum conversion digit of parameter zero in this macro. If the macro does not use parameter zero, the second word of the header contains -1.

Each call on a formal parameter in the input text appears as < mct param flag > dc ($0 \leq d \leq 9$, $0 \leq c \leq 9$ if $d = 0$; otherwise $c = 0,1$). Such a call is held internally as the two integers -6-d and c. This form saves one word in each parameter call and allows easy identification of parameter calls. Moreover, the first integer is just the negative of the index of the actual parameter store in the array.

The three environment routines are INPUT(i), PRINT(j,k) and PUNCH(j,k). If the parameter of the input routine call is nonzero, the value returned will be the integer which corresponds to the next input character. When the parameter is zero, a zero value is returned and the input is advanced to the beginning of the next line.

The two output routines are identical in behavior, except that one writes on the punch, the other on the printer. If the first argument is negative, the current line is written out and subsequent output begins on a new line. Successive calls with j negative produce blank output lines. If $j \geq 0$, it is converted to a character if $k = 0$. Otherwise it is written as a decimal integer. When the output is to be written on cards, it is often important to be able to fix the number of digits, regardless of the specific character, to ensure correct positioning of the various fields. Since there are 48 characters available on the IBM 7044, the output routines for this machine always write the integer as ~~at least~~ two digits. This means that a parameter call of the form < flag > dl will always result in exactly two digits.

The control characters are read from the flag line into positions 1 through 5 of the array by the DO-loop at statement 1. The variable K is set to point to position 16, the first available for macro definitions, and the definition read routine is entered.

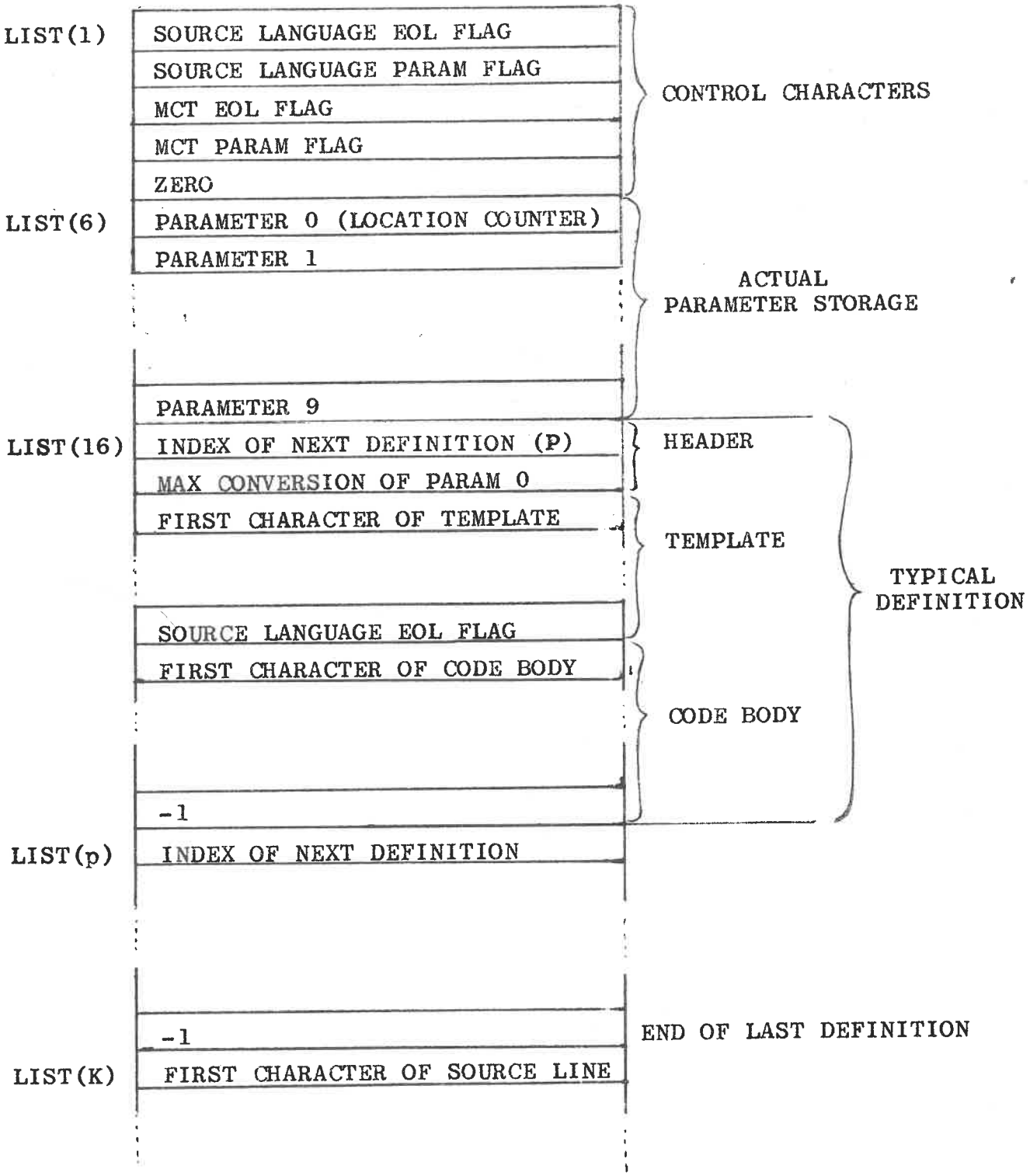

LIST(1)
SOURCE LANGUAGE EOL FLAG
SOURCE LANGUAGE PARAM FLAG
MCT EOL FLAG
MCT PARAM FLAG
ZERO
CONTROL CHARACTERS
LIST(6)
PARAMETER 0 (LOCATION COUNTER)
PARAMETER 1
ACTUAL PARAMETER STORAGE
PARAMETER 9
LIST(16)
INDEX OF NEXT DEFINITION (P)
MAX CONVERSION OF PARAM 0
HEADER
FIRST CHARACTER OF TEMPLATE
TEMPLATE
SOURCE LANGUAGE EOL FLAG
FIRST CHARACTER OF CODE BODY
CODE BODY
TYPICAL DEFINITION
-1
LIST(p)
INDEX OF NEXT DEFINITION
-1
END OF LAST DEFINITION
LIST(K)
FIRST CHARACTER OF SOURCE LINE

FIGURE IV-1

While a definition is being read, K addresses the first word of the header and L the second. I is used as a running index for storing the characters as they are read. The second header word is initially set to -1, its value if the macro does not use parameter 0. The loop at statement 3 reads the template until a source language EOL flag is found. The code body is then brought in, and each character checked for MCT parameter flag and MCT EOL flag. An MCT EOL flag is replaced by -1 in the statement following 12, and a check is made for a void line. If the line is void (which means that the definition is complete), the first word of the header is set to point to the available space. If the second character of the line is also an MCT EOL flag, the definitions are complete. Otherwise control returns to statement 6 for a new definition.

When an MCT parameter flag is read, I is not moved. The next character is read and the integer -6-d is computed. I is then advanced and the conversion digit read and converted to the integer c. If the parameter is 0 (i.e. if -6-d = -6), then the second word of the header is checked against the conversion digit and altered if necessary.

After all definitions have been read, the statement input routine is entered at 20 and the first statement is placed in the array starting above the last definition. The remainder of the array is available for statement storage, but the DO-loop which reads the statement is interrupted by the occurrence of a source language EOL flag. If the routine attempts to read past the end of the array, control passes to 100 where a dump of the last 200 positions is taken.

The translation routine compares the input line with each template in turn, defining the actual parameters as those characters which match the parameter flags in the template. If no match can be found, the offending line is printed out by the error routine at 50, and a new statement is read. If a match <u>is</u> found, the MCT output routine is entered at 41.

The MCT output routine sequences through the code body, comparing each integer to -1. If it is not less than -1, the punch routine is called to write it out as a character. Recall that a negative integer causes the punch routine to finish the current line--this is the reason for replacing each MCT EOL flag by -1.

When an integer less than -1 is detected, it must be a call on a parameter. At 40, L is pointed to the parameter store and J is advanced to the conversion specification. If the call is for parameter 0, the correct integer is output by the routine at 46. Otherwise, the value of the actual parameter is written with the correct conversion by the statement just before 41.

Each time that a normal character (not a parameter call) is written, a check is made for the end of the code body. This is done by testing the current value of the pointer against the location of the first headword of the next macro (the first headword of a macro contains the address of the first headword of the next). If the end has been reached, parameter 0 is updated by the contents of the second headword of the current macro, plus 1. This ensures that the integers generated in future calls of parameter 0 will be unique.

The comments of the program indicate how it should be modified if no unique, generated labels are required. These modifications eliminate all tests of parameter 0, as well as the mechanism for updating LIST(6). The second headword of the macro definitions will also be eliminated by these changes, thus saving one word per definition.

should fit on 16K machine

```
      DIMENSION LIST(15000)
        COMMON LIST
        DATA KMAX/15000/
C   IF NO GENERATED LABELS ARE REQUIRED, DELETE THE FOLLOWING CARD.
        LIST(6)=100
C
C       READ CONTROL CHARACTERS, IN THE ORDER -
C                    SOURCE EOL, SOURCE PARAM, MCT EOL, MCT PARAM, ZERO.
C
        DO 1 I=1,5
      1 LIST(I)=INPUT(1)
        K=16
C
C       READ MACRO DEFINITIONS
C
      2 J=INPUT(0)
C   IF NO GENERATED LABELS ARE REQUIRED, REPLACE THE FOLLOWING 3 CARDS
C   WITH -
C       I=K
        L=K+1
        LIST(L)=-1
        I=L
      3 IF (I .GE. KMAX)  GO TO 100
        I=I+1
        LIST(I)=INPUT(1)
        IF (LIST(I) .NE. LIST(1))  GO TO 3
      6 J=INPUT(0)
        J=I+1
      7 I=I+1
        IF (I .GE. KMAX)  GO TO 100
        LIST(I)=INPUT(1)
        IF (LIST(I) .NE. LIST(4))  GO TO 12
        LIST(I)=LIST(5)-INPUT(1)-6
C   IF NO GENERATED LABELS ARE REQUIRED, DELETE THE FOLLOWING CARD.
        J=I
        I=I+1
        LIST(I)=INPUT(1)-LIST(5)
C   IF NO GENERATED LABELS ARE REQUIRED, DELETE THE FOLLOWING 2 CARDS.
        IF (LIST(J) .NE. (-6))  GO TO 7
        IF (LIST(I) .GT. LIST (L))  LIST(L)=LIST(I)
        GO TO 7
     12 IF (LIST(I) .NE. LIST(3))  GO TO 7
        LIST(I)=-1
        IF (I .GT. J)  GO TO 6
        LIST(K)=I
        K=I
        J=INPUT(1)
        IF (J .NE. LIST(3))  GO TO 2
C
C       READ A SOURCE STATEMENT
C
     20 J=INPUT(0)
        DO 21 I=K,KMAX
        LIST(I)=INPUT(1)
        IF(LIST(I) .EQ. LIST(1))  GO TO 30
     21 CONTINUE
        GO TO 100
```

```
C
C      TRANSLATE ONE STATEMENT
C
   30 IF (I .EQ. K) CALL FXIT
      M=16
   31 L=7
      DO 34 J=K,I
C  IF NO GENERATED LABELS ARE REQUIRED, REPLACE THE FOLLOWING CARD WITH-
C      N=J-K+M+1
      N=J-K+M+2
      IF (LIST(N) .EQ. LIST(2))  GO TO 33
      IF(LIST(N) .EQ. LIST(J))  GO TO 34
      M=LIST(M)
      IF(M-K)31,50,50
   33 LIST(L)=LIST(J)
      L=L+1
   34 CONTINUE
      J=N
      GO TO 41
C
C      PUNCH MACHINE CODE TRANSLATION
C
   40 L=-LIST(J)
      J=J+1
C  IF NO GENERATED LABELS ARE REQUIRED, DELETE THE FOLLOWING CARD.
      IF (L .EQ. 6)  GO TO 46
      CALL PUNCH (LIST(L),LIST(J))
   41 J=J+1
      IF (LIST(J) .LT. (-1))  GO TO 40
      CALL PUNCH (LIST(J),0)
C  IF NO GENERATED LABELS ARE REQUIRED, REPLACE THE FOLLOWING 7 CARDS
C WITH -
C      IF (J+1-LIST(M)) 41,20,20
      IF (J+1 .LT. LIST(M))  GO TO 41
      M=M+1
      LIST(6)=LIST(6)+LIST(M)+1
      GO TO 20
   46 L=LIST(6)+LIST(J)
      CALL PUNCH (L,1)
      GO TO 41
C
C      PRINT OUT AN ERROR
C
   50 DO 51 J=K,I
   51 CALL PRINT(LIST(J),0)
      CALL PRINT(-1,0)
      GO TO 20
C
C      SYSTEM ERROR MESSAGES
C
  100 I = KMAX-200
      CALL DUMP(LIST(I),LIST(KMAX),0)
      END
```

Appendix V

The Basic Wisp Compiler, BASCMP

BASCMP is written using a subset of the operations of Basic Wisp. The format of the program conforms to the restrictions imposed by SIMCMP, and no dynamic storage allocation is required. The environment must contain I/O (the same as that used for SIMCMP) and the storage maintenance routines described in section IV(D).

(1) Definitions required by BASCMP

```
* = *.
* = '*.
* = CAR *.
* = CDR *.

CAR * = *.                    CDR * = *.
CAR * = '*.
CAR * = NIL.                  CDR * = NIL.
CAR * = CAR *.
CAR * = CDR *.                CDR * = CDR *.

**.
TO **.
STOP.

TO ** IF CAR * = '*.          TO ** IF CAR * ≠ '*.
TO ** IF CAR * = NIL.         TO ** IF CAR * ≠ NIL.
TO ** IF CAR * = ATOM.
TO ** IF CAR * = CAR *.       TO ** IF CAR * ≠ CAR *.

TO ** IF CDR * = NIL.

PUSH DOWN *.

START LIST *.

TO ** AND BACK.               RETURN.

SAVE *.                       RECALL *.

    TØ ** IF STACK EMPTY.
```

```
                    CAR * = INPUT.
                    NEXT LINE ON INPUT.
PRINT '*.                          PUNCH CAR *.
PRINT CAR *.                       FINISH LINE ON PUNCH.
FINISH LINE ON PRINT.              PUNCH DECIMAL CAR *.
                    ASSEMBLE.
```

Several of the above operations have not been discussed previously: "TO ** IF STACK EMPTY." tests the pointers TOP and ADDR for equality, and if they are equal control passes to location **. Consideration of Figures 26 and 27 will reveal that this condition occurs if and only if the subroutine in control has no information stored in the stack.

Type 1 conversion is implemented in BASCMP by means of the operation "PUNCH DECIMAL CAR *.". This operation is just a call on PUNCH(j,k) with k = 1 and j being the integer corresponding to the character *. (A more detailed description of PUNCH(j,k) is given in Appendix IV.)

The macro "ASSEMBLE." is executed when the compilation is complete and no errors have been found. It is used to call in the translator for the symbolic assembly language of the target machine to complete translation of a program placed on some intermediate storage device by BASCMP. Effective use of this operation requires that Wisp be integrated with the overall operating system used by a particular installation. Unfortunately very little can be said about this challenging problem, because of the variety of possibilities. The basic procedure is to modify PUNCH(j,k) in such a way that it stacks its output on some intermediate storage device. When "ASSEMBLE." is executed, the assembly and eventual running of the stacked program should be initiated. If the compiler executes "STOP.", then it has detected an error and the stacked program should be abandoned. If such a procedure is not possible then the output must be punched, and the operations "STOP." and "ASSEMBLE." are synonymous.

(2) The Macro Test Program

The following program is designed to check the operation of the macros required by BASCMP. The program assumes that the operations "**.", "TO **." and "STOP." are correct. The input data consists of two lines, as follows:

```
THE QUICK FOX.1
THE QUICK FOX.2
```

The printed output should be

```
THE QUICK BROWN FOX JUMPED.
OK
```

The punched output should be:

```
THE QUICK FOX.
OK - n
```

The lower case "n" in the above output should be the integer resulting from a type 1 conversion of the character "period".

The correct output should not appear if any macro contains an error. Each symbol is dependent primarily upon one or two macros, as shown in the program's comments. Any error may mean that future errors will be compounded or masked. The operation "START LIST *." is basic to the entire system and must be working correctly to allow the test program to even begin execution. (There is also a specific sequence to test out the subtler aspects of this operation.) An error in START LIST will seriously upset the remainder of the program, so that if no correct results are obtained this macro should be checked first.

When the output routines PRINT(j,k) and PUNCH(j,k) were described, there was no mention of writing out the buffers on exit. For SIMCMP and BASCMP this is not required because both compilers use FINISH LINE operations at the end of each output line. (FINISH LINE clears the output buffer, passing the information to the output device or system.) For the test program, and any user program, failure to use FINISH LINE causes the loss of any information remaining in the buffer at the time STOP is executed. If the test program gives no output at all, then it is perhaps wise to examine the FINISH LINE macros carefully.

```
START LIST A.
START LIST B.
START LIST C.
START LIST D.
START LIST E.
START LIST F.
START LIST G.
START LIST H.
START LIST I.
START LIST J.
START LIST K.
START LIST L.
START LIST M.
START LIST N.
START LIST O.
START LIST P.
START LIST Q.
START LIST R.
START LIST S.
START LIST T.
START LIST U.
START LIST V.
START LIST W.
START LIST X.
START LIST Y.
START LIST Z.
START LIST 0.
START LIST 1.
START LIST 2.
START LIST 3.
START LIST 4.
START LIST 5.
START LIST 6.
START LIST 7.
START LIST 8.
START LIST 9.
CAR A = INPUT.
PRINT CAR A.                    T     CAR * = INPUT, PRINT CAR *.
CAR B = INPUT.
C = B.
PRINT CAR C.                    H     * = *.
CAR D = INPUT.
CAR E = CAR D.
PRINT CAR E.                    E     CAR * = CAR *.
TO 24 IF CAR F NE NIL.
TO 25 IF CAR A NE NIL.
24.
PRINT CAR A.
TO 17.
25.
CAR F = INPUT.
PRINT CAR F.                    SPACE TO ** IF CAR * NE NIL.
G = CDR F.
CAR G = INPUT.
F = CDR F.
PRINT CAR F.                    Q     * = CDR *.
CAR I = INPUT.
CAR H = INPUT.
J = CAR H.
K = CDR J.
```

```
PRINT CAR K.                    U       * = CAR *.
CAR L = J.
PRINT CAR L.                    I       CAR * = *.
CAR M = INPUT.
CDR N = M.
N = CDR N.
PRINT CAR N.                    C       CDR * = *.
TO 17 IF CAR A = 'P.
TO 02 IF CAR A = 'T.
TO 17.
02.
CAR O = INPUT.
PRINT CAR O.                    K       TO ** IF CAR * = '*.
TO 17 IF CAR A = CAR B.
TO 04 IF CAR D = CAR E.
PRINT CAR A.
TO 17.
04.
CAR P = INPUT.
PRINT CAR P.                    SPACE TO ** IF CAR * = CAR *.
PRINT 'B.                       B       PRINT '*.
CAR Q = 'R.
PRINT CAR Q.                    R       CAR * = '*.
T = 'O.
CAR U = T.
PRINT CAR U.                    O       * = '*.
CAR V = CDR W.
V = CAR V.
CAR V = 'W.
CAR X = CDR W.
X = CAR X.
PRINT CAR X.                    W       CAR * = CDR *.
TO 17 IF CAR X = NIL.
TO 06 IF CAR Y = NIL.
TO 17.
06.
PRINT 'N.                       N       TO ** IF CAR * = NIL.
TO 17 IF CDR W = NIL.
TO 08 IF CDR Y = NIL.
PRINT 'Z.
TO 17.
08.
PRINT ' .                       SPACE TO ** IF CDR * = NIL.
CAR X = NIL.
TO 09 IF CAR X = NIL.
PRINT 'Z.
TO 17.
09.
CAR X = INPUT.
PRINT CAR X.                    F       CAR * = NIL.
CDR W = NIL.
TO 10 IF CDR W = NIL.
TO 17.
10.
CAR Z = INPUT.
PRINT CAR Z.                    O       CDR * = NIL.
CAR Y = Z.
TO 17 IF CAR Y = ATOM.
TO 12 IF CAR X = ATOM.
TO 17.
```

```
12.
CAR 0 = INPUT.
PRINT CAR 0.                    X       TO ** IF CAR * = ATOM.
CDR 0 = 1.
CDR 0 = CDR 1.
TO 14 IF CDR 0 = NIL.
PRINT 'Z.
TO 17.
14.
PRINT ' .                       SPACE   CDR * = CDR *.
TO 17 IF CAR 0 NE 'X.
TO 16 IF CAR 1 NE 'X.
TO 17.
16.
PRINT 'J.                       J       TO ** IF CAR * NE '*.
TO 17 IF CAR 1 NE CAR 2.
TO 18 IF CAR 0 NE CAR 1.
17.
FINISH LINE ON PRINT.
FINISH LINE ON PUNCH.
STOP.
18.
PRINT 'U.                       U       TO ** IF CAR * NE CAR *.
CAR 1 = 'M.
2 = 1.
PUSH DOWN 2.
PRINT CAR 2.
2 = CDR 2.
PRINT CAR 2.                    M       PUSH DOWN *.
T = 'P.
SAVE T.
T = 'X.
SAVE T.
T = 'V.
RECALL T.
CAR U = T.
TO 17 IF CAR U NE 'X.
RECALL T.
CAR U = T.
PRINT CAR U.                    P       SAVE *, RECALL *.
4 = 1.
CAR 1 = 'M.
CDR 1 = T.
START LIST 1.
TO 20 IF CAR 1 = NIL.
TO 17.
20.
TO 23 IF CDR 1 = NIL.
TO 17.
23.
TO 17 IF CAR 4 NE 'M.
PRINT 'E.                       E       START LIST *.
T = 'D.
SAVE T.
TO 21 AND BACK.
CAR U = T.
TO 17 IF CAR U NE 'X.
RECALL T.
CAR U = T.
PRINT CAR U.                    D       TO ** AND BACK.
```

```
TO 22 IF STACK EMPTY.
TO 17.
21.
T = 'X.
SAVE T.
RETURN.
22.
SAVE T.
TO 17 IF STACK EMPTY.
CAR U = INPUT.
PRINT CAR U.                          .     TO ** IF STACK EMPTY.
FINISH LINE ON PRINT.
PRINT 'O.                             O     FINISH LINE ON PRINT.
NEXT LINE ON INPUT.
CAR 5 = INPUT.
TO 17 IF CAR 5 = '1.
PRINT 'K.                             K     NEXT LINE ON INPUT.
26.
PUNCH CAR 5.                          PUNCH THE SECOND INPUT LINE AS IT STANDS.
CAR 5 = INPUT.
TO 26 IF CAR 5 NE '2.
FINISH LINE ON PUNCH.
PUNCH CAR Z.                          O     FINISH LINE ON PUNCH.
PUNCH CAR O.
PUNCH CAR F.
PUNCH DECIMAL CAR U.                  DECIMAL EQUIVALENT OF PERIOD.
TO 17.
```

(3) The Basic Compiler

The compiler consists of 9 modules, each of which performs one particular function. The references within each module begin with the same digit, as follows:

0 - Supervisor

1 - Macro definition input

2 - Statement input

3 - Statement recognition

4,5 - Machine code output

6 - Reference label recognition

7 - Reference label generation and list definition

8 - Reference table output

9 - Program listing output

The supervisor handles initialization and the normal sequencing of "read a statement", "translate", "output code" and "print". The macro definition input routine uses the definitions supplied by the programmer to create the definition we described in Section IV(D). This routine is entered by the supervisor at the beginning of the compilation, and also by the machine code output routine when parameter S0 is encountered.

The statement input routine reads a single statement onto list I, editing it by removing all redundant spaces and replacing the terminating delimiter (period or comma) with the atom NIL. Statement recognition compares the statement to the macro tree as described in Section IV(D), defining all actual parameters. The machine code output routine is analogous to that of SIMCMP, but with more complex behavior for reference label conversions. The mechanisms for defining and referencing labels are found in the label recognition routine, while label generation provides the unique integers.

At the end of the compilation, reference tables are written by the reference table output routine. Program listing is provided by the last routine, which handles error printouts as well as printing of correct statements under the control of S2 and S3.

Each error detected by BASCMP causes an error flag to appear on the program listing. These flags are printed at the left margin, and their meanings are:

0 Operation code error (invalid Wisp statement)

L List reference error (list name is more than one character and was not previously defined)

C Compiler switch error (the macro uses a non-existent conversion of parameter S)

U Undefined reference label

M Multiply defined reference label.

```
START LIST A.                    A, B, C, AND D ARE FLAG LISTS.
START LIST B.
START LIST C.
START LIST D.
START LIST I.                    INPUT STRING.
START LIST K.                    COMPILER ERROR FLAG.
START LIST L.                    LABEL GENREATOR.
START LIST M.                    TEMPORARY LOCATION AND ERROR FLAG.
START LIST N.                    LIST OF DEFINED LABELS.
START LIST P.                    FLAG FOR PRINT ROUTINE.
START LIST 1.                    PARAMETER LIST HEADERS.
2 = CDR 1.
3 = CDR 2.
4 = CDR 3.
5 = CDR 4.
6 = CDR 5.
7 = CDR 6.
8 = CDR 7.
9 = CDR 8.
0 = CDR 9.
F = '1.                          SET UP THE PARAMETER LISTS.
01.
F = CDR F.                       STEP TO THE NEXT PARAMETER LIST.
START LIST G.
CAR F = G.
TO 01 IF CAR F NE CAR 0.         ARE WE DONE YET, NO.
CAR A = INPUT.                   WISP END-OF-LINE FLAG.
CAR B = INPUT.                   MCT PARAMETER FLAG.
CAR C = INPUT.                   MCT END-OF-LINE FLAG.
CAR D = INPUT.                   NUMBER OF DIGITS IN A LABEL.
CAR I = CAR A.                   SET FOR A NEW LINE.
TO 20 AND BACK.                  READ THE FIRST TEMPLATE.
F = 'E.
CDR F = CDR I.                   ATTACH THE TEMPLATE TO THE TREE.
CDR I = NIL.                     REMOVE IT FROM THE INPUT LIST.
02.                              READ THE FIRST STANDARD FORM.
F = CDR F.
TO 02 IF CAR F NE NIL.           HAVE WE REACHED THE END, NO.
TO 12 AND BACK.                  YES, GET THE MCT AND READ ALL DEFINITIONS.
03.                              RE-ENTRY FOR ANOTHER COMPILATION.
O = L.                           SET UP THE LABEL GENERATOR.
TO 04 IF CAR D = '0.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '1.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '2.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '3.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '4.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '5.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '6.
```

```
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '7.
CAR O = '0.
O = CDR O.
TO 04 IF CAR D = '8.
CAR O = '0.
O = CDR O.
04.
CAR O = 'U.                          SET THE FLAG ENDING THE GENERATOR.
O = N.                               SET THE POINTER TO THE LABEL LIST.
START LIST T.                        START THE LABEL TREE.
CAR K = '+.                          SET THE PUNCH SWITCH.
CAR P = '2.                          TURN OFF PRINTING.
CAR I = CAR A.                       SET TO SKIP TO THE FIRST PROGRAM LINE.
05.                                  COMPILER LOOP.
TO 20 AND BACK.                      READ ONE INPUT STATEMENT.
TO 30 AND BACK.                      COMPILE ONE STATEMENT.
TO 05 IF CAR M NE ' .                WERE THERE ERRORS, YES.
TO 41 AND BACK.                      NO, PUT OUT MACHINE CODE.
Z = CAR O.                           RESET PARAMETER ZERO.
CAR Z = NIL.
TO 05 IF CAR M NE ' .                WAS THE STATEMENT CORRECT, NO.
TO 05 IF CAR P = '2.                 YES, SHOULD WE PRINT, NO.
TO 90 AND BACK.                      YES, PRINT IT.
TO 05.
10.                                  READ STANDARD FORMS ONTO THE OPS TREE.
F = 'E.                              INITIALIZE THE POINTER.
TO 20 AND BACK.                      READ ONE INPUT LINE.
G = I.                               SET G ONTO THE INPUT.
11.                                  MATCH THE INPUT TO THE CURRENT BRANCH.
F = CDR F.
G = CDR G.
TO 14 IF CAR F NE CAR G.             DOES THE INPUT MATCH THE TREE, NO.
09.
TO 11 IF CAR F NE NIL.               ARE WE FINISHED WITH THIS LINE, NO.
12.                                  READ THE MACHINE CODE TRANSLATION.
F = CDR F.
NEXT LINE ON INPUT.
CAR F = INPUT.
TO 13 IF CAR F NE CAR C.             IS THE FIRST CHARACTER AN EOL, NO.
CAR M = INPUT.                       YES, GET THE SECOND.
TO 10 IF CAR M NE CAR C.             HAVE WE REACHED THE END OF THE DEFINITIONS.
RETURN.                              YES, GET OUT.
13.                                  READ ONE LINE OF THE MCT.
F = CDR F.
CAR F = INPUT.
TO 13 IF CAR F NE CAR C.             IS THIS THE END OF THE MCT LINE, NO.
TO 12.
14.                                  A MISMATCH HAS OCCURRED.
TO 19 IF CAR F = ATOM.               ARE WE AT A NODE, NO.
TO 15 IF CAR G NE '*.                YES, IS THE CHARACTER A PARAMETER, NO.
F = CDR F.
TO 11 IF CAR F = '*.                 IS THERE A PARAMETER HERE IN THE TREE, YES.
TO 18.
15.
F = CAR F.
16.
TO 09 IF CAR F = CAR G.
TO 17 IF CAR F = ATOM.
```

```
X = CAR F.
F = CDR F.
TO 16 IF CAR G NE CAR X.
F = X.
TO 09.
17.                                  CREATE A NEW BRANCH AT THIS NODE.
START LIST X.                        GET A NEW ELEMENT.
PUSH DOWN F.
CAR F = X.                           ATTACH IT TO THE TREE.
F = X.                               SET THE POINTER ONTO THE NEW BRANCH.
18.                                  FILL THE NEW BRANCH.
CAR F = CAR G.                       COPY ONE INPUT CHARACTER.
TO 12 IF CAR F = NIL.                HAVE WE REACHED THE END, YES.
F = CDR F.                           NO, GET THE NEXT ATOM.
G = CDR G.
TO 18.
19.                                  WE MUST CREATE A NODE.
TO 17 IF CAR F = '*.                 IS THERE A PARAMETER ON THE TREE, YES.
PUSH DOWN F.                         NO, SAVE THE SPACE FOR ONE
CAR F = CDR F.
CDR F = NIL.
TO 15 IF CAR G NE '*.                IS THE INPUT A PARAMETER FLAG, NO.
F = CDR F.                           YES, PLACE IT ON THE PARAMETER BRANCH.
CAR F = '*.
TO 18.
20.                                  READ ONE INPUT STATEMENT.
G = CDR I.
TO 21 IF CAR I NE CAR A.             SHOULD WE GO TO THE NEXT LINE, NO.
NEXT LINE ON INPUT.
CAR I = ',.
21.                                  ELIMINATE ANY LEADING SPACES.
CAR G = INPUT.
TO 21 IF CAR G = ' .
22.                                  CHECK AN INPUT CHARACTER.
TO 25 IF CAR G = ' .
TO 28 IF CAR G = ',.
TO 27 IF CAR G = CAR A.
23.                                  CHECK A CHARACTER WHICH IS NOT A
TO 24 IF CAR G NE ''.                SPACE OR DELIMITER.
G = CDR G.                           DO NOT CHECK A CHARACTER WHICH IS
CAR G = INPUT.                       PRECEEDED BY A QUOTE.
24.                                  READ AND CHECK THE NEXT CHARACTER.
G = CDR G.
CAR G = INPUT.
TO 22.
25.                                  ELIMINATE REDUNDANT SPACES.
CAR M = INPUT.
TO 25 IF CAR M = ' .
TO 28 IF CAR M = ',.
TO 27 IF CAR M = CAR A.
G = CDR G.                           THE CHARACTER IS NOT A SPACE OR A DELIMITER
CAR G = CAR M.
TO 23.
27.
CAR I = CAR A.                       SET TO SKIP TO THE NEXT LINE.
28.                                  THE CHARACTER IS AN END-OF-STATEMENT.
CAR G = NIL.
29.                                  COMMON SYSTEM RETURN POINT.
RETURN.
30.                                  COMPILE ONE STATEMENT.
```

```
F = E.                          SET THE OPERATIONS TREE POINTER.
J = 1.                          RESET THE PARAMETER POINTER
CAR M = ' .                     AND ERROR FLAG.
G = CDR I.                      SET THE STATEMENT POINTER.
TO 33.
32.                             MOVE TO THE NEXT CHARACTER.
TO 29 IF CAR F = NIL.           HAVE WE MATCHED THE WHOLE STATEMENT, YES.
F = CDR F.                      NO, STEP ALONG THE CURRENT BRANCH.
G = CDR G.                      AND STATEMENT.
33.
TO 37 IF CAR F = '*.            SHOULD THIS BE A PARAMETER, YES.
TO 32 IF CAR F = CAR G.         NO, DOES THE STATEMENT STILL MATCH, YES.
TO 36 IF CAR F = ATOM.          NO, ARE WE AT A NODE, NO.
TO 34 IF CDR F = NIL.           YES, IS THERE A PARAMETER POSSIBLE, NO.
SAVE F.                         YES, SAVE THE POINTER.
SAVE J.                         PARAMTER POINTER.
SAVE G.                         INPUT POINTER.
34.                             CHECK A NODE.
F = CAR F.                      MOVE ONTO THE NODE.
35.                             SCAN A NODE.
TO 32 IF CAR F = CAR G.         DOES THE STATEMENT MATCH, YES.
TO 36 IF CAR F = ATOM.          NO, ARE WE STILL AT THE NODE, NO.
X = CAR F.                      YES, CHECK THE NEXT BRANCH.
F = CDR F.
TO 35 IF CAR G NE CAR X.        DOES IT MATCH, NO.
F = X.                          YES, SCAN IT.
TO 32.
36.                             WE HAVE HAD A MISMATCH.
TO 98 IF STACK EMPTY.           ARE THERE ANY MORE POSSIBILITIES, NO.
RECALL G.                       YES, RECALL THE INPUT POINTER.
RECALL J.                       PARAMETER POINTER.
RECALL F.                       TREE POSITION.
F = CDR F.                      STEP TO THE PARAMETER BRANCH.
37.                             DEFINE A PARAMETER.
TO 36 IF CAR G = NIL.           THE LAST CHARACTER CANNOT BE A PARAMETER.
Z = CAR J.                      POINT Z TO THE CURRENT PARAMETER STORE.
J = CDR J.                      INCREMENT THE PARAMETER STORE.
F = CDR F.
38.                             GET THE PARAMETER STRING.
CAR Z = CAR G.
TO 33 IF CAR G = NIL.           HAS THE PARAMETER ENDED, YES.
G = CDR G.
Z = CDR Z.
TO 38 IF CAR G NE ' .           HAS THE PARAMETER ENDED, NO.
CAR Z = NIL.                    TERMINATE THE PARAMETER
TO 33.
40.                             MCT OUTPUT ROUTINE.
TO 41 IF CAR K = '-.            IS PUNCHING SUSPENDED, YES.
TO 41 IF CAR K = '+.            NO, ARE WE AT THE BEGINNING OF A LINE, YES.
FINISH LINE ON PUNCH.
CAR K = '+.
41.                             PUNCH ONE LINE OF THE MCT.
F = CDR F.
TO 29 IF CAR F = CAR C.         IS THE MCT COMPLETE, YES.
42.                             CHECK ONE CHARACTER.
TO 43 IF CAR F = CAR B.         IS IT A PARAMETER FLAG, YES.
TO 40 IF CAR F = CAR C.         NO, IS IT THE END OF THE CURRENT LINE, YES.
TO 45 IF CAR K = '-.            NO, IS PUNCHING SUSPENDED, YES.
PUNCH CAR F.                    NO, PUT IT OUT.
CAR K = 'P.                     SET THE PUNCH FLAG.
```

```
F = CDR F.                          CHECK THE NEXT.
TO 42.
43.                                 EVALUATE A PARAMETER.
F = CDR F.                          GET THE PARAMETER NUMBER.
TO 46 IF CAR F = 'S.                IS IT A COMPILER SWITCH, YES.
Z = CAR F.                          NO, SET Z TO THE PARAMETER DEFINITION.
F = CDR F.                          GET THE TYPE OF CONVERSION REQUIRED.
Z = CDR Z.
TO 51 IF CAR Z = CAR 0.             IS THIS A CALL ON PARAMETER ZERO, YES.
Z = CAR Z.
TO 49 IF CAR F = '1.                LIST REFERENCE.
TO 66 IF CAR F = '2.                LABEL REFERENCE.
TO 66 IF CAR F = '3.                LABEL DEFINITION.
TO 45 IF CAR K = '-.                WAS PUNCHING SUSPENDED. YES.
44.                                 COPY THE ARGUMENT.
PUNCH CAR Z.
Z = CDR Z.
TO 44 IF CAR Z NE NIL.
CAR K = 'P.                         SET THE PUNCH FLAG.
45.
F = CDR F.                          STEP TO THE NEXT CHARACTER OF THE MCT.
TO 42.
46.                                 THE PARAMETER IS A COMPILER SWITCH.
F = CDR F.                          GET THE SWITCH VALUE.
TO 10 IF CAR F = '0.                DEFINE A NEW STANDARD FORM.
TO 47 IF CAR F = '1.                COPY MACHINE CODE TO A VOID LINE.
TO 96 IF CAR F = '2.                SET TO PRINT ALL SOURCE TEXT.
TO 97 IF CAR F = '3.                SUPPRESS PRINTING OF CORRECT SOURCE TEXT.
TO 80 IF CAR F = '4.                WRAP UP THE COMPILATION.
TO 78 IF CAR F = '5.                DEFINE NEW LISTS.
CAR M = 'C.                         SET THE COMPILER SWITCH ERROR FLAG.
TO 99.
47.                                 COPY MACHINE CODE.
NEXT LINE ON INPUT.
CAR M = INPUT.
TO 45 IF CAR M = CAR C.
TO 47 IF CAR K = '-.                WAS PUNCHING SUSPENDED, YES.
TO 48 IF CAR K = '+.                NO, ARE WE AT THE BEGINNING OF A LINE, YES.
FINISH LINE ON PUNCH.               NO, FINISH THE CURRENT ONE.
CAR K = '+.
48.                                 COPY ONE LINE.
PUNCH CAR M.
CAR M = INPUT.
TO 48 IF CAR M NE CAR C.            HAVE WE COME TO EOL, NO.
FINISH LINE ON PUNCH.
TO 47.                              YES, CHECK THE NEXT LINE.
49.                                 LOOK UP A DEFINED LIST REFERENCE.
Y = CDR Z.
TO 50 IF CAR Y NE NIL.              IS THIS A SPECIAL LIST, YES.
TO 45 IF CAR K = '-.                NO, HAS PUNCHING BEEN SUSPENDED, YES.
PUNCH DECIMAL CAR Z.                NO, PUNCH ITS RELATIVE ADDRESS.
CAR K = 'P.                         SET THE PUNCH FLAG.
TO 45.
50.
TO 60 AND BACK.
TO 69 IF CAR Y = 'D.                IS THE LIST PROPERLY DEFINED, YES.
CAR M = 'L.                         NO, SET THE LIST NAME ERROR FLAG.
TO 99.
51.                                 PARAMETER ZERO CONVERSION.
Y = CAR Z.                          CHECK THE NEXT LABEL.
```

```
TO 69 IF CAR F = CAR Y.          IS THIS THE ONE, YES.
Z = CDR Z.                       NO, MOVE ALONG THE LIST OF LABELS.
TO 51 IF CAR Y NE NIL.           HAVE WE RUN OUT OF LABELS, NO.
TO 52 IF CAR Z = NIL.            YES, IS THERE SPACE ON THE LIST, NO.
Z = CAR Z.                       YES, GO AND DELETE THE NEXT LABEL.
CAR Z = NIL.
TO 53.
52.                              START A NEW SUBLIST.
START LIST X.
CAR Z = X.
53.                              GENERATE AND ATTACH A NEW LABEL.
CAR Y = CAR F.                   SET THE NAME.
TO 70 AND BACK.                  SET UP THE NEXT LABEL FROM THE GENERATOR.
TO 69.                           COPY IT ONTO THE OUTPUT STREAM.
60.                              LOOK UP A LABEL.
Y = T.
TO 64 IF CDR Y = NIL.            HAS THE LABEL TREE BEEN STARTED, NO.
61.
TO 62 IF CAR Y = CAR Z.          NO, ARE WE STILL ON THE RIGHT BRANCH, YES.
TO 63 IF CAR Y = ATOM.           NO, MUST WE CREATE A NEW BRANCH, YES.
X = CAR Y.                       NO, LOOK AT THE CURRENT BRANCH.
Y = CDR Y.
TO 61 IF CAR X NE CAR Z.         DOES IT MATCH, NO.
Y = X.
62.
TO 65 IF CAR Y = NIL.            HAVE WE MATCHED THE TREE BRANCH, YES.
Y = CDR Y.                       CONTINUE SCANNING.
Z = CDR Z.
TO 61.
63.                              WE MUST CREATE A NEW BRANCH.
PUSH DOWN Y.
START LIST X.
CAR Y = X.
Y = X.
TO 65 IF CAR Z = NIL.            HAVE WE REACHED THE END, YES.
64.
CAR Y = CAR Z.                   COPY ONE CHARACTER OF THE PARAMETER.
Y = CDR Y.
Z = CDR Z.
TO 64 IF CAR Z NE NIL.
65.
Y = CDR Y.                       STEP TO THE FLAG.
RETURN.
66.                              DEAL WITH A LABEL.
TO 60 AND BACK.
TO 67 IF CDR Y = NIL.            IS THIS A NEW LABEL, YES.
TO 68 IF CAR Y = 'U.             HAS IT BEEN DEFINED, NO.
TO 45 IF CAR Y = 'M.             YES, HAS IT BEEN MULTIPLY DEFINED, YES.
TO 69 IF CAR F = '2.             IS THIS A MULTIPLE DEFINITION, NO.
CAR Y = 'M.                      YES, SET THE MULTIPLE DEFINITION FLAG.
CAR K = '-.                      SUSPENDED PUNCHING.
TO 45.                           CONTINUE SCANNING THE MCT.
67.
CAR O = Y.                       ATTACH A NEW LABEL TO THE LIST.
O = CDR O.
TO 70 AND BACK.                  GET A NEW LABEL.
CAR Y = 'U.                      SET THE 'UNDEFINED' FLAG.
68.
TO 69 IF CAR F = '2.             IS THIS A LABEL DEFINITION, NO.
CAR Y = ' .                      YES, MARK THE LABEL AS DEFINED.
```

```
69.
Z = CDR Y.
TO 44 IF CAR K NE '-.          HAS PUNCHING BEEN SUSPENDED, NO.
TO 45.                         YES, CONTINUE SCANNING THE MCT.
70.                            CREATE A NEW LABEL.
Z = L.
71.                            BUMP THE CONTENTS OF LIST Z.
SAVE X.
X = '1.
TO 72 IF CAR Z = 'U.
TO 73 IF CAR Z = '0.
X = '2.
TO 73 IF CAR Z = '1.
X = '3.
TO 73 IF CAR Z = '2.
X = '4.
TO 73 IF CAR Z = '3.
X = '5.
TO 73 IF CAR Z = '4.
X = '6.
TO 73 IF CAR Z = '5.
X = '7.
TO 73 IF CAR Z = '6.
X = '8.
TO 73 IF CAR Z = '7.
X = '9.
TO 73 IF CAR Z = '8.
X = '0.
CAR Z = '0.
Z = CDR Z.                     THERE IS A CARRY.
TO 71.
72.                            ADD A DIGIT TO LIST Z.
CAR Z = '1.
Z = CDR Z.
CAR Z = 'U.
SAVE X.
TO 75.
73.
CAR Z = X.
74.                            COMPLETE THE LABEL.
SAVE X.
Z = CDR Z.
X = CAR Z.
TO 74 IF CAR Z NE 'U.          HAVE WE REACHED THE END, NO.
75.                            YES, COPY THE LABEL IN THE RIGHT ORDER.
Z = CDR Y.
76.
RECALL X.
TO 29 IF STACK EMPTY.          HAVE WE GOT EVERYTHING, YES.
CAR Z = X.
Z = CDR Z.
TO 76.
78.                            DEFINE NEW LISTS.
TO 20 AND BACK.                READ THE LIST NAME.
Z = CDR I.
TO 45 IF CAR Z = NIL.          TERMINATE ON A VOID LINE.
TO 60 AND BACK.                LOOK UP THE NAME.
CAR Y = 'D.
CAR O = Y.                     ATTACH A NEW LABEL TO THE LABEL LIST.
O = CDR O.
```

```
TO 20 AND BACK.
CDR Y = CDR I.
79.                                  ATTACH THE DEFINITION.
Y = CDR Y.
TO 79 IF CAR Y NE NIL.
CDR I = CDR Y.
CDR Y = NIL.
TO 78.
80.                                  COMPILATION COMPLETE.
TO 08 IF CAR K NE 'P.
FINISH LINE ON PUNCH.
08.
TO 88 IF CDR T = NIL.                ARE THERE ANY LABELS, NO.
FINISH LINE ON PRINT.
FINISH LINE ON PRINT.
FINISH LINE ON PRINT.
PRINT 'R.
PRINT 'E.
PRINT 'F.
PRINT 'S.
PRINT '-.
Y = T.                               PREPARE TO REVERSE THE LABEL TREE.
T = A.
TO 81 AND BACK.                      PUT OUT THE LABELS.
88.
TO 89 IF CAR K = '-.                 WERE THERE COMPILATION ERRORS, YES.
ASSEMBLE.
89.
STOP.                                YES, DO NOT ASSEMBLE THE PROGRAM.
81.                                  SCAN A TREE BRANCH.
TO 82 IF CAR Y = ATOM.
SAVE T.                              MUST RETURN TO THE LOWER BRANCH.
SAVE Y.
Y = CDR Y.                           MOVE ALONG AN UPPER BRANCH.
TO 81.
82.                                  REVERSE THE POINTER OF AN ATOM.
TO 83 IF CAR Y = NIL.                HAVE WE REACHED THE END OF THE USERS LABEL,
Z = CDR Y.                           NO, REVERSE THE POINTER.
CDR Y = T.
T = Y.                               STEP ALONG THE BRANCH.
Y = Z.
TO 81.
83.                                  WE HAVE REACHED A LEAF OF THE TREE.
Y = CDR Y.
Y = CDR Y.                           MOVE PAST THE FLAG.
PUSH DOWN Y.                         ATTACH THE USER'S LABEL.
CAR Y = T.
TO 84 IF STACK EMPTY.                HAVE WE REACHED ALL BRANCHES, YES.
RECALL Y.                            NO, RETURN TO THE LAST ONE SAVED.
RECALL T.
Y = CAR Y.                           MOVE ONTO IT.
TO 81.
84.                                  PRINT THE NEXT LABEL.
FINISH LINE ON PRINT.
TO 29 IF CAR N = ATOM.               HAVE WE FINISHED THE REFERENCE TABLES, YES.
Y = CAR N.
N = CDR N.                           REPLACE BY 'POP UP N' TO SAVE SPACE.
PRINT CAR Y.                         PRINT THE DEFINITION FLAG.
CAR M = CAR Y.
PRINT ' .
```

```
PRINT ' .
Y = CDR Y.                          REPLACE BY 'POP UP Y' TO SAVE SPACE.
Z = CAR Y.                          SAVE THE POINTER TO THE USERS LABEL.
TO 85 IF CAR M = ' .                IS THIS LABEL PROPERLY DEFINED, YES.
TO 85 IF CAR M = 'D.                NO, IS IT A DEFINED LIST, YES.
CAR K = '-.                         NO, DELETE ASSEMBLY.
85.                                 PRINT THE COMPILER-GENERATED LABEL.
Y = CDR Y.                          REPLACE BY 'POP UP Y' TO SAVE SPACE.
PRINT CAR Y.
TO 85 IF CAR Y NE NIL.
PRINT ' .
86.                                 COPY THE USERS LABEL IN THE RIGHT ORDER.
SAVE Z.
Z = CDR Z.
TO 86 IF CAR Z NE CAR A.            ARE WE AT THE END, NO.
87.                                 PRINT THE USERS LABEL.
TO 84 IF STACK EMPTY.
RECALL Z.
PRINT CAR Z.
TO 87.
90.                                 STATEMENT PRINT ROUTINE.
TO 92 IF CAR P = '1.                YES, ARE WE AT THE BEGINNING OF A LINE, NO.
91.                                 YES, START A NEW LINE.
PRINT CAR M.                        ERROR FLAG.
PRINT ' .
TO 92 IF CAR P = '2.                IS PRINTING OF CORRECT STATEMENTS DESIRED,
CAR P = '1.                         YES, SET THE SWITCH FOR NON-EMPTY LINE.
92.
PRINT ' .
G = CDR I.
TO 94 IF CAR G = NIL.               IS THERE ANY STATEMENT, NO.
93.                                 PRINT ONE STATEMENT.
PRINT CAR G.                        PRINT THE NEXT CHARACTER.
G = CDR G.                          STEP ALONG.
TO 93 IF CAR G NE NIL.              ARE WE FINISHED, NO.
94.                                 WE HAVE COMPLETED A STATEMENT.
PRINT CAR I.                        PRINT THE TERMINATOR.
TO 95 IF CAR I = CAR A.             WAS THIS THE LAST STATEMENT OF THE LINE,YES
TO 29 IF CAR P = '1.                ARE WE PRINTING CORRECT STATEMENTS, YES.
FINISH LINE ON PRINT.               NO, PRINT ONLY ONE ERROR STATEMENT PER LINE
RETURN.
95.                                 WRAP UP A PRINT LINE.
FINISH LINE ON PRINT.
TO 29 IF CAR P = '2.
CAR P = '0.
RETURN.
96.                                 COMMENCE PRINTING ALL STATEMENTS.
CAR P = '0.
TO 45.
97.                                 SUSPEND PRINTING OF CORRECT STATEMENTS.
FINISH LINE ON PRINT.
FINISH LINE ON PRINT.
CAR P = '2.
TO 45.
98.                                 ERROR - UNRECOGNIZABLE STATEMENT.
CAR M = '0.
99.                                 ERROR PRINT OUT.
CAR K = '-.                         SUSPEND PUNCHING.
TO 91 IF CAR P NE '1.               ARE WE IN A LINE, NO.
FINISH LINE ON PRINT.               YES, TERMINATE IT.
```

TO 91.

Appendix VI

Dynamic Storage Allocator

```
        ENTRY INITAL.
        TO +ENDIN IF FREE NE NIL.        DONT INITIALIZE TWICE
        P1 = LOLIM.                      INITIALIZE SPACE POINTER.
        P2 = CAR FREE.                   PREPARE TO START LISTS.
+STRTL,CDR P2 = P1.                      START A LIST.
        CAR P1 = NIL, CDR P1 = NIL.      CLEAN THE NEW ELEMENT.
        STEP P1, P2 = CAR P2.            ADVANCE THE POINTERS.
        TO +STRTL IF P2 NE NIL.          ARE THERE MORE LISTS, YES.
+FREE, CDR P2 = P1.                      ATTACH ELEMENT TO FREE LIST.
       .P2 = P1, STEP P1.
        TO +FREE IF P1 NE UPLIM.         ARE WE AT THE TOP, NO.
        CDR P2 = NIL.                    YES, TERMINATE THE FREE LIST.
        LPRIME = P1, CDR P1 = NIL.       START THE RETURN STACK.
+ENDIN,EXIT INITAL.
        ENTRY SUBJMP.
        CAR LPRIME = P1.                 SAVE THE RETURN ADDRESS.
        PUSH DOWN LPRIME.
        EXIT SUBJMP.
        ENTRY RETURN.
        TO +NORET IF CDR LPRIME = NIL.   CHECK FOR AN EMPTY RETURN STACK.
        POP UP LPRIME.                   POP THE RETURN STACK.
        P1 = CAR LPRIME.                 RECALL THE RETURN ADDRESS.
        EXIT RETURN.
+NORET,MESSAGE 4.                        NO RETURNS LEFT.
        ENTRY RESTOR.
```

```
        CDR P1 = FREE, FREE = P1.       ATTACH ELEMENT TO FREE LIST.
        EXIT RESTOR.
        ENTRY GETNEW.
        TO +UNFRM IF FREE = NIL.        CHECK FOR UNFORMED FREE LIST.
        TO +ELOK IF CDR FREE NE NIL.    CHECK FOR EMPTY FREE LIST.
        TO +LISTR IF CDR LPRIME = NIL.  CHECK FOR EMPTY RETURN STACK.
        P1 = LPRIME.
+STKTR,P1 = CDR P1, P1 = ATOM.          MARK AN ELEMENT OF LPRIME.
        TO +STKTR IF CDR P1 NE NIL.     ARE WE DONE, NO.
+LISTR,P1 = NIL.
+SEQLS,FLAG P1, P1 = CAR P1.           SEQUENCE THROUGH LISTS.
        TO +CROOK IF P1 IS FLAGGED.     CHECK FOR CROOK BASE REG CHAIN.
        USE TRACER.                     TRACE AND MARK ONE LIST.
        TO +SEQLS IF CAR P1 NE NIL.     ARE WE DONE, NO.
        P1 = LOLIM, P2 = NIL.           YES, FORM FREE LIST.
+TSTFL,TO +CLRFL IF P1 = ATOM.          IS THIS FREE, NO.
        CDR P2 = P1, P2 = P1.           YES, STEP ALONG.
        TO +ADVPT.
+CLRFL,DEATOM P1.                       CLEAR THE ATOM FLAG.
+ADVPT,STEP P1.
        TO +TSTFL IF P1 NE UPLIM.       ARE WE DONE, NO.
        TO +GRBFL IF CDR FREE = NIL.    DID WE RECLAIM ANY SPACE, NO.
        P1 = NIL.                       RESTORE ALL ATOM FLAGS.
+FIXAT,P1 = ATOM, UNFLAG P1, P1 = CAR P1.
        TO +FIXAT IF CAR P1 NE NIL.
+ELOK, P1 = FREE, FREE = CDR FREE.      SATISFY THE CLAIM REQUEST.
        CDR P1 = NIL.
        EXIT GETNEW.
+UNFRM,MESSAGE 1.                       FREE LIST NEVER FORMED.
+GRBFL,MESSAGE 2.                       FREE LIST EXHAUSTED.
+CROOK,MESSAGE 3.                       BASE REGISTER CHAIN DESTROYED.
        ENTRY TRACER.
```

```
        TO +ENDTR IF CDR P1 = ATOM.     CHECK FOR EMPTY LIST.
        P2 = P1.
+FORWD,P3 = CDR P1, CDR P1 = P2.        STEP FORWARD.
+MARK, P3 = ATOM, P2 = P1, P1 = P3.     MARK AND MOVE.
        TO +FORWD IF CDR P1 NE ATOM.    ARE WE DONE, NO.
+REV,   TO +BRNCH IF CAR P1 NE ATOM.    CHECK FOR BRANCH.
+CHKBR,TO +ENDBR IF P2 IS FLAGGED.      ARE WE ON A BRANCH, YES.
        P3 = CDR P2, CDR P2 = P1.       RESTORE POINTERS.
        P1 = P2, P2 = P3.               STEP BACK.
        TO +REV IF P1 NE P2.            ARE WE DONE, NO.
+ENDTR,EXIT TRACER.
+BRNCH,P3 = CAR P1, CAR P1 = P2.        SAVE BACK POINTER IN CAR.
        FLAG P1, TO +MARK.              SET THE BPF AND TRACE THE BRANCH.
+ENDBR,P3 = CAR P2, CAR P2 = P1.        RESET THE CAR POINTER.
        UNFLAG P2.                      RESET THE BPF.
        P1 = P2, P2 = P3, TO +CHKBR.    STEP BACK.
.       END OF DYNAMIC STORAGE ALLOCATOR.
```

Appendix VII

List Dump Routine

```
        ENTRY SNAPR2.
        FINISH LINE ON PRINT.
        PRINT '*, PRINT 'R, PRINT 'S.     RETURN STACK HEADING.
        TO +ENDRS IF CDR LPRIME = NIL.    CHECK FOR EMPTY RETURN STACK.
        P3 = NIL, FINISH LINE ON PRINT.   START A NEW DUMP LINE.
        P1 = LPRIME.                      SCAN THE RETURN STACK.
+STKTR,P1 = CDR P1, P2 = CAR P1.          SET POINTER FOR THE DUMP.
        USE DUMPER, USE TABOUT.           PUT OUT AN ELEMENT.
        TO +STKTR IF CDR P1 NE NIL.       ARE WE DONE, NO.
+ENDRS,P3 = NIL, FINISH LINE ON PRINT.    START A NEW DUMP LINE.
        EXIT SNAPR2.
        ENTRY SNAPL2.
        FINISH LINE ON PRINT.
        PRINT '*, PRINT 'B, PRINT 'R.     BASE REGISTER HEADING.
        P3 = NIL, FINISH LINE ON PRINT.   START A NEW DUMP LINE.
        P1 = NIL.                         START BY DUMPING BASE REGISTERS.
+NEXBR,P1 = CAR P1, P2 = P1.              SEQUENCE THROUGH BASE REGISTERS.
        USE DUMPER.                       PUT OUT THE CAR FIELD.
        TO +NILBR IF CDR P1 = NIL.        DOES THE BR POINT TO NIL, YES.
        P2 = CDR P1, TO +OUTBR.           NO, GRAB ITS VALUE.
+NILBR,P2 = NIL.
+OUTBR,USE DUMPER, USE TABOUT.            PUT OUT THE CDR FIELD.
        DEATOM P1.                        MAKE SURE IT ISNT PUT OUT AGAIN.
        TO +NEXBR IF CAR P1 = ATOM.       ARE WE DONE, NO.
        P3 = NIL, FINISH LINE ON PRINT.   START A NEW DUMP LINE.
```

```
        FINISH LINE ON PRINT.           INSERT A BLANK LINE.
        PRINT '*, PRINT 'E, PRINT 'L.   ELEMENT HEADING.
        FINISH LINE ON PRINT.
        P1 = LOLIM.
+TSTFL,TO +DUMPL IF P1 = ATOM.          IS THIS MARKED, YES.
+ADVPT,STEP P1.                        NO, ADVANCE TO THE NEXT
        TO +TSTFL IF P1 NE UPLIM.       ARE WE DONE, NO.
+FIXFL,P3 = NIL, FINISH LINE ON PRINT. START A NEW DUMP LINE.
        P1 = NIL.
+REATM,P1 = CAR P1, P1 = ATOM.          RESET THE ATOM FLAGS ON BASES.
        TO +REATM IF CAR P1 NE ATOM.    ARE WE DONE, NO.
        EXIT SNAPL2.
+DUMPL,DEATOM P1, P2 = CAR P1.          REMOVE THE MARK FROM THE ELEMENT.
        USE DUMPER.                     PRINT THE CAR FIELD.
        TO +NILCD IF CDR P1 = NIL.      IS THE CDR NIL, YES.
        P2 = CDR P1, TO +OUTEL.
+NILCD,P2 = NIL.
+OUTEL,USE DUMPER, USE TABOUT.         PUT OUT THE CDR FIELD.
        STEP P1.                        ADVANCE TO THE NEXT.
        TO +FIXFL IF P1 = UPLIM.        ARE WE DONE, YES.
        TO +DUMPL IF P1 = ATOM.         IS THE NEXT MARKED, YES.
        P3 = NIL, FINISH LINE ON PRINT. START A NEW DUMP LINE.
        TO +ADVPT.
.
```

SYMBOLIC EDITOR

General Description

The symbolic editor was prepared to aid the SDS 920 programmer in the preparation of symbolic source tapes. The Editor may be used to prepare symbolic source tapes for any language whose source input is of a line structure.

The basic unit of data for the Symbolic Editor is a line--a string of characters (less than a fixed length--usually 80 characters) terminated by the character 'carriage return'. Individual lines or groups of lines may be input (or output) via the Teletype or punched paper tape.

Instructions are typed in to the Editor and immediately executed. Instructions may specify input or output of data, or perform certain housekeeping functions (such as PUNCH LEADER, PUNCH END-OF-TAPE CHARACTER, or CHANGE LENGTH).

Use

The Symbolic Editor is loaded into the SDS 920 by standard fill procedure. A standard restart at any time will return control to the Editor for instruction type-in.

INSTRUCTION DEFINITION

Instructions to the Editor may be defined in Backus form as:

<INSTRUCTION> ::= <MNEMONIC><ARGUMENT SET>

<ARGUMENT SET> ::= <ARGUMENT> CR|<ARGUMENT>,<ARGUMENT>CR|

<ARGUMENT> ::= <SYMBOL><INTEGER>

<INTEGER> ::= <UNSIGNED INTEGER> | +<UNSIGNED INTEGER>| - <UNSIGNED INTEGER>|

<UNSIGNED INTEGER> ::= <DIGIT>|<UNSIGNED INTEGER><DIGIT>

<DIGIT> ::= 0|1|2|3|4|5|6|7|8|9

<MNEMONIC> ::= A|C|D|F|I|K|L|N|P|Q|R|W|+|-

<SYMBOL> ::= T|B|.|

Note that the argument set, either argument, or either symbol may be null.

The mnemonic may never be null.

All spaces, tabs, and blanks within an instruction are ignored.

The text is maintained in memory in a linear list structure. A moving pointer is maintained which points to an element of the list. A fixed pointer points to the top of the list, another to the bottom.

The symbol 'T' refers to the top of the list. The symbol 'B' refers to the bottom of the list. The symbol '.' refers to the location in the list presently pointed to by the moving pointer.

STANDARD TWO-ARGUMENT FUNCTIONS

The arguments of a standard two argument function (see VALID INSTRUCTIONS AND THEIR INTERPRETATION) are interpreted as follows:

1) If the first argument is a pointer (if the symbol of the first argument is non-null) then the first argument is interpreted to indicate a position in the list to which the moving pointer is to be moved before execution of the function.

   EXAMPLE: T+3 refers to the third line of the text,

   B-4 refers to the fourth line up from the bottom of the text,

   .+6 refers to the line six lines below the present location of the pointer.

   A) If the symbol of the second argument is non-null, it is interpreted as a pointer to the line on which to stop execution of the function.

   B) If the symbol of the second argument is null, the function is executed N times (where N is the number of the second argument). If N is zero or null the function is executed once.

2) If the symbol of the first argument is null, then the number of the first argument is interpreted as the number of times the function is to be executed. If the number is zero or null, the function is executed once. The starting position is given by the present location of the moving pointer. In this case the second argument is ignored.

VALID INSTRUCTIONS AND THEIR INTERPRETATION

A - ATTACH TO FRONT OF LIST

This instruction accepts lines from the Teletype keyboard and attaches them to the beginning of the list.

The symbol and sign of the first argument are ignored. The entire second argument is ignored. The number of the first argument, N, is taken to be the number of lines to be accepted from the Teletype keyboard. If the number N is null or zero, an indefinite number of lines will be accepted from the keyboard.

A line with a 'record mark' in column 1 will terminate input and return control to the Editor for another instruction at any time. The line with the record mark in column 1 will be deleted from the text. The moving pointer is left pointing to the last line typed in.

C - CHANGE LINES

Lines are deleted from the text and replaced by lines accepted from the Teletype keyboard. This is a standard two-argument function. The pointer is left pointing to the last line which was changed.

D - DELETE LINES

Lines are deleted from the text. This is a standard two-argument function. The moving pointer is left pointing to the line following the deleted line(s) (unless the deleted line was the last one of the text, in which case the pointer points to the last line of the text).

I - INSERT LINES

Lines are accepted from the Teletype and entered into the text. If the first argument is a pointer (if the symbol of the first argument is non-null) the moving pointer is moved to that position in the list indicated by the first argument and lines typed in are inserted following that line. The number of the second argument is interpreted as the number of lines to be inserted.

If this number is null or zero, an indefinite number of lines are accepted.

If the first argument is not a pointer, then it is interpreted as the number of lines to be inserted and the second argument is ignored. The lines typed in are inserted following the line presently pointed to by the moving pointer. A line with a record mark in column 1 will terminate input and return control to the Editor for instruction type-in at any time. The moving pointer is left pointing to the Last Line typed in.

K - KILL LIST

The storage area is cleared and memory is reinitalized.

L - LOOK AT

Lines are typed out on the console Teletype. This is a standard two-argument function.

N - CHANGE LENGTH

This function changes the length of the maximum allowable data input line. An attempt to execute this function

will result in self-explanatory instructions being typed out. Normally the length of the the maximum allowable data input line is 79 characters (not counting the 'carriage return'). CAUTION execution of this function reinitalizes memory and destroys any previous text which was stored in memory.

P - PUNCH LINES

lines of text are punched out on paper tape. This is a standard two-argument function. The pointer is left pointing to the last line punched out.

Q - PUNCH LEADER

If the number of the first argument is null or zero, a standard length (about six inches) of leader will be punched out. If the number of the first argument is non-null and non-zero, approximately one and one half times that number of inches of leader will be punched. Symbol and sign of the first argument are ignored. The second argument is ignored.

R - READ LINES FROM PAPER TAPE

Lines are read from the tape and inserted following the line pointed to by the pointer. The pointer is left pointing to the last line read from the tape. If the number of the first argument is non-null and non-zero, then that number (sign ignored) of lines are read from the tape. Symbol, sign, and the entire second argument are ignored.

W - WRITE THE ENTIRE LIST

The entire text is dumped out on the console Teletype.

+ - TYPE OUT NEXT LINE

The pointer is moved down the list one line and that is typed out on the console Teletype.

- - TYPE OUT PRECEDING LINE

The pointer is moved up the list one line and that line is typed out on the console teletype.

ERROR INDICATIONS

BEGINNING-OF-FILE

An attempt was made to move the pointer beyond the beginning of the list. This is a non-fatal error. The pointer will be left pointing to the first line of the text and the function will execute as normal.

END-OF-FILE

An attempt was made to move the pointer beyond the end of the list. This is a non-fatal error and the pointer will be left pointing to the last element of the list. Execution of the function will stop at the bottom of the list.

MAXIMUM LENGTH OF INPUT LINE EXCEEDED

This indication may occur on instruction of data input. The type-out which follows this will be self-explanatory. They will be one of the following:

RETYPE LAST INSTRUCTION or

RETYPE LAST LINE or

LINE IN ERROR DIVIDED INTO TWO PARTS AND REMAINS IN TEXT.

BUFFER ERROR ON TAPE INPUT or
BUFFER ERROR ON KEYBOARD INPUT

A parity error has occurred on input. This indication will be accompanied by another line which will indicate the action which will be taken. These lines will be one of the following:

LINE IN ERROR REMAINS IN TEXT or

RETYPE LAST LINE or

RETYPE INSTRUCTION

ILLEGAL ARGUMENT

The argument field of an instruction to the Editor was either of an incorrect format, contained illegal characters, or the combination of arguments was such that the user attempted to execute a function specifying a finish point which was closer to the beginning of the list than his starting point. Control is returned to the Editor for another instruction.

ILLEGAL MNEMONIC

An undefined mnemonic was specified. Control is returned to the Editor for another instruction type line.

MEMORY FULL

The storage space has overflowed. Before any more lines may be entered into the text, some will have to be destroyed. One may punch a number of them on paper tape, and then delete them. Control is returned to the Editor for instruction type-in following this type-out.