Solving Common Programming Problems With an Applicative Programming Language

by

Thomas M. Grismer
Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT No. 109
SOLVING COMMON PROGRAMMING PROBLEMS WITH
AN APPLICATIVE PROGRAMMING LANGUAGE

THOMAS M. GRISMER
JUNE, 1981

This material is based upon work supported by the National Science Foundation under Grants MCS77-22325 and MCS 79-04183.

SOLVING COMMON PROGRAMMING PROBLEMS WITH A PURELY APPLICATIVE PROGRAMMING LANGUAGE

Thomas Markle Grismer

Abstract:

An applicative programming language is used to solve some common programming problems. The language used is one developed by Daniel P. Friedman and David S. Wise of the Department of Computer Science at Indiana University. With respect to the use of the language in this paper, its significant aspects include purely applicative control structures, dynamic allocation of storage, functions which return multiple values, and capabilities for applying functions in parallel; a "suspended evaluation" [FW1] scheme allows a programmer to manipulate and create infinite list structures.

Programs written are based upon structured FORTRAN programs presented in the book, Software Tools, by Kernighan and Plauger [KP]. Among the programs presented in the paper are a text editor, a text formatter, a macro expander, and a structured FORTRAN to FORTRAN translator.

Introduction.

This paper has been written to demonstrate the use of an applicative programming language for solving "real world" programming problems. In order to achieve this goal, several programs are presented and discussed. The language used to write these programs is a programming language developed by Daniel P. Friedman and David S. Wise of Indiana University, Bloomington, Indiana. In this thesis, the implementation of that language is called INTERPRETER [SJ].

For years the business computing community has placed primary emphasis upon the use of programming languages which run efficiently on the existing machine architecture. Recently, however, the rapidly decreasing cost of computer hardware has resulted in an increasing amount of emphasis being placed upon the human costs of developing and maintaining computer software.

Current trends in computer hardware architecture, moreover, allow the implementation and maintenance of applicative programming languages to be less expensive. These trends make it appropriate to discuss the extension of the use of applicative programming languages beyond the enthusiastic following of academic computer scientists which they now enjoy [FW, SJ].

The programs written in this paper address problems presented in the book, Software Tools, by Kernighan and Plauger [KP]*. In fact, the problems addressed in this paper correspond

* References to Software Tools are indicated by page number enclosed in square brackets.

chapter-for-chapter with those presented there. The reader is encouraged to obtain a copy of Software Tools to assist in understanding the subjects discussed herein.

Experience with the programming language, LISP, has convinced me that an applicative programming language can be an effective tool for expressing algorithms as computer programs [MC]. A trait shared by INTERPRETER and pure-LISP is that the semantics nearly requires the programmer to write structured, modular code. The only control structure is the application of a function to a list of arguments.

INTERPRETER, as LISP, provides for the dynamic allocation of storage, freeing the programmer of the responsibility of reserving and managing the data space. INTERPRETER, however, has features which give it an expressive power beyond that of pure-LISP. Programs written in this paper make extensive use of functions which return multiple values. They use functional combination [FW2] to apply functions in parallel to lists of arguments, allowing the programmer to simultaneously manipulate a number of values without having to resort to the use of free variables or iterative control structures. Another aspect of INTERPRETER, which is utilized in the programs presented in this paper, is a suspended evaluation scheme which allows the programmer to manipulate infinite list structures and list structures containing divergent elements.

An aspect of INTERPRETER which has not been utilized in the programs presented in this paper is the "multiset", an unordered, nondeterministic data structure. This feature was

in the development stages when programs presented here were being written.

For an introduction to INTERPRETER, it is essential that the reader obtain a copy of the master's thesis of Steven D. Johnson, An Interpretive Model for a Language based on Suspended Construction [SJ]. This paper contains a user's manual for the language, and a discussion of its implementation.

The programs and examples presented in Chapters 1 through 6 of this paper were reproduced from programs and examples which were machine compiled and tested on a version of INTERPRETER which has been implemented on a CDC6600 computer at Indiana University. Due to physical limitations of the current implementation of the language on the CDC6600, it was not possible to extensively test the higher level functions presented in Chapter 6. The programs presented in Chapters 7 through 9 have not been implemented on a machine.

My thanks go to my mother, Mrs. Arabelle Hoyt, for her help in preparing this manuscript. I give thanks also to Steve Johnson for his work in implementing and maintaining INTERPRETER on the CDC6600, and for helping me to use and understand it. The suggestions and assistance of Anne Kohlstaedt have been invaluable. I am very grateful to Daniel Friedman and David Wise for their support and guidance of my work. Finally, I thank the National Science Foundation for funding this research.

Table of Contents.

	page number
Introduction	i-iii
Chapter 1--Simple functions	1
Chapter 2--Filters	13
Chapter 3--Files	35
Chapter 4--Sorting	72
Chapter 5--Pattern Matching	93
Chapter 6--Text Editor	104
Chapter 7--Text Formatter	170
Chapter 8--Macro-Processor	220
Chapter 9--RATFOR-FORTRAN Translator	242
Function and Constant Index	274
References	309

CHAPTER 1.

Introduction:

Five simple functions are presented in this chapter. Kernighan and Plauger use these first programs to familiarize the reader with the basics of the programming language they are using; similarly I use this chapter as a introduction to some of the programming ideas and structures which occur in the chapters to follow.

First, let's look at four functions, which show much similarity in form.

COPY simply creates an output file which is identical to its input file, TEXT. (A file is a list of zero or more characters).

```
DEFINE COPY TEXT
    <1*>:<TEXT>
.
 -=>COPY

COPY:<>.
 -=>()

COPY:"(0).
 -=> (0)

COPY:"(0 N C E).
 -=> (0 N C E)
```


Next, we have three counting functions, CHARCT, LINECT and WORDCT. CHARCT returns the total number of characters in its input file, TEXT.

```
DEFINE CHARCT TEXT
    IF NULL:TEXT THEN 0
    ELSE ADD1:CHARCT:REST:TEXT
.
 -=>CHARCT

CHARCT:<>.
 -=>0

CHARCT:"(F).
 -=>1

CHARCT:"(F 0).
 -=>2

CHARCT:"(F 0 U).
 -=>3

CHARCT:"(F 0 U R).
 -=>4
```

The function LINECT, returns the number of lines in its input file, TEXT. A line is a sequence of zero or more characters followed by a special newline character (declared here as the constant, NL), or by the end of the file. Actually, then, what LINECT does is to count the number of newline characters in TEXT and add one to the result.

```
DEFINE LINECT TEXT
    IF NULL:TEXT  THEN  1
    ELSEIF NLQ:FIRST:TEXT  THEN  ADD1:LINECT:REST:TEXT
    ELSE LINECT:REST:TEXT
.
 -=>LINECT
```

A file has at least one line. Zero characters followed by the end of the file comprise an empty line.

```
DEFINE NLQ CHAR
    SAME:<CHAR NL>.
.
 -=>NLQ
```

```
LINECT:<>.
 -=>1
LINECT:"(L I N E).
 -=>1

LINECT:"(L O T S NL O F NL L I N E S).
 -=>3
```


WORDCT counts the number of words in its input file, TEXT. A word is a sequence of non-"whitespace" characters which begins either at the beginning of a file or immediately following a "whitespace" character, and which ends either at the end of a file or immediately preceding a "whitespace" character. A "whitespace" character is a tab character (declared as the constant, TAB) or a blank character (BLANK) or a newline character.

```
DEFINE WORDCT TEXT
    IF NULL:TEXT THEN 0
    ELSEIF OUTWORD:FIRST:TEXT THEN WORDCT:REST:TEXT
    ELSE ADD1:WORDCT:EATWORD:REST:TEXT
.
 -=>WORDCT
```

```
DEFINE OUTWORD CHAR
    OR:<NLQ:CHAR BLANKQ:CHAR TABQ:CHAR>
.
 -=>OUTWORD
```

```
DEFINE BLANKQ CHAR
    SAME:<CHAR BLANK>
.
 -=>BLANKQ
```

```
DEFINE TABQ CHAR
    SAME:<CHAR TAB>
.
 -=>TABQ
```

```
DEFINE OR L
    IF NULL:L THEN FALSE
    ELSEIF FIRST:L THEN TRUE
    ELSE OR:REST:L
.
 -=>OR
```

After WORDCT has encountered a word in TEXT, EATWORD is called to trim the leading non-whitespace characters from TEXT so that WORDCT may look for a new word.

```
DEFINE EATWORD TEXT
    IF NULL:TEXT THEN TEXT
    ELSEIF OUTWORD:FIRST:TEXT THEN REST:TEXT
    ELSE EATWORD:REST:TEXT

-=>EATWORD
```

```
WORDCT:"(T H R E E BLANK L I T T L E TAB W O R D S).
 -=>3

WORDCT:"(BLANK BLANK BLANK).
 -=>0
```


The definitions of the four functions, COPY, CHARCT, LINECT and WORDCT have similar form. In the case of the three counting functions discussed, an integer result is obtained by applying the function, ADD1, to the recursive invocation of the defining function. Each function terminates with the expression 0 or 1, when the input file TEXT is found to be empty -- when the predicate IF NULL:TEXT evaluates to TRUE.

In general, we will see that a function terminates when one of its predicates, for which the predicate's corresponding expression does not involve a recursive call of the defining function, evaluates to TRUE. An expression "involves" a recursive call of the defining function if either the expression is a recursive call, or the expression is a function whose definition contains a recursive call of the defining function. We say that function A "calls" function B if either A is an explicit invocation of the function B or A invokes a function which calls B.

An expression which results in termination of a defining function shall be referred to as a "ground expression", and its corresponding predicate shall be referred to as a "ground condition".

Another thing to note about the three counting functions is that within each recursive invocation of a defining function is a call to the function REST. The invocation of REST:TEXT insures that progress is made toward satisfaction of the

ground condition IF NULL:TEXT. I shall refer to functions such as this as "progress functions".

Now, to detect the similarity in form of COPY to the three counting functions, note that the definition of COPY is equivalent to the following.

```
DEFINE COPY TEXT
    IF NULL:TEXT THEN <>
    ELSE CONS:<FIRST:TEXT COPY:REST:TEXT>
.
```

This definition of COPY differs from the definitions of the counting functions in two significant ways. First, the function CONS is used where in the other functions, ADD1 was used. This difference is due to the fact that a list is being built in this case rather than an integer. Functions like ADD1 and CONS, which are used to build results will be referred to as "constructors". The ground expression of COPY, '<>', also differs from that of the counting functions, again due to the nature of the result being built.

Although these differences exist, when one recognizes that ADD1 and CONS are functions of the same class, constructors, and that '0' and '<>' are expressions of the same class, ground expressions, one can see that the definition of COPY is similar in form to the three counting functions. In fact, the definition of COPY is conceptually and structurally identical to that of CHARCT.

Now let's look at DETAB, whose definition is a slight variation on the form we have seen so far.

DETAB takes as input a text file, TEXT, and two lists of tab stops, TABS and XTABS. The purpose of the function is to create an output file identical to TEXT except that one or more BLANKs are inserted into the file wherever a TAB character occurs in TEXT.

Initially, the lists, TABS and XTABS, are identical with each containing tab stops for one line of TEXT. The value of XTABS is unaltered throughout the program, and is bound to TABS whenever a new line is encountered in TEXT.

```
DEFINE DETAB (TEXT TABS XTABS)
    IF NULL:TEXT THEN TEXT
    ELSEIF NL?:FIRST:TEXT
        THEN CONS:<NL DETAB:<REST:TEXT XTABS XTABS>>
    ELSEIF TAB?:FIRST:TEXT
        THEN TABBLNKS:<REST:TEXT TABS XTABS>
    ELSE CONS:<BLANK TABBLNKS:<TEXT REST:TABS XTABS>>
.
-=>DETAB
```

When a tab character is encountered in the input file, DETAB calls TABBLNKS to add blanks to the output file. The number of blanks to be added is determined by the contents of TABS. TABS and XTABS each consist of a list of the values TRUE and FALSE. A TRUE is considered to be a tab stop. A blank is added to the output file for each leading FALSE in TABS. Then one more blank is added when a tab stop is

encountered.

Example 1: Each tab character adds at least one blank to the output.

```
DETAB:<<TAB> <TRUE FALSE FALSE>
             <TRUE FALSE FALSE>>.
 -=> (BLANK)
```

Example 2: When a tab character is encountered in the TEXT file, a blank is added for each leading FALSE in TABS and one more is added for the tab stop.

```
DETAB:<<TAB> <FALSE FALSE FALSE TRUE FALSE>
             <FALSE FALSE FALSE TRUE FALSE>>.
 -=> (BLANK BLANK BLANK BLANK)
```

Example 3: When a tab is encountered in TEXT in the example below, FIRST:TABS is TRUE; hence one blank is added after the B in the output.

```
DETAB:<"(A B TAB NL) <FALSE FALSE TRUE TRUE>
                     <FALSE FALSE TRUE TRUE>>.
 -=> (A B BLANK NL)
```


Example 4: In the example below, when the second tab is encountered in TEXT, a fresh copy of TABS, identical to XTABS exists, so three blanks are added, two for the leading FALSEs, and one for the tab stop (TRUE).

```
DETAB:<"(A B TAB NL TAB C NL)
       <FALSE FALSE TRUE TRUE>
       <FALSE FALSE TRUE TRUE>>.
 -=> (A B BLANK NL BLANK BLANK BLANK C NL)
```

Like the first four functions described in this chapter, DETAB goes through a list character by character, building its result through the application of a constructor to a recursive invocation of the defining function. Again, the ground condition is IF NULL:TEXT.

The call to TABBLNKS in the definition of DETAB is our first example of a recursive function call which is not an explicit invocation of the defining function. However, as has been the case with the previous examples, the call involves an invocation of the progress function, REST:TEXT.

The expression in the first line of TABBLNKS looks something like the ground expressions of the previous examples in that it contains no invocation of the defining function. However, depending upon the contents of TEXT, this expression may either be a ground expression or a recursive function call.

```
DEFINE TABBLNKS (TEXT TABS XTABS)
    IF FIRST:TABS
         THEN CONS:<BLANK DETAB:<TEXT REST:TABS XTABS>>
    ELSE CONS:<BLANK TABBLNKS:<TEXT REST:TABS XTABS>>

'
-=>TABBLNKS
```

TABBLNKS' argument, TABS is assumed to contain a tab stop, and so, progress is made toward satisfaction of the condition IF FIRST:TABS through the invocation of REST:TABS. A call to REST:TABS is the progress function for TABBLNKS since the ground condition will be encountered in a call to DETAB, which is called when FIRST:TABS is TRUE.

TABBLNKS could have been written so that it just created a list of BLANKs which would then be appended to the output file being built by DETAB. Then the definition of TABBLNKS would be independent of the definition of DETAB.

However, the function performed by TABBLNKS is so specialized, that it is unlikely that it would be needed other than as a help function to DETAB. In this case, it is worthwhile to sacrifice modularity in order to save the expense of first building a list of blanks and then appending it to the output.

Constants Introduced in Chapter 1.

```
DECLARE NL "NL.
 -=>NL

DECLARE BLANK "BLANK.
 -=>BLANK

DECLARE TAB "TAB.
 -=>TAB
```

CHAPTER 2

Kernighan and Plauger refer to the programs in Chapter 2 as "filters". Each program takes a text file as input, and copies it to the output, making changes in the input stream as it passes through.

With regard to the programming constructs discussed in Chapter 1, this implies that each of the functions has a ground condition "IF NULL: . . .". Each uses the progress function, REST, and each uses CONS as a constructor of its output file.

ENTAB is the function which complements DETAB. It "replaces strings of blanks by equivalent tabs and blanks." [35] The lists of tab stops are just like those in DETAB. Also, like in DETAB, the value of XTABS is bound to TABS when a new line is encountered.

```
DEFINE ENTAB (TEXT TABS XTABS)
    IF NULL:TEXT THEN TEXT
    ELSEIF NLQ:FIRST:TEXT
         THEN CONS:<FIRST:TEXT  ENTAB:<REST:TEXT REST:TABS XTABS>>
    ELSEIF BLANKQ:FIRST:TEXT
         THEN CKBLANKS:<ENTAB:<REST:TEXT REST:TABS XTABS> TABS>
    ELSE CONS:<FIRST:TEXT ENTAB:<REST:TEXT REST:TABS XTABS>>

-=>ENTAB
```

CKBLANKS is our first introduction to a very useful and often used function type, the "lookahead" function.

A lookahead function is a function which is called with a recursive invocation of the calling function as one of its arguments. The value of this argument at least partly determines

the action to be taken by the lookahead function.

CKBLANKS is invoked by ENTAB when a BLANK is encountered in TEXT. CKBLANKS has two arguments; 1) the result of a recursive invocation of ENTAB, OUTFILES, and 2) TABS. A tab is to be substituted for any string of blanks for which C1) the last blank occurs in the same position in a line as a tab stop, or C2) the last blank occurs at the end of a line .

Example 1: C1 is true. . .

```
ENTAB:<"(BLANK BLANK I N D) <FALSE TRUE TRUE TRUE TRUE>
       <FALSE TRUE TRUE TRUE TRUE>>.
 -=> (TAB I N D)
```

Example 2: C2 is true . . .

```
ENTAB:<<BLANK BLANK BLANK> <FALSE FALSE FALSE>
       <FALSE FALSE FALSE>>.
 -=> (TAB)
```

Example 3: Neither C1 nor C2 is true . . .

```
ENTAB:<"(A B C D E) <TRUE TRUE TRUE TRUE TRUE>
       <TRUE TRUE TRUE TRUE TRUE>>.
 -=> (A B C D E)
```

If the first element in TABS is a tab stop, then CKBLANKS can add a blank to the output file with no other problems. Otherwise, CKBLANKS must inspect the list which is the result of the recursive invocation of ENTAB to know what to do. If this list is empty, condition C2 has been met and a tab character is added to the output file. If the first element of the list is a tab character, condition C1 has been met, and CKBLANKS

adds nothing to the output file. Otherwise neither condition has been met, and a blank must be added to correspond to the blank in the input file.

```
DEFINE CKBLANKS (OUTFILE TABS)
    IF NULL:OUTFILE THEN CONS:<TAB OUTFILE>
    ELSEIF FIRST:TABS THEN CONS:<TAB OUTFILE>
    ELSEIF TABQ:FIRST:OUTFILE THEN OUTFILE
    ELSE CONS:<BLANK OUTFILE>

-=>CKBLANKS
```

OSTRIKE is used to simulate backspaces on a printer which has no mechanism for backspacing, but which can do carriage returns without line feeds. When a string of one or more backspace characters is encountered in a text file, a new line is written on top of the current line with an appropriate number of leading blanks inserted, so that the character following the backspace character(s) will be positioned as though backspacing had occurred.

The algorithm used here for OSTRIKE is very similar to that used in Kernighan and Plauger's book.[40] First, a help function OSTRIKE1 is called. Its arguments are an integer, N, which indicates the number of characters in the current line which have been printed, and an input file, TEXT, with leading backspace characters (BS) removed. A backspace past the beginning of a line is ignored. The count is reset to zero whenever a new line is encountered in the input file.


```
DEFINE OSTRIKE TEXT
    OSTRIKE1:<MINUS:1 TRIMBS:TEXT>

-=>OSTRIKE

DEFINE OSTRIKE1 (N TEXT)
    IF NULL:TEXT THEN <>
    ELSEIF NLQ:FIRST:TEXT THEN NEWSKIP:REST:TEXT
    ELSEIF BSQ:FIRST:TEXT THEN BLANKOVR:<N REST:TEXT>
    ELSE CONS:<FIRST:TEXT OSTRIKE1:<ADD1:N REST:TEXT>>

-=>OSTRIKE1


DEFINE BSQ CHAR
    SAME:<CHAR BS>

-=>BSQ
```

Here are some examples of OSTRIKE in action. The first shows the result of backspacing at the beginning of a line (the backspaces have no effect.). The second shows the result of backspacing at the end of a line (again no effect). The third shows the effects of backspacing in mid-line.

```
OSTRIKE:"(BS BS A B C).
 -=> (A B C)

OSTRIKE:"(A B D E E BS BS BS).
 -=> (A B D E E)

OSTRIKE:"(A B C BS D NL E F G BS BS H NL).
 -=> (A B C NL NS BLANK BLANK D NL SKIP E F G NL NS BLANK H NL SKIP)
```

When OSTRIKE1 encounters a backspace character, the function BLANKOVR is called. BLANKOVR checks to see if the string of one or more BSs comes at the end of a line or at the end of the input file. In that case, the backspace characters have no effect. Otherwise, BLANKOVR calls BLANKOUT to

begin a new line of the output file. A special NOSKIP (NS) character follows the NL character at the beginning of the new line to indicate to the printer that there should be a carriage return with no line feed. BLANKOUT adds an appropriate number of leading blanks to the new line in the output file.

```
DEFINE BLANKOVR (N TEXT)
    IF NULL:TEXT THEN <>
    ELSEIF NLQ:FIRST:TEXT THEN NEWSKIP:REST:TEXT
    ELSEIF BSQ:FIRST:TEXT
         THEN BLANKOVR:<SUB1:N REST:TEXT>
    ELSE CONCAT:<<NL NS> BLANKOUT:MAX:<N 0>
                 OSTRIKE1:<MAX:<N 0> TEXT>>
.
-=>BLANKOVR

DEFINE BLANKOUT N
    IF ZEROP:N THEN <>
    ELSE CONS:<BLANK BLANKOUT:SUB1:N>
.
-=>BLANKOUT

DEFINE NEWSKIP TEXT
    CONS:<NL CONS:<SKIP OSTRIKE1:<MINUS:1 TRIMBS:TEXT>>>
.
-=>NEWSKIP
DEFINE MAX (N1 N2)
    IF GREAT:<N1 N2> THEN N1
    ELSE N2
.
-=>MAX
```


The function CONCAT, called by BLANKOVR, merges a list of lists into one list. The function APPEND, a helper to CONCAT, merges two lists into one.

```
DEFINE CONCAT L
    IF NULL:L THEN <>
    ELSE APPEND:<FIRST:L CONCAT:REST:L>
.
-=>CONCAT

DEFINE APPEND (L1 L2)
    IF NULL:L1 THEN L2
    ELSE CONS:<FIRST:L1 APPEND:<REST:L1 L2>>
.
-=>APPEND
```

TRIMBS, a help function to OSTRIKE and to OSTRIKE1, removes leading backspace characters from its input file, TEXT.

```
DEFINE TRIMBS TEXT
    IF NULL:TEXT THEN <>
    ELSEIF BSQ:FIRST:TEXT THEN TRIMBS:REST:TEXT
    ELSE TEXT
.
-=>TRIMBS
```

Unlike the pattern set in the previous examples, BLANKOVR's call to OSTRIKE1 contains no explicit invocation of a progress function. Everything works out though, because OSTRIKE1's call to BLANKOVR contains as its argument, "REST:TEXT". So, we can still say that BLANKOVR holds true to the general rule that the recursive call to a function results in an invocation of a progress function for that function.

The result of BLANKOUT is appended (via a call to CONCAT) to the result of the invocation of OSTRIKE1. This is in opposition to the earlier decision, made in DETAB's call to TABBLNKS, to sink the recursive call down another level to save having to append a list of characters to the result of the recursive invocation of DETAB. We decide differently this time because BLANKOUT is a useful funntion in its own right, and because the number of blanks to append is likely to be fairly small (always less than the length of a line of text.).

We are going to see this decision come up again and again in functions which build lists. When we are just interested in adding a character at a time to a result, then we may just apply the constructor, CONS, to a recursive call of the function building the list. We could use APPEND as a list constructor when we wish to add more than one character to a list at a time, in the same manner in which we have used CONS as an atomic constructor to add one character to a list at a time, and we often will;

but the expense of using the APPEND - - an extra pass through the list to be appended to the building result - - must be considered.

So, before deciding whether or not to use an APPEND (when it may be avoided by sinking a recursive function call to a lower level), we must consider the following (in no particular order):

1) Is the list short? - - If so, we don't lose much by appending.
2) Can the function used to build the list serve as a useful tool in other circumstances besides this particular one? If so, we may not wish to decrease its use as a general tool by inserting within it a call to a special function.
3) How is the readability of the code effected? In my opinion, sinking a recursive function call to a lower level function detracts from the apparent logical flow of the code.

- - - - - -

The function COMPRESS compresses a file by replacing all runs of N or more consecutive characters by a compression code which indicates the length of the run and the character involved. All characters not involved in such runs are left unaltered.

The arguments to COMPRESS are a file, TEXT, and THRESH, which is the maximum number of times a character may occur consecutively before the run of that character is replaced by a compression code.

For example, if THRESH equals three then a run of four

consecutive A's would result in the following code:

NCHARS A 4

NCHARS is a special character which signals that what follows is a compression code.

```
DEFINE COMPRESS (THRESH TEXT)
    COMPRES1:<THRESH 0 TEXT>

'-=>COMPRESS

DEFINE COMPRES1 (THRESH COUNT TEXT)
    IF NULL:TEXT THEN <>
    ELSEIF OR:<NULL:REST:TEXT DIFFER:TEXT>
         THEN ADDCHARS:<THRESH ADD1:COUNT TEXT>
    ELSE CKREST:<FIRST:TEXT
                 COMPRES1:<THRESH ADD1:COUNT REST:TEXT>>

'-=>COMPRES1
DEFINE DIFFER (A B)
    NOT:SAME:<A B>

'-=>DIFFER
```

In the first example of COMPRESS below, THRESH equals three and the longest string of consecutive characters is only two, so the input is unaltered.

In the second example, THRESH is again three, but there is a string of three identical characters ('B'), so a compression code is inserted.

```
COMPRESS:<3 "(A B B C D D E)>.
 -=> (A B B C D D E)

COMPRESS:<3 "(A B B B C C D E)>.
 -=> (A NCHARS B (3) C C D E)
```


As Kernighan and Plauger have suggested, it helps when writing this code, to think of TEXT as a file containing a series of runs of one or more identical characters. The argument, COUNT, in COMPRES1 is used to keep track of the length of the current run. When there is only one character left in text or the first and second characters differ, then ADDCHARS is called to see if the length of the current run is greater than THRESH; if this is so, a compression code is generated.

```
DEFINE ADDCHARS (THRESH COUNT TEXT)
    IF NOT:LESS:<COUNT THRESH>
         THEN LCONS:<NCHARS FIRST:TEXT ITOC:COUNT
                    COMPRES1:<THRESH 0 REST:TEXT>>
    ELSE CONS:<FIRST:TEXT COMPRES1:<THRESH 0 REST:TEXT>>

'-=>ADDCHARS

DEFINE LCONS L
    IF NULL:REST:L THEN FIRST:L
    ELSE CONS:<FIRST:L LCONS:REST:L>

'-=>LCONS
```

If the first and second characters in TEXT are identical, then CKREST is called. CKREST is another lookahead function. The arguments to CKREST are CHAR, the first character in TEXT, and TEXT, the result of a recursive invocation of COMPRES1. If the first character of TEXT is NCHARS, then CHAR was part of a run of more than THRESH identical characters and so, TEXT is left alone; otherwise, CHAR is added to TEXT.

```
DEFINE CKREST (CHAR TEXT)
    IF NCHARSQ:FIRST:TEXT THEN TEXT
    ELSE CONS:<CHAR TEXT>

*
-=>CKREST

DEFINE NCHARSQ CHAR
    SAME:<CHAR NCHARS>

*
-=>NCHARSQ
```

There is not much new to see in COMPRESS et. al. The invocation of ADDCHARS in COMPRES1 has no invocation of the progress function, 'REST:TEXT', but this is taken care of in the definition of ADDCHARS.

ITOC converts an integer to a string of one or more digits (plus MINUS sign if the integer is negative). PUTDEC outputs the string into a given field width. Leading blanks are inserted if the width of the string is less than the given field width. The string is not truncated if it is wider than the given field width, but is instead output without any leading blanks.

```
DEFINE ITOC N
    IF MINUSP:N THEN CONS:<MINUS ITOC1:MINUS:N>
    ELSE ITOC1:N

*
-=>ITOC

DEFINE ITOC1 N
    IF LESS:<N 10> THEN <DIGVAL:N>
    ELSE SNOC:<DIGVAL:MOD:<N 10> ITOC1:DIV:<N 10>>

*
-=>ITOC1

DEFINE PUTDEC (N W)
    APPEND:<PUTBLNKS:<CHARCT:ITOC:N W> ITOC:N>

*
-=>PUTDEC
```


Here are two examples of the execution of PUTDEC. In each example, PUTDEC converts an integer to a character string, and outputs the string within a five column field.

```
PUTDEC:<34 5>.
 -=>(BLANK BLANK BLANK 3 4)

PUTDEC:<-34 5>.
 -=>(BLANK BLANK MINUS 3 4)
```

ITOC1 calls the help function, SNOC, which works like CONS in reverse. It places an atom at the end of a list. ITOC1 also calls the help function MOD, which returns the remainder of the integer division of its argument, N1, by its other argument, N2.

```
DEFINE SNOC (A L)
    IF NULL:L THEN <A>
    ELSE CONS:<FIRST:L SNOC:<A REST:L>>

*
-=>SNOC

DEFINE MOD (N1 N2)
    DIFF:<N1 TIMES:<DIV:<N1 N2> N2>>

*
-=>MOD
```

```
DEFINE PUTBLNKS (WN N)
    IF LESSEQ:<N WN> THEN <>
    ELSE CONS:<BLANK PUTBLNKS:<ADD1:WN N>>

•
-=>PUTBLNKS

DEFINE MINUS N
    DIFF:<0 N>

•
-=>MINUS

DEFINE MINUSP N
    LESS:<N 0>

•
-=>MINUSP

DEFINE DIGVAL N
    PROJ:<ADD1:N DIGITS>

•
-=>DIGVAL
DEFINE PROJ (N LIST)
    IF NULL:LIST THEN <>
    ELSEIF ONEP:N THEN FIRST:LIST
    ELSE PROJ:<SUB1:N REST:LIST>

•
-=>PROJ
DEFINE ONEP N
    SAME:<N 1>

•
-=>ONEP

DEFINE LESSEQ (N1 N2)
    OR:<LESS:<N1 N2> SAME:<N1 N2>>

•
-=>LESSEQ
```

EXPAND is the complement function to COMPRESS. Whenever the key character, NCHARS, is encountered in a text file, PUTCHARS is called to substitute for the compression code, a series of characters as is indicated by that code.

```
DEFINE EXPAND TEXT
    IF OR:<NULL:TEXT NULL:REST:TEXT NULL:RREST:TEXT>
        THEN TEXT
    ELSEIF NCHARSQ:FIRST:TEXT THEN CK3:TEXT
    ELSE CONS:<FIRST:TEXT EXPAND:REST:TEXT>

•
-=>EXPAND
```



```
EXPAND:"(A NCHARS B (3) C C D E).
 -=> (A B B B C C D E)
```

CK3 checks to see if the third element of TEXT is a list. If it is, it is assumed to be a list of digits. This list is converted to an integer by CTOI. The result returned by CTOI is used by PUTCHARS to EXPAND the compression code to a sequence of identical characters.

```
DEFINE CK3 TEXT
    IF ATOM:3:TEXT
        THEN CONS:<FIRST:TEXT EXPAND:REST:TEXT>
    ELSE APPEND:<PUTCHARS:<CTOI:3:TEXT 2:TEXT>
                 EXPAND:RREST:TEXT>
•
-=>CK3

DEFINE CTOI STRING
    IF MINUSQ:FIRST:STRING
        THEN MINUS:2:POWERVAL:REST:STRING
    ELSE 2:POWERVAL:STRING
•
-=>CTOI

CTOI:"(1 3 5).
 -=>135

DEFINE POWERVAL STRING
    IF NULL:REST:STRING
        THEN <10 MAPDIGIT:FIRST:STRING>
    ELSE SUBTOTL:<MAPDIGIT:FIRST:STRING
                  POWERVAL:REST:STRING>
•
-=>POWERVAL

DEFINE SUBTOTL (N (POWER VAL))
    <TIMES:<POWER 10> PLUS:<TIMES:<POWER N> VAL>>
•
-=>SUBTOTL
```

```
DEFINE MINUSQ CHAR
    SAME:<CHAR MINUS>
.
-=>MINUSQ

DEFINE MAPDIGIT CHAR
    SUB1:INDEX:<CHAR DIGITS>
.
-=>MAPDIGIT

DEFINE RREST LIST
    REST:REST:LIST
.
-=>RREST
DEFINE RRREST LIST
    REST:REST:REST:LIST
.
-=>RRREST
```

PUTCHARS is a function which is useful in its own right, and so its definition is left independent of that of its calling function.

```
DEFINE PUTCHARS (COUNT CHAR)
    IF ONEP:COUNT THEN <CHAR>
    ELSE CONS:<CHAR PUTCHARS:<SUB1:COUNT CHAR>>
.
-=>PUTCHARS
```

INDEX takes as arguments a character, CHAR, and a list of zero or more characters, LCHARS. It returns the index of the first character in LCHARS that matches CHAR. Zero is returned if no match is found.

```
DEFINE INDEX (CHAR LCHARS)
    IF NULL:LCHARS THEN 0
    ELSEIF SAME:<CHAR FIRST:LCHARS> THEN 1
    ELSE INQ:INDEX:<CHAR REST:LCHARS>
.
-=>INDEX
```


INQ, called by INDEX, is another lookahead function. The decision of whether to add 1 to the recursive invocation of INDEX depends upon whether CHAR exists somewhere in LCHARS. INQ determines this by inspecting its argument, NUM.

```
DEFINE INQ NUM
    IF ZEROP:NUM THEN NUM
    ELSE ADD1:NUM
.
-=>INQ

DEFINE ZEROP NUM
    SAME:<NUM 0>
.
-=>ZEROP

INDEX:"(Z (A B C D)).
 -=>0

INDEX:"(Z (A B Z D)).
 -=>3
```

XLATE performs transliteration of certain characters in an input file, leaving all other characters in the file unchanged. XLATE takes as arguments a text file, TEXT, a list of character sets, LSETS, a boolean, ALLBUT, and two other lists of characters, FROM and TO. The main idea is to translate characters in the text file belonging to FROM to the corresponding characters in TO; but there are variations to this. If ALLBUT evaluates to TRUE, then the transliteration is applied to all characters in TEXT which don't match any characters in FROM. If TO is empty, then a function BREMBER is called to delete from TEXT all characters which either belong to (if ALLBUT evaluates to FALSE) or don't belong to (if ALLBUT evaluates to TRUE) the set FROM. If TO is non-empty, but contains fewer characters than FROM contains, it is assumed that the last

character in TO is repeated until TO contains as many characters as does FROM.

```
DEFINE XLATE (LSETS TEXT ALLBUT FROM TO)
    IF NULL:TEXT THEN <>
    ELSEIF NULL:TO
         THEN AREMBER:<TEXT ALLBUT MAKESET:<LSETS FROM>>
    ELSE XLATE1:<TEXT ALLBUT MAKESET:<LSETS FROM>
                             MAKESET:<LSETS TO>>

'-=>XLATE

      XLATE:<<> "(A B C D E) FALSE "(C D E F G) "(Q R S T U)>.
       -=> (A B Q R S)

      XLATE:<<> "(A B C D E) TRUE "(C D E F G) "(Q)>.
       -=>(Q Q C D E)

      XLATE:<<> "(T E X T F I L E) TRUE "(T) <>>.
       -=> (T T)
```

If strings like 'A-G' or '2-7' are found in FROM or TO, MAKESET calls DODASH to translate these shorthand strings to complete character strings. For example, if the string, 'A-H', is encountered in FROM or TO, a list of character sets, LSETS, is searched for a list of characters which contains a string beginning with an A and ending with an H. If no such string is found in any of the lists in LSETS, then the literal string 'A-H' is left unaltered. Also, MAKESET leaves unmodified all character strings which don't take the above mentioned shorthand form.


```
DEFINE MAKESET (LSETS STR)
    IF NULL:LSETS THEN STR
    ELSEIF OR:<NULL:STR NULL:REST:STR> THEN STR
    ELSEIF ESCAPEQ:FIRST:STR
         THEN CONS:<2:STR MAKESET:<LSETS RREST:STR>>
    ELSEIF NULL:RREST:STR THEN STR
    ELSEIF DASHQ:2:STR
         THEN APPEND:<DODASH:<1:STR 2:STR 3:STR LSETS>
                      MAKESET:<LSETS RRREST:STR>>
    ELSE CONS:<FIRST:STR MAKESET:<LSETS REST:STR>>

'-=>MAKESET

DEFINE DASHQ CHAR
    SAME:<CHAR DASH>

'-=>DASHQ

DEFINE ESCAPEQ CHAR
    SAME:<CHAR ESCAPE>

'-=>ESCAPEQ

DEFINE DODASH (A B C LSETS)
    IF NULL:LSETS THEN <A B C>
    ELSEIF CHOP:<A C FIRST:LSETS>
         THEN CHOP:<A C FIRST:LSETS>
    ELSE DODASH:<A B C REST:LSETS>

'-=>DODASH

MAKESET:"(((A B C D E)(H I J)) (A B C DASH E)).
 -=> (A B C D E)

MAKESET:"(((A B C D E))(X DASH Y Z)).
 -=> (X DASH Y Z)
```

MAKESET allows a dash to be escaped so that a literal dash is copied to the output.

```
MAKESET:"(((A B C D E))(A ESCAPE DASH E)).
 -=> (A DASH E)
```

CHOP looks through SET for a list of characters starting with some character A and ending with some character C. A lookahead function, IFADD, is used by BEFORE so that a null list may still be returned if a string is encountered which begins with A (of CHOP) but which doesn't end with B.

```
DEFINE CHOP (A C SET)
    BEFORE:<C AFTER:<A SET>>
.
-=>CHOP

DEFINE BEFORE (CHAR SET)
    IF NULL:SET THEN SET
    ELSEIF SAME:<CHAR FIRST:SET> THEN <CHAR>
    ELSE IFADD:<FIRST:SET BEFORE:<CHAR REST:SET>>
.
-=>BEFORE

DEFINE AFTER (CHAR SET)
    IF NULL:SET THEN SET
    ELSEIF SAME:<CHAR FIRST:SET> THEN SET
    ELSE AFTER:<CHAR REST:SET>
.
-=>AFTER

DEFINE IFADD (CHAR RESULT)
    IF NULL:RESULT THEN RESULT
    ELSE CONS:<CHAR RESULT>
.
-=>IFADD
```


Note that IFADD is structurally and logically equivalent to the function, INQ, using CONS as the constructor rather than ADD1.

As was already explained, BREMBER is called by XLATE if TO is empty. BMEMBER, a help function to BREMBER, returns CHAR if either 1) ALLBUT evaluates to FALSE and CHAR matches a character in TEXT, or 2) ALLBUT evaluates to TRUE and CHAR does not match any of the characters in TEXT.

```
DEFINE BREMBER (TEXT ALLBUT FROM)
    TCONS:<BMEMBER*>:<
            TEXT
            <ALLBUT*>
            <FROM*>>
.
-=>BREMBER

DEFINE BMEMBER (CHAR ALLBUT TEXT)
    IF XOR:<MEMBER:<CHAR TEXT> ALLBUT> THEN CHAR
    ELSE <>
.
-=>BMEMBER

DEFINE XOR (A B)
    IF A THEN NOT:B ELSE B
.
-=>XOR

DEFINE TCONS L
    IF NULL:L THEN L
    ELSEIF FIRST:L THEN CONS:<FIRST:L TCONS:REST:L>
    ELSE TCONS:REST:L
.
-=>TCONS

DEFINE MEMBER (CHAR LIST)
    OR:<SAME*>:<
        <CHAR*>
         LIST>
.
-=>MEMBER
```

If both TO and FROM are non-empty, then XLATE calls XLATE1. For each character in TEXT, XLATE1 calls XCHANGE which, with its help functions PICK and CONDPICK, decides whether or not the character should undergo transliteration. Since XCHANGE treats each character as a unit,depending upon the contents of the lists, FROM and TO, and the boolean ALLBUT to aid in transliteration of the character, it is easy to express XLATE1 as a star function.

```
DEFINE XLATE1 (TEXT ALLBUT FROM TO)
    <XCHANGE*>:<
     TEXT
    <ALLBUT*>
    <FROM*>
    <TO*>>

 -=>XLATE1

DEFINE XCHANGE (CHAR ALLBUT FROM TO)
    IF NULL:FROM THEN PICK:<CHAR ALLBUT TO>
    ELSEIF NULL:REST:TO
         THEN CONDPICK:<CHAR ALLBUT FROM TO>
    ELSEIF SAME:<FIRST:FROM CHAR>
         THEN PICK:<CHAR NOT:ALLBUT TO>
    ELSE XCHANGE:<CHAR ALLBUT REST:FROM REST:TO>

 -=>XCHANGE

DEFINE PICK (CHAR ALLBUT LCHARS)
    IF ALLBUT THEN FIRST:LCHARS
    ELSE CHAR

 -=>PICK

DEFINE CONDPICK (CHAR BOOL FROM TO)
    IF MEMBER:<CHAR FROM>
         THEN PICK:<CHAR NOT:BOOL TO>
    ELSE PICK:<CHAR BOOL TO>

 -=>CONDPICK
```


Constants Introduced in Chapter 2.

```
DECLARE NCHARS "NCHARS.
 -=>NCHARS

DECLARE BS "BS.
 -=>BS

DECLARE NS "NS.
 -=>NS

DECLARE SKIP "SKIP.
 -=>SKIP

DECLARE DIGITS <0 1 2 3 4 5 6 7 8 9>.
 -=> (0 1 2 3 4 5 6 7 8 9)

DECLARE MINUS "MINUS.
 -=>MINUS

DECLARE PLUS "PLUS.
 -=>PLUS

DECLARE ESCAPE "ESCAPE.
 -=>ESCAPE

DECLARE DASH "DASH.
 -=>DASH
```

CHAPTER 3.

In this chapter, some functions are presented which deal with the problem of file I/O. Kernighan and Plauger's discussion of this subject omits any discussion of the intricacies of the file system being used. For the most part, I have done the same, although for the purposes of demonstration, I have developed some very simple aspects of a file system.

The chapter begins with two logically similar functions which are used for comparing character strings.

LDIFFER takes two strings as arguments and returns TRUE if they aren't identical. LDIFFER knows the two strings differ if its help function, LOCDIFF returns a value greater than zero. LOCDIFF compares two strings, returning an integer which indicates the position of the first character at which they differ; if the two strings are identical, LOCDIFF returns zero. LOCDIFF's help function, ISDIF is identical to the function INQ which was a help function to INDEX (see page 27).

```
DEFINE LOCDIFF (STR1 STR2)
    IF AND:<NULL:STR1 NULL:STR2> THEN 0
    ELSEIF OR:<NULL:STR1 NULL:STR2> THEN 1
    ELSEIF SAME:<FIRST:STR1 FIRST:STR2>
        THEN ISDIF:LOCDIFF:<REST:STR1 REST:STR2>
    ELSE 1
.
-=>LOCDIFF

DEFINE LDIFFER (STR1 STR2)
    GREAT:<LOCDIFF:<STR1 STR2> 0>
.
-=>LDIFFER

DEFINE ISDIF NUM
    IF ZEROP:NUM THEN NUM
    ELSE ADD1:NUM
.
-=>ISDIF
```



```
DEFINE AND L
   IF NULL:L THEN TRUE
   ELSEIF NOT:FIRST:L THEN FALSE
   ELSE AND:REST:L
.
-=>AND
```

Here are three examples, each of which tests a different ground condition of the function LOCDIFF.

Example 1: The strings are identical; IF AND:<NULL:STR1 NULL:STR2> evaluates to TRUE.

```
LOCDIFF:"((A B C)(A B C)).
 -=>0
```

Example 2: The two strings are identical through their third characters, but the second string is longer than the first. IF OR:<NULL:STR1 NULL:STR2> evaluates to TRUE.

```
LOCDIFF:"((A B C)(A B C D)).
 -=>4
```

Example 3: The strings differ at the second element. ELSE 1 is the ground expression here.

```
LOCDIFF:"((A B C)(A X C)).
 -=>2
```

The function COMPARE takes two files as input and

returns TRUE if they are identical. COMPARE determines whether the two files are identical by checking the result of a help function, DOCOMPAR. DOCOMPAR, useful in its own right, compares two files, line by line to see if they differ. If at any time they differ, then DOCOMPAR returns a list containing the line number of the first line at which the files differ, and copies of the two differing lines. A list containing zero, and two empty files is returned if the two files are identical. A help function, DOCOMPQ, is similar in function to ISDIF. (see page 34).

```
DEFINE COMPARE (TEXT1 TEXT2)
    ZEROP:1:DOCOMPAR:<TEXT1 TEXT2>

-=>COMPARE

DEFINE DOCOMPAR (TEXT1 TEXT2)
    IF AND:<NULL:TEXT1 NULL:TEXT2> THEN <0 <> <>>
    ELSEIF LDIFFER:<1:GETLINE:TEXT1 1:GETLINE:TEXT2>
         THEN <1 1:GETLINE:TEXT1 1:GETLINE:TEXT2>
    ELSE DOCOMPQ:DOCOMPAR:<2:GETLINE:TEXT1 2:GETLINE:TEXT2>

-=>DOCOMPAR

DEFINE DOCOMPQ (NUM TEXT1 TEXT2)
    IF ZEROP:NUM THEN <NUM TEXT1 TEXT2>
    ELSE <ADD1:NUM TEXT1 TEXT2>

-=>DOCOMPQ
```


Example 1: TEXT1 and TEXT2, the two files being COMPAREd, are identical.

```
DOCOMPAR:"((A NL B NL)(A NL B NL)).
 -=> (0 () ())
```

Example 2: TEXT1 and TEXT2 differ at the second line.

```
DOCOMPAR:"((A NL B NL)(A NL C NL)).
 -=> (2 (B NL) (C NL))
```

The function GETLINE, a help function to DOCOMPAR, is used to isolate the first line of a file. GETLINE returns two lists, the first list being the first line of the file, and the second being the rest of the file.

The algorithm used here is similar to that of LINECT, which we saw in Chapter 1. If the line isolated by GETLINE is terminated by a newline (NL) character, that character is retained as the last character of the first list returned by the function.

```
DEFINE GETLINE TEXT
    IF NULL:TEXT THEN <<><>>
    ELSEIF NLQ:FIRST:TEXT THEN <<FIRST:TEXT> REST:TEXT>
    ELSE <CONS       1>:<
         <FIRST:TEXT  #>
         GETLINE:REST:TEXT>

-=>GETLINE
```

Example:

```
GETLINE:"(A L I N E NL R E S T).
 -=> ((A L I N E NL) (R E S T))
```

The function OPEN will be used frequently throughout the rest of this chapter. The purpose of the function is to provide for a user a means of accessing files from a file library. The user specifies the name of a file and an access mode, and if the name and mode are in order, OPEN returns the contents of the specified file.

OPEN has three arguments, a file name FNAME, an access mode indicator MODE, and a list of ordered triples (a file library) LFILES. Each triple in LFILES consists of: 1) a file name; 2) an access mode indicator; and 3) the contents of the file associated with the file name.

```
DEFINE OPEN (FNAME MODE LFILES)
    IF NULL:LFILES THEN <ERROR <>>
    ELSEIF EQSTR:<FNAME FIRST:FIRST:LFILES>
        THEN MTCHMODE:<MODE REST:FIRST:LFILES>
    ELSE OPEN:<FNAME MODE REST:LFILES>

-=>OPEN

DEFINE EQSTR (STR1 STR2)
    NULL:LDIFFER:<STR1 STR2>

-=>EQSTR

DEFINE MTCHMODE (MODE PAIR)
    IF SAME:<MODE 1:PAIR> THEN <OK 2:PAIR>
    ELSEIF READWRTQ:1:PAIR> THEN <OK 2:PAIR>
    ELSE <ERROR 1:PAIR>

-=>MTCHMODE

DEFINE READWRTQ STR
    SAME:<STR READWRIT>

-=>READWRTQ
```


LFILES is searched for a triple whose first element matches FNAME and whose second element matches or subsumes MODE. An access mode indicator must be one of the following symbolic constants; READ, WRITE, or READWRIT, indicating that a file is read-only, write-only, or both. READWRIT subsumes both READ and WRITE. If the first and second elements of a triple in LFILES match up with FNAME and MODE, then OPEN returns a status indicator (the symbolic constant OK) and the third element of the triple (the contents of file FNAME).

If there is no triple in LFILES whose first element matches FNAME (there is no file FNAME in the library), then a list containing the symbolic constant ERROR and an empty list is returned. If the file FNAME is in the library, but an improper MODE has been indicated, then a list containing ERROR and the proper access-mode indicator for the file is returned.

Example 1: Attempt to open a file which doesn't exist in the file library LFILES.

```
OPEN:"((B O O) READ ()).
 -=> (ERROR ())
```

Example 2: The file to be opened is in the file library, but the mode of the file in the library (READ) does not match or 'subsume' the MODE given in the attempt to open the file.

```
OPEN:"((B O O) READWRIT (((B O O) READ (F I L E)))).
 -=> (ERROR READ)
```

Example 3: The file is successfully opened.

```
OPEN:"((B O O) READ (((B O O) READWRIT (F C O N T E N T S)))).
 -=> (OK (F C O N T E N T S))
```

OPEN and its help functions don't provide elaborate error messages as they aren't likely to be used as top level functions.

At this point, let's boil down what LDIFFER, COMPARE, GETLINE, and OPEN are about. In each case, a list (or lists) is searched, element by element -- an element may either be a character or a list -- until 1) the list(s) is empty or 2) some property evaluates to TRUE about the first element of the list(s).

So, for each of the functions (or of the help functions which actually do the searching) there are two ground

conditions; a check for the null list, and "IF P", where P is a predicate concerning a property of the first element of the list(s).

FCOMPARE is a function almost identical to DOCOMPAR except that it must first OPEN the files it is to compare. If both files are successfully opened, then DOCOMPAR is called to compare the files; otherwise, an error message is returned.

```
DEFINE FCOMPARE (F1 F2 LFILES)
    IF ERRORQ:FIRST:OPEN:<F1 READ LFILES> THEN CANT:F1
    ELSEIF ERRORQ:FIRST:OPEN:<F2 READ LFILES>
         THEN CANT:F2
    ELSE DOCOMPAR:<2:OPEN:<F1 READ LFILES>
                  2:OPEN:<F2 READ LFILES>>
.
 -=>FCOMPARE

DEFINE ERRORQ CHAR
    SAME:<CHAR ERROR>
.
 -=>ERRORQ

DEFINE CANT STR
    APPEND:<"(NL CANT OPEN NL) STR>
.
 -=>CANT

DEFINE APPEND (L1 L2)
    IF NULL:L1 THEN L2
    ELSE CONS:<FIRST:L1 APPEND:<REST:L1 L2>>
.
 -=>APPEND
```

The function INCLUDE makes use of the function OPEN to provide a user with a convenient way to access an arbitrary number of library files. One may cause the contents of a file from a file library to be inserted into a text file by having in the text file a line beginning with the word INCLUDE, followed by a file name.

If a file with the given file name exists in the file library and has READ or READWRIT as its access mode, it is inserted into the text file, replacing the line which began with INCLUDE. Then this inclusion process is applied to the newly included file. The result of this is appended to the application of the inclusion to the original text file (minus its first line).

```
DEFINE INCLUDE (FILE LFILES)
    CKANS:<INCLUDE1:<1 "(TOPLEVEL) FILE LFILES> FILE>
.
 -=>INCLUDE

DEFINE INCLUDE1 (LEVEL FNAME FILE LFILES)
    IF NULL:FILE
        THEN <OK APPEND:<"(NL END OF FILE ) FNAME> FILE>
    ELSE FEXPAND:<LEVEL FNAME 1:GETLINE:FILE 2:GETLINE:FILE LFILES>
.
 -=>INCLUDE1

DEFINE FEXPAND (LEVEL FNAME LINE FILE LFILES)
    IF INCLUDEQ:GETWORD:LINE
        THEN ADDFILE:<LEVEL FNAME INCLUDEQ:GETWORD:LINE FILE LFILES>
    ELSE <1     1     APPEND>:<
         <#     #       LINE>
         INCLUDE1:<LEVEL FNAME FILE LFILES>>
.
 -=>FEXPAND
```


The first thing we need to make this process work is a function to extract the first word from a line--(as was explained in Chapter 1; a word is a sequence of non-whitespace characters). The function GETWORD accomplishes this. It takes a list as its argument and returns two lists, the first containing the first word encountered (leading white-space characters are lost) and the second containing the rest of the original list.

```
DEFINE GETWORD FILE
    GETSTR:TRIM:FILE
.
 -=>GETWORD

DEFINE GETSTR FILE
    IF NULL:FILE THEN <<><>>
    ELSEIF OUTWORD:FIRST:FILE THEN <<> FILE>
    ELSE <CONS          1>:<
         <FIRST:FILE    #>
          GETSTR:REST:FILE>
.
 -=>GETSTR

DEFINE OUTWORD CHAR
    OR:<TABQ:CHAR BLANKQ:CHAR NLQ:CHAR>
.
 -=>OUTWORD

DEFINE TRIM FILE
    IF NULL:FILE THEN <>
    ELSEIF OUTWORD:FIRST:FILE THEN TRIM:REST:FILE
    ELSE FILE
.
 -=>TRIM
```

Example:

```
GETWORD:"(M Y BLANK W O R D NL).
 ==> ((M Y) (BLANK W O R D NL))
```

The function INCLUDEQ uses GETWORD to check the first word of a line to see if it matches the keyword INCLUDE. If it does, the second word of the line, presumably a file name, is returned. An empty list is returned by INCLUDEQ if the first word of the line isn't INCLUDE.

```
DEFINE INCLUDEQ (WORD LINE)
    IF EQSTR:<WORD "(I N C L U D E)>
        THEN 1:GETWORD:LINE
    ELSE <>

.
==>INCLUDEQ
```

If INCLUDEQ returns a non-empty list, the function ADDFILE is called from FEXPAND to attempt to open a file, using the list returned by INCLUDEQ as a file name.

If ADDFILE's attempt to open a file fails, a message indicating this is appended to a message file which is one of the results of the function, INCLUDE. The occurrence of this error does not terminate the inclusion process; INCLUDE1 is called with FILE (what remains of the original input FILE to INCLUDE) as one of its arguments.

Since included files may INCLUDE other files, a counter, LEVEL, is used to keep track of how deeply embedded INCLUDEd files have become. If LEVEL exceeds the integer constant, DEEP, the program terminates with the ground expression, ERROR.

If ADDFILE successfully opens a new file and the level of includes is not too deep, then the inclusion process is applied to the newly opened file via a recursive call to INCLUDE1. The lookahead CKRESULT checks for a fatal error condition before appending an "included" file to the result of the application of the inclusion process to what is left of the original input FILE for the function INCLUDE.

```
DEFINE ADDFILE (LEVEL OLD NEW FILE LFILES)
    IF ERROR:FIRST:OPEN:<NEW READ LFILES>
        THEN <1          APPEND          1>:<
              <#         CANT:NEW        #>
               INCLUDE1:<LEVEL OLD FILE LFILES>>
    ELSEIF SAME:<LEVEL DEEP>
        THEN <ERROR <> <>>
    ELSE CKRESULT:<INCLUDE1:<ADD1:LEVEL NEW 2:OPEN:<NEW READ LFILES>
                             LFILES> LEVEL OLD FILE LFILES>

.
==>ADDFILE

DEFINE CKRESULT (ANS LEVEL FNAME FILE LFILES)
    IF ERRORQ:1:ANS THEN ANS
    ELSE <1          APPEND     APPEND>:<
          <#         2:ANS      3:ANS >
           INCLUDE1:<LEVEL FNAME FILE LFILES>>

.
==>CKRESULT
```

CKANS is a lookahead function invoked within the definition of INCLUDE. It is needed because there may be many messages in the message file and the original input FILE may have undergone many changes before detection of a fatal error condition. If a fatal error has occurred, then CKANS returns the message file of the aborted run, and the original copy of FILE. The message file contains a trace of the progress of the program up to the occurrence of the error.

If no error has occurred, then CKANS returns the message file and the altered text file.

```
DEFINE CKANS (ANS FILE)
    IF ERRORQ:1:ANS
        THEN <APPEND:<"(INCLUDE TOO DEEP NL) 2:ANS> FILE>
    ELSE REST:ANS
.
-=>CKANS
```


Example 1: There is no line in the input file beginning with the key word INCLUDE.

```
INCLUDE:"((A NL F I L E)
          (((N A M E) READ (C O N T E N T S)))).
 -=> ((NL END OF FILE TOPLEVEL) (A NL F I L E))
```

Example 2: There is a line beginning with the key word INCLUDE, but the attempt to open the specified file fails.

```
INCLUDE:"((I N C L U D E BLANK B O O)
          (((N E E K T) READ (B R Z Z Z O T)))).
 -=> ((NL CANT OPEN B O O NL END OF FILE TOPLEVEL) ())
```

Example 3: There is a line beginning with the key word INCLUDE and the file is successfully opened.

```
INCLUDE:"((I N C L U D E BLANK B O O NL O L D F I L E)
          (((B O O) READ (NL B O O F I L E)))).
 -=> ((NL END OF FILE B O O NL END OF FILE TOPLEVEL)  (NL B O O F I L E
  O L D F I L E))
```

Example 4: The level of included files has become too deep and the program aborts on a fatal error condition. (I have set DEEP to 1 for this run, so one INCLUDE will cause the error).

```
INCLUDE:"((I N C L U D E BLANK B O M B)
          (((B O M B) READ (U N S E E N)))).
 -=> ((INCLUDE TOO DEEP NL) (I N C L U D E BLANK B O M B))
```


FCONCAT concatenates a list of files into one file. One may think of FCONCAT as "a version of INCLUDE that takes all its file names from an argument list instead of from lines saying include". [77] A message file is kept to record all unsuccessful attempts to open files.

```
DEFINE FCONCAT (FNAMES LFILES)
    IF NULL:FNAMES THEN <<><>>
    ELSE FETCHONE:<FIRST:FNAMES OPEN:<FIRST:FNAMES READ LFILES>
                  REST:FNAMES LFILES>
.
-=>FCONCAT

DEFINE FETCHONE (NAME (STATUS FILE) FNAMES LFILES)
    IF ERROR0:STATUS
        THEN <APPEND        1>:<
             <CANT:NAME     #>
              FCONCAT:<FNAMES LFILES>>
    ELSE <1      FCOPY>:<
         <#       FILE>
          FCONCAT:<FNAMES LFILES>>
.
-=>FETCHONE

DEFINE FCOPY (FILE1 FILE2)
    APPEND:<FILE1 FILE2>
.
-=>FCOPY

FCONCAT:"(((F I L E 1)(F I L E 2)(B O M B))
         (((F I L E 1) READ (F 1 C O N T E N T S NL))
          ((F I L E 2) READ (F 2 C O N T E N T S NL)))).
 -=> ((NL CANT OPEN B O M B) (F 1 C O N T E N T S NL F 2 C O N T E N T S
  NL))
```

PRETTY is invoked to list the contents of one or more files in a readable format; it prints the files with the file name and page number at the top of each page. Each new file begins on a new page. [78] PRETTY also keeps an error log of unsuccessful attempts to open files.

```
DEFINE PRETTY (FNAMES LFILES)
    IF NULL:FNAMES THEN <<><>>
    ELSE PRETTY1:<FIRST:FNAMES OPEN:<FIRST:FNAMES READ LFILES>
                 REST:FNAMES LFILES>

-=>PRETTY


     DEFINE PRETTY1 (NAME (STATUS FILE) FNAMES LFILES)
        IF ERRORQ:STATUS
            THEN <APPEND          1>:<
                 <CANT:NAME       #>
                  PRETTY:<FNAMES LFILES>>
        ELSE <1                    APPEND>:<
             <#    FPRETTY:<NAME FILE>>
              PRETTY:<FNAMES LFILES>>

     -=>PRETTY1
```

Structurally, PRETTY is almost identical to FCONCAT. The only difference is that before a file is added to the list of files built by PRETTY, it is formatted by the function FPRETTY.

FPRETTY is a primitive version of FORMAT, a text formatter which is presented in Chapter 7. FPRETTY creates

output in pages. A page has PAGESIZE (an integer constant) lines. There are MARG1 blank lines at the beginning of a page, followed by a header line, which contains a page number and a file name; after the header line come MARG2 blank lines. Text from a file begins at the next line (at line number MARG1 + MARG2 + 2). The last line of a page upon which the contents of a file may be written is line number BOTTOM. The rest of the page is filled with blank lines.

```
DEFINE FPRETTY (NAME FILE)
    FPRETTY1:<0 0 NAME FILE>

-=>FPRETTY

DEFINE FPRETTY1 (LINO PAGENO FNAME TEXT)
   IF NULL:TEXT THEN MAKUPDIF:LINO
   ELSE FPRETTY2:<LINO PAGENO FNAME 1:GETLINE:TEXT 2:GETLINE:TEXT>

-=>FPRETTY1

DEFINE FPRETTY2 (LINO PAGENO FNAME LINE TEXT)
    IF ZEROP:LINO
         THEN APPEND:<TOPPAGE:<PAGENO FNAME LINE>
                      FPRETTY1:<SUM:<MARG1 MARG2 3> ADD1:PAGENO
                                FNAME TEXT>>
    ELSEIF LESS:<LINO BOTTOM>
         THEN APPEND:<LINE FPRETTY1:<ADD1:LINO PAGENO FNAME TEXT>>
    ELSE CONCAT:<LINE SKIP:DIFF:<PAGESIZE LINO>
                 FPRETTY1:<0 PAGENO FNAME TEXT>>

-=>FPRETTY2

DEFINE SUM L
   IF NULL:REST:L THEN FIRST:L
   ELSE PLUS:<FIRST:L SUM:REST:L>

-=>SUM
```

FPRETTY2 uses the help function TOPPAGE to insert margins and the header line at the top of each new page. FPRETTY1 calls MAKUPDIF to fill in the last page of a file with blank lines. PUTDEC, called by HEAD, was presented in Chapter 2.

```
DEFINE MAKUPDIF LINO
    IF ZEROP:LINO THEN <>
    ELSE SKIP:DIFF:<PAGESIZE LINO>

-=>MAKUPDIF

DEFINE TOPPAGE (PAGENO FNAME LINE)
    CONCAT:<SKIP:MARG1 HEAD:<FNAME PAGENO> SKIP:MARG2 LINE>

-=>TOPPAGE

DEFINE SKIP NUM
    IF ZEROP:NUM THEN <>
    ELSE CONS:<NL SKIP:SUB1:NUM>

-=>SKIP
DEFINE HEAD (NAME PAGENO)
    CONCAT:<NAME "(BLANK P A G E) PUTDEC:<PWIDTH PAGENO> "(NL)>

-=>HEAD
```


For the following examples of PRETTY, I have created a short page, with MARG1 = 1, MARG2 = 2, BOTTOM = 10 and PAGESIZE = 12. The first list returned by PRETTY is a message file for error messages.

```
PRETTY:"(((N E L L I E) (L U I S))
         (((N E L L I E) WRITE (S E C O N D))
          ((L U I S) READ (S H O R T)))).
-=> ((NL CANT OPEN N E L L I E) (NL L U I S BLANK P A G E BLANK BLANK
BLANK 0 NL NL S H O R T NL NL NL NL NL NL NL NL NL))

PRETTY:"(((F I L E 1)) (((F I L E 1) READ
                         (C NL O NL N NL T NL E NL N NL T NL S
                          NL O NL F NL F NL I NL L NL E NL 1 NL)))).
-=> ((() (NL F I L E 1 BLANK P A G E BLANK BLANK BLANK 0 NL NL C NL O NL
N NL T NL E NL N NL T NL NL NL NL F I L E 1 BLANK P A G E BLANK BLANK
BLANK 1 NL NL S NL O NL F NL F NL I NL L NL E NL NL NL NL F I L E 1
BLANK P A G E BLANK BLANK BLANK 2 NL NL 1 NL NL NL NL NL NL NL NL NL))
```

So far, using OPEN, we have been concerned with retrieving the contents of a file from a file library. MAKECOPY replaces the contents of a given file ONAME, in a file library, LFILES, with the contents of another file, INAME also in LFILES. If there is no INAME in LFILES, an error message is returned and the function terminates. If there is no ONAME in LFILES, but there is an INAME, then an entry in the library for ONAME is created.

```
DEFINE MAKECOPY (INAME ONAME LFILES)
    SUBFILE:<INAME OPEN:<INAME READ LFILES> ONAME LFILES>
.
-=>MAKECOPY

DEFINE SUBFILE (INAME (STATUS IFILE) ONAME LFILES)
    IF ERRORQ:STATUS THEN <ERROR LFILES>
    ELSE CREATE:<<ONAME READWRIT IFILE> LFILES>
.
-=>SUBFILE

MAKECOPY:"(((I N A M E) (O N A M E)
          (((I N A M E READWRIT (S T U F F NL M O R E NL
          S T U F F)))).
-=> (OK (((I N A M E) READWRIT (S T U F F NL M O R E NL S T U F F)) ((O
 N A M E) READWRIT (S T U F F NL M O R E NL S T U F F))))
```

CREATE inserts a new ordered triple (filename, mode indicator, file contents) into a file library LFILES. The algorithm is:

1) If LFILES is empty, then insert the ordered triple.
2) If the file name of the new ordered triple matches the file name of the first triple in the library, then a help function, MODECHEK, is called. MODECHEK compares the access mode of the new ordered triple

with the mode of the first triple in the library. If the modes match, the new triple replaces the first triple in the library, and OK status is returned. If the modes don't match, ERROR status is returned along with an unaltered copy of the file library.

```
DEFINE CREATE (TRIPLE LFILES)
    IF NULL:LFILES THEN <OK CONS:<TRIPLE LFILES>>
    ELSEIF FMATCH:<TRIPLE FIRST:LFILES>
         THEN MODECHEK:<TRIPLE FIRST:LFILES LFILES>
    ELSE <1                CONS>:<
         <#        FIRST:LFILES>
          CREATE:<TRIPLE REST:LFILES>>
.
-=>CREATE

DEFINE FMATCH (TRIP1 TRIP2)
    EQSTR:<1:TRIP1 1:TRIP2>
.
-=>FMATCH

DEFINE MODECHEK (TRIP1 TRIP2 LFILES)
    IF SAME:<2:TRIP1 2:TRIP2> THEN <OK CONS:<TRIP1 REST:LFILES>>
    ELSE <ERROR LFILES>
.
-=>MODECHEK

CREATE:"(((L U I S) READ (A P A R I C I O))
        (((E R N I E) READ (B A N K S)))).
-=> (OK (((E R N I E) READ (B A N K S)) ((L U I S) READ (A P A R I C I
 O))))

CREATE:"(((B O O) READ (U G H)) (((B O O) READ (H A)))).
-=> (OK (((B O O) READ (U G H))))

CREATE:"(((N E W) READ (L U I S))
        (((N E W) READWRIT (J O R G E)))).
-=> (ERROR (((N E W) READWRIT (J O R G E))))
```

The function ARCHIVE " is a library maintainer whose purpose is to collect sets of arbitrary files into one big file and to maintain that file as an archive. Files can be extracted from the archive, new ones can be added, old ones can be deleted or replaced by updated versions, and data about the contents may be listed". [85]

The first line of a file in an archive is a header line, which contains a file name followed by an indicator of the number of characters in the file (minus the header).

The arguments to the function ARCHIVE are a command COM, the name of an archive ARCHNAME, a list of file names, FNAMES, and a file library LFILES. ARCHIVE produces two output files; the contents of the first output file, the standard output file, are intended to be displayed on a line printer or other output device. The contents of the second file vary, depending upon COM.

The archive commands are:

UPDATE--(U)--"update named members or add at end"

DELETE--(D)--"delete named members from archive"

LIST--(L)-- list contents of named members

EXTRACT--(E)--"extract named files from archive"

TABLE--(T)-- list table of archive contents. [86]

For all archive commands, an attempt is made to open some file ARCHNAME from a file library, LFILES (the access mode varies depending upon COM). COM is applied to all files in the archive which are named in the file, FNAMES.

If FNAMES is empty, COM is applied to all files in the archive. Before processing begins for a command, duplicate file names in FNAMES are weeded out and noted in the standard output file.

```
DEFINE ARCHIVE (COM ARCHNAME FNAMES LFILES)
    IF UPDATEQ:COM
        THEN UPDATE:<OPEN:<ARCHNAME READWRIT LFILES> ARCHNAME
                    FNAMES LFILES>
    ELSEIF DELETEQ:COM
        THEN TRYOPEN:<OPEN:<ARCHNAME READWRIT LFILES> COM
                    ARCHNAME FNAMES LFILES>
    ELSE TRYOPEN:<OPEN:<ARCHNAME READ LFILES> COM ARCHNAME
                FNAMES LFILES>
.
-=>ARCHIVE

DEFINE UPDATEQ COM
    SAME:<COM UPDATE>
.
-=>UPDATEQ
DEFINE DELETEQ COM
   SAME:<COM DELETE>
.
-=>DELETEQ
```

If COM is an UPDATE, an attempt is made to open file ARCHNAME in READWRIT access mode. READWRIT mode is used since the UPDATE command may result in both the inspection and alteration of information in an archive.

```
DEFINE UPDATE ((STATUS ARCHIVE) ARCHNAME FNAMES LFILES)
    IF ERRMODE:<STATUS ARCHIVE>  THEN <CANT:ARCHNAME LFILES>
    ELSE CKNAMES:<ARCHNAME ARCHIVE FNAMES LFILES>
.
-=>UPDATE

DEFINE ERRMODE (STATUS FILE)
   AND:<ERRORQ:STATUS FILE>
.
-=>ERRMODE
```

The function UPDATE does the following:

1) If the attempt to open file ARCHNAME fails due to improper access mode, then UPDATE returns an error message and an unaltered copy of the file library, LFILES.
2) If there is no file ARCHNAME in LFILES, then an empty one is created.
3) If FNAMES, the list of file names; is empty, then each file in the archive is replaced with a new one of the same name from LFILES. An appropriate header line is added to each new file in the archive. (see discussion of functions CKNAMES and ALLNEW below).
4) If FNAMES is not empty, then for each file name FNAME in the list of names FNAMES, an attempt is made to open a file of that name in LFILES and replace a file of that name in the archive with the newly opened (in READ mode) file. If there is no ordered triple in LFILES identified by the name FNAME, then a message to this effect is appended to a message file. If a file FNAME is opened, but there is no file FNAME in the archive, then the newly opened file is added to the archive. (See discussion of functions CKNAMES, UPDATE2 and PUTFILE below).

The function CKNAMES is called after a file ARCHNAME has been opened or an empty one created. CKNAMES checks to see if the list of file names, FNAMES, is empty, and calls the appropriate help function (refer to steps 3 and 4 above).


```
DEFINE CKNAMES (ARCHNAME ARCHIVE FNAMES LFILES)
    IF NULL:FNAMES
         THEN <<> 2:CREATE:<<ARCHNAME READWRIT
                              ALLNEW:<ARCHIVE LFILES>> LFILES>>
    ELSE  <APPEND                        1>:<
          <1:GETNAMES:FNAMES             #>
           UP2CREAT:<UPDATE2:<ARCHIVE 2:GETNAMES:FNAMES LFILES>
                     ARCHNAME LFILES>>
.
-=>CKNAMES
```

If FNAMES is not empty, the function GETNAMES is called to weed out duplicate file names from the list of names. Two lists are returned by GETNAMES; the first, which is appended to the standard output file, contains messages signalling the existence of duplicate file names in FNAMES. The second list consists of FNAMES with duplicates removed.

```
DEFINE GETNAMES FNAMES
    IF NULL:FNAMES THEN <<> <>>
    ELSEIF NULL:REST:FNAMES THEN <<> FNAMES>
    ELSEIF DUPNAME:<FIRST:FNAMES REST:FNAMES>
         THEN <APPEND                    1>:<
              <DUPMESS:FIRST:FNAMES #>
               GETNAMES:REST:FNAMES>
    ELSE <1                  CONS>:<
         <#        FIRST:FNAMES>
          GETNAMES:REST:FNAMES>
.
-=>GETNAMES

DEFINE DUPNAME (NAME FNAMES)
    OR:<EQSTR*>:<
       <NAME*>
        FNAMES>
.
-=>DUPNAME

DEFINE DUPMESS NAME
    CONS:<NL APPEND:<"(DUPLICAT  ) NAME>>
.
-=>DUPMESS

GETNAMES:"((C A R M E N) (A L) (C A R M E L I T A) (A L)).
 -=> ((NL DUPLICAT A L) ((C A R M E N) (C A R M E L I T A) (A L)))
```

The function UPDATE2 is called to replace files in the archive which are named in FNAMES.

```
DEFINE UPDATE2 (ARCHIVE FNAMES LFILES)
    IF NULL:FNAMES
         THEN <<> ARCHIVE>
    ELSE PUTFILE:<OPEN:<FIRST:FNAMES READ LFILES>
                  FIRST:FNAMES ARCHIVE
                  REST:FNAMES LFILES>
-=>UPDATE2
```

The process of updating an archive, using UPDATE2, goes as follows:

1) For each file name FNAME in the list of files FNAMES, the function PUTFILE is called. PUTFILE checks to see if an attempt to open a file FNAME from the file library LFILES has succeeded. If not, then an error message is output.

```
DEFINE PUTFILE ((STATUS NEWFILE) FNAME ARCHIVE FNAMES LFILES)
    IF ERRORQ:STATUS
         THEN <APPEND         1>:<
              <CANT:FNAME     #>
               UPDATE2:<ARCHIVE FNAMES LFILES>>
    ELSE <1                        CONS>:<
         <#     MAKHDR:<FNAME NEWFILE>>
          UPDATE2:<2:GRABONE:<FNAME ARCHIVE>
                   FNAMES LFILES>>

-=>PUTFILE
```


2) If a file FNAME has been successfully opened, then the function GRABONE is called to extract a file of the same name from the archive. The second argument of GRABONE is the contents of the archive minus file FNAME.

```
DEFINE GRABONE (NAME ARCHIVE)
    IF NULL:ARCHIVE THEN <<> ARCHIVE>
    ELSEIF FOUNDQ:<NAME FIRST:ARCHIVE>
         THEN <FIRST:ARCHIVE REST:ARCHIVE>
    ELSE <1              CONS>:<
         <#     FIRST:ARCHIVE>
          GRABONE:<NAME REST:ARCHIVE>>

-=>GRABONE
```

```
GRABONE:"((A F I L E)
          ((C F I L E BLANK 8 NL C F I L E C O N)
           (B F I L E BLANK 8 NL B F I L E C O N)
           (A F I L E BLANK 8 NL A F I L E C O N))).
-=> ((A F I L E BLANK 8 NL A F I L E C O N) ((C F I L E BLANK 8 NL C F
I L E C O N) (B F I L E BLANK 8 NL B F I L E C O N)))
```

```
DEFINE FOUNDQ (NAME FILE)
    EQSTR:<NAME 1:GETWORD:FILE>

-=>FOUNDQ
```

3) MAKHDR is called to add a header line to the newly opened file FNAME, and the file is appended to the second result being built by UPDATE2, the new archive.

```
DEFINE MAKHDR (NAME FILE)
    CONCAT:<NAME PUTDEC:<PWIDTH CHARCT:FILE> CONS:<NL FILE>>
.
-=>MAKHDR
```

4) UPDATE2 returns two results, a message file containing error messages signalling any failures to open files from FNAMES, and a new archive file.

```
UPDATE2:"(((X BLANK 9 NL X C O N T E N T S)
      (Y BLANK 9 NL Y C O N T E N T S)
     (Z Z BLANK 1 0 NL Z Z C O N T E N T S))
      ((X) (Y))
      (((X) READ (N E W X C O N T E N T S))
     ((Y) WRITE (N E W Y C O N T E N T S)))).
 -=> ((NL CANT OPEN Y) ((X BLANK BLANK 1 2 NL N E W X C O N T E N T S)
 (Y BLANK 9 NL Y C O N T E N T S) (Z Z BLANK 1 0 NL Z Z C O N T E N T S)
 ))
```


5) UP2CREAT, called in CKNAMES, returns the message file from UPDATE2, and a new file library with an updated version of the archive, ARCHNAME.

```
DEFINE UP2CREAT ((MESS FILE) ARCHNAME LFILES)
    <MESS 2:CREATE:<<ARCHNAME READWRIT FILE> LFILES>>
.
-=>UP2CREAT
```

If the UPDATE command is used and the list of file names, FNAMES, is empty (see definition of CKNAMES-- IF NULL:FNAMES . . .), a function ALLNEW is called to update all files in the archive. ALLNEW builds a new archive from the old version in the following way:

1) GETNAME is called to extract the file name, NAME (the first word of the header line), of each file in the archive.
2) An attempt is made to open file NAME from the file library, LFILES.
3) If the attempt was successful, then the contents of this new file are added to the archive being built, replacing the old version of the file in the archive.
4) Otherwise, the old archive file is added to the new archive, unaltered.

After ALLNEW has created a new version of an archive, CREATE is called to replace the old version of the archive in the file library with the new one. (See definition of CKNAMES).

```
DEFINE ALLNEW (ARCHIVE LFILES)
    IF NULL:ARCHIVE THEN <>
    ELSE CONS:<NEWFILE:<GETNAME:FIRST:ARCHIVE FIRST:ARCHIVE LFILES>
              ALLNEW:<REST:ARCHIVE LFILES>>

.
 -=>ALLNEW

DEFINE GETNAME FILE
    1:GETWORD:FILE

.
 -=>GETNAME

DEFINE NEWFILE (NAME FILE LFILES)
    MAKEFILE:<OPEN:<NAME READWRIT LFILES> NAME FILE>

.
 -=>NEWFILE

DEFINE MAKEFILE ((STATUS NEWFILE) NAME OLDFILE)
    IF ERRORQ:STATUS THEN OLDFILE
    ELSE MAKHDR:<NAME NEWFILE>

.
 -=>MAKEFILE

ALLNEW:"(((B O O BLANK 1 1 NL B O O C O N T E N T S)
         (H A R O L D BLANK 1 4 NL H A R O L D C O N T E N T S)
         (H A BLANK 1 0 NL H A C O N T E N T S))
         (((B O O) READWRIT (N E W B O O))
          ((H A R O L D) READWRIT (N E W H A R O L D)))).
-=> ((B O O BLANK BLANK BLANK 6 NL N E W B O O) (HAROLD BLANK BLANK
BLANK 9 NL N E W H A R O L D) (H A BLANK 1 0 H A C O N T E N T S))
```


If an archive command is anything but an UPDATE, the function TRYOPEN is called. (See definition of ARCHIVE). If for any reason, the attempt to open the archive, ARCHNAME, fails then TRYOPEN terminates with an error message; otherwise, the function GETCOM is called to determine what the command is and to perform the appropriate command.

```
DEFINE TRYOPEN ((STATUS ARCHIVE) COM ARCHNAME FNAMES LFILES)
    IF ERRORQ:STATUS THEN <CANT:ARCHNAME LFILES>
    ELSE <APPEND              1>:<
         <1:GETNAMES:FNAMES #>
         GETCOM:<COM ARCHIVE 2:GETNAMES:FNAMES ARCHNAME LFILES>>

.
 -=>TRYOPEN

DEFINE GETCOM (COM ARCHIVE FNAMES ARCHNAME LFILES)
    IF BADCOM:COM THEN <ERRCOM:COM ARCHIVE>
    ELSE GETCOM1:<COM FDISPLAY:<ARCHIVE FNAMES> ARCHNAME LFILES>

.
 -=>GETCOM

DEFINE BADCOM COM
    NOT:MEMBER:<COM ARCHCOMS>

.
 -=>BADCOM

DEFINE ERRCOM COM
    CONCAT:<<NL BLANK> <COM> "(I S BLANK N O T BLANK A BLANK
                               P R O P E R BLANK C O M M A N D)>.

 -=>ERRCOM
```

The performance of each command involves a call to FDISPLAY. The purpose of FDISPLAY is to extract from an archive all files named in a list of file names, FNAMES (duplicate file names are removed from the list through a call to GETNAMES in the definition of TRYOPEN). FDISPLAY returns three output files: 1) a log of all file names from FNAMES for which there are no entries in the archive; 2) a list of all files extracted from the archive; and 3) the archive, less the files which have been extracted.

In the special case that FNAMES is empty, the second file is the whole original archive, and the other two files are empty.

```
DEFINE FDISPLAY (ARCHIVE FNAMES)
    IF NULL:FNAMES THEN <<> ARCHIVE <>>
    ELSE FDISP1:<ARCHIVE FNAMES>

-=>FDISPLAY

DEFINE FDISP1 (ARCHIVE FNAMES)
    IF NULL:FNAMES THEN <<><> ARCHIVE>
    ELSE ONEDISP:<FIRST:FNAMES GRABONE:<FIRST:FNAMES ARCHIVE>
                  REST:FNAMES>

-=>FDISP1

DEFINE ONEDISP (NAME PAIR FNAMES)
    IF NULL:1:PAIR
         THEN <APPEND          1      1>:<
              <NFOUND:NAME     #      #>
               FDISP1:<2:PAIR FNAMES>>
    ELSE <1       CONS       1>:<
         <#     1:PAIR       #>
          FDISP1:<2:PAIR FNAMES>>

-=>ONEDISP

DEFINE NFOUND NAME
   CONCAT:<<NL BLANK> "(F I L E BLANK) NAME
          "(BLANK N O T BLANK F O U N D)>

-=>NFOUND
```



```
FDISPLAY:"(((X BLANK 5 NL X F I L E)
            (Y BLANK 5 NL Y F I L E)
            (Z BLANK 5 NL Z F I L E)
            (8 BLANK 5 NL 8 F I L E))  ((X) (Z))).
-=> (() ((X BLANK 5 NL X F I L E) (Z BLANK 5 NL Z F I L E)) ((Y BLANK 5
 NL Y F I L E) (8 BLANK 5 8 F I L E)))
```

Most of the commands are performed within the body of GETCOM1. A DELETE command returns the standard output file and a copy of the file library with the old version of the archive replaced by a new version with deletions, OTHERS. The LIST command returns the standard output file, containing all files which have been extracted from the archive, and an empty list. The EXTRACT command returns the standard output file, and a list of all files extracted from the archive. The TABLE command returns the standard output file and a list of the header lines from all files extracted from the archive. In all the above cases, the "standard output file" contains all error messages accumulated during processing of the ARCHIVE command.

```
DEFINE GETCOM1 (COM (MESS TARGFILS OTHERS) ARCHNAME LFILES)
   IF DELETEQ:COM
        THEN <MESS 2:CREATE:<<ARCHNAME READWRIT OTHERS> LFILES>
   ELSEIF LISTQ:COM
        THEN <CONCAT:<MESS <NL NL> TARGFILS> <>>
   ELSEIF EXTRACTQ:COM
        THEN <MESS TARGFILS>
   ELSE <MESS TLIST:TARGFILS>

-=>GETCOM1

DEFINE LISTQ COM
    SAME:<COM LIST>

-=>LISTQ
```

```
DEFINE EXTRACTQ COM
    SAME:<COM EXTRACT>
.
 -=>EXTRACTQ

DEFINE TLIST LFILES
    CONCAT:<PHEAD*>:<LFILES>
.
 -=>TLIST

DEFINE PHEAD FILE
    1:GETLINE:FILE
.
 -=>PHEAD
```


Constants Introduced in Chapter 3.

```
DECLARE READ "READ.
 -=>READ

DECLARE READWRIT "READWRIT.
 -=>READWRIT

DECLARE WRITE "WRITE.
 -=>WRITE

DECLARE ERROR "ERROR.
 -=>ERROR

DECLARE DEEP 4.
 -=>4

DECLARE PWIDTH 4.
 -=>4

DECLARE ARCHCOMS <UPDATE DELETE EXTRACT LIST TABLE>.
 -=> (U D E L T)

DECLARE UPDATE "U.
 -=>U

DECLARE DELETE "D.
 -=>D

DECLARE EXTRACT "E.
 -=>E

DECLARE LIST "L.
 -=>L

DECLARE TABLE "T.
 -=>T
.
DECLARE MARG1 1.
 -=>1

DECLARE MARG2 1.
 -=>1
.
DECLARE BOTTOM 10.
 -=>10

DECLARE PAGESIZE 12.
 -=>12
```

```
DECLARE ERROR "ERROR.
 -=>ERROR

DECLARE OK "OK.
 -=>OK
```


CHAPTER 4.

This chapter is made up of a group of increasingly elaborate sorting algorithms First, algorithms for sorting lists of numbers are presented, and later, extensions on these algorithms are used for sorting text files.

My Shell sort is a direct translation of the algorithm used by Kernighan and Plauger in their book[105] The quick sort presented here was suggested by another paper [FW] .

The function BUBBLE does a bubble sort on a list of numbers. The algorithm presented here differs slightly from that presented in Kernighan and Plauger's book. In their algorithm, there is a loop which "rearranges out-of-order adjacent elements on each pass; by the end of the pass, the largest element has been bubbled to the top". [105]
My version of BUBBLE has a help function, SINKSMAL which also "rearranges out-of-order adjacent elements on each pass", but which results in the smallest element being located at the top of the list.

```
DEFINE BUBBLE L
     IF NULL:L THEN <>
     ELSE CONS:<FIRST:SINKSMAL:L BUBBLE:REST:SINKSMAL:L>
.
 -=>BUBBLE

DEFINE SINKSMAL L
    IF NULL:REST:L THEN L
    ELSE ORDER:<FIRST:L SINKSMAL:REST:L>
.
 -=>SINKSMAL
```

ORDER is a lookahead function used to order adjacent elements in a list.

```
DEFINE ORDER (N L)
    IF LESS:<N FIRST:L> THEN CONS:<N L>
    ELSE CONS:<FIRST:L CONS:<N REST:L>>
.
-=>ORDER


BUBBLE:<1 3 5 2 1 4 6>
 -=>(1 1 2 3 4 5 6)
```

SHELL does a Shell sort on a list of numbers. "The basic idea of the Shell sort is that in the early stages far-apart elements are compared instead of adjacent ones . . . Gradually, the interval between compared elements is decreased, until it reaches one at which point it effectively becomes an adjacent interchange method." [106]

```
DEFINE SHELL L
    SHELL1:<CHARCT:L DIV:<CHARCT:L 2> L>
.
-=>SHELL

DEFINE SHELL1 (N1 N2 L)
    IF ZEROP:N2 THEN L
    ELSE SHELL1:<N1 DIV:<N2 2> SHELL2:<N1 ADD1:N2 N2 L>>
.
-=>SHELL1
```



```
DEFINE SHELL2 (N1 N2 N3 L)
    IF GREAT:<N2 N1> THEN L
    ELSE SHELL2:<N1 ADD1:N2 N3 SHELL3:<DIFF:<N2 N3> N3 L>>
.
-=>SHELL2

DEFINE SHELL3 (N1 N2 L)
    IF LESSEQ:<N1 0> THEN L
    ELSEIF CLESSEQ:<PROJ:<N1 L> PROJ:<PLUS:<N1 N2> L>> THEN L
    ELSE SHELL3:<DIFF:<N1 N2> N2 EXCHANGE:<L N1 N2>>
.
-=>SHELL3

SHELL:<3 6 2 8 1 3>
 -=>(1 2 3 3 6 8)

SHELL:<<3 6><2 8><1><1 3>>.
 -=> ((1) (1 3) (2 8) (3 6))
```

SHELL uses EXCHANGE to exchange two elements in a list. The functions EXCHANG1 and SWITCH work together to have an effect similar to that of ORDER, a help function to BUBBLE; ELT is placed in the N<u>th</u> position of a list L, and the N<u>th</u> element of L is shuffled to the front of the list via N interchanges of adjacent elements of L.

```
DEFINE EXCHANGE (L N1 N2)
    IF ONEP:N1 THEN EXCHANG1:<FIRST:L N2 REST:L>
    ELSE CONS:<FIRST:L EXCHANGE:<REST:L SUB1:N1 N2>>
.
-=>EXCHANGE

DEFINE EXCHANG1 (ELT N L)
    IF ONEP:N THEN SWITCH:<ELT L>
    ELSE SWITCH:<FIRST:L EXCHANG1:<ELT SUB1:N REST:L>>
.
-=>EXCHANG1

DEFINE SWITCH (ELT LIST)
    CONS:<FIRST:LIST CONS:<ELT REST:LIST>>
.
-=>SWITCH
```

The function CLESSEQ can compare either two numbers or two lists of numbers. It returns TRUE if the first number (list) is less than or equal to the second.

Lists of numbers are compared by comparing the elements of each list one by one, from left to right. If somewhere along the line an element from one list is greater than the corresponding element from another list, we say that the first list is greater than the second list. If the first list is of length N and the second list is of length greater than N, and the two lists are equal through their first N elements, then the second list is greater than the first.

```
DEFINE CLESSEQ (A B)
    OR:<CLESS:<A B> CEQ:<A B>>
.
-=>CLESSEQ

DEFINE CEQ (A  B)
    IF NOT:ATOM:A THEN EQSTR:<A B>
    ELSE SAME:<A B>
.
-=>CEQ

DEFINE CLESS (A B)
    IF NOT:ATOM:A THEN LLESS:<A B>
    ELSE LESS:<A B>
.
-=>CLESS

DEFINE LLESS (A B)
    IF NULL:A THEN B
    ELSEIF NULL:B THEN FALSE
    ELSEIF LESS:<FIRST:A FIRST:B> THEN TRUE
    ELSEIF GREAT:<FIRST:A FIRST:B> THEN FALSE
    ELSE LLESS:<REST:A REST:B>
.
-=>LLESS
```


Due to the versatility of the function CLESSEQ, a slight variation of SHELL, LSHELL, may be used to lexically order lines of text in a file. First, a function CONVERT is called to convert the file into a list of lists, with each containing a line of text. CONVERT also maps the characters of each of the lines into integers, according to some convenient standard. Then SHELL, using CLESSEQ to compare lines of the file, arranges these lists of integers into ascending order. Finally REVERT converts the integers in the lists back into characters and the lists are merged into one file.

```
DEFINE LSHELL TEXT
    REVERT:SHELL:CONVERT:TEXT
.
-=>LSHELL

DEFINE CONVERT L
    ALCHRVL:SEPLINES:L
.
-=>CONVERT

DEFINE ALCHRVL L
    <LINECONV*>:<L>
.
-=>ALCHRVL

DEFINE LINECONV (L)
    <CHRVAL*>:<L>
.
-=>LINECONV

DEFINE CHRVAL (AT)
    SUB1:INDEX:<AT ALPHANUM>
.
-=>CHRVAL

DEFINE SEPLINES TEXT
    IF NULL:TEXT THEN <>
    ELSE CONS:<1:GETLINE:TEXT SEPLINES:2:GETLINE:TEXT>
.
-=>SEPLINES
```

```
DEFINE REVERT L
    CONCAT:ALASCII:L
.
-=>REVERT

DEFINE ALASCII L
    <LASCII*>:<L>
.
-=>ALASCII

DEFINE LASCII (L)
    <ASCII*>:<L>
.
-=>LASCII

DEFINE ASCII (N)
    PROJ:<ADD1:N ALPHANUM>
.
-=>ASCII
```

```
LSHELL:"(C D NL E NL A D C NL A D B NL).
 -=> (A D B NL A D C NL C D NL E NL)
```


SORTINT, an "in memory" sorting algorithm, uses LSHELL to sort files whose length is less than some integer constant, MAXFILES.

```
DEFINE SORTINT TEXT
    IF GREAT:<CHARCT:TEXT MAXFILES>
         THEN "(NL T O O BLANK B I G NL)
    ELSE LSHELL:TEXT
.
-=>SORTINT


SORTINT:"(A B C D E F G H I).
 -=> (NL T O O BLANK B I G NL)

SORTINT:"(A B NL A A NL)>
 -=> (A A NL A B NL)
```

QUICK is a quicksort algorithm. Like SHELL, it can sort a list of integers or a list of lists of integers. Until the list is empty, each invocation of QUICK calls LTEQGT to divide the list into three lists, the first containing all elements of the original list which are less than some N, the second containing all elements equal to N, and the third containing all elements greater than N.

```
DEFINE QUICK (L)
    IF NULL:L THEN <>
    ELSE CONCAT:<QUICK       1     QUICK>:<
                 LTEQGT:<FIRST:L REST:L>>
.
-=>QUICK
```

```
DEFINE LTEQGT (N L)
    IF NULL:L THEN <<> <N> <>>
    ELSEIF CLESS:<N FIRST:L>
         THEN <1      1      CONS>:<
              <#      #   FIRST:L>
               LTEQGT:<N REST:L>>
    ELSEIF CEQ:<N FIRST:L>
         THEN <1      CONS      1>:<
              <#     FIRST:L    #>
               LTEQGT:<N REST:L>>
    ELSE <CONS      1      1>:<
         <FIRST:L   #      #>
          LTEQGT:<N REST:L>>

-=>LTEQGT

QUICK:<<8 3 9 3 6 2 4>>.
 -=> (2 3 3 4 6 8 9)

QUICK:<<<8 3><9 3><2 4>>>.
 -=> ((2 4) (8 3) (9 3))

LTEQGT:<3 <10 9 8 7 7 3 3 2 1 8 4>>.
 -=> ((2 1) (3 3 3) (10 9 8 7 7 8 4))
```

LQUICK is to QUICK as LSHELL is to SHELL. LQUICK uses QUICK to lexically sort lines of text within a file.

```
DEFINE LQUICK TEXT
    REVERT:QUICK:<CONVERT:TEXT>

-=>LQUICK

LQUICK:"(C Z NL A B NL Q V 2 NL).
 -=> (A B NL C Z NL Q V 2 NL)
```


SORTX uses a quicksort algorithm to sort a very large file --ie. a file which is too large to fit into memory in one piece. "The essential idea of most external sorting methods is simple: chunks of the input (as big as possible) are sorted internally and copied onto intermediate files; each chunk is called a run. When the entire input has been split into sorted runs the runs are merged, typically into further intermediate files. Eventually all the data winds up merged on one file; this final run is the sorted output." [116]

```
DEFINE SORTX FILE
    MERGEGP:QUICKRNS:FILE

-=>SORTX

SORTX:"(Q R V NL S T A NL A B NL C D NL).
 -=> (A B NL C D NL Q R V NL S T A NL)
```

QUICKRNS is used to divide a file into sorted 'runs'. GTEXT is used to divide a file into chunks of length less than or equal to some integer MAXCHARS. Each of these chunks is then CONVERTed to lists of integers which are sorted by QUICK.

```
DEFINE QUICKRNS FILE
    IF NULL:FILE THEN <>
    ELSE CONS:<QUICK:<CONVERT:1:GTEXT:FILE> QUICKRNS:2:GTEXT:FILE>
.
-=>QUICKRNS

DEFINE GTEXT TEXT
    GTEXT1:<0 TEXT>
.
-=GTEXT

DEFINE GTEXT1 (LEN TEXT)
    IF NULL:TEXT THEN <<> <>>
    ELSE ADDLINE:<PLUS:<LEN LINELEN:TEXT> TEXT>
.
-=>GTEXT1

DEFINE LINELEN TEXT
    CHARCT:1:GETLINE:TEXT
.
-=>LINELEN

DEFINE ADDLINE (LEN TEXT)
    IF GREAT:<LEN MAXCHARS> THEN <<> TEXT>
    ELSE <APPEND          1>:<
         <1:GETLINE:TEXT  #>
          GTEXT1:<LEN 2:GETLINE:TEXT>>
.
-=>ADDLINE

QUICKRNS:"(Q R V NL S T A NL A B NL C D NL).
-=> (((26 27 31 36) (28 29 10 36)) ((10 11 36) (12 13 36)))
```


MERGEGP handles the merging of the lists created by QUICKRNS. When all the lists have been merged into one list, the contents of this list are converted back to their original form by REVERT--that is, the list of lists of numbers is changed back to a text file.

```
DEFINE MERGEGP LFILES
    IF NULL:LFILES THEN <>
    ELSEIF NULL:REST:LFILES THEN REVERT:FIRST:LFILES
    ELSE MERGEGP:MRGCHNKS:LFILES
.
-=>MERGEGP
```

MRGCHNKS is a progress function for MERGEGP, reducing the number of lists in LFILES each time it is called.

```
DEFINE MRGCHNKS LFILES
    IF NULL:LFILES THEN <>
    ELSE CONS:<MERGE:1:GETSOME:<0 LFILES>
               MRGCHNKS:2:GETSOME:<0 LFILES>>
.
-=>MRGCHNKS

DEFINE GETSOME (N LFILES)
    IF NULL:LFILES THEN <<><>>
    ELSEIF SAME:<N FLIMIT> THEN <<> LFILES>
    ELSE <CONS          1>:<
         <FIRST:LFILES  #>
          GETSOME:<ADD1:N REST:LFILES>>
.
-=>GETSOME

MRGCHNKS:<<<26 27 31 36><28 29 10 36>>
         <<10 11 36><12 13 36>>>.
-=> (((10 11 36) (12 13 36) (26 27 31 36) (28 29 10 36)))
```

FLIMIT is an integer constant indicating the maximum number of files which GETSOME may gather into one file.

MERGE does a merge sort on a list of sorted, CONVERTed files. The algorithm for merge is very similar to the algorithm for BUBBLE. The function MERGE corresponds to the function BUBBLE, the function LORDER corresponds to the function ORDER, and LOLINE corresponds to SINKSMAL.

```
DEFINE MERGE LFILES
   IF NULL:LFILES THEN <>
   ELSE CONS:<FIRST:FIRST:LOLINE:LFILES
              MERGE:TCONS1:<REST:FIRST:LOLINE:LFILES
                           REST:LOLINE:LFILES>>
 -=>MERGE

DEFINE LOLINE LFILES
    IF NULL:REST:LFILES THEN LFILES
    ELSE LORDER:<FIRST:LFILES LOLINE:REST:LFILES>
.
 -=>LOLINE

DEFINE LORDER (FILE LFILES).
    IF LLESS:<FIRST:FILE FIRST:FIRST:LFILES>
        THEN CONS:<FILE LFILES>
    ELSE CONS:<FIRST:LFILES CONS:<FILE REST:LFILES>>
.
 -=>LORDER

DEFINE TCONS1 (A L)
   IF NULL:A THEN L
   ELSE CONS:<A L>
.
 -=>TCONS1

MERGE:<<<10 13><23 44>><<2 5><3>><<13 99 77><44>>>.
 -=> ((2 5) (3) (10 13) (13 99 77) (23 44) (44))

MERGE:<<<10 13 3><14 2>><<2 7><3 5><8 9><12>>>
 -=> ((2 7) (3 5) (8 9) (10 13 3) (12) (14 2))
```


UNIQUE is a function which can be useful if used in combination with a sort. It compares adjacent lines in a text file and removes duplicate lines. Applied to a sorted file, UNIQUE will remove all duplicate lines from that file.

```
DEFINE UNIQUE FILE
    IF NULL:FILE THEN <>
    ELSE CKPAIR:GETLINE:FILE
.
 -=>UNIQUE


DEFINE CKPAIR (F1 F2)
    IF EQLINE:<F1 F2> THEN UNIQUE:F2
    ELSE APPEND:<F1 UNIQUE:F2>
.
 -=>CKPAIR

DEFINE EQLINE (LINE TEXT)
    EQSTR:<LINE 1:GETLINE:TEXT>
.
 -=>EQLINE


UNIQUE:"(A R N O L D NL W I L L I E NL W I L L I E NL
      H E A T H E R NL).
 -=> (A R N O L D NL W I L L I E NL H E A T H E R NL)

UNIQUE:"(T H R E E NL T H R E E NL T H R E E NL).
 -=> (T H R E E NL)
```

The function KWIC (Key Word In Context) is used to create a permuted index for a text file. This index is created by KWIC in the following manner. ALLROT is called to apply ROTLINE to each line of an input file, FILE. For a line of N words, ROTLINE creates N lines of text, with each line beginning with a different word of the original line. The lines are created by rotating the original line, one word at a time, as in a circular shift.

After all the lines have been rotated, LQUICK is called to sort the lines of text. Finally, ALUNROT is called to rotate each line so that its first word begins at the center of a page (were the line to be printed out). For example, the line

I lead and you follow you lead and I follow

becomes

you lead and I follow I lead and you follow .

```
DEFINE KWIC TEXT
    ALUNROT:ALASCII:QUICK:<ALCHRVAL:REMEOLNS:ALLROT:TEXT>
.
-=>KWIC

DEFINE ALLROT TEXT
    CONCAT:<ROTLINE*>:<
            SEPLINES:TEXT>
.
-=>ALLROT

DEFINE ROTLINE (LINE)
    CKNULL:TRIM:LINE
.
-=>ROTLINE

ALLROT:"(A BLANK B NL D BLANK E).
 -=> ((B NL FOLD A BLANK) (A BLANK B NL) (E FOLD D BLANK) (D BLANK E))

ROTLINE:<"(A BLANK B BLANK C NL)>.
 -=> ((B BLANK C NL FOLD A BLANK) (C NL FOLD A BLANK B BLANK) (A BLANK B
  BLANK C NL))
```


TRIM is called by ROTLINE to remove non-alfa characters from the front of a line. (An alfa character is a letter or a digit). These just get in the way and are unnecessary to our permuted index. CKNULL then weeds out empty lines. A special FOLD character is added to the end of each line before it is rotated. This marks the end of a line for a line with no NL character.

```
DEFINE TRIM LINE
    IF OR:<NULL:LINE ALFA:FIRST:LINE> THEN LINE
    ELSE TRIM:REST:LINE
.
-=>TRIM

DEFINE CKNULL LINE
    IF NULL:LINE THEN <>
    ELSE ROTLINE1:ROTATE:SNOC:<FOLD LINE>
.
-=>CKNULL
```

ROTLINE1 is the function which takes a line of text and creates from that line a list of ROTATEd lines. ROTATE takes the first word from the front of a line and tacks it to the end. ROTLINE1 terminates when a FOLD or NL character is encountered as the first character in the line.

```
DEFINE ROTLINE1 LINE
    IF NLQ:FIRST:LINE THEN <RREST:LINE>
    ELSEIF FOLDQ:FIRST:LINE THEN <REST:LINE>
    ELSE CONS:<LINE ROTLINE1:ROTATE:LINE>
.
 -=>ROTLINE1

DEFINE ROTATE LINE
    ROTBLNKS:ROTWORD:LINE
.
 -=>ROTATE

ROTATE:"(F I R S T NL S E C O N D BLANK T H I R D BLANK).
 -=> (T H I R D BLANK F I R S T NL S E C O N D BLANK)

DEFINE ROTWORD LINE
    IF NOT:ALFA:FIRST:LINE THEN LINE
    ELSE ROTWORD:SNOC:<FIRST:LINE REST:LINE>
.
 -=>ROTWORD

DEFINE ALFA CHAR
    AND:<POS:INDEX:<CHAR ALPHANUM>
         LESS:<INDEX:<CHAR ALPHANUM> TOPALFA>
.
 -=>ALFA

DEFINE FOLDQ CHAR
    SAME:<CHAR FOLD>
.
 -=>FOLDQ

DEFINE ROTBLNKS TEXT
    IF OR:<NLQ:FIRST:TEXT FOLDQ:FIRST:TEXT ALFA:FIRST:TEXT>
        THEN TEXT
    ELSE ROTBLNKS:SNOC:<FIRST:TEXT REST:TEXT>
.
 -=>ROTBLNKS

DEFINE SNOC (A L)
    IF NULL:L THEN <A>
    ELSE CONS:<FIRST:L SNOC:<A REST:L>>
.
 -=>SNOC
```


After all the lines have been rotated, REMEOLNS is called to remove from each line the characters used to mark the end of the line. These characters would get in the way of our sort and would clutter our permuted index; for example, after rotation, a NL character would be in the middle of a "line".

```
DEFINE REMEOLNS TEXT
    <CLEANLIN*>:<
     TEXT>
.
 -=>REMEOLNS

DEFINE CLEANLIN (LINE)
    IF NLQ:FIRST:LINE THEN CKADD:RREST:LINE
    ELSEIF FOLDQ:FIRST:LINE THEN CKADD:REST:LINE
    ELSE CONS:<FIRST:LINE CLEANLIN:<REST:LINE>>
.
 -=>CLEANLIN

 DEFINE CKADD L
     IF NULL:L THEN <>
     ELSEIF BLANKQ:FIRST:L THEN L
     ELSE CONS:<BLANK L>
.
  -=>CKADD

REMEOLNS:"((A B NL FOLD C)(A B C NL FOLD)(A B NL FOLD BLANK C)).
 -=> ((A B BLANK C) (A B C) (A B BLANK C))
```

Finally, ALUNROT "unrotates" all the lines, so that the first word of each line is at its center. SETRIGHT fills the right side of a line with as many words from a rotated line as is possible ($\leq$ HALFLINE characters, where HALFLINE is the length of half a printed line of text). It is assumed that no one word will contain more than HALFLINE characters. A NL character is added to the end of the line.

```
    DEFINE ALUNROT TEXT
        CONCAT:<UNROT*>:<
                TEXT>
    .
     -=>ALUNROT

ALUNROT:"((B A D BLANK A BLANK) (A BLANK B O B) ).
 -=> (BLANK BLANK A BLANK BLANK B A D BLANK NL BLANK B O B BLANK A BLANK
  BLANK BLANK NL)

    DEFINE UNROT (LINE)
        SETLEFT:SETRIGHT:<HALFLINE LINE>
    .
     -=>UNROT

    DEFINE SETRIGHT (N LINE)
        IF NULL:LINE THEN <<> SNOC:<NL FILBLNKS:N>>
        ELSE ADCHNKS:<N LINE NEXCHNK:<N LINE>>
    .
     -=>SETRIGHT

    UNROT:<"(B A D BLANK A BLANK)>.
     -=> (BLANK BLANK A BLANK BLANK B A D BLANK NL)

    UNROT:<"(B I T)>.
     -=> (BLANK BLANK BLANK BLANK BLANK B I T BLANK NL)
```


SETRIGHT adds strings of characters to the output a word at a time. ADCHUNKS and NEXCHNK attempt to add one word to the output. If the value LEFTOVER, returned by NEXCHNK is negative, then the next word of input won't fit within the boundaries of the current line of output; so the line minus that word (CHUNK) is output. FILBLNKS pads the line with blanks.

```
DEFINE ADCHNKS (N LINE (LEFTOVER CHUNK RESTLINE))
    IF MINUSP:LEFTOVER
        THEN <LINE SNOC:<NL FILBLNKS:N>>
    ELSE <1       APPEND>:<
         <#        CHUNK>
          SETRIGHT:<LEFTOVER RESTLINE>>
.
 -=>ADCHNKS

DEFINE NEXCHNK (N LINE)
    IF NULL:LINE THEN <N <> <>>
    ELSEIF ALFA:FIRST:LINE
        THEN <1          CONS       1>:<
             <#      FIRST:LINE    #>
              NEXCHNK:<SUB1:N REST:LINE>>
    ELSE ADBLNKS:<N LINE>
.
 -=>NEXCHNK


DEFINE FILBLNKS N
    IF ZEROP:N THEN <>
    ELSE CONS:<BLANK FILBLNKS:SUB1:N>
.
 -=>FILBLNKS
```

NEXCHNK is designed to output a word plus as many of its following whitespace characters as is possible. The function ADBLNKS puts out the trailing whitespace characters until 1) the maximum line length has been output; or 2) there are no more characters in the line or 3) an alfa character is encountered.

```
DEFINE ADBLNKS (N LINE)
    IF OR:<NULL:LINE NOT:POS:N ALFA:FIRST:LINE>
        THEN <N <> LINE>
    ELSE <1      CONS      1>:<
         <#  FIRST:LINE  #>
          ADBLNKS:<SUB1:N REST:LINE>>
.
-=>ADBLNKS


DEFINE POS N
    GREAT:<N 0>
.
-=>POS
```

SETLEFT outputs the left half of a line of text, including whatever is left of the line after it has been processed by SETRIGHT. An extra blank is added between the two halves of the line.

```
DEFINE SETLEFT (LEFT RIGHT)
    CONCAT:<FILBLNKS:DIFF:<HALFLINE CHARCT:LEFT> LEFT
           CONS:<BLANK RIGHT>>
.
-=>SETLEFT
```


Constants Introduced in Chapter 4.

```
DECLARE ALPHANUM
    "(0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T
      U V W X Y Z NL FOLD).
-=> (0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T
      U V W X Y Z NL FOLD).

DECLARE MAXFILES 6.
-=>6

DECLARE MAXCHARS 8.
-=>8

DECLARE FLIMIT 2.
-=>2

DECLARE FOLD "FOLD.
-=>FOLD

DECLARE HALFLINE 4.
-=>4

DECLARE TOPALFA 38.
-=>38
```

CHAPTER 5.

In this chapter, some relatively simple functions, which will be used as tools in the chapters to follow, are presented. The functions deal with the problem of matching, isolating and manipulating strings of characters in a text file, a task for which this language is very well suited.

The function FIND and its help functions are used for locating and matching patterns in a file. The following rules define valid "text patterns" used by FIND for pattern matching.

1) "Any literal character, like 'A' is a text pattern that matches the same character in the text being scanned." [136]
2) The character '?', represented by the symbolic constant ANY, is "a text pattern that matches any single character except a newline". [137]
3) A list within a pattern "forms a 'character class', that is, a text pattern that matches any character from the . . . list." [137] A list which has as its first character, NCCL, matches any character not contained in the rest of the list. (An exception to this is that a negated character class will not match a newline.) The contents of a character class may be expressed using the same shorthand as was used in the function XLATE (see ch. 2), eg. 'A-Z' or '0-9'. The function MAKESET is called to expand the notation.
4) By preceding any of the above mentioned

text patterns with the character, CLOS, we end up with a text pattern which matches "zero or more successive occurrences of the single character pattern. FIND matches the longest possible string even when a null string match would be equally valid." CLOS (A-Za-z) ". . . matches an entire word (which may be a null string) and . . ." CLOS ANY " . . . matches a whole line (which may be a null string)." [137]

5) The special character BOL is a text pattern which matches the null string at the beginning of a line. BOL, however, only has special meaning if it occurs at the beginning of a scan pattern; otherwise it is taken literally.
6) The special character EOLN is a text pattern which matches the null string at the end of a line (before a newline). If the EOLN character does not occur at the end of a scan pattern, it loses its special meaning.
7) "A text pattern followed by another text pattern forms a new text pattern that matches anything matched by the first followed by anything matched by the second." [136]

"Any ambiguity in deciding which part of a line matches a pattern will be resolved by choosing the match beginning with the leftmost character, then choosing the longest possible match at that point." [137]

The function FIND takes as arguments a description of a pattern, PAT, a list of character sets, LSETS, and a file, TEXT. A scan pattern is built from PAT, and TEXT is searched for a line matching the scan pattern.

The function FILSET is used to build the scan pattern from PAT using a function which was introduced in chapter 2, MAKESET.

```
DEFINE FIND (LSETS PAT TEXT)
    FINDX:<FILSET:<LSETS PAT> TEXT>
.
-=>FIND

DEFINE FILSET (LSETS STR)
    IF NULL:STR THEN STR
    ELSEIF ATOM:FIRST:STR
         THEN CONS:<FIRST:STR FILSET:<LSETS REST:STR>>
    ELSE CONS:<MAKESET:<LSETS FIRST:STR>
               FILSET:<LSETS REST:STR>>
.
-=>FILSET

FIND:<EXSETS "(CLOS ANY EOLN) "(Y O U NL N A M E NL I T)>.
 -=> (() (Y O U NL N A M E NL I T))

FIND:<EXSETS "(Q R V) "(N O T NL I N S A N E)>.
 -=> ((N O T NL I N S A N E) ())

FILSET:<EXSETS "(C (A DASH E) T)>.
 -=> (C (A B C D E) T)

FILSET:<EXSETS "(C (M DASH E) T)>.
 -=> (C (M DASH E) T)

FIND:<EXSETS "(C (A DASH E) T)
             "(C U T NL C A T NL R E S T NL)>.
 -=> ((C U T NL) (C A T NL R E S T NL))
```


The help function FINDX looks through TEXT line by line, calling the function MATCH to see if the current line contains a string which matches the scan pattern, PAT. FINDX returns two files as output, one containing all lines in TEXT which precede the one which contains the matching string, and the other containing the sought for line, and all following lines.

The definition of FINDX is similar in form to that of its help function, GETLINE, which I will write down again here to refresh your memory. The only differences are 1) that FINDX searches a line at a time while GETLINE searches a character at a time, and 2) the final element scanned by FINDX is placed in its second output file while the final element scanned by GETLINE goes in its first output file.

```
DEFINE FINDX (PAT TEXT)
    IF NULL:TEXT THEN <<><>>
    ELSEIF MATCH:<PAT 1:GETLINE:TEXT>
         THEN <<> TEXT>
    ELSE <APPEND                1>:<
         <1:GETLINE:TEXT        #>
          FINDX:<PAT 2:GETLINE:TEXT>>
.
-=>FINDX

DEFINE GETLINE TEXT
   IF NULL:TEXT THEN <<><>>
   ELSEIF NLQ:FIRST:TEXT THEN <<NL> REST:TEXT>
   ELSE <CONS                1>:<
        <FIRST:TEXT          #>
         GETLINE:REST:TEXT>
.
-=>GETLINE
```

The function MATCH searches through a line, character by character, calling AMATCH to see if the scan pattern PAT matches a string beginning with the character in the line which is currently being scanned. If the first character in PAT is BOL, then MATCH returns TRUE only if PAT matches a string which begins at the start of a line.

```
DEFINE MATCH (PAT LINE)
    IF NULL:PAT THEN TRUE
    ELSEIF BOLQ:FIRST:PAT THEN AMATCH:<REST:PAT LINE>
    ELSE MATCH1:<PAT LINE>
.
-=>MATCH

DEFINE MATCH1 (PAT LINE)
    IF NULL:LINE THEN FALSE
    ELSEIF AMATCH:<PAT LINE> THEN TRUE
    ELSE MATCH1:<PAT REST:LINE>
.
-=>MATCH1

DEFINE BOLQ CHAR
    SAME:<CHAR BOL>
.
-=>BOLQ


MATCH:"((BOL S T A R T) (N O S T A R T NL)).
 -=>()

MATCH:"((BOL S T A R T) (S T A R T O K NL)).
 -=>TRUE

MATCH:"((A B BOL C) (A B BOL C)).
 -=>TRUE
```


MATCH and MATCH1 have separate definitions because of the condition that a BOL has special meaning only at the beginning of a scan pattern. So, after it has been tested for once in MATCH, MATCH1 is called to avoid testing for it again.

The logic of MATCH1 is like that of MEMBER (Chapter 2). Given a typical element of a list, search the list from left to right until the list is empty or the element matches the first element in the list.

The function AMATCHX returns three results. The first is a list consisting of the portion of a scan pattern PAT for which no matching characters were found in a LINE. The second result contains the portion of the LINE which matched PAT. The third result contains the rest of the line.

The function AMATCH checks the first result of AMATCHX to see if the match succeeded. The match has succeeded if this first result of AMATCHX is an empty list. (Though we don't need all the results returned by AMATCHX now, we will need them in functions to come).

```
DEFINE AMATCH (PAT LINE)
    NULL:1:AMATCHX:<PAT LINE>
.
-=>AMATCH

DEFINE AMATCHX (PAT LINE)
    IF NULL:PAT THEN <PAT <> LINE>
    ELSEIF CLOSQ:FIRST:PAT
         THEN CLOSMACH:<PAT LINE>
    ELSEIF NOT:OMATCH:<FIRST:PAT LINE>
         THEN <PAT <> LINE>
    ELSE <1        CONS       1>:<
         <#     FIRST:LINE    #>
          AMATCHX:<REST:PAT REST:LINE>>
.
-=>AMATCHX
```

```
AMATCHX:<<> "(L I N E 1 NL)>.
 -=> (() () (L I N E 1 NL))

AMATCHX:"((G O N E) (L O N G G O N E)).
 -=> ((G O N E) () (L O N G G O N E))

AMATCHX:"((G O N E) (G O N E L O N G)).
 -=> (() (G O N E) (L O N G))

AMATCHX:"((CLOS A B) (B)).
 -=> (() (B) ())

AMATCHX:"((CLOS A B) (A B)).
 -=> (() (A B) ())

AMATCHX:"((CLOS A B) (A A A A B)).
 -=> (() (A A A A B) ())

AMATCHX:"((CLOS ANY) (A B NL)).
 -=> (() (A B) (NL))

AMATCHX:"((CLOS ANY EOLN) (A B NL)).
 -=> (() (A B NL) ())
```

CLOSMACH is called by AMATCH to find the longest possible string which will match a pattern whose first literal character (or character class) is preceded by the special character CLOS.

```
DEFINE CLOSMACH (PAT LINE)
    IF OMATCH:<2:PAT LINE>
         THEN CKREST:<PAT LINE AMATCHX:<PAT REST:LINE>>
    ELSE AMATCHX:<RREST:PAT LINE>
.
 -=>CLOSMACH

DEFINE CKREST (PAT LINE ANS)
    IF NULL:FIRST:ANS
         THEN <1        CONS        1>:<
              <#     FIRST:LINE     #>
               ANS>
    ELSE AMATCHX:<RREST:PAT LINE>
.
 -=>CKREST
```


OMATCH returns TRUE if the first literal character (or character class) in the scan pattern PAT matches the first character in the file TEXT. OMATCHAT is called to match a character, and OMATCHL is called to match a character class.

Note that it is possible for EOLN to match the null string before the end of a line, even though it may not be at the end of a scan pattern. This would seem to contradict rule #6 in the definition of regular expressions; however, if the match occurs and there is still more of the scan pattern left to be matched, then AMATCH will fail when it attempts to continue matching the scan pattern.

```
DEFINE OMATCH (PAT TEXT)
    IF NULL:TEXT THEN FALSE
    ELSEIF NLQ:FIRST:TEXT THEN EOLNQ:PAT
    ELSEIF ATOM:PAT THEN OMATCHAT:<PAT TEXT>
    ELSE OMATCHL:<PAT TEXT>
.
 -=>OMATCH

DEFINE EOLNQ PAT
    .AND:<ATOM:PAT SAME:<PAT EOLN>>
.
 -=>EOLNQ

DEFINE CLOSQ CHAR
    SAME:<CHAR CLOS>
.
 -=>CLOSQ

DEFINE OMATCHAT (PAT LINE)
    OR:<ANYQ:PAT SAME:<FIRST:LINE PAT>>
.
 -=>OMATCHAT

DEFINE ANYQ CHAR
    SAME:<CHAR ANY>
.
 -=>ANYQ
```

```
DEFINE OMATCHL (PAT LINE)
   IF NULL:PAT THEN FALSE
   ELSEIF NCCLQ:FIRST:PAT
        THEN NOT:MEMBER:<FIRST:LINE REST:PAT>
   ELSE MEMBER:<FIRST:LINE PAT>
.
-=>OMATCHL


OMATCH:<EOLN "(NL)>.
 -=>TRUE

OMATCH:"(A (A B C D NL)).
 -=>TRUE

OMATCH:"((L I S T) (L A S T NL)).
 -=>TRUE
```

CHANGE searches line by line through a file, TEXT, replacing each string matched by a scan pattern, PAT, with a substitute string SUB. If SUB is null, then CHANGE deletes from TEXT all strings which match PAT. CHANGE may also be used to insert strings of characters into TEXT. To do this, a special character DITTO is used. The occurrence of the character DITTO in SUB causes a duplicate of the current string matching PAT to be inserted into TEXT.

```
DEFINE CHANGE (SUB LSETS PAT TEXT)
    CHANGE1:<SUB FILSET:<LSETS PAT> TEXT>
.
-=>CHANGE1

DEFINE CHANGE1 (SUB PAT TEXT)
    IF NULL:TEXT THEN <>
    ELSE APPEND:<ALCHANGE:<SUB PAT 1:GETLINE:TEXT>
                 CHANGE1:<SUB PAT 2:GETLINE:TEXT>>
.
-=>CHANGE1
```


The function ALCHANGE is used to replace all strings in a line which match PAT, with the substitute pattern SUB. MAKESUB is called to do each replacement.

Remember that if AMATCHX succeeds in finding a match, its second result is the string which matched the scan pattern. MAKESUB uses this result when expanding occurrences of DITTO in SUB.

```
DEFINE ALCHANGE (SUB PAT LINE)
    IF NULL:LINE THEN <>
    ELSEIF AMATCH:<PAT LINE>
         THEN APPEND:<MAKESUB:<SUB 2:AMATCHX:<PAT LINE>>
                      ALCHANGE:<SUB PAT 3:AMATCHX:<PAT LINE>>>
    ELSE CONS:<FIRST:LINE ALCHANGE:<SUB PAT REST:LINE>>
.
-=>ALCHANGE

DEFINE MAKESUB (SUB PAT)
    IF NULL:SUB THEN SUB
    ELSEIF DITTOQ:FIRST:SUB
         THEN APPEND:<PAT MAKESUB:<REST:SUB PAT>>
    ELSE CONS:<FIRST:SUB MAKESUB:<REST:SUB PAT>>
.
-=>MAKESUB

DEFINE DITTOQ CHAR
    SAME:<CHAR DITTO>
.
-=>DITTOQ

DEFINE NCCLQ CHAR
    SAME:<CHAR NCCL>
.
-=>NCCLQ

CHANGE:<<> EXSETS "(S I N G)
         "(S I N G L E NL B L E S S I N G NL T W E E T)>.
 -=> (L E NL B L E S NL T W E E T)

CHANGE:<"(DITTO DASH DITTO) EXSETS "(S I N G)
         "(I NL L O V E BLANK T O NL S I N G)>.
 -=> (I NL L O V E BLANK T O NL S I N G DASH S I N G)

CHANGE:<"(B A L L) EXSETS "(R O L L)>.
         "(B A L L BLANK B A L L NL R O L L NL)>.
 -=> (B A L L BLANK B A L L NL B A L L NL)
```

Constants Introduced in Chapter 5.

```
DECLARE EXSETS
    "((A B C D E F G H I J K L M N O P Q R S T
       U V W X Y Z)).
 -=>((A B C D E F G H I J K L M N O P Q R S T U V W X Y Z))


      DECLARE BOL "BOL.
       -=>BOL

      DECLARE CLOS "CLOS.
       -=>CLOS

      DECLARE EOLN "EOLN.
       -=>EOLN

      DECLARE ANY "ANY.
       -=>ANY

      DECLARE DITTO "DITTO.
       -=>DITTO

      DECLARE NCCL "NCCL.
       -=>NCCL
```


CHAPTER 6

Introduction

In this chapter a text editor is discussed. The editor is primarily "line oriented", though provisions have been made for character manipulations as well.

Given a file name, FNAME, and a file library, LFILES, as input, the editor creates a working text file (also sometimes referred to as the edit or editor file) either by opening file FNAME from the file library or by creating a new file. Commands are then read from a command file, COMS. Usually, these commands involve manipulations of the text file; the specific actions of the commands will be discussed in some detail shortly.

Changes in the text file may be "permanently" recorded by writing portions of the file out to the file library. LFILES may undergo many changes throughout the editing session. The current copy of LFILES when the editing session ends becomes one of the results of the editor.

The other output file created by the editor is a 'message' file which may contain echoed commands, error messages, and/or portions of the text file.

This editor is suitable for interactive use on a full duplex teletype where the command file consists of an input stream from the teletype keyboard, and the message file produces

output for the teletype printer.

An edit command specifies an 'action' which is to take place, and, optionally, a segment of the text file over which that action is to take place (I shall call this segment the 'range' of a command).

Specifying the Range of a Command.

1) line numbers: A user may specify the range of a command by referring to parts of the text file by line number; for example, the command

(((7)) P)

causes line seven of the text file to be printed out (the 'P' is a print command). The following command

(((3), (7)) P)

causes lines three through seven of the text file to be printed out.

The line numbers are relative to what is currently the first line of the file; if, say, the first three lines are deleted from a file, then what used to be line 4 is now line 1.

2) relative indicators of position:

a. A current line pointer is kept to point at the line in the text file most recently affected by or inspected by the editor. Initially the first line of the text file

is the current line.

In specifying the range of a command, one may use the symbol '.' (CURLN) to refer to the current line. For example

(((.)) P)

causes the current line to printed out.

b. The last line of the text file may be referred to by using the symbol, '$'; so,

(($)) P)

causes the last line of the file to printed out and

(((1),($)) P)

causes the whole text file to be printed out.

3) One may also specify the range of a command by using a context search. The command

(((>(A B C))) P)

causes the editor to look forward (indicated by '>') through the text file, beginning at the line after the current line.

Searches may also go backward (indicated by '<') as well as forward.

A context search may 'wrap around' the end of the text file; it only fails if the search has gone full circle from the line after the current line up to the current line. So the command

(((<(X Y Z)),(>(B O O))) P)

prints out all lines of the text file from the first line

preceding the current line which contains the pattern 'X Y Z' to the first line following the current line which contains the pattern 'B O O'.

4) To enhance the power of the above methods of specifying the range of a command, one may also use, in combination with each of these techniques, line number arithmetic.

For example, the command

(((.-5),(.+5)) P)

causes a section of text beginning five lines before the current line and ending five lines after the current line to be printed out.

The command

(($+5)) P)

is illegal, and results in an error message. The arithmetic portion of a command (in this case '+5') may not cause a wrap around the end (or beginning) of the text file.

5) "A semicolon may be used to separate line numbers just as a comma does, but it has the additional effect of setting dot (the current line) to the latest line number before evaluating the next argument." [170] For example,

(((>(A B C));(.+1)) P)

"scans forward to the next line containing 'A B C', then prints that line and the one following it (.+1)." [170] (I have

substituted our syntax for that of Kernighan and Plauger within this and following quotations.

There may be arbitrarily many components in a range specification, "so long as the last one or two are legal for the particular command." [171]

(((>(A B C));(>(A B C));(>(A B C))) P)

prints from the second line following the current line and containing 'A B C' to the third.

6) If no range specifications precede an 'action', the action is usually interpreted to apply to the current line.

NOTE: There is a special line in an editor file which marks the beginning of the file. This line contains only the special character, END, followed by a newline (NL). This line will be referred to as line zero.

It is illegal to print or delete line zero. One may not specify line zero in a range specification. Line zero is the current line only if it is the only line in the file (the file is 'empty').

ACTIONS.

1) A range specification without any following action specifications causes the last line indicated in the range specification to be printed out. The current line is set to

that line, so

((> (A B C));(> (A B C)))

causes the second line containing 'A B C' following the current line to be printed out, and the current line is reset.

A null command, with neither a range specification nor an action specification, causes the current line to be advanced one line and the new current line is printed out. If the current line is the last line of the file, then a null command results in an error message, and the current line is unchanged. (Remember, line zero may not be printed).

2) We have already seen examples of the print command, 'P'. A print command may be preceded by zero range components (in which case the current line is printed) or one component (the line indicated by that component is printed) or more than one component (the range of lines specified by the last two components is printed). The current line is set to the last line printed by the print command.

If line zero is within the range of lines to be printed, an error message is output and the current line is unchanged.

3) The symbol '=' is used to cause the line number of the current line to be printed out. It may be followed by the print command to cause the current line to be printed also.

4) An append command, 'A', causes the text which follows the 'A' to be added to the text file after the last line indicated in the range specification. The current line is set to point at the last line appended. For example,

(((\$)) A (M O R E NL M O R E NL E N O U G H NL))

causes the string 'MORE NL MORE NL ENOUGH NL' to be added to the end of the file. The current line is set to point to 'ENOUGH NL', now the last line of the file.

Text may be added to an empty file by using an append command with no range specifications -- so text is added after the current line, line zero. A '.' specification may also be used.

5) The insert command, 'I', is exactly the same as the append command, except that the text following the 'I' is inserted before the last line indicated by the range specification. The current line is set to point to the last line of text inserted.

An insert, like an append, may be used to add text to an empty file, by using the command with no range specifications, or with '.' as a range specification.

6) The delete command, 'D', is used to delete from the text file all lines indicated by the range specifications. The current line indicator is set to point to the next line of the

file following the last deleted line -- unless the last line is deleted, in which case the current line becomes the line preceding the first deleted line.

A trailing 'P' causes the new current line to be printed out. Line zero may not be deleted.

(((1),(7)) DP)

causes lines one through seven of a file to be deleted, and the new current line to be printed out.

7) The format for a substitute command, 's', is

(RANGE-SPECIFICATIONS S PAT NEW)

for each line in the section of text indicated by the range specifications, the first occurrence of the pattern, PAT, is replaced by the pattern indicated by the scan pattern, NEW. A pattern is built from NEW using the function, MAKESUB, which we saw in Chapter 5.

The current line becomes the last line in which a substitution was made. If no line in the range of the command was changed, then the current line becomes the first line of the range. Example:

(((2),($)) S (H A) (B O O))

changes the first occurrence of 'B O O' in each line (from the second line of the editor file through the last) to 'H A'.

The command may be followed by a letter, 'G', which indicates that all occurrences of PAT in the range of the command

are to be changed to NEW. A trailing 'P' causes the current line to be printed out after the execution of the substitute command.

8) The change command, 'C', is used to replace one or more lines of the editor file by some new text. Example:

(((3),(7)) C (O N C E NL A G A I N NL))

causes lines three through seven to be replaced by the two lines "ONCE NL AGAIN NL".

The current line becomes the last line of the replacement text. The section of text to be changed may not contain line zero.

9) The enter command, 'E', is used to replace the text file with the contents of some new file from the file library. For example,

(E (G E O R G E))

causes the file named 'GEORGE' to be opened (in READ mode) and the editor file is replaced by the contents of 'GEORGE'. The first line of 'GEORGE' becomes the current line.

If, for any reason, the attempt to open 'GEORGE' fails, then an error message is issued and no change is made in the text file.

The file name used in initiating the editing session is remembered. Because of this, one may use the 'E' command

without an argument, and the 'remembered' file name is used by default. If the enter command has an argument, then this file name is remembered, and becomes the new, default file name.

10) The following command

(F (S U E))

causes the name 'SUE' to be printed out. This name is remembered and replaces the old, default file name.

An 'F' command without an argument prints the default file name. An 'F' command must not be preceded by any range specifications.

11) The command

(((3)) R (P E A C H E S))

causes the contents of the file 'PEACHES' to be appended after line three of the editor file. The last line of 'PEACHES' becomes the current line. (If 'PEACHES' is empty, then the current line is line three).

If the 'R' (READ) command has no argument, then the default file name is used. The contents of the file specified in the 'R' command are obtained by opening the file in READ access mode. If the attempt to open the file fails, an error message is issued, and the editor file is unaltered. If the file was successfully opened, a message is printed indicating the number of lines in the file read in.

12) The write command, 'W', is the only means by which the contents of an edited text file may be saved on the file library.

The command

(((3),(25)) W (R A S T A))

causes lines three through twenty-five of the editor file to be written to the file, 'RASTA'. If no file 'RASTA' exists in the file library, then one is created, with READWRIT access mode.

If the 'W' is preceded by no range specifications, then the default range is the whole editor file. If no file name is specified, then the default file name is used.

If the attempt to replace or create a file fails, then an error message is issued. The 'W' command has no effect upon the current line, nor does it alter the editor file. A message is printed indicating the number of lines written.

13) The quit command 'Q', causes termination of the editing session. A message is issued indicating that the session is over. The 'Q' command must not be preceded by any range specifications. This command does not save the editor file in the permanent file library.

Global Commands. "Any editor command except" Q, E and F "may be preceded by a GLOBAL PREFIX:

(G PATTERN COMMAND)

specifies that COMMAND is to be performed for each line of text which contains an instance of PATTERN. 'G', like 'W', has a default range of the whole file, but a smaller range can be given." [170]

The command

(((1),(7)) G (A B C) P)

causes all lines between and including lines one through seven which contain 'A B C' to be printed.

(G ((O F R M A T)) S (O F R M A T) (F O R M A T) G P)

instructs the editor to "find all OFRMATS, fix them and print each corrected line as a check." [170]

"There is also an 'X' command which is identical to 'G' except that is operates only on those files that do not contain the pattern (X is for 'exclude')". [170] For example

(X ((A B C)) P)

causes all lines not containing 'A B C' to be printed out.

Now, here is a semi-formal definition of the command syntax.

NOTATION USED IN DEFINITION OF SYNTAX:

1) Strings within double quote marks are descriptions, not literal definitions.

2) Strings within single quotes ARE to be taken literally.

3) OR means 'exclusive or'.

4) '=' means 'is equivalent to'

5) Braces are used to delimit one part of a definition from the rest of the definition. They are not to be taken literally.

6) Parentheses are to be taken literally.

7) Lower case strings not contained within single or double quotes are to be or have been defined in terms either of the notation of (1-6) or of other such strings.

DEFINITION OF COMMAND SYNTAX

command = () OR (range) OR (action) OR (g-range r-action)
OR (range r-action)

grange = global range

global = 'g' OR 'x'

range = (speclist)

speclist = (component) OR (component) delim speclist

delim = ',' OR ';'

component = spec OR spec signed-integer

spec = ' ' OR '$' OR {>(pattern)} OR {<(pattern)}
OR positive-integer

signed-integer = sign integer

sign = '+' OR '-'

integer = 0 OR {positive-integer} OR {0 integer}

positive-integer = posdigit OR 0 positive-integer
OR posdigit positive-integer

posdigit = 1 OR 2 OR 3 OR 4 OR 5 OR 6 OR 7 OR 8 OR 9

pattern = "a text pattern-- see Chapter 5"

```
action = 'q' OR entercom OR printfcom OR r-action
r-action = deletecom OR printcom
           OR subcom OR writecom OR readcom
           OR appendcom OR printccom OR changecom
           OR insertcom
entercom = 'e' OR {'e' (fname)}
fname = alpha OR {alpha l-alphanum}
alpha = a OR b OR . . . OR z OR A OR B OR . . . OR Z
alphanum = alpha OR integer
l-alphanum = alphanum OR {alphanum l-alphanum}
printfcom = 'f' OR {'f' (fname)}
appendcom = 'a' text
text = () OR (lchars)
lchars = character OR {character lchars}
character = "any ascii character"
changecom = 'c' text
insertcom = 'i' text
deletecom = 'd' OR{'d' printcom}
printcom = 'p'
writecom = 'w' OR{'w' (fname)}
readcom = 'r' OR {'r' (fname)}
printccom = '=' OR {'=' printcom}
subcom = subst OR {subst 'p'}
subst = {'s' (new) (old)} OR {'s' (new (old) global}
old = "a text pattern      - -see chapter 5"
```



```
new = lchars
```

THE CODE

By far the most complicated task of this editor is the location of the range of a command; so, let's look at this part of the code first. Later, we will proceed with a top-down discussion of the editor as a whole.

LOCATING THE RANGE OF A COMMAND

GETLIST locates the range of a command. Its arguments are a list of range specifications, TARGET and a text file, TEXT.

As we shall see, the top level functions of the editor divide the lines of the edit file into separate lists. At the time GETLIST is called, FIRST of the text file is the current line; so, the first line of the file is not necessarily line number one (the first line after line zero).

The desired output of the function includes two lists. The first list contains all lines in the file which fall within the range of the current command, except for the last line in the range, which is the first line of the second list. The rest of the second list consists of the remaining lines in the text file. If there is an error due either to bad syntax or failure to meet the range specifications, then the second list will be empty.

In testing the editor, some special (hopefully mnemonic) atoms have been substituted for some of the symbols used in the examples. Here is a list of the characters for which we have had to substitute, and their corresponding atoms.

SYMBOL	ATOM
','	COMMA
';'	SEMI
'<'	BACK
'$'	LAST
'+'	PLUS
'-'	MINUS
'.'	CURLN
'>'	FORE

```
GETLIST:"(((FORE (A T))) ((D O G NL)(C A T NL)(END NL))).
 -=> (() () ((C A T NL) (END NL) (D O G NL)))

GETLIST:"(((FORE (A T))) ((D O G NL)(M O U S E NL)(END NL))).
 -=> (() () ())
```

If we were to concatenate list one (the second result) returned by GETLIST with list two (the third result-- and no error condition had occurred), we would see that the "circular ordering" of the text file had been maintained. We define the "circular ordering" of the text file as follows:

1) Every line in the file (including line zero) precedes line zero.

2) Line A (not line zero) precedes line B (not line zero)

if, in the sequential order of the lines in TEXT, either a) line A comes before line B and line B comes before line zero, or b) line B comes before line zero and line zero comes before line A.

So, it will be the case that if line A precedes line B before the call to GETLIST, it shall do so after the call. This is also true of the help functions of GETLIST.

The other result of GETLIST is a boolean default indicator. If this indicator evaluates to TRUE, this is a signal that no range specifications were used in this command and that default range specifications are to apply.

```
DEFINE GETLIST (TARGET TEXT)
    IF NULL:TARGET THEN <TRUE <> TEXT>
    ELSE CONS:<FALSE GETLIST1:<TARGET TEXT>>
.
 -=>GETLIST

DEFINE GETLIST1 (TARGET TEXT)
    IF NULL:REST:TARGET
        THEN ONEADD:GETSPOT:<FIRST:TARGET TEXT>
    ELSE GETONE:<TARGET TEXT>
.
 -=>GETLIST1
```

Only if TARGET is empty when GETLIST is called is the default flag set to TRUE. To isolate this test for NULL:TARGET,

we must have separate function definitions for GETLIST and GETLIST1.

For most commands, the default range is the current line, so when no range is specified, the result of GETLIST is a list containing the default flag (TRUE), followed by an empty list, followed by an unaltered copy of TEXT.

Example

```
GETLIST:<<> "(B 0 0)>.
 -=> (TRUE () (B 0 0))
```

Each range component is processed by GETSPOT. GETSPOT returns two lists, which together make up the edit file. If GETSPOT is successful the second list begins with the line specified by the range component, TARGET.

If list two returned by GETSPOT is empty, then the attempt to locate the line specified by the range specification has failed.

```
DEFINE GETSPOT (TARGET TEXT)
    PROCREST:PROCFRST:<TARGET TEXT>

-=>GETSPOT
```



```
GETSPOT:"((FORE (A)) ((A NL)(END NL)(Z NL))).
 -=> (((Z NL)) ((A NL) (END NL)))

GETSPOT:"((BACK (Z)) ((A NL)(END NL)(Z NL))).
 -=> (() ((Z NL) (A NL) (END NL)))


DEFINE PROCFRST (TARGET TEXT)
    IF OR:<ATOM:TARGET NOT:ATOM:FIRST:TARGET>
        THEN <TARGET <> TEXT>
    ELSEIF CURLNQ:FIRST:TARGET
        THEN <REST:TARGET <> TEXT>
    ELSEIF ENDQ:FIRST:TEXT
        THEN <TARGET <> <>>
    ELSEIF DIGITQ:FIRST:TARGET
        THEN CONS:<2:DIGVAL:TARGET
                   FINDLINE:<SUB1:1:DIGVAL:TARGET INORDER:TEXT>>
    ELSEIF LASTQ:FIRST:TARGET
        THEN CONS:<REST:TARGET FINDEND:TEXT>
    ELSEIF FORWARDQ:FIRST:TARGET
        THEN FORESCAN:<REST:TARGET CKLSHIFT:LSHIFT:TEXT>
    ELSEIF BAKWARDQ:FIRST:TARGET
        THEN BACKSCAN:<REST:TARGET CKRSHIFT:RSHIFT:TEXT>
    ELSE <TARGET <> <>>

-=>PROCFRST
```

GETSPOT calls PROCFRST to process the 'spec' (see page 110) part of a range component. PROCFRST returns a list containing what is left of TARGET after the 'spec' part, and two lists which comprise the editor file. The second begins with the line corresponding to the 'spec' part of the range component, TARGET.

Here are some examples of PROCFRST:

```
PROCFRST:<<> "((END NL))).
 -=> (() () ((END NL))).

PROCFRST:"((7) ((END NL))).
 -=> ((7) () ())

PROCFRST:"((2) ((A NL)(B NL)(C NL)(END NL))).
 -=> (() ((A NL)) ((B NL) (C NL) (END NL)))
```

If the TARGET begins with a digit (IF DIGITQ:FIRST:TARGET), then a line number specification has been used. DIGVAL is then called; it returns two results. The first is an integer indicating the value of the leading string of one or more digits in TARGET. The second result is a list containing what remains of TARGET after the leading string of digits.

```
DEFINE DIGVAL LIST
    <CTOI       1>:<
     GETDIGS:LIST>
.
 -=>DIGVAL

DEFINE GETDIGS LIST
    IF NULL:LIST THEN <<><>>
    ELSEIF DIGITQ:FIRST:LIST
         THEN <CONS          1>:<
              <FIRST:LIST   #>
               GETDIGS:REST:LIST>
    ELSE <<> LIST>
.
 -=>GETDIGS

DIGVAL:"(1 2 A B C).
 -=> (12 (A B C))
```

We have seen CTOI in Chapter 2.

If a line number specification is used, then FINDLINE locates the proper line. INORDER is first called to order the text file so that it begins with line 1 (if it isn't empty). FINDLINE is just a variation of the function PROJ, which we have seen in previous chapters.

Both FINDLINE and INORDER have the ground condition of ENDQ:FIRST:TEXT; PROCFRST has already determined that there is

at least one line besides line zero in the file, so INORDER won't go on LSHIFTing forever (definition of LSHIFT is to follow).

```
DEFINE INORDER TEXT
    IF ENDQ:FIRST:TEXT THEN LSHIFT:TEXT
    ELSE INORDER:LSHIFT:TEXT
.
 -=>INORDER


DEFINE FINDLINE (NUM TEXT)
    IF ENDQ:FIRST:TEXT THEN <<><>>
    ELSEIF NOT:POS:NUM THEN <<> TEXT>
    ELSE <CONS          1>:<
         <FIRST:TEXT    #>
          FINDLINE:<SUB1:NUM REST:TEXT>>
.
 -=>FINDLINE

FINDLINE:<3 "((A NL)(B NL)(C NL)(D NL)(END NL))>.
 -=> (((A NL)(B NL)(C NL))((D NL)(END NL)))
```

FINDEND is called in response to a '$' specification.

```
DEFINE FINDEND TEXT
    IF ENDQ:2:TEXT THEN <<> TEXT>
    ELSE <CONS          1>:<
         <FIRST:TEXT        #>
          FINDEND:REST:TEXT>
.
 -=>FINDEND

FINDEND:"((A NL)(B NL)(END NL)(J NL)(E E K NL)).
 -=> (((A NL))((B NL)(END NL)(J NL)(E E K NL)))
```

FINDEND knows TEXT isn't empty since this was ascertained

in PROCFRST. The definition of FINDEND follows the same form of GETLINE, FINDX, etc.

FORESCAN handles forward context searches. LSHIFT is called before the call to FORESCAN to rotate the text file as in a circular left shift so that the context search starts with the line after the current line and ends with the current line. CKLSHIFT insures that the text file doesn't begin with line zero unless it is empty. (Remember, line zero can't be the current line of a non-empty file).

```
DEFINE FORESCAN (TARGET TEXT)
    IF OR:<NULL:TARGET ATOM:FIRST:TARGET>
          THEN <TARGET <> <>>
    ELSE CONS:<REST:TARGET FINDX:<FIRST:TARGET TEXT>>

'
-=>FORESCAN

DEFINE LSHIFT LIST
    SNOC:<FIRST:LIST REST:LIST>

'
-=>LSHIFT

DEFINE CKLSHIFT TEXT
    IF ENDQ:FIRST:TEXT THEN LSHIFT:TEXT
    ELSE TEXT

'
-=>CKLSHIFT


FORESCAN:"(((A)) ((X NL)(A NL)(END NL)(B NL))).
 -=> (() ((X NL)) ((A NL) (END NL) (B NL)))
```


FORESCAN uses a version of FINDX which differs from that used in Chapter 5 in that it deals with lines which have already been divided into separate lists.

```
DEFINE FINDX (PAT TEXT)
    IF NULL:TEXT THEN <<><>>
    ELSEIF MATCH:<PAT FIRST:TEXT>
         THEN <<> TEXT>
    ELSE <CONS        1>:<
         <FIRST:TEXT  #>
          FINDX:<PAT REST:TEXT>>

'
-=>FINDX
```

BACKSCAN handles backward context searches. FINDX is applied to the REVERSEd text file and REV2 restores the elements of TEXT to their original order, insuring that the first line of the second list returned begins with the line "found" by FINDX.

```
DEFINE BACKSCAN (TARGET TEXT)
    IF OR:<NULL:TARGET ATOM:FIRST:TARGET>
         THEN <TARGET <> <>>
    ELSE <REST:TARGET REV2:FINDX:<FIRST:TARGET REVERSE:TEXT>>

'
-=>BACKSCAN

DEFINE REV2 (TEXT1 TEXT2)
    IF NULL:TEXT2 THEN <REVERSE:TEXT2 TEXT1>
    ELSE <REVERSE:REST:TEXT2 CONS:<FIRST:TEXT2 REVERSE:TEXT1>>

'
-=>REV2

DEFINE REVERSE TEXT
    IF NULL:TEXT THEN <>
    ELSE SNOC:<FIRST:TEXT REVERSE:REST:TEXT>

'
-=>REVEPSE
```

RSHIFT and CKRSHIFT are used to insure that the backward scan begins with the line preceding the current line.

```
DEFINE CKRSHIFT TEXT
    IF ENDQ:FIRST:TEXT THEN RSHIFT:TEXT
    ELSE TEXT
.
 -=>CKRSHIFT

DEFINE RSHIFT TEXT
    CONS:<RAC:TEXT RDC:TEXT>
.
 -=>RSHIFT


DEFINE RAC TEXT
    IF NULL:REST:TEXT THEN FIRST:TEXT
    ELSE RAC:REST:TEXT
.
 -=>RAC

DEFINE RDC TEXT
    IF NULL:REST:TEXT THEN <>
    ELSE CONS:<FIRST:TEXT RDC:REST:TEXT>
.
 -=>RDC
```

After PROCFRST has finished processing the 'spec' part of a range component, PROCREST is used to process the 'signed-integer' portion. This portion of a range component has to do with line number arithmetic.

PROCREST returns two lists which comprise the final output of GETSPOT.

```
DEFINE PROCREST (TARGET TEXT1 TEXT2)
    IF NULL:TEXT2 THEN <<><>>
    ELSEIF NULL:TARGET THEN <TEXT1 TEXT2>
    ELSEIF NOT:ATOM:FIRST:TARGET
         THEN <<><>>
    ELSEIF PLUSQ:FIRST:TARGET
         THEN PLUSMOVE:<REST:TARGET APPEND:<TEXT2 TEXT1>>
    ELSEIF MINUSQ:FIRST:TARGET
         THEN MINUSMOV:<REST:TARGET APPEND:<TEXT2 TEXT1>>
    ELSE <<><>>
.
 -=>PROCREST
```


If the first character of TARGET is '+' (PLUS), then PLUSMOVE is called to handle line number addition.

```
DEFINE PLUSMOVE (N TEXT)
    PLUSMOV1:<DIGVAL:N TEXT>
.
 -=>PLUSMOVE

DEFINE PLUSMOV1 ((N TARG) TEXT)
    IF OR:<NULL:N TARG> THEN <<><>>
    ELSEIF ZEROP:N THEN <<> TEXT>
    ELSE FINDLINE:<N TEXT>
.
 -=>PLUSMOV1


PROCREST:"((PLUS 2) ((A NL)) ((B NL)(C NL)(D NL)(END NL))).
 -=> (((B NL) (C NL)) ((D NL) (END NL) (A NL)))


DEFINE PLUSQ CHAR
   SAME:<CHAR PLUS>
.
 -=>PLUSQ
```

The test for ZEROP:N in PLUSMOV1 precedes the call to FINDLINE since some prankster may wish to use '((. + 0))' as the range specification of a command when the text file is empty; FINDLINE returns an error message as soon as it sees that line zero is the first line of TEXT. Since line number arithmetic may not cause a wrap around line zero, FINDLINE serves well as a help function to PLUSMOVE.

MINUSMOV is used to handle line number subtraction. Its relation to PLUSMOVE parallels that of BACKSCAN to FORESCAN.

```
DEFINE MINUSMOV (TARGET TEXT)
    SETMINUS:PLUSMOVE:<TARGET REVERSE:TEXT>
.
-=>MINUSMOV


DEFINE SETMINUS (TEXT1 TEXT2)
    IF NULL:TEXT2 THEN <<><>>
    ELSE REV2:<TEXT1 TEXT2>
.
-=>SETMINUS
```

ONEADD is called when there is only one component to a range specification. Its arguments are two lists returned by GETSPOT. In this case, the range of the command is to be the line of TEXT indicated by the range component; so, the first list returned by ONEADD is empty and the second list begins with the line which is the range of the command.


```
DEFINE ONEADD (TEXT1 TEXT2)
    IF NULL:TEXT2 THEN <<><>>
    ELSE <<> APPEND:<TEXT2 TEXT1>>
.
-=>ONEADD
```

Now, we have seen how the range of a command is located when zero or one range component is used in a range specification. When more than one are used, GETLIST calls GETONE.

```
DEFINE GETONE (TARGET TEXT)
    IF NOT:ATOM:2:TARGET THEN <<><>>
    ELSEIF COMMAQ:2:TARGET
        THEN SETUPC:<FIRST:TARGET RREST:TARGET TEXT>
    ELSEIF SEMIQ:2:TARGET
        THEN SETUPS:<FIRST:TARGET RREST:TARGET TEXT>
    ELSE <<><>>
.
-=>GETONE

DEFINE SEMIQ CHAR
    SAME:<CHAR SEMI>
.
-=>SEMIQ

DEFINE COMMAQ CHAR
    SAME:<CHAR COMMA>
.
-=>COMMAQ
```

Range components must be separated by a comma or a semicolon. If the first two range components of a range specification are separated by a comma, then SETUPC is called.

```
DEFINE SETUPC (TARGET1 TARGET2 TEXT)
    IF NULL:TARGET2 THEN <<><>>
    ELSEIF NULL:REST:TARGET2
         THEN FINALSET:<TARGET1 FIRST:TARGET2 TEXT>
    ELSEIF NULL:2:GETSPOT:<TARGET1 TEXT>
         THEN <<><>>
    ELSE GETONE:<TARGET2 TEXT>

-=>SETUPC
```

If there are more than two range components, then SETUPC just checks to see if the first specifies a line within the bounds of the editor file, and calls GETONE with the rest of the range specifications and an unaltered TEXT as arguments. TEXT is unaltered because the current line is not moved after the location of the line corresponding to a range component that is followed by a comma.

If there are exactly two range components separated by a comma, then the complexity is increased. Our goal is to isolate a section of text which lies between the line specified by the first component and the line specified by the second, inclusive. (If both components specify the same line, then that line is to be located.)

Since the lines corresponding to both components must be located in relation to the same current line, we must mark the current line before locating the line indicated by the first range component. This is because the editor file may be rotated

during location of a line corresponding to a range component (e.g., in context searches, and when line number specifications are used). The line which was the current line at the beginning of a call to GETSPOT may not be at the front of the first list it returns.

```
DEFINE FINALSET (TARGET1 TARGET2 TEXT)
    GETLAST:<TARGET2 GETFIRST:GETSPOT:<TARGET1 PUTCUR:TEXT>>

-=>FINALSET

DEFINE PUTCUR TEXT
    CONS:<SNOC:<CURLN FIRST:TEXT> REST:TEXT>

-=>PUTCUR
```

After the first of the two components has been processed, the line which has been found (the first line of the second list returned by GETSPOT) is marked, and the text file is rearranged so that it again begins with the current line (the current line marker is then removed).

```
DEFINE GETFIRST (TEXT1 TEXT2)
    IF NULL:TEXT2 THEN <>
    ELSE CURBEGIN:APPEND:<TEXT1
                          CONS:<PUTFIRST:FIRST:TEXT2 REST:TEXT2>>

-=>GETFIRST
```

Mark the first line.

```
DEFINE PUTFIRST LINE
    IF SAME:<RAC:LINE CURLN>
        THEN SNOC:<RAC:LINE SNOC:<FIRSTAD RDC:LINE>>
    ELSE SNOC:<FIRSTAD LINE>

-=>PUTFIRST
```

Order the file, beginning with the current line.

```
DEFINE CURBEGIN TEXT
    REVCONC2:LOCMRKER:<CURLN TEXT>

-=>CURBEGIN

CURBEGIN:"((A NL)(B NL CURLN)(C NL)(END NL)).
 -=> ((B NL) (C NL) (END NL) (A NL))


DEFINE LOCMRKER (MARK FILE)
    IF NULL FILE THEN <<><>>
    ELSEIF SAME:<RAC:FIRST:FILE MARK>
        THEN <<> CONS:<RDC:FIRST:FILE REST:FILE>>
    ELSE <CONS          1>:<
         <FIRST:FILE   #>
          LOCMRKER:<MARK REST:FILE>>

-=>LOCMRKER

DEFINE REVCONC2 (TEXT1 TEXT2)
    APPEND:<TEXT2 TEXT1>

-=>REVCONC2
```


After the second component has been processed, the text file is again rearranged so that it consists of two lists; the first list begins with the line corresponding to the first component, and the second list begins with the line corresponding to the second component.

Locate line 2.

```
DEFINE GETLAST (TARGET TEXT)
    IF NULL:TEXT THEN <<><>>
    ELSE CKFINAL:GETSPOT:<TARGET TEXT>

-=>GETLAST


DEFINE CKFINAL (TEXT1 TEXT2)
    IF NULL:TEXT2 THEN <<><>>
    ELSE IFFIND1:<GETFSTAD:TEXT1 TEXT2>

-=>CKFINAL
```

Locate the first line (marked by FIRSTAD).

```
DEFINE GETFSTAD TEXT
    LOCMRKER:<FIRSTAD TEXT>

-=>GETFSTAD
```

Order the two lists so the range is isolated.

```
DEFINE IFFIND1 ((BEFORE AFTER) TEXT)
    IF AFTER THEN <AFTER APPEND:<TEXT BEFORE>>
    ELSE IFFIND2:<GETFSTAD:TEXT BEFORE>

-=>IFFIND1

DEFINE IFFIND2 ((BEFORE AFTER) TEXT)
    <APPEND:<AFTER TEXT> BEFORE>

-=>IFFIND2
```

The symbol '.' (CURLN) is used to mark the current line. The symbol 'F' (FIRSTAD) is used to mark the line associated with the first range component. All marks are SNOCed to the end of a line, after the NL character, so that there is no interference with pattern matching.

If two range specifications are separated by a semicolon, then each time a line corresponding to a range component is found, that line becomes the current line.

SETUPS handles specifications in which the first two components are separated by semicolons. For each component, GETSPOT is called. If there are more than two components in the range specification, then GETONE is called, with the second list returned by GETSPOT appended to the first (indicating that the line corresponding to the most recently processed range component is now the current line).

As was the case when range components were separated by commas, when there are exactly two range components separated by semicolons, our goal is to return two lists, with the first list beginning with the line of the text file associated with the first component and with the line associated with the second component at the beginning of the second list. (Again, if both components specify the same line, then we want an empty list followed by a list beginning with the line specified by both components).

In this case we needn't mark the current line because the current line is changed to the line associated with the first component after that line has been located. Before locating the line associated with the second component, however, we must mark the line associated with the first (with FIRSTAD). After the second line has been located, the two desired lists are returned via a call to CKFINAL, which we have recently seen.

```
DEFINE SETUPS (TARGET1 TARGET2 TEXT)
    IF NULL:TARGET2 THEN <<><>>
    ELSE SETUPS1:<TARGET2 GETSPOT:<TARGET1 TEXT>>
.
-=>SETUPS

DEFINE SETUPS1 (TARGET (TEXT1 TEXT2))
    IF NULL:TEXT2 THEN <<><>>
    ELSEIF NULL:REST:TARGET
        THEN CKFINAL:GETSPOT:<FIRST:TARGET
                              APPEND:<CONS:<PUTFIRST:FIRST:TEXT2
                                            REST:TEXT2> TEXT1>>
    ELSE GETONE:<TARGET APPEND:<TEXT2 TEXT1>>
.
-=>SETUPS1
SETUPS:"((2) ((3)) ((A NL)(B NL)(C NL)(END NL)(D NL))).
-=> (((A NL)) ((B NL) (C NL) (END NL) (D NL)))
```

Exactly what causes GETONE to terminate may not be immediately apparent for the reader. Each recursive call to GETONE (via invocation of SETUPC or SETUPS) results in invocation of the progress function, REST:TARGET. Termination results when SETUPC or SETUPS detect erroneous syntax, or there are exactly two range components in TARGET, resulting in a call to FINALSET.

Now that we have discussed the issue of locating the range of a command, let's begin the discussion of the editor as a whole.

TOP LEVEL FUNCTIONS (TEXTED, et al.)

The editing session begins with a call to TEXTED, with arguments COMS, FNAME, and LFILES. TEXTED attempts to open an edit file, FNAME, in READ access mode (it is not assumed at this point that the file will be written to the file library).

As was mentioned in the introduction to this chapter, the editor creates two output files, a message file and a file library. If TEXTED attempts to open a file, FNAME, and the attempt fails due to an access mode violation, the editing session terminates with its output being an error message in the message file and an unaltered copy of the file library. (Note that the file library takes the same form as have the file libraries used in the functions of Chapter 3.)

```
TEXTED:<"((Q)) "(B 0 0) "(((B 0 0) WRITE (H A)))>.
 -=> ((NL C A N T BLANK O P E N BLANK B 0 0) (((B 0 0) WRITE (H A))))
```

If the attempt to open file FNAME fails because no file FNAME was found in the file library, then an empty one is created.

```
TEXTED:<"((Q)) "(B 0 0) <>>.
 -=> ((NL Q NL E X I T NL) ())
```

If the edit file is successfully opened, then EDIT is called to read and process commands. SEPLINES, which we saw in Chapter 4, is called to separate the lines of the file into

lists, and ADDEND is called to add line zero to the end of the file.

```
DEFINE TEXTED (COMS FNAME LFILES)
    TEXTED1:<COMS OPEN:<FNAME READ LFILES> FNAME LFILES>
.
-=>TEXTED

DEFINE TEXTED1 (COMS (STATUS TEXT) FNAME LFILES)
    IF NOT:ERRORQ:STATUS
         THEN EDIT:<COMS ADDEND:SEPLINES:TEXT FNAME LFILES>
    ELSEIF TEXT THEN <CANT:FNAME LFILES>
    ELSE EDIT:<COMS <> FNAME LFILES>
.
-=>TEXTED1

DEFINE ERRORQ STATUS
    SAME:<STATUS ERROR>
.
-=>ERRORQ

DEFINE CANT NAME
    APPEND:<"(NL C A N T BLANK O P E N BLANK) NAME>
.
-=>CANT

DEFINE ADDEND TEXT
    SNOC:<<END NL> TEXT>
.
-=>ADDEND
```

Drivers (EDIT, et al.).

EDIT reads the first command in its command file, COMS. Once EDIT has been called, the editor is designed to terminate only when EDIT encounters a quit command, 'Q'. The command file is treated as an infinite list. The progress function for EDIT is REST:COMS.

```
DEFINE EDIT (COMS TEXT FNAME LFILES)
    IF QUITQ:FIRST:COMS
        THEN <APPEND:<CONS:<NL FIRST:COMS> "(NL E X I T NL)>
             LFILES>
    ELSE <APPEND                    1>:<
         <<FIRST:COMS NL PR>        #>
          EDIT1:<1:GETTARG:FIRST:COMS
                 2:GETTARG:FIRST:COMS
                 TEXT FNAME LFILES REST:COMS>>
.
-=>EDIT
```

```
DEFINE QUITQ COM
   SAME:<FIRST:COM QUIT>
.
-=>QUITQ
```

If the first command is not a quit command, then EDIT1 is called to determine whether or not the first command is a global command. GETTARG is used to separate a command into its 'range' and 'action' components.


```
DEFINE GETTARG COM
    IF NULL:COM THEN <<> <>>
    ELSEIF ATOM:FIRST:COM THEN <<> COM>
    ELSE <FIRST:COM REST:COM>
.
-=>GETTARG
```

```
GETTARG:"(((1) COMMA (7)) P).
 -=> (((1) COMMA (7)) (P))
```

```
DEFINE EDIT1 (TARG COM TEXT FNAME LFILES LCOMS)
    IF GLOBALQ:COM
        THEN GCKERR:<DOGLOB:<TARG TEXT GVAL:FIRST:COM
                             REST:COM FNAME LFILES LCOMS>
                            TEXT FNAME LFILES LCOMS>
    ELSE CKERR:<DOSINGLE:<TARG COM TEXT FNAME LFILES>
                          TEXT FNAME LFILES LCOMS>
.
-=>EDIT1

DEFINE GLOBALQ COM
    AND:<COM ATOM:FIRST:COM OR:<SAME:<FIRST:COM GLOBAL>
                                SAME:<FIRST:COM NOGLOB>>>
.
-=>GLOBALQ

DEFINE GVAL COM
    SAME:<COM GLOBAL>
.
-=>GVAL
```

We'll be looking at GCKERR and CKERR shortly.

Non-global Commands.

The execution of non-global commands begins with a call to DOSINGLE. A call to DOSINGLE results in the creation of 4 lists; 1) a file containing output for the printer (a message file); 2) a (possibly altered copy of the editor file; 3) a file name; and 4) a file library.

```
DEFINE DOSINGLE (TARG TEXT FNAME LFILES)
    IF ENTERQ:<TARG COM> THEN DOENTER:<REST:COM FNAME LFILES>
    ELSEIF PRINTFNQ:<TARG COM>
        THEN DOPRINTF:<REST:COM TEXT FNAME LFILES>
    ELSE SETARGS:<EXEC:<GETLIST:<TARG TEXT> COM FNAME LFILES
                        TEXT> FNAME>

-=>DOSINGLE
```

The 'E' and 'F' commands may not be preceded by any range specifications. It is for this reason that the functions to carry out these commands are called from DOSINGLE, before a call to GETLIST is made to locate the range of a command.

DOENTER executes the 'E' command. The test for NULL:TARG occurs in ENTERQ because the 'E' command must have no range specifications. ENTER evaluates to 'E'.


```
DEFINE ENTERQ (TARG COM)
    AND:<NULL:TARG ATOM:FIRST:COM SAME:<FIRST:COM ENTER>>

-=>ENTERQ

DEFINE DOENTER (NEWNAME FNAME LFILES)
    IF NULL:NEWNAME
        THEN CKOPEN:<OPEN:<FNAME READ LFILES> FNAME LFILES>
    ELSEIF OR:<REST:NEWNAME ATOM:FIRST:NEWNAME
                NOT:ISLAT:FIRST:NEWNAME>
        THEN <<><><><>>
    ELSE CKOPEN:<OPEN:<FIRST:NEWNAME READ LFILES>
                    FIRST:NEWNAME LFILES>

-=>DOENTER
```

The file must be converted to an edit file with SEPLINES and ADDEND.

```
DEFINE CKOPEN ((STATUS FILE) NEWNAME LFILES)
    IF ERRORQ:STATUS THEN <><><><>>
    ELSE <<> ADDEND:SEPLINES:FILE NEWNAME LFILES>

-=>CKOPEN

DEFINE ISLAT LIST
    IF NULL:LIST THEN TRUE
    ELSEIF NOT:ATOM:FIRST:LIST THEN FALSE
    ELSE ISLAT:REST:LIST

-=>ISLAT
```

DOENTER looks at NEWNAME to see if a file name has been specified in the command. If not, FNAME is used as the default file name. Four results are returned by DOENTER, an empty message file, a text file, a file name and the file library. Four empty lists result from erroneous syntax or the failure to open a new text file in READ mode.

```
DOENTER:"(((B O O)) (S O X) (((B O O) READ (G O)))),
 -=> (() (G O) (B O O) (((B O O) READ (G O))))
```

DOPRINTF executes the 'F' command. As was the case with the enter command, PRINTFNQ must check to see if any range specifications precede the 'F' command.

```
DEFINE PRINTFNQ (TARG COM)
    AND:<NULL:TARG ATOM:FIRST:COM SAME:<FIRST:COM PRINTFN>>

-=>PRINTFNQ

DEFINE DOPRINTF (COM TEXT FNAME LFILES)
    IF NULL:COM THEN <CONS:<NL FNAME> TEXT FNAME LFILES>
    ELSEIF OR:<REST:COM ATOM:FIRST:COM NOT:ISLAT:FIRST:COM>
        THEN <<><><><>>
    ELSE <<NL FIRST:COM> TEXT FIRST:COM LFILES>

-=>DOPRINTF
```

DOPRINTF returns four results, a message file containing a file name, the text file, a file name and the file library. Four empty lists result from erroneous command syntax.

```
DOPRINTF:"(((A F I L E)) ((END NL)) (D F I L E)
        (((B O O) READ (G O)))),
 -=> ((NL A F I L E) ((END NL)) (A F I L E) (((B O O) READ (G O))))
```


EXEC is called if the current command was neither an 'E' nor an 'F'. One of its arguments is the result of a call to GETLIST. None of the commands called after the invocation of EXEC affects the default file name. A call to EXEC returns three lists, a message file, an editor file and a file library. SETARGS adds the file name to the result returned by EXEC.

```
DEFINE EXEC ((DEFAULT TEXT1 TEXT2) COM FNAME LFILES XTEXT)
    IF NULL:TEXT2 THEN <<><><>>
    ELSE DOCOM:<XTEXT COM DEFAULT TEXT1 TEXT2 FNAME LFILES>

-=>EXEC

DEFINE SETARGS ((MESS TEXT LFILES) FNAME)
    <MESS TEXT FNAME LFILES>

-=>SETARGS
```

If GETLIST succeeds in locating the range of a command, then DOCOM is called. DOCOM and its help function, DOCOM1, decide what the current command is and call a section of code to execute that particular command.

```
DEFINE DOCOM (XTEXT COM DEFAULT TEXT1 TEXT2 FNAME LFILES)
    IF WRITECQ:COM
        THEN DOWRITE:<XTEXT DEFAULT REST:COM TEXT1 TEXT2
                    FNAME LFILES>
    ELSE SETANS:<DOCOM1:<DEFAULT COM TEXT1 TEXT2 FNAME LFILES>
                LFILES>

-=>DOCOM
```

```
DEFINE DOCOM1 (DEFAULT COMMAND TEXT1 TEXT2 FNAME LFILES)
    IF NULL:COMMAND THEN PDEFAULT:<DEFAULT TEXT1 TEXT2>
    ELSEIF NOT:ATOM:FIRST:COMMAND
         THEN <<><>>
    ELSEIF READCQ:FIRST:COMMAND
         THEN DOREAD:<COMMAND TEXT1 TEXT2 FNAME LFILES>
    ELSEIF APPENDQ:FIRST:COMMAND
         THEN DOAPPEND:<REST:COMMAND TEXT1 TEXT2>
    ELSEIF INSERTQ:FIRST:COMMAND
         THEN DOINSERT:<REST:COMMAND TEXT1 TEXT2>
    ELSEIF PRINCQ:FIRST:COMMAND
         THEN DOPRINC:<REST:COMMAND TEXT1 TEXT2>
    ELSEIF ATEND:<TEXT1 TEXT2>
         THEN <<><>>
    ELSEIF CHANGEQ:FIRST:COMMAND
         THEN DOCHANGE:<REST:COMMAND REST:TEXT2>
    ELSEIF DELETEQ:FIRST:COMMAND
         THEN DODELETE:<REST:COMMAND REST:TEXT2>
    ELSEIF PRINTQ:FIRST:COMMAND
         THEN DOPRINT:<REST:COMMAND TEXT1 TEXT2>
    ELSEIF SUBSTQ:FIRST:COMMAND
         THEN DOSUBST:<REST:COMMAND TEXT1 TEXT2>
    ELSE <<><>>
.
-=>DOCOM1
```

(A list of the trivial help functions to DOCOM1 is on page 167 .)

Recall that GETLIST returns three results, a default indicator (DEFAULT), and two lists (TEXT1 and TEXT2). If DEFAULT evaluates to TRUE, then default range specifications apply for all commands.

If DEFAULT is FALSE and TEXT2 is non-empty, then the range of the command consists of TEXT1 plus the first line of TEXT2. If TEXT2 is empty, there has either been an error in syntax or the range specifications were not met. (That is, the lines associated with one or more range components were not found).

The outcome of a call to DOCOM is a list of three results, a message file, a text file and the file library. The only

command which may alter the file library is the write command ('W'); for this reason it is treated separately from the other commands, which are called from DOCOM1.

As we shall see, the code to execute the individual commands is (with the exception of the 'S' command) for the most part, simple and non-recursive. The work has been done in GETLIST.

```
DEFINE WRITECQ COM
    AND:<COM ATOM:FIRST:COM SAME:<FIRST:COM WRITEC>>
.
-=>WRITEQ

DEFINE DOWRITE (XTEXT DEFAULT COM TEXT1 TEXT2 FNAME LFILES)
    IF ENDQ:FIRST:TEXT2
         THEN <<><><>>
    ELSEIF DEFAULT
         THEN EDWRITE:<XTEXT COM RDC:INORDER:XTEXT FNAME
                       LFILES>
    ELSEIF BEHINDQ:TEXT1
         THEN <<><><>>
    ELSE EDWRITE:<XTEXT COM SNOC:<FIRST:TEXT2 TEXT1> FNAME
                  LFILES>
.
-=>DOWRITE
```

XTEXT is a copy of the text file as it existed before the call to GETLIST in EXEC. This file is one of the results of DOWRITE, since a 'W' command does not alter the text file.

If the DEFAULT flag is TRUE, then the default range for the 'W' command is the whole file. INORDER is called to order

the text file. The RDC of the result of INORDER is the text file without line zero.

BEHINDQ checks to see if the section of text to be written contains line zero; if so, the command fails.

```
DEFINE BEHINDQ TEXT
    LMEMBER:<END TEXT>
.
 -=>BEHINDQ

DEFINE LMEMBER (A LIST)
    OR:<MEMBER*>:<
       <A*>
        LIST>
.
 -=>LMEMBER
```

Now, EDWRITE creates a file containing text from the editor file. The text must be restored to its original form (the lists of lines must be concatenated).

```
DEFINE EDWRITE (XTEXT COM TEXT FNAME LFILES)
    IF NULL:COM
        THEN EDCREATE:<XTEXT TEXT
                      CREATE:<<FNAME READWRIT CONCAT:TEXT>
                               LFILES>>
    ELSEIF OR:<ATOM:FIRST:COM REST:COM NOT:ISLAT:FIRST:COM>
        THEN <<><><>>
    ELSE EDCREATE:<XTEXT TEXT
                  CREATE:<<FIRST:COM READWRIT CONCAT:TEXT>
                           LFILES>>
.
 -=>EDWRITE
```



```
DEFINE EDCREATE (XTEXT TEXT (STATUS LFILES))
    IF ERRORQ:STATUS THEN <><><>>
    ELSE <WTCOUNT:TEXT XTEXT LFILES>
.
 -=>EDCREATE
```

WTCOUNT yields a message indicating the number of lines which have been written from the text file.

```
DEFINE WTCOUNT TEXT
    CONS:<NL PUTDEC:<PDWIDTH LENGTH:TEXT>>
.
 -=>WTCOUNT
```

```
DOWRITE:<"((A NL)(B NL)(C NL)(D NL)(END NL))
         FALSE "((B 0 0 P)) "((B NL)(C NL))
        "((D NL)(END NL)(A NL)) "(F N A M E)
        "(((B 0 0 P) READWRIT (0 L D NL S T U F F NL)))>.
-=> ((NL BLANK BLANK BLANK BLANK 3) ((A NL) (B NL) (C NL) (D NL) (END N
L)) (((B 0 0 P) READWRIT (B NL C NL D NL))))
```

```
DEFINE LENGTH TEXT
    IF NULL:TEXT THEN 0
    ELSE ADD1:LENGTH:REST:TEXT
.
 -=>LENGTH
```

SETANS just creates a list containing the results of DOCOM1 (a message file and a text file) and a file library.

```
DEFINE SETANS ((MESS TEXT) LFILES)
    <MESS TEXT LFILES>
.
-=>SETANS
```

PDEFAULT does a default PRINT command if no action has been specified for the current command. If the default flag is on, CKEND rotates the text file so that the current line is advanced one line, and that line is printed out. An error results from an attempt to print line zero.

```
DEFINE PDEFAULT (DEFAULT TEXT1 TEXT2)
    IF ENDQ:FIRST:TEXT2 THEN <<><>>
    ELSEIF DEFAULT THEN CKEND:TEXT2
    ELSE <FIRST:TEXT2 APPEND:<TEXT2 TEXT1>>
.
-=>PDEFAULT

DEFINE CKEND TEXT
    IF ENDQ:RAC:TEXT THEN <<><>>
    ELSE <RAC:TEXT LSHIFT:TEXT>
.
-=>CKEND

PDEFAULT:<<> "((A NL)(B NL)) "((C NL)(D NL)(END NL))>.
 -=> ((C NL) ((C NL) (D NL) (END NL) (A NL) (B NL)))
```

DOREAD executes the 'R' command. READANS arranges for the last line of the file read in to be the new current line. WTCOUNT prints a message indicating the number of lines in the file

which has been read.

```
DEFINE DOREAD (COM TEXT1 TEXT2 FNAME LFILES)
    IF NULL:COM
        THEN EDREAD:<OPEN:<FNAME READ LFILES> TEXT1 TEXT2
                    LFILES>
    ELSEIF OR:<NOT:ATOM:FIRST:COM REST:COM NOT:ISLAT:FIRST:COM>
        THEN <<><>>
    ELSE EDREAD:<OPEN:<FIRST:COM READ LFILES> TEXT1 TEXT2
                LFILES>
.
-=>DOREAD

DEFINE EDREAD ((STATUS FILE) TEXT1 TEXT2 LFILES)
    IF ERRORQ:STATUS THEN <<><>>
    ELSEIF NULL:FILE THEN <WTCOUNT:FILE APPEND:<TEXT2 TEXT1>>
    ELSE READANS:<SEPLINES:FILE TEXT1 TEXT2 LFILES>
.
-=>EDREAD
.
DEFINE READANS (FILE TEXT1 TEXT2 LFILES)
    <WTCOUNT:FILE EDARANGE:<FILE SNOC:<FIRST:TEXT2 TEXT1>
                            REST:TEXT2>>
.
-=>READANS

DEFINE EDARANGE (NEW TEXT1 TEXT2)
    CONS:<RAC:NEW CONCAT:<TEXT2 TEXT1 ROC:NEW>>
.
-=>EDARANGE


DOREAD:"(((F N A M E))((A NL)(B NL)) ((C NL)(END NL))
    (O L D) (((F N A M E) READ (N E W NL S T U F NL)))).
 -=> ((NL BLANK BLANK BLANK BLANK 2) ((S T U F NL) (END NL) (A NL) (B NL
 ) (C NL) (N E W NL)))
```

DOAPPEND executes the 'A' command. It is just like DOREAD except that it doesn't have to worry about file I/O, and it puts nothing into the message file.

```
DEFINE DOAPPEND (COM TEXT1 TEXT2)
    IF NULL:COM THEN <<> APPEND:<TEXT2 TEXT1>>
    ELSEIF OR:<ATOM:FIRST:COM REST:COM NOT:ISLAT:FIRST:COM>
        THEN <<><>>
    ELSE EDAPPEND:<SEPLINES:FIRST:COM SNOC:<FIRST:TEXT2 TEXT1>
                REST:TEXT2>
.
 -=>DOAPPEND

DEFINE EDAPPEND (NEW TEXT1 TEXT2)
    <<> EDARANGE:<NEW TEXT1 TEXT2>>
.
 -=>EDAPPEND

DOAPPEND:"(((N E W NL S T U F NL))((A NL)) ((B NL)(END NL))).
 -=> (() ((S T U F NL) (END NL) (A NL) (B NL) (N E W NL)))
```

Again, DOINSERT, which executes the 'I' command, is a variation on the theme of the 'R' and 'A' commands, except that NEW (see EDINSERT) is inserted before TEXT2 rather than after its first line.

```
DEFINE DOINSERT (COM TEXT1 TEXT2)
    IF NULL:COM THEN <<> APPEND:<TEXT2 TEXT1>>
    ELSEIF OR:<ATOM:FIRST:COM REST:COM NOT:ISLAT:FIRST:COM>
        THEN <<><>>
    ELSE EDINSERT:<SEPLINES:FIRST:COM TEXT1 TEXT2>
.
 -=>DOINSERT

DEFINE EDINSERT (NEW TEXT1 TEXT2)
    <<> EDARANGE:<NEW TEXT1 TEXT2>>
.
 -=>EDINSERT


DOINSERT:"(((N E W NL S T U F NL))((A NL)) ((B NL)(END NL))).
 -=> (() ((S T U F NL) (B NL) (END NL) (A NL) (N E W NL)))
```


DOPRINC executes the '=' command.

```
DEFINE DOPRINC (COM TEXT1 TEXT2)
    CHECKP:<COM FIRST:TEXT2 PRINC:<TEXT1 TEXT2>
        APPEND:<TEXT2 TEXT1>>
.
 -=>DOPRINC

DEFINE PRINC (TEXT1 TEXT2)
    PUTNO:GETNO:<TEXT1 TEXT2 2:FINDX:<<END> TEXT1>>
.
 -=>PRINC

DEFINE GETNO (TEXT1 TEXT2 PREFIX)
    IF PREFIX THEN LENGTH:PREFIX
    ELSE PLUS:<LENGTH:TEXT1 LENGTH:2:FINDX:<<END> REST:TEXT2>>
.
 -=>GETNO

DEFINE PUTNO N
    CONCAT:<<NL BLANK> PUTDEC:<N PDWIDTH> <NL>>
.
 -=>PUTNO

DOPRINC:<<> "((A NL)(END NL)) "((B NL)(C NL))>.
 -=> ((NL BLANK BLANK BLANK BLANK BLANK 1 NL) ((B NL) (C NL) (A NL) (END
 NL)))
```

Finding the current line number involves locating line zero in either TEXT1 or TEXT2 and calculating the position of FIRST:TEXT2, the current line, in relation to that line. PRINC and GETNO perform this function.

CHECKP prints the new current line if there is a trailing 'P' command after any command for which it is legal to have a trailing 'P'. If the command which has the trailing 'P' has output for the message file, CHECKP causes this to be printed out also (it puts it in the message file it creates).

```
DEFINE CHECKP (COM LINE INSERT TEXT)
    IF NULL:COM THEN <INSERT TEXT>
    ELSEIF OR:<REST:COM NOT:ATOM:FIRST:COM
               NOT:PRINTQ:FIRST:COM ENDQ:LINE>
          THEN <<><>>
    ELSE <APPEND:<INSERT CONS:<NL LINE>> TEXT>
.
-=>CHECKP
```

```
DOPRINC:"((P) ((A NL)(END NL)) ((B NL)(C NL))).
 -=> ((NL BLANK BLANK BLANK BLANK BLANK 1 NL NL B NL) ((B NL) (C NL) (A
 NL) (END NL)))
```

The rest of the commands require both that TEXT1 not contain line zero and that TEXT2 not begin with line zero (i.e., that the file is not empty). Hence, we have the line

```
ELSEIF ATEND:<TEXT1 TEXT2> THEN
```

in DOCOM1.

```
DEFINE ATEND (TEXT1 TEXT2)
    OR:<NULL:TEXT2 ENDQ:FIRST:TEXT2 BEHINDQ:TEXT1>
.
-=>ATEND
```

DOCHANGE executes a 'C' command. In effect, it deletes the lines indicated by the range specifications and 'appends' some text to what is left of the original editor file. Since the lines indicated in the range specifications are no longer

needed, DOCOM1 just passes REST:TEXT2 to DOCHANGE as the text file.

```
DEFINE DOCHANGE (COM TEXT)
    IF OR:<NULL:COM REST:COM ATOM:FIRST:COM NOT:ISLAT:FIRST:COM>
          THEN <<><>>
    ELSE EDCHANGE:<SEPLINES:FIRST:COM TEXT>
.
-=>DOCHANGE

DEFINE EDCHANGE (NEW TEXT)
    <<> CARRANGE:<NEW TEXT>>
.
-=>EDCHANGE

DEFINE CARRANGE (NEW TEXT)
    CONS:<RAC:NEW APPEND:<TEXT RDC:NEW>>
.
-=>CARRANGE
```

```
DOCHANGE:"(((N E W NL S T U F NL)) ((END NL))).
 -=> (() ((S T U F NL) (END NL) (N E W NL)))
```

DODELETE, which executes the 'D' command, is just like DOCHANGE except that nothing replaces the lines deleted from

the text file. CHECKP handles a possible trailing 'P' command.

```
DEFINE DODELETE (COM TEXT)
    IF ENDQ:FIRST:TEXT
         THEN CHECKP:<COM RAC:TEXT <> RSHIFT:TEXT>
    ELSE CHECKP:<COM FIRST:TEXT <> TEXT>
.
-=>DODELETE


DODELETE:<<> "((A NL) (END NL))>.
 -=> (() ((A NL)(END NL)))
```

DOPRINT, which executes a 'P' command is very straightforward.

```
DEFINE DOPRINT (COM TEXT1 TEXT2)
    IF COM THEN <<><>>
    ELSE <PRINTOUT:<TEXT1 TEXT2> APPEND:<TEXT2 TEXT1>>
.
-=>DOPRINT


DEFINE PRINTOUT (TEXT1 TEXT2)
    CONS:<NL CONCAT:SNOC:<FIRST:TEXT2 TEXT1>>
.
-=>PRINTOUT


DOPRINT:<<> "((O U T NL)(P U T NL)) "((M A Y B E NL)(B A B Y))>.
 -=> ((NL O U T NL P U T NL M A Y B E NL) ((M A Y B E NL) (B A B Y) (O U
 T NL) (P U T NL)))
```


The code for a substitute command, 'S', is lengthy. DOSUBST's argument COM should contain a replacement string and a scan pattern.

```
DEFINE DOSUBST (COM TEXT1 TEXT2)
    IF OR:<NULL:COM NULL:REST:COM ATOM:1:COM ATOM:2:COM>
         THEN <<><>>
    ELSE SETSUB:CHECKG:<RREST:COM 1:COM 2:COM
                        SNOC:<FIRST:TEXT2 TEXT1> REST:TEXT2>
.
-=>DOSUBST
```

CHECKG checks to see if this is a global substitution, in which case all occurrences of OLD in the range of the command are replaced by NEW; if this is not a global substitution, then the first occurrence of OLD in each line in the range is replaced by NEW.

```
DEFINE CHECKG (COM NEW OLD TEXT1 TEXT2)
    IF NULL:COM THEN <COM SUBST:<FALSE NEW OLD TEXT1 TEXT2>>
    ELSEIF SGLOBALQ:FIRST:COM
         THEN <REST:COM SUBST:<TRUE NEW OLD TEXT1 TEXT2>>
    ELSE <COM SUBST:<FALSE NEW OLD TEXT1 TEXT2>>
.
-=>CHECKG


DEFINE SGLOBALQ COM
   AND:<ATOM:COM GVAL:COM>
.
-=>SGLOBALQ
```

SUBST and its help functions do the substitutions.

```
DEFINE SUBST (GLOB NEW OLD TEXT1 TEXT2)
    SARRANGE:<SUBST1:<GLOB NEW OLD TEXT1> TEXT2>
.
-=>SUBST
```

SUBST1 looks for a line containing OLD, by calling FINDX.

```
DEFINE SUBST1 (GLOB NEW OLD TEXT)
    IF NULL:2:FINDX:<OLD TEXT> THEN <<> TEXT>
    ELSE SUBST2:<GLOB NEW OLD 1:FINDX:<OLD TEXT>
                 2:FINDX:<OLD TEXT>>
.
-=>SUBST1

DEFINE SUBST2 (GLOB  NEW OLD TEXT1 TEXT2)
    IF GLOB
        THEN SUBST3:<GLOB NEW OLD
                     SNOC:<ALCHANGE:<NEW OLD FIRST:TEXT2>
                          TEXT1> REST:TEXT2>
    ELSE SUBST3:<GLOB NEW OLD
                 SNOC:<ONECHANG:<NEW OLD FIRST:TEXT2> TEXT1>
                 REST:TEXT2>
.
-=>SUBST2
```

The substitute command is unique in that it is the only command which provides for changes of characters within a line without replacing the contents of the whole line. As the editor is designed mainly to be "line-oriented", one pays a price

for such a command.

The scan pattern, OLD must be matched at least twice for each line in which a substitution is to be made; once to locate the line (FINDX), and once for each substitution made within the line (AMATCHX).

The routine for a global substitution within a line, ALCHANGE, and its helper, AMATCHX, are from chapter 5; so is the function, MAKESUB, which makes substitutions of NEW for OLD.

```
DEFINE ONECHANG (NEW OLD LINE)
    IF NULL:LINE THEN <>
    ELSEIF NULL:1:AMATCHX:<OLD LINE>
        THEN APPEND:<MAKESUB:<NEW 2:AMATCHX:<OLD LINE>>
                     3:AMATCHX:<OLD LINE>>
    ELSE CONS:<FIRST:LINE ONECHANG:<NEW OLD REST:LINE>>
.
-=>ONECHANG

DEFINE SUBST3 (GLOB SUB PAT TEXT1 TEXT2)
    <APPEND          1>:<
    <TEXT1           #>
    SUBST1:<GLOB SUB PAT TEXT2>>
.
-=>SUBST3
```

SARRANGE arranges so that the last line in which a substitution was made is the current line of the edit file.

```
DEFINE SARRANGE ((FOUND NFOUND) TEXT)
    IF NULL:FOUND THEN APPEND:<NFOUND TEXT>
    ELSE CONS:<RAC:FOUND CONCAT:<NFOUND TEXT RDC:FOUND>>
.
-=>SARRANGE
```

```
DEFINE SETSUB (COM TEXT)
    CHECKP:<COM FIRST:TEXT <> TEXT>
.
-=>SETSUB


DOSUBST:"(((N E W)(O L D)) ((O L D NL)(O L D BLANK O L D NL))
          ((A NL) (END NL))).
 -=> (() ((N E W BLANK O L D NL) (A NL) (END NL) (N E W NL)))


DOSUBST:"(((N E W)(O L D) G) ((O L D NL)(O L D BLANK O L D NL))
          ((A NL)(END NL))).
 -=> (() ((N E W BLANK N E W NL) (A NL) (END NL) (N E W NL)))
```

It is true of all commands that an empty text file is returned if there is an error due to bad syntax or some other reason that causes unsuccessful termination of the command. The function, CKERR, which was called by EDIT1, notes this condition if it exists and enters an error message into the message file. The old values of TEXT, FNAME, and LFILES from before execution of the most recent command are used in the recursive

call to EDIT.

If the most recent command was executed successfully (the text file is not empty), then the message created by the individual command is appended to the message file, and the new values for the text file, file name and file library are used in the recursive call to EDIT.

```
DEFINE CKERR ((MESS XTEXT XFNAME XLFILES) TEXT FNAME LFILES LCOMS)
    IF NULL:XTEXT
        THEN <APPEND                        1>:<
             <"(NL E R R O R NL PR)         #>
              EDIT:<LCOMS TEXT FNAME LFILES>>
    ELSE <APPEND2                           1>:<
         <MESS                              #>
         <<NL PR NL>                        #>
          EDIT:<LCOMS XTEXT XFNAME XLFILES>>
.
-=>CKERR
```

Global Commands.

Recall that a global command has the following format

'G' PATTERN COMMAND

or 'X' PATTERN COMMAND

The sequence of steps gone through by the code which executes global commands is as follows:

1) Locate the range over which the global commands are to apply. (Default is the whole file).

```
DEFINE DOGLOB (TARG TEXT GLOBTYPE COM FNAME LFILES LCOMS)
    IF OR:<NULL:COM ATOM:FIRST:COM> THEN <<><>>
    ELSEIF NULL:TARG
         THEN PRCGLOBS:<1:GETTARG:REST:COM 2:GETTARG:REST:COM>
                       GLOBDEF:<GLOBTYPE FIRST:COM TEXT>
                       FNAME LFILES LCOMS>
    ELSE SETGRANG:<GETLIST:<TARG TEXT> GLOBTYPE COM FNAME
                  LFILES LCOMS>

•
-=>DOGLOB
```

Set default range.

```
DEFINE GLOBDEF (GLOBTYPE COM TEXT)
    INORDER:PUTMARKS:<GLOBTYPE COM TEXT>

•
-=>GLOBDEF
```

2) Within this range, mark each line containing an occurrence of PATTERN (or not containing PATTERN if an 'X' is the global specification--GLOBTYPE is true if a 'G' was the global specification ; otherwise it is false) with the global marker, GMARK. We must mark lines (see PUTMARKS, MKGLOB, MARKRANG, GMARK) to avoid the possibility of infinite loops when commands which add text to the edit file are applied globally. For

example, an 'A' command appends a line containing PATTERN to the edit file each time it is called. We must be able to distinguish between such lines and the lines which contained PATTERN at the beginning of execution of the global command.

```
DEFINE SETGRANG ((DEFAULT TEXT1 TEXT2) GLOBTYPE COM
                 FNAME LFILES LCOMS)
    PRCGLOBS:<1:GETTARG:REST:COM 2:GETTARG:REST:COM
              MKGLOB:<GLOBTYPE FIRST:COM TEXT1 TEXT2>
              FNAME LFILES LCOMS>

•
-=>SETGRANG

DEFINE PUTMARKS (GLOBTYPE PAT TEXT)
    MARKRANG:<GLOBTYPE PAT BFIND:<GLOBTYPE PAT TEXT>>

•
-=>PUTMARKS


PUTMARKS:"(TRUE (A) ((A NL) (B NL) (C A NL) (END NL))).
 -=> ((A NL M) (B NL) (C A NL M) (END NL))

PUTMARKS:"(FALSE (A) ((A NL) (B NL) (C A NL) (END NL))).
 -=> ((A NL) (B NL M) (C A NL) (END NL))


DEFINE MARKRANG (GLOBTYPE PAT (TEXT1 TEXT2))
    IF NULL:TEXT2 THEN TEXT1
    ELSE APPEND:<TEXT1 CONS:<GLOMARK:FIRST:TEXT2
                             PUTMARKS:<GLOBTYPE PAT REST:TEXT2>>>
•
-=>MARKRANG

DEFINE GLOMARK LINE
    SNOC:<GMARK LINE>
•
-=>GLOMARK

DEFINE MKGLOB (GLOBTYPE PAT TEXT1 TEXT2)
    APPEND:<PUTMARKS:<GLOBTYPE PAT SNOC:<FIRST:TEXT2 TEXT1>>
            REST:TEXT2>
•
-=>MKGLOB
```

If GLOBTYPE is TRUE then locate a line containing PAT; otherwise, locate a line not containing PAT.

```
DEFINE BFIND (GLOBTYPE PAT TEXT)
    IF NULL:TEXT THEN <<><>>
    ELSEIF BMATCH:<GLOBTYPE PAT FIRST:TEXT>
         THEN <<> TEXT>
    ELSE <CONS            1>:<
         <FIRST:TEXT      #>
          BFIND:<GLOBTYPE PAT REST:TEXT>>
.
-=>BFIND

DEFINE BMATCH (BOOL PAT LINE)
    XOR:<NOT:BOOL MATCH:<PAT LINE>>
.
-=>BMATCH

DEFINE XOR (A B)
    IF A THEN NOT:B ELSE B
.
-=>XOR
```

3) Go through the text file looking for marked lines. When one is found, make this line the current line (it goes at the front of the editor file) and execute COM.

```
DEFINE PRCGLOBS (TARG COM TEXT FNAME LFILES LCOMS)
    IF 2:GETMARK:TEXT
        THEN PROC1COM:<TARG COM REVCONC2:GETMARK:TEXT
                       FNAME LFILES LCOMS>
    ELSE <APPEND        1>:<
         <<NL PR NL>    #>
          EDIT:<LCOMS TEXT FNAME LFILES>>
.
-=>PRCGLOBS

DEFINE GETMARK TEXT
    LOCMRKER:<GMARK TEXT>
.
-=>GETMARK
```



```
GETMARK:"((A NL) (B NL M) (C NL) (END NL)).
-=> (((A NL)) ((B NL) (C NL) (END NL)))
```

PROC1COM calls EXEC to execute a single command; note that the non-global commands, 'E' and 'F' have been bypassed.

```
DEFINE PROC1COM (TARG COM TEXT FNAME LFILES LCOMS)
    IFOKGO:<EXEC:<GETLIST:<TARG TEXT> COM FNAME LFILES TEXT>
          FNAME TARG COM LCOMS>
.
-=>PROC1COM

DEFINE IFOKGO ((MESS TEXT LFILES) FNAME TARG COM LCOMS)
    IF NULL:TEXT THEN <<><>>
    ELSE CKAPPEND:<MESS PRCGLOBS:<TARG COM TEXT FNAME LFILES
                                  LCOMS>>
.
-=>IFOKGO

DEFINE CKAPPEND (MESS (MESSFILE LFILES))
    IF NULL:MESSFILE THEN <MESSFILE LFILES>
    ELSE <APPEND:<MESS MESSFILE> LFILES>
.
-=>CKAPPEND
```

The lookahead function, CKAPPEND, adds whatever output has been created for the message file through single execution of a command only if the whole global command has succeeded. It keys on the message file for the whole global command; if the command has succeeded, there will be at least a new line and prompt added to the message file.

The function which checks the result of a global command, GCKERR, also keys on the message file of the command. If the message file is empty, an error message is issued. GCKERR differs from its companion function, CKERR, in that there is no recursive call to EDIT if there is no error. This is because the recursive call has been sunk within the code which executes the global commands (see PRCGLOBS). Having the recursive call in PRCGLOBS allows data to be added to the message file after each single execution of a global command rather than after a whole lot of data has gathered after several single command executions. This arrangement saves having to do an extra pass through all this data.

```
DEFINE GCKERR ((MESS NEWFILES) TEXT FNAME LFILES LCOMS)
    IF NULL:MESS
         THEN <APPEND              1>!<
              <<NL ERROR NL PR> #>
               EDIT:<LCOMS TEXT FNAME LFILES>>
    ELSE <MESS NEWFILES>
.
-=>GCKERR
```


Trivial Help Functions to PROCFRST.

```
DEFINE CURLNQ CHAR
    SAME:<CHAR CURLN>
.
-=>CURLNQ

DEFINE ENDQ LINE
    SAME:<FIRST:LINE END>
.
-=>ENDQ

DEFINE DIGITQ CHAR
    MEMBER:<CHAR DIGITS>
.
-=>DIGITQ

DEFINE LASTQ CHAR
    SAME:<CHAR LAST>
.
-=>LASTQ

DEFINE FORWARDQ CHAR
    SAME:<CHAR FORE>
.
-=>FORWARDQ

DEFINE BAKWARDQ CHAR
    SAME:<CHAR BACK>
.
-=>BAKWARDQ
```

Trivial Help Functions to DOCOM1.

```
DEFINE READCQ COM
    AND:<COM ATOM:FIRST:COM SAME:<FIRST:COM READC>>
.
 -=>READCQ

DEFINE CHANGEQ CHAR
    SAME:<CHAR CHANGE>
.
 -=>CHANGEQ
.
DEFINE PRINCQ CHAR
    SAME:<CHAR PRINC>
.
 -=>PRINCQ

DEFINE INSERTQ CHAR
    SAME:<CHAR INSERT>
.
 -=>INSERTQ

DEFINE APPENDQ CHAR
    SAME:<CHAR APPEND>
.
 -=>APPENDQ

DEFINE DELETEQ CHAR
    SAME:<CHAR DELETE>
.
 -=>DELETEQ

DEFINE PRINTQ CHAR
    SAME:<CHAR PRINT>
.
 -=>PRINTQ

DEFINE SUBSTQ CHAR
    SAME:<CHAR SUBST>
.
 -=>SUBSTQ
```


Constants Introduced in Chapter 6.

```
DECLARE ERROR "ERROR.
 -=>ERROR


DECLARE APPEND "A.
 -=>A

DECLARE INSERT "I.
 -=>I

DECLARE PRINC "=.
 -=>=

DECLARE CHANGE "C.
 -=>C

DECLARE DELETE "D.
 -=>D

DECLARE PRINT "P.
 -=>P

DECLARE SUBST "S.
 -=>S

DECLARE GMARK "M.
 -=>M

DECLARE PR "PR.
 -=>PR

DECLARE POWIDTH 5.
 -=>5
```

```
DECLARE FIRSTAD "F.
 -=>F

DECLARE CURLN "CURLN.
 -=>CURLN

DECLARE END "END.
 -=>END

DECLARE LAST "LAST.
 -=>LAST

DECLARE FORE "FORE.
 -=>FORE

DECLARE BACK "BACK.
 -=>BACK

DECLARE PLUS "PLUS.
 -=>PLUS

DECLARE MINUS "MINUS.
 -=>MINUS

DECLARE COMMA "COMMA.
 -=>COMMA

DECLARE SEMI "SEMI.
 -=>SEMI

DECLARE GLOBAL "G.
 -=>G

DECLARE NOGLOB "X.
 -=>X

DECLARE PRINTF "F.
 -=>F

DECLARE ENTER "E.
 -=>E

DECLARE WRITEC "W.
 -=>W

DECLARE READC "R.
 -=>R
```


Chapter 7

Introduction.

In this chapter, a simple text formatter is discussed. This formatter "provides a bare minimum of formatting controls, those which we have observed people actually use when preparing documents. It produces output for devices like terminals and line printers, with automatic right margin justification, pagination (skipping over the fold in the paper), page numbering and titling, centering, indenting, and multiple line spacing". [219]

There are a number of formatting commands which a user may intersperse within a text file to achieve the above mentioned ends. "A command consists of a period, a 2-letter name, and perhaps some optional information. Each command must appear at the beginning of a line, with nothing on the line but the command and its arguments.

For instance,

```
.CE
```

centers the next line of output, and

```
.SP 3
```

generates 3 spaces (blank lines)". [219]

The default behavior of the formatter is such that even without explicit formatting commands, a nicely formatted text file will be produced. Default features include "filled" lines, in which as many words as possible are gathered into one output

line before it is output; and automatic right margin justification, so all lines end in the same column.

The Commands.

1. .NF -- (means "no fill") This command negates the default behavior of outputting "filled" lines. As long as "no fill" is in effect, each line of text input to the formatter is passed to the output unaltered. "When a .NF is encountered, there may be a partial line collected but not yet output. The .NF will force this line out before anything else happens. The action of forcing out a partially collected line is called a 'break'." [220]
2. .BR -- (break) causes a "break".
3. .FI -- (fill) causes the "fill" option to be in effect.
4. .SP -- (space) "causes a break, then produces a blank line. To get 'n' blank lines use

   .SP n

   (a space is always required between a command and its argument.) If the bottom of a page is reached before all the blank lines have been printed, the excess ones are thrown away, so that all pages will normally start at the same first line." [220]
5. .LS n -- (line spacing) sets line spacing to 'n' lines. The integer argument, 'n', is optional in this command, and in all other commands having an integer argument.

6. .BP n -- (begin page) causes a break and a skip to the top of a new page. The 'n' is the new page number. By default (if no 'n' is present) the previous page number is incremented by one.
7. .PL n -- (page length) changes the page length to 'n' lines.
8. .CE n -- (center) causes the next 'n' lines of output to be centered on the page. A break is caused and the line-filling option is disabled until all 'n' lines have been centered.
9. .UL n -- (underline) "causes the next 'n' lines to be underlined upon output. But .UL does not cause a break, so words in filled text may be underlined by

   ```
   WORDS AND WORDS AND
   .UL
   LOTS MORE
   WORDS
   ```

   to get WORDS AND WORDS AND LOTS MORE WORDS" [221]
10. .IN n -- (indent) "causes all subsequent output to be indented 'n' positions. (Normally, they are indented by 0)." [222] A break is not caused.
11. .RM n -- (right margin) sets the right margin at column 'n'.
12. .TI n -- (temporary indent) causes a break and "sets the indent to 'n' for one output line only. If 'n' is less than the current indent, the indent is backwards (a 'hanging indent')." [222]

13. .HE line -- (header) causes a header with the contents of 'line' to be put at the top of every page. For example, a header might consist of the following

        # Software Tools Chapter 7

    The '#' is to be replaced by the current page number.

14. .FO line -- (footer) causes a footer with the contents of 'line' to be put on the bottom of every page. The .HE and .FO commands do not cause a break.

"Since absolute numbers are often awkward, FORMAT allows relative values as command arguments. All commands that allow a numeric argument, 'n' will also allow '+n' or '-n' instead, to signify a change in the current value. For instance

```
.RM-10
.IN+10
```

shrinks the right margin by 10 from its current value, and moves the indent 10 spaces further to the right. Thus

```
.RM 10
```

and

```
.RM+10
```

are quite different." [222]

Now, here is a summary of the commands. It indicates for each command, the command mnemonic, the function of the command, the default value if no argument is given for the command, and whether or not the command causes a break.

COMMAND	BREAK?	DEFAULT	FUNCTION
.BP n	yes	n = +1	begin page numbered n
.BR	yes		cause break
.CE n	yes	n = 1	center next n lines
.FI	yes		start filling
.FO	no	empty	footer title
.HE	no	empty	header title
.IN n	no	n = 0	indent n spaces
.LS n	no	n = 1	line spacing is n
.NF	yes		stop filling
.PL n	no	n = 66	set page length to n
.RM n	no	n = 60	set right margin to n
.SP n	yes	n = 1	space down n lines
.TI n	yes	n = 0	temporary indent of n
.UL n	no	n = 1	underline words from next n lines

[223]

If a line of the file begins with a blank, one of two things can occur:

1) If the line has no non-blank characters, then a break is caused and a number of blank lines equal to the current line spacing is output;
2) otherwise, a break is caused and the indent for this line is equal to the number of leading blanks in the line.

The Code.

Due to the large number of user defined options available

with this formatter, an important design decision is how to keep track of all these values. To represent each of these variables as a formal parameter would result in some very bulky, difficult to read function definitions (this language allows no free variables).

The solution which has been chosen is to maintain all the variable parameters and their corresponding values in a list of pairs, which is referred to in the following function definitions as PARAMS. For example, if the current line spacing is two (double-spacing), there will be contained within PARAMS an ordered pair <LS 2> . Simple functions have been written to extract and alter the values of the variables in PARAMS.

At the top level, PARAMS is bound to the constant, CPARAMS, which gives initial default values to the variables contained within.

```
DEFINE FORMAT TEXT
    IF NULL:TEXT THEN <>
    ELSE FORMAT1:<TEXT CPARAMS>
.
-=>FORMAT

DEFINE FORMAT1 (TEXT PARAMS)
    IF NULL:TEXT THEN LASTPAGE:PARAMS
    ELSE ADDLINE:PUTLINE:<FALSE FALSE TEXT PARAMS>
.
-=>FORMAT1
```


The function FORMAT1 and its help function ADDLINE use the constructor APPEND to build a formatted output file. Like other list building functions we have seen, FORMAT1's ground condition is "IF NULL:TEXT". APPEND is applied to the recursive invocation of FORMAT1 within the definition of ADDLINE.

```
DEFINE ADDLINE (INDENT SPACING BREAKQ OLINE TEXT PARAMS)
    IF NULL:OLINE THEN FORMAT1:<TEXT SETPAG:PARAMS>
    ELSE APPEND:<1:PUT:<INDENT SPACING BREAKQ OLINE PARAMS>
                FORMAT1:<TEXT SETPAG:2:PUT:<INDENT SPACING
                                            BREAKQ OLINE PARAMS>>>
.
-=>ADDLINE
```

The arguments to ADDLINE are the output of the function PUTLINE, called in FORMAT1. INDENT is an integer indicating the number of columns the current output line should be indented: SPACING is an integer indicating the current line spacing; eg. if SPACING equals two, double spacing is in effect. BREAKQ indicates whether or not a break has occurred. As has been mentioned, several commands may cause

a break; also a line with leading blanks in the input file causes a break. OLINE is the current output line.

As we shall see, the occurrence of a break results in the satisfaction of a ground condition for PUTLINE. It is possible that as a result of processing two successive input lines containing commands which cause breaks, the output line, OLINE, may be empty. In this case, ADDLINE ignores it

"IF NULL:OLINE THEN FORMAT1:<TEXT SETPAG:PARAMS> ".

The other arguments to ADDLINE are TEXT, which is what remains of the input file after it has been processed by PUTLINE, and PARAMS, the parameter list.

ADDLINE uses the information it receives as input to its help function PUT. Every line added to the output file is processed by PUT and its help functions. PUT is responsible for indenting lines, right justifying "filled" lines, outputting header and footer lines, inserting top and bottom margins, and line spacing. A detailed discussion of PUT and its help functions is deferred until later.

Another help function to ADDLINE, SETPAG resets the temporary indent value and page width previous to each recursive call to FORMAT1. Discussion of the details of SETPAG will also be deferred.

The function PUTLINE provides the arguments for ADDLINE. PUTLINE and its helpers inspect the input file, TEXT, to see if the current input line being inspected contains a command. If it does, a command processing routine is called.

The main function of this routine is to alter the parameter list and to indicate the presence of or absence of a break condition.

If the current line being inspected doesn't contain a command, a function is called to use the data from the input file to build an output line.

```
DEFINE PUTLINE (BREAKQ OUTQ TEXT PARAMS)
    IF NULL:TEXT
        THEN <TIVAL:PARAMS LSVAL:PARAMS TRUE <> TEXT PARAMS>
    ELSEIF BREAKQ
        THEN <TIVAL:PARAMS LSVAL:PARAMS TRUE <> TEXT
              SETBOL:<TRUE PARAMS>>
    ELSEIF OUTQ
        THEN <TIVAL:PARAMS LSVAL:PARAMS FALSE <> TEXT
              SETBOL:<FALSE PARAMS>>
    ELSEIF POS:SPACEP:PARAMS
        THEN <0 1 TRUE <NL> TEXT SUBSPACE:PARAMS>
    ELSE BLDLINE:CKLINE:<PARAMS TEXT>

-=>PUTLINE
```

If PUTLINE detects an empty input file, TEXT, it behaves as though a break has occurred. TIVAL is used to extract from the parameter list the current temporary indent value, and LSVAL extracts the current line spacing. A break flag is set to TRUE. An empty list is returned as the output line.

A break also occurs if the break flag has been set by a lower level function (a command was processed which causes a break, or a line with a leading newline character or leading blanks was encountered). A pair <BOL val> in PARAMS indicates whether or not (VAL= TRUE or FALSE) the first character in the input file is the beginning of an input line. If the

break flag is TRUE, we know we are at the beginning of a line, so the function SETBOL is called to set "VAL" to TRUE.

If OUTQ is TRUE, and BREAKQ is not, it indicates that a "filled" line of text is to be output. This indicates that we are not at the beginning of an input line; SETBOL is called to set the beginning of line indicator to FALSE. For the output line, an empty list is returned.

SPACEP checks a pair in PARAMS <SP n> , returning n. This value is set by the SP, (space) command. If the value is positive, an empty line is to be output; the list <NL> is returned as the output line. The indent value is set to zero, so that PUT and its helpers won't bother to indent the empty line. The line spacing value is set at one, so only one empty line is output. SUBSPACE decrements the line spacing value.

The output of lines as a result of a space command is handled within PUTLINE rather than by a special spacing routine because we don't want to duplicate code already available for outputting lines, and more importantly, because we don't want the routines which handle the processing of commands to be involved with outputting lines.

```
DEFINE SETBOL (BOOL PARAMS)
    REPVAL:<BOOL LOCVALS:<BOL PARAMS>>
.
-=>SETBOL
```



```
DEFINE SUBSPACE PARAMS
    REPVAL:<SUB1:SPACEP:PARAMS LOCVALS:<SP PARAMS>>
.
-=>SUBSPACE
```

When PUTLINE is called within FORMAT1, the initial values for OUTQ and BREAKQ are FALSE, so the call to PUTLINE always results in execution of

BLDLINE:CKLINE: <PARAMS TEXT> .

CKLINE inspects the input file, TEXT, and determines whether to call a command processing routine, DOCOM or to call a routine to build an output line, OTEXT. CKLINE returns a list containing what remains of TEXT after processing by DOCOM or OTEXT, and a list containing an output line, an indicator of whether or not a break has occurred, an indicator of whether a line is to be output, and the parameter list.

```
DEFINE CKLINE (PARAMS TEXT)
    IF NOT:BOLQ:PARAMS
        THEN <FIRST:OTEXT:<TEXT PARAMS> REST:OTEXT:<TEXT PARAMS>>
    ELSEIF COMQ:FIRST:TEXT
        THEN <2:GETLINE:REST:TEXT
              CONS:<<> CONS:<1:DOCOM:<1:GETLINE:REST:TEXT PARAMS>
                               DOCOM:<1:GETLINE:REST:TEXT PARAMS>>>>
    ELSE <FIRST:OTEXT:<TEXT PARAMS> REST:OTEXT:<TEXT PARAMS>>
.
-=>CKLINE

DEFINE COMQ CHAR
    SAME:<CHAR COM>
.
-=>COMQ
```

Before checking to see if a special command character (declared as the constant, COM) begins the output file, CKLINE checks the beginning of line indicator to see if that portion of the input file is at the beginning of an input line

IF NOT:BOLQ:PARAMS . . . .

This assures that COM will have special meaning only at the beginning of a line of input.

BLDLINE uses the results of CKLINE as input to a recursive call to PUTLINE. If any output text has been built by CKLINE and its helpers, this is appended to the output being built by PUTLINE. The text which is appended in the call to PUTLINE is all or a portion of an input line.

ULVAL is called to see if the current input line being inspected is to be underlined. The pair <UL value> is inspected to see if the integer "value" is positive, indicating underlining is in effect. (This value is set by the underline --UL-- command). If the value is positive the function UNDER is called to "underline" the line.

```
DEFINE BLDLINE (TEXT (OLINE BREAKQ OUTQ PARAMS))
    IF NULL:OLINE
        THEN PUTLINE:<BREAKQ OUTQ TEXT PARAMS>
    ELSEIF POS:ULVAL:PARAMS
        THEN <1          1    1     APPEND     1     1>:<
             <#          #    #   UNDER:OLINE ##>
              PUTLINE:<BREAKQ OUTQ TEXT
                       SUBUNDER:<BREAKQ OUTQ PARAMS>>>
    ELSE <1    1    1     APPEND     1     1>:<
         <#    #    #     OLINE      #     #>
          PUTLINE:<BREAKQ OUTQ TEXT PARAMS>>
.
-=>BLDLINE
```



```
DEFINE UNDER OLINE
    IF NLNULL:OLINE THEN <>
    ELSEIF OR:<TABQ:FIRST:OLINE BLANKQ:FIRST:OLINE
               BSQ:FIRST:OLINE>
        THEN CONS:<FIRST:OLINE UNDER:REST:OLINE>
    ELSE CONS:<FIRST:OLINE CONS:<BS CONS:<USCR UNDER:REST:OLINE>>>
.
-=>UNDER

        DEFINE NLNULL L
           OR:<NULL:L NLQ:FIRST:L>
        .
        -=>NLNULL
```

UNDER "converts each character which is not a blank, tab, or backspace into

character BACKSPACE UNDERLINE"

[243]

SUBUNDER is responsible for decrementing the "value" which indicates the number of input lines remaining to be underlined. If the parameter BREAKQ is TRUE, it indicates that the current string being added to the output is being flushed as a result of a break condition, or the "fill" option is not in effect. In this case, we may assume that the end of an input line has been reached, and the "value" associated with underlining is decremented by one.

If the parameter OUTQ is FALSE, it indicates that the fill option is in effect, and that the lower level routines responsible for building "filled" output lines have reached the end of an input line without having built a line of a given length indicated by a pair <LL value> in the parameter list. In this case also, the value indicating the number of lines left to be underlined, is decremented by SUBUNDER.

```
DEFINE SUBUNDER (BREAKQ OUTQ PARAMS)
    IF OR:<BREAKQ NOT:OUTQ>
        THEN REPVAL:<SUB1:ULVAL:PARAMS LOCVALS:<UL PARAMS>>
    ELSE PARAMS
.
-=>SUBUNDER
```

REPVAL replaces a value, OLDVAL, in a parameter pair " <COMTYP OLDVAL> " with a new value NEWVAL and CONSs the pair to the parameter list PARAMS. LOCVALS isolates a parameter pair containing the token COMTYP and returns that pair and the rest of the parameter list PARAMS.

```
DEFINE REPVAL (NEWVAL ((COMTYP OLDVAL) PARAMS))
    CONS:<<COMTYP NEWVAL> PARAMS>
.
-=>REPVAL

DEFINE LOCVALS (COMTYP PARAMS)
    IF SAME:<COMTYP FIRST:FIRST:PARAMS>
        THEN <FIRST:PARAMS REST:PARAMS>
    ELSE <1                 CONS>:<
         <#       FIRST:PARAMS>
          LOCVALS:<COMTYP REST:PARAMS>>
.
-=>LOCVALS
```

DOCOM, a help function to CKLINE, is responsible for command processing. Processing of a command involves (possibly) altering the parameter list; also, processing of some commands may cause a break. DOCOM and its helpers do not have anything to do with the actual building of output text.


```
DEFINE DOCOM (LINE PARAMS)
    PROCCOM:<PARSECOM:LINE PARAMS>
.
-=>DOCOM
```

PARSECOM uses GETCOM to divide an input line into two lists. The first list contains the first two characters in the input line following the already removed COM character (if there aren't at least two non-blank characters immediately following the COM character, an empty list is returned). PARSECOM calls GCOMTYP to translate this list of characters into a command token; a special token, UNKNOWN, is returned for an unrecognized command. The second list returned by GETCOM serves as the argument to the formatting command. This list consists of whatever is left of the input line after the first string of non-blank characters and any trailing whitespace characters.

For example, if the current input line contains

INDENT 7 NL

GETCOM returns

((IN)(7 NL))

This feature gives the user the option of representing a command in a longhand form, and not as just a two character string.

```
DEFINE PARSECOM LINE
    <GCOMTYP        1>:<
     GETCOM:LINE>
.
 -=>PARSECOM


DEFINE GETCOM LINE
    IF OR:<NLNULL:LINE NLNULL:REST:LINE>
          THEN <<><>>
    ELSE <<1:LINE 2:LINE> TRIMLINE:RREST:LINE>
.
 -=>GETCOM

DEFINE TRIMLINE LINE
    EATBLNKS:EATWORD:LINE
.
 -=>TRIMLINE

DEFINE EATWORD LINE
    IF NULL:LINE THEN <>
    ELSEIF OUTWORD:FIRST:LINE THEN LINE
    ELSE EATWORD:REST:LINE
.
 -=>EATWORD

DEFINE EATBLNKS LINE
    IF NULL:LINE THEN <>
    ELSEIF BLANKQ:FIRST:LINE THEN EATBLNKS:REST:LINE
    ELSE LINE
.
 -=>EATBLNKS


DEFINE GCOMTYP STR
    IF NULL:STR THEN UNKNOWN
    ELSEIF ISFO:STR THEN FO
    ELSEIF ISHE:STR THEN HE
    ELSEIF ISIN:STR THEN IN
    ELSEIF ISFI:STR THEN FI
    ELSEIF ISNF:STR THEN NF
    ELSEIF ISLS:STR THEN LS
    ELSEIF ISPL:STR THEN PL
    ELSEIF ISRM:STR THEN RM
    ELSEIF ISUL:STR THEN UL
    ELSEIF ISBP:STR THEN BP
    ELSEIF ISBR:STR THEN BR
    ELSEIF ISCE:STR THEN CE
    ELSEIF ISSP:STR THEN SP
    ELSEIF ISTI:STR THEN TI
    ELSE UNKNOWN
.
 -=>GCOMTYP
```


Here, in one piece, is the definition of the function, PROCCOM, a command processing routine. It is discussed in some detail in the following pages.

```
DEFINE PROCCOM ((COMTYP ARG) PARAMS)
    IF UNKNOWNQ:COMTYP THEN <FALSE PARAMS>
    ELSEIF BRQ:COMTYP THEN <TRUE PARAMS>
    ELSEIF BPQ:COMTYP
         THEN <TRUE SETNUM:<GETVAL:ARG ADD1:PAGENO:PARAMS
                                MINUS:HUGE HUGE
                                LOCVALS:<PN BIGSPACE:PARAMS>>>
    ELSEIF FILLQ:COMTYP
         THEN <TRUE REPVAL:<TRUE LOCVALS:<COMTYP PARAMS>>>
    ELSEIF NFILLQ:COMTYP
         THEN <TRUE REPVAL:<FALSE LOCVALS:<FI PARAMS>>>
    ELSEIF OR:<HEADERQ:COMTYP FOOTERQ:COMTYP>
         THEN <FALSE REPVAL:<STRIP:ARG LOCVALS:<COMTYP PARAMS>>>
    ELSEIF CENTERQ:COMTYP
         THEN <TRUE SETNUM:<GETVAL:ARG CTRDEF 0 HUGE
                               LOCVALS:<COMTYP PARAMS>>>
    ELSEIF ULQ:COMTYP
         THEN <FALSE SETNUM:<GETVAL:ARG ULDEF 1 HUGE
                                LOCVALS:<COMTYP PARAMS>>>
    ELSEIF INDENTQ:COMTYP
         THEN <FALSE SETNUM:<GETVAL:ARG INDEF 0
                                SUB1:RMVAL:PARAMS
                                LOCVALS:<COMTYP PARAMS>>>
    ELSEIF LSQ:COMTYP
         THEN <FALSE SETNUM:<GETVAL:ARG LSDEF 1 HUGE
                                LOCVALS:<COMTYP PARAMS>>>
    ELSEIF PLQ:COMTYP
         THEN <FALSE SETPL:<GETVAL:ARG LOCVALS:<COMTYP PARAMS>>>
    ELSEIF RMQ:COMTYP
         THEN <FALSE SETNUM:<GETVAL:ARG RMDEF ADD1:TIVAL:PARAMS
                                HUGE LOCVALS:<COMTYP PARAMS>>>
    ELSEIF SPQ:COMTYP
         THEN <TRUE SETNUM:<GETVAL:ARG SPDEF 0 MAXSPACE:PARAMS
                               LOCVALS:<COMTYP PARAMS>>>
    ELSE <TRUE SETNUMTI:<GETVAL:ARG TIDEF 0 RMVAL:PARAMS
                           LOCVALS:<TI2 PARAMS>>>
.
 -=>PROCCOM
```

PROCCOM, using as arguments the result of PARSECOM, "(COMTYP ARG)" and the parameter list PARAMS, actually does the command processing. The result of PROCCOM is a boolean indicator to flag a break, and the (possibly) altered parameter list.

If the command type is UNKNOWN, PROCCOM returns FALSE (no break) and the unaltered parameter list.

```
IF  UNKNOWNQ:COMTYP  THEN  <FALSE  PARAMS>
```

If a break command, BR, is encountered, the value TRUE and the unaltered parameter list is returned.

```
IF  BRQ:COMTYP  THEN  <TRUE  PARAMS>
```

If a "begin page", BP, command is encountered, two things are done to the parameter list. The current page number value must be updated, and a value associated with the "space" parameter <SP value> , must be set to a very large number, so the rest of the current page will be filled (by PUTLINE) with blank lines. The break flag is set to TRUE.

BIGSPACE sets the space value to a very big number, HUGE.

```
DEFINE BIGSPACE PARAMS
    REPVAL:<HUGE LOCVALS:<SP PARAMS>>
.
-=>BIGSPACE
```

GETVAL is used to convert the character string ARG, the argument to a formatting command, to an integer (possibly preceded

by a sign) or to an empty list (if ARG does not begin with a numeric character string).

```
DEFINE GETVAL ARG
    IF NLNULL:ARG THEN <>
    ELSEIF OR:<PLUSQ:FIRST:ARG MINUSQ:FIRST:ARG>
        THEN CCONS:<FIRST:ARG NUMVAL:SEPDIGS:REST:ARG>
    ELSE NUMVAL:SEPDIGS:ARG
.
-=>GETVAL
```

SEPDIGS extracts a list of characters which represent decimal digits from the beginning of a string, ARG.

```
DEFINE SEPDIGS ARG
   1:GETDIGS:ARG
.
-=>SEPDIGS
```

If the list returned by SEPDIGS is non-empty, NUMVAL uses CTOI (presented in chapter 2) to convert the list of characters to an integer. NUMVAL returns either an empty list or a list containing an integer.

```
DEFINE NUMVAL ARG
    IF NULL:ARG THEN <>
    ELSE <CTOI:ARG>
.
-=>NUMVAL
```

If GETVAL's ARG begins with a PLUS or MINUS sign, CCONS

either appends the sign to the result of NUMVAL (if non-empty) or leaves the result of NUMVAL unchanged.

```
DEFINE CCONS (A L)
    IF NULL:L THEN <>
    ELSE CONS:<A L>
.
-=>CCONS
```

SETNUM calls NEWVAL to update the value of a parameter pair, which has been extracted from PARAMS by LOCVALS. SETNUM then CONSs the new parameter pair which is the result of NEWVAL, to the parameter list.

```
DEFINE SETNUM (ARG DEFAULT MIN MAX (LVAL PARAMS))
    CONS:<NEWVAL:<ARG DEFAULT MIN MAX LVAL> PARAMS>
.
-=>SETNUM
```

Every parameter pair has associated with it a default value, a minimum possible value and a maximum possible value. In the case of the BP command, the default value is the current page number plus 1, the minimum value is negative HUGE, and the maximum value is HUGE. NEWVAL uses these values (DEFAULT, MIN and MAX) and the value ARG, returned by GETVAL to assign a new value to the parameter pair associated with the command currently being processed-- in this case <PN value> .


```
DEFINE NEWVAL (ARG DEFAULT MIN MAX (COMTYP CURVAL))
    IF ARG
        THEN <COMTYP CKBNDS:<SETVAL:<ARG CURVAL> MIN MAX>>
    ELSE <COMTYP CKBNDS:<DEFAULT MIN MAX>>
.
-=>NEWVAL
```

If ARG is non-empty, SETVAL uses the current value of a parameter pair, CURVAL, and ARG to determine a new value for a parameter pair. Note that if the user supplies no argument to the BP command, or the user supplies unrecognizeable data as an argument, the command is processed using its default value.

```
DEFINE SETVAL (ARG VAL)
    IF PLUSQ:1:ARG
        THEN PLUS:<2:ARG VAL>
    ELSEIF MINUSQ:1:ARG THEN DIFF:<VAL 2:ARG>
    ELSE 1:ARG
.
-=>SETVAL
```

If the value returned by SETVAL is below the minimum possible for a particular command, then CKBNDS replaces the value with the minimum possible value. If the value exceeds the maximum possible, CKBNDS returns the maximum value. Otherwise, CKBNDS leaves the value unchanged. In the case of the BP command, CKBNDS isn't really crucial.

```
DEFINE CKBNDS (VAL MIN MAX)
    IF LESS:<VAL MIN> THEN MIN
    ELSEIF GREAT:<VAL MAX> THEN MAX
    ELSE VAL

'==>CKBNDS
```

Getting back to the definition of PROCCOM, if a fill command, FI, is encountered, the value in the parameter pair <FI value> , is set to TRUE, causing line filling to be put into effect. The nofill command, NF, causes this value to be set to FALSE. Each command causes a break.

```
ELSEIF FILLQ:COMTYP
    THEN <TRUE TRUE REPVAL:<TRUE LOCVALS:<COMTYP PARAMS>>
ELSEIF NFILLQ:COMTYP . . .
```

The header command, HE, causes the pattern for the header line, contained in the parameter pair <HE pattern> to be replaced with a new pattern, obtained from ARG, the argument to a formatting command. The footer command, FO, replaces the pattern contained in the pair <FO pattern> .

To create a header or footer pattern with leading blanks, the argument of the HE or FO command should be preceded with a single or double quote character (SQUOTE or DQUOTE); otherwise, GETCOM will "eat" the leading blanks. PROCCOM calls STRIP to remove a leading quote or double quote character from the list ARG.


```
DEFINE STRIP LINE
    IF NLNULL:LINE THEN <>
    ELSEIF OR:<SQUOTEQ:FIRST:LINE DQUOTEQ:FIRST:LINE>
        THEN REST:LINE
    ELSE LINE

'==>STRIP

DEFINE SQUOTEQ CHAR
    SAME:<CHAR SQUOTE>

'==>SQUOTEQ

DEFINE DQUOTEQ CHAR
    SAME:<CHAR DQUOTE>

'==>DQUOTEQ
```

If a center command, CE, is encountered by PROCCOM, SETNUM is called to update the parameter pair <CE value> , which indicates the number of subsequent output lines to be centered on a page. A break is caused.

SETNUM updates the parameter pair <UL value> if an underline command is processed. No break is caused.

The indent command, IN, causes SETNUM to be invoked to update the parameter pair, <IN value> . As each new output line is begun, the temporary indent value for the next line is determined by the value in the parameter pair. The value indicates the number of columns the next line is to be indented. This command does not cause a break. The default value for the indent command, INDEF, is zero.

The LS command resets the current line spacing. No break is caused. The default value for this command, LSDEF, is 1.

SETPL is called if the page length command, PL, is encountered. SETPL calls SETNUM to reset the parameter pair <PL value> , which indicates the current page length. The line number of the last line of output on a page is obtained from this new page length (PLVAL:PARAMS) by the function SETBOTTM. The value of this bottom line number, contained in the parameter pair <BO value> is equal to the page length minus the number of lines in the bottom margins, MARG3 (which represents the number of blank lines preceding the optional footer line) and MARG4 (which indicates the number of blank lines following the footer, plus 1 if there is a footer line).

```
DEFINE SETPL (ARG (LVAL PARAMS))
    SETBOTTM:SETNUM:<ARG PLDEF SUM:<1 MARG1 MARG2 MARG3 MARG4>
                    HUGE <LVAL PARAMS>>

-=>SETPL

DEFINE SETBOTTM PARAMS
    REPVAL:<DIFF:<PLVAL:PARAMS PLUS:<MARG3 MARG4>>
           LOCVALS:<BO PARAMS>>

-=>SETBOTTM
```

The "right margin" command resets the value in the parameter pair <RM value> , which indicates the number of columns between the left edge of a page and the right margin of an output line. The lower bound for the right margin is the current temporary indent value plus 1 (ADD1:TIVAL:PARAMS), so there will always be at least one column of output per line. No break is caused by the RM command.

The SP command causes a break and resets the value of the parameter pair <SP value> to the value indicated by the argument of the command. As has been mentioned, setting this value to a positive integer (its lower bound and default value are zero) will cause PUTLINE to output that number of blank lines.

A function MAXSPACE determines the upper bound for the value set by the SP command. The upper bound is set so that spacing will not occur past what should be the last printed line on a page (BOTTOM:PARAMS).

```
DEFINE MAXSPACE PARAMS
    IF NOT:LESS:<LINO:PARAMS BOTTOM:PARAMS>
        THEN DIFF:<BOTTOM:PARAMS PLUS:<MARG1 MARG2>>
    ELSE DIFF:<BOTTOM:PARAMS LINO:PARAMS>

-=>MAXSPACE
```

The final line of PROCCOM is used to process the temporary indent (TI) command. A break is caused and the value in a parameter pair, <TI2 value> is updated with a new value determined by SETNUMTI from the current temporary indent value and the value, ARG, supplied as an argument to the TI command. The value in the pair <TI2 value> will then be used to set the temporary indent value after the current output line has been flushed.

```
DEFINE SETNUMTI (ARG DEF MIN MAX (LVAL PARAMS))
    SETNUM:<ARG DEF MIN MAX <<TI2 TIVAL:PARAMS> PARAMS>>

-=>SETNUMTI
```

Now, let us turn to the functions which produce an output line. The function OTEXT is called by CKLINE, if the data currently being inspected in TEXT is not at the beginning of an input line

```
IF  NOT:BOLQ:PARAMS
```

or if the current input line being inspected doesn't begin with the special character, COM. The check for beginning of line comes before the check for the command character so that the command character will only have special meaning at the beginning of a line.

```
DEFINE OTEXT (TEXT PARAMS)
    IF IDMARKQ:PARAMS
        THEN CKFILL:<TEXT ZIPTI:INCLL:REMIDMK:PARAMS>
    ELSEIF OR:<NLQ:FIRST:TEXT BLANKQ:FIRST:TEXT>
        THEN <TEXT <> TRUE TRUE SETIDMK:PARAMS>
    ELSE CKFILL:<TEXT PARAMS>
.
-=>OTEXT
```

OTEXT is responsible for building an output line. The first problem OTEXT handles is the processing of empty lines and lines with leading blanks. If such a line is encountered at the beginning of the input file TEXT

```
OR: <NLQ:FIRST:TEXT  BLANKQ:FIRST:TEXT>
```

a ground condition has occurred. OTEXT returns the input file, an empty list (which is the ground expression for the output line), an indicator that a break has occurred (TRUE),

an indicator that a line is to be output (TRUE-- this value sets a ground condition in the higher level function, PUTLINE) and the parameter list. The value in the parameter pair <IDMARK value> , is set to TRUE, indicating that the next output line is to be manually indented.

The next time OTEXT is invoked, TEXT is unchanged, but OTEXT sees, via IDMARKQ, that a break has already occurred, so a ground condition doesn't exist. The function CKFILL is called to commence with the building of the output line. REMIDMK resets the value of <IDMARK value> to FALSE. ZIPTI sets the temporary indent to zero so that when the output line being built by OTEXT is appended to the output file being built by the higher level functions (ADDLINE, PUT), no extra blanks will be added to the front of the line.

INCLL resets the maximum line length, <LL value> , to its current value plus the temporary indent value.

```
DEFINE REMIDMK PARAMS
    REPVAL:<FALSE LOCVALS:<IDMARK PARAMS>>
.
-=>REMIDMK

DEFINE SETIDMK PARAMS
    REPVAL:<TRUE LOCVALS:<IDMARK PARAMS>>
.
-=>SETIDMK

DEFINE ZIPTI PARAMS
    REPVAL:<0 LOCVALS:<TI PARAMS>>
.
-=>ZIPTI
```

```
DEFINE INCLL PARAMS
    REPVAL:<PLUS:<LLVAL:PARAMS TIVAL:PARAMS> LOCVALS:<LL PARAMS>>
.
-=>INCLL
```

Centered lines are not "filled" even if the fill option is in effect. It is due to this that CKFILL first checks the parameter pair <CE value> to see if the next output line is to be centered.

IF POS:CEVAL:PARAMS

If this next output line is to be centered, the first line in the input file is returned as the output line, and the break flag and output flag are set to TRUE. SETCTR is called to subtract one from the value in the pair, <CE value> , and to set the temporary indent value so that the output line will be centered between the left and right margins. The right margin value is extracted by the function, RMVAL.

```
DEFINE CKFILL (TEXT PARAMS)
    IF POS:CEVAL:PARAMS
        THEN <2:GETLINE:TEXT 1:GETLINE:TEXT TRUE TRUE
              SETCTR:<1:GETLINE:TEXT PARAMS>>
    ELSEIF ALLBLNKS:TEXT
        THEN <2:GETLINE:TEXT <NL> TRUE TRUE PARAMS>
    ELSEIF FILVAL:PARAMS
        THEN PUTWORDS:<TEXT LLVAL:PARAMS PARAMS>
    ELSE <2:GETLINE:TEXT 1:GETLINE:TEXT TRUE TRUE PARAMS>
.
-=>CKFILL

DEFINE SETCTR (LINE PARAMS)
    REPVAL:<SUB1:CEVAL:PARAMS
            LOCVALS:<CE REPVAL:<CENTER:<LINE TIVAL:PARAMS
                                        RMVAL:PARAMS>
                      LOCVALS:<TI PARAMS>>>>>
.
-=>SETCTR
```



```
DEFINE CENTER (LINE INDENT RMARG)
    MAX:<DIV:<DIFF:<PLUS:<INDENT RMARG> OWIDTH:LINE> 2> 0>
.
-=>CENTER
```

CKFILL calls ALLBLNKS to see if the first line in the text file contains only blanks, or consists only of a newline character. If this is so, a break is caused, and a list containing a newline character is returned as the output line.

```
DEFINE ALLBLNKS TEXT
    IF NLNULL:TEXT THEN TRUE
    ELSEIF NOT:BLANKQ:FIRST:TEXT THEN FALSE
    ELSE ALLBLNKS:REST:TEXT
.
-=>ALLBLNKS
```

FILVAL:PARAMS returns TRUE if the "fill" option is in effect. If filling is in effect, PUTWORDS is called to output data from the input file a word at a time; otherwise, the first line of the input file is returned as the output line, and a break is caused.

```
DEFINE PUTWORDS (TEXT LINLEN PARAMS)
    IF ZEROP:LINLEN
        THEN <TEXT <NL> FALSE TRUE SETLEN:<0 PARAMS>>
    ELSEIF NOT:BLANKQ:FIRST:TEXT
        THEN ADWORDS:<LINLEN TEXT PARAMS>
    ELSE <1     CONS    1   1   1>:<
          <#    BLANK   #   #   #>
          PUTWORDS:<REST:TEXT SUB1:LINLEN PARAMS>>
.
-=>PUTWORDS
```

PUTWORDS takes as arguments, LINLEN, an indicator of the number of characters left to be output in the current output line, TEXT, the input file, and the parameter list, PARAMS.

PUTWORDS first appends leading blanks to the output one at a time, decrementing LINLEN by one each time. It is possible that an input line, while not containing only blanks, will have a number of leading blanks greater than or equal to the maximum line length of a filled output line; hence, the existence of the line

```
IF  ZEROP:LINLEN
    THEN  <TEXT <NL> FALSE  TRUE  SETLEN: <0 PARAMS>>
```

The list <NL> is returned as the output line; FALSE indicates no break has occurred, and TRUE indicates a line is to be output.

The function SETLEN resets the value of the parameter pair, <LL value> to the current value of LINLEN. A higher level function, PUT, uses this value in right justifying "filled" output lines.

```
DEFINE SETLEN (LINLEN PARAMS)
    REPVAL:<LINLEN LOCVALS:<LL PARAMS>>

'=>SETLEN
```


Once leading blanks are accounted for, ADWORDS is called to begin adding "words" to the output. ADWORDS uses GETWORD to extract a word from TEXT, the input file. AD1WORD, a helper to ADWORDS, checks to see if the new word can be added to the output without exceeding the specified length for an output line. If not, a ground condition is reached. Otherwise, CAPPEND2 is called to add the new word and one blank to the output. ADWORDS is called to continue processing the input; EATBLNKS removes extra blanks between words.

```
DEFINE ADWORDS (LINLEN TEXT PARAMS)
    IF NULL:TEXT
        THEN <<> <NL> TRUE TRUE PARAMS>
    ELSEIF NLQ:FIRST:TEXT
        THEN <REST:TEXT <> FALSE FALSE
                SETLEN:<LINLEN SETBOL:<TRUE PARAMS>>>
    ELSE AD1WORD:<LINLEN GETWORD:TEXT TEXT PARAMS>

'=>ADWORDS

DEFINE AD1WORD (LINLEN (WORD TEXT) XTEXT PARAMS)
    IF MINUSP:DIFF:<LINLEN OWIDTH:WORD>
        THEN <XTEXT <NL> FALSE TRUE SETLEN:<LINLEN PARAMS>>
    ELSE <1   CAPPEND2        1    1    1>:<
         <#   WORD            #    #    #>
         <#   <BLANK>         #    #    #>
          ADWORDS:<DIFF:<LINLEN OWIDTH:WORD> EATBLNKS:TEXT
                   PARAMS>>

'=>AD1WORD
```

ADWORDS terminates if the input file is empty (a break is caused), or a newline character is encountered (SETBOL is called to indicate that a new input line is to be processed).

CAPPEND2 takes three lists as arguments. If the third is null or begins with a newline, then the first is appended to the third; otherwise, the first is appended to the second, which is then appended to the third. The point of all this is to avoid adding a trailing blank to the last word in an output line.

```
DEFINE CAPPEND2 (L1 L2 L3)
    IF NLNULL:L3
        THEN APPEND:<L1 L3>
    ELSE APPEND2:<L1 L2 L3>

-=>CAPPEND2
```

OWIDTH determines the "output width" of a string, the number of characters in a string as it would appear after being displayed on an output device. An NL character adds nothing to the output width of a string, and a backspace character subtracts one from the output width.

```
DEFINE OWIDTH STRING
    IF NULL:STRING THEN 0
    ELSEIF BSQ:FIRST:STRING
        THEN MAX:<0 SUB1:OWIDTH:REST:STRING>
    ELSEIF NLQ:FIRST:STRING
        THEN OWIDTH:REST:STRING
    ELSE ADD1:OWIDTH:REST:STRING

-=>OWIDTH
```

As has been mentioned earlier, the output of PUTLINE and its helpers serves as the input to the function ADDLINE. If the output line built is non-empty, ADDLINE calls PUT to do

additional processing of the output line before appending it to the output file.

```
DEFINE PUT (INDENT SPACING BREAKQ OLINE PARAMS)
    IF BREAKQ
        THEN  HEADQ:<INDENT SPACING OLINE PARAMS>
    ELSE HEADQ:<INDENT SPACING 1:SPREAD:<OLINE LLVAL:PARAMS PARAMS>
               2:SPREAD:<OLINE LLVAL:PARAMS PARAMS>>

-=>PUT
```

If a break condition exists (if BREAKQ is TRUE) no further processing of the output line, OLINE, occurs. HEADQ is called to add a top page margin and a header line, if appropriate.

If BREAKQ is FALSE, a filled line is being output. SPREAD is called to right justify a filled line. SPREAD right justifies a line by adding extra blanks between words in the line until it fills the space between the right and left margins. The value in the parameter pair <LL PARAMS> , extracted by the function LLVAL, is the number of blanks needed to fill the output line.

```
DEFINE SPREAD (LINE NEXTRA PARAMS)
    IF OR:<NLNULL:LINE ONEWORD:LINE NOT:POS:NEXTRA>
        THEN <LINE PARAMS>
    ELSE SKIPBLKS:<NEXTRA LINE PARAMS>

-=>SPREAD


DEFINE ONEWORD LINE
    NLNULL:EATBLNKS:EATWORD:EATBLNKS:LINE

-=>ONEWORD
```

If a line contains only one word, or the line is empty, or NEXTRA, the number of blanks to be added to an output line is zero, then the line is unaltered. Otherwise, SKIPBLKS is called to skip the leading blanks in the output line.

```
DEFINE SKIPBLKS (NEXTRA LINE PARAMS)
    IF NLNULL:LINE
        THEN <LINE PARAMS>
    ELSEIF BLANKQ:FIRST:LINE
        THEN <CONS          1>:<
             <FIRST:LINE    #>
              SKIPBLKS:<NEXTRA REST:LINE PARAMS>>
    ELSE <CONDREV:<DIRQ:PARAMS SPREAD1:<CONDREV:<DIRQ:PARAMS LINE>
                                       NEXTRA>> CHANGDIR:PARAMS>
.
 -=>SKIPBLKS
```

As Kernighan and Plauger have done, I have written SPREAD and its helpers so that for each output line, extra blanks are added between words starting alternatively from the left side or the right side of a line. A parameter pair <DIR value> indicates whether blanks are to be added between words going from left to right (if "value" is LEFT) or right to left ("value" is RIGHT). CHANGDIR is called to change the value of <DIR value> each time a filled line is output. CONDREV reverses a line if the value of DIR is RIGHT.

```
    DEFINE CHANGDIR PARAMS
        IF LEFTQ:DIRQ:PARAMS
            THEN REPVAL:<RIGHT LOCVALS:<DIR PARAMS>>
        ELSE REPVAL:<LEFT LOCVALS:<DIR PARAMS>>
    .
     -=>CHANDIR

    DEFINE LEFTQ DIR
        SAME:<DIR LEFT>
    .
     -=>LEFTQ
```



```
DEFINE CONDREV (DIR LINE)
    IF RIGHTQ:DIR THEN REVERSE:LINE
    ELSE LINE
.
 -=>CONDREV

DEFINE RIGHTQ DIR
    SAME:<DIR RIGHT>
.
 -=>RIGHTQ
```

The point of alternating the direction from which blanks are added between words in the output lines is to "avoid 'rivers' of whitespace down one margin or another" [240] . SPREAD1, with its helpers SLIPBLKS and ADDBLNK, actually "fills" the output line by adding blanks between the words in the line.

```
DEFINE SPREAD1 (LINE NEXTRA)
    IF OR:<NLNULL:LINE ZEROP:NEXTRA> THEN LINE
    ELSE SPREAD1:SLIPBLKS:<LINE NEXTRA>
.
 -=>SPREAD1

DEFINE SLIPBLKS (LINE NEXTRA)
    IF OR:<NLNULL:LINE ZEROP:NEXTRA> THEN <LINE NEXTRA>
    ELSEIF BLANKQ:FIRST:LINE
        THEN ADDBLNK:<LINE SUB1:NEXTRA>
    ELSE <CONS          1>:<
         <FIRST:LINE    #>
          SLIPBLKS:<REST:LINE NEXTRA>>
.
 -=>SLIPBLKS

DEFINE ADDBLNK (LINE NEXTRA)
    IF NLNULL:LINE THEN <LINE NEXTRA>
    ELSEIF BLANKQ:FIRST:LINE
        THEN <CONS          1>:<
             <FIRST:LINE    #>
              ADDBLNK:<REST:LINE NEXTRA>>
    ELSE <CONS          1>:<
         <BLANK         #>
          SLIPBLKS:<LINE NEXTRA>>
.
 -=>ADDBLNK
```

HEADQ is called by PUT to output a top page margin and header line, if the current line number is greater than "BOTTOM:PARAMS", the line number specified in the parameter list as the last printed line on a page.

```
DEFINE HEADQ (INDENT SPACING OLINE PARAMS)
    IF GREAT:<LINO:PARAMS BOTTOM:PARAMS>
        THEN PUTHEAD:<PHEAD:PARAMS INDENT SPACING OLINE>
    ELSE APINDENT:<INDENT SPACING OLINE PARAMS>
.
-=>HEADQ


DEFINE PUTHEAD ((HDBLOK PARAMS) INDENT SPACING OLINE)
    <APPEND                    1>:<
    <HDBLOK                    #>
     APINDENT:<INDENT SPACING OLINE PARAMS>>
.
-=>PUTHEAD
```

PHEAD sandwiches a header line between two margins, one having MARG1 blank lines, and the other containing MARG2 blank lines. The header line is considered to be the last line of the top MARG1 lines, so a header can be suppressed by using the value zero for the constant, MARG1. PHEAD sets the current line number (the value in the parameter pair <LN value>) to MARG1+MARG2+1.


```
DEFINE PHEAD PARAMS
    IF ZEROP:MARG1
        THEN <SKIP:MARG2 REPVAL:<ADD1:MARG2 LOCVALS:<LN PARAMS>>>
    ELSE <APPEND2                                          1>:<
         <PUTTL:<PAGENO:PARAMS HEADER:PARAMS>              #>
         <SKIP:MARG2    REPVAL:<ADD1:PLUS:<MARG1 MARG2>
                        LOCVALS:<LN PARAMS>>>>

.
-=>PHEAD

DEFINE APPEND2 (L1 L2 L3)
    APPEND:<L1 APPEND:<L2 L3>>

.
-=>APPEND2
```

PUTTL creates a header line from the value in the parameter pair <HE value> , extracted from the parameter list by the function HEADER. The current page number (PAGENO:PARAMS) is substituted for any occurrence of the constant, PMARK, in the header line.

```
DEFINE PUTTL (PAGENO LINE)
    IF NLNULL:LINE THEN <NL>
    ELSEIF PMARKQ:FIRST:LINE
        THEN APPEND:<PUTDEC:<PNWIDTH PAGENO> PUTTL:REST:LINE>
    ELSE CONS:<FIRST:LINE PUTTL:REST:LINE>
.
-=>PUTTL


DEFINE PMARKQ CHAR
    SAME:<CHAR PMARK>
.
-=>PMARKQ
```

APINDENT, called in HEADQ and PUTHEAD, is responsible for indenting a line. It simply CONSs INDENT blanks to an output line.

```
DEFINE APINDENT (INDENT SPACING OLINE PARAMS)
    IF ZEROP:INDENT
        THEN <APPEND          1>:<
             <OLINE           #>
              LSPACE:<SPACING PARAMS>>
    ELSE <CONS           1>:<
         <BLANK          #>
          APINDENT:<SUB1:INDENT SPACING OLINE PARAMS>>
.
-=>APINDENT
```

LSPACE uses SKIP to effect line spacing. Note that LSPACE never adds spaces past the limit, BOTTOM:PARAMS. ENDPUT is called to put out a footer line and bottom margins, if the end of a printed page has been reached.

```
DEFINE LSPACE (SPACING PARAMS)
   <APPEND                                                          1>:<
   <SKIP:MIN:<SUB1:SPACING DIFF:<BOTTOM:PARAMS ADD1:LINO:PARAMS>>   #>
    ENDPUT:<PLUS:<LINO:PARAMS SPACING> PARAMS>>
.
-=>LSPACE

DEFINE ENDPUT (LNUM PARAMS)
  IF NOT:LESS:<LNUM BOTTOM:PARAMS>
      THEN PFOOT:PARAMS
  ELSE <<> REPVAL:<LNUM LOCVALS:<LN PARAMS>>>
.
-=>ENDPUT
```

ENDPUT calls PFOOT to output the footer lines and bottom margin, if necessary; otherwise ENDPUT just updates the current line number.

```
DEFINE PFOOT PARAMS
    <APPEND                          1>:<
    <SKIP:MARG3                      #>
    <PFOOT1:<PAGENO:PARAMS  PARAMS> UPLINPAG:LOCVALS:<PN PARAMS>>>
.
-=>PFOOT
```



```
DEFINE PFOOT1 (PAGENO PARAMS)
    IF ZEROP:MARG4 THEN <>
    ELSE APPEND:<PUTTL:<PAGENO FOOTER:PARAMS>
                 SKIP:SUB1:MARG4>
.
-=>PFOOT1
```

PFOOT calls UPLINPAG to add one to the current page number (the value in the parameter pair <PN value>) and to set the current line number to HUGE. Setting the current line number to HUGE assures that the next line of output will cause a new page to begin (even if a PL command sets the page length to a higher value than it is currently).

```
DEFINE UPLINPAG ((TOK VAL) PARAMS)
    CONS:<<TOK ADD1:VAL> REPVAL:<HUGE LOCVALS:<LN PARAMS>>>
.
-=>UPLINPAG
```

As was the case with the header line, the footer line is sandwiched between two margins, one containing MARG3 lines and the other containing MARG4 lines. The footer line is built from the value in the parameter pair <FO value> . The footer line is the first line in the bottom margin, so setting MARG4 to zero will suppress output of the footer line.

The function, LASTPAGE, is called by FORMAT1 to fill the last page in the output file with blank lines. If the end of a page hasn't already been reached, LSPACE is called to fill the page.

```
DEFINE LASTPAGE PARAMS
    IF GREAT:<LINO:PARAMS BOTTOM:PARAMS>
        THEN <>
    ELSE 1:LSPACE:<HUGE PARAMS>

'-=>LASTPAGE
```

Recall that in the function ADDLINE, a call is made to the function SETPAG to set the temporary indent and line length before output of a new line begins.

```
DEFINE SETPAG  PARAMS
    SETLL:SETTIVAL:PARAMS

'-=>SETPAG
```

SETTIVAL sets the new value for the temporary indent, indicated by the value in the parameter pair <TI value> . If the "value" in the parameter pair <TI2 value> (set by the TI command) is an integer, this value is used for the new temporary indent value; otherwise, the value in the parameter pair <IN value> , is used for the new temporary indent.

```
DEFINE SETTIVAL PARAMS
    SETIVAL1:LOCVALS:<TI2 PARAMS>

'-=>SETTIVAL

DEFINE SETIVAL1 ((TOK VAL) PARAMS)
    IF NOT:VAL
        THEN REPVAL:<INDENTP:PARAMS
                    LOCVALS:<TI CONS:<<TOK VAL> PARAMS>>>
    ELSE REPVAL:<VAL LOCVALS:<TI CONS:<<TOK FALSE> PARAMS>>>

'-=>SETIVAL1
```


SETLL sets the value of the parameter pair <LL value> , the line length, using the values for the right margin (RMVAL:PARAMS) and the temporary indent.

```
DEFINE SETLL PARAMS
    REPVAL:<DIFF:<RMVAL:PARAMS TIVAL:PARAMS>
                LOCVALS:<LL PARAMS>>

'-=>SETLL
```

Functions Used To Extract Values From The Parameter List.

The function, GET, is used as a help function to functions which extract values from the parameter list. Its arguments are a token, TOK, and the parameter list. It returns a value from a parameter pair <TOK value> .

```
DEFINE GET (TOK PARAMS)
    IF SAME:<TOK 1:FIRST:PARAMS>
        THEN 2:FIRST:PARAMS
    ELSE GET:<TOK REST:PARAMS>

'-=>GET
```

Here are the definitions of the functions which call GET.

```
DEFINE DIRQ PARAMS
    GET:<DIR PARAMS>

'-=>DIRQ

DEFINE PAGENO PARAMS
    GET:<PN PARAMS>

'-=>PAGENO
```

```
DEFINE BOLQ PARAMS
    GET:<BOL PARAMS>
.
-=>BOLQ

DEFINE HEADER PARAMS
    GET:<HE PARAMS>
.
-=>HEADER

DEFINE FOOTER PARAMS
    GET:<FO PARAMS>
.
-=>FOOTER

DEFINE LINO PARAMS
    GET:<LN PARAMS>
.
-=>LINO

DEFINE BOTTOM PARAMS
    GET:<BO PARAMS>
.
-=>BOTTOM

DEFINE INDENTP PARAMS
    GET:<IN PARAMS>
.
-=>INDENTP

DEFINE SPACEP PARAMS
    GET:<SP PARAMS>
.
-=>SPACEP

DEFINE TIVAL PARAMS
    GET:<TI PARAMS>
.
-=>TIVAL

DEFINE LSVAL PARAMS
    GET:<LS PARAMS>
.
-=>LSVAL

DEFINE CEVAL PARAMS
    GET:<CE PARAMS>
.
-=>CEVAL
```



```
DEFINE FILVAL PARAMS
    GET:<FI PARAMS>
.
-=>FILVAL

DEFINE LLVAL PARAMS
    GET:<LL PARAMS>
.
-=>LLVAL

DEFINE ULVAL PARAMS
    GET:<UL PARAMS>
.
-=>ULVAL

DEFINE RMVAL PARAMS
    GET:<RM PARAMS>
.
-=>RMVAL

DEFINE IDMARKQ PARAMS
    GET:<IDMARK PARAMS>
.
-=>IDMARKQ

DEFINE PLVAL PARAMS
    GET:<PL PARAMS>
.
-=>PLVAL
```

Trivial Help Functions To GCOMTYP.

The following functions check a two character string, STR, for a particular combination of characters. They use the help function MATCH2.

```
DEFINE ISFO STR
    MATCH2:<STR "(F O)>.

'-=>ISFO

DEFINE ISHE STR
    MATCH2:<STR "(H E)>

'-=>ISHE

DEFINE ISIN STR
    MATCH2:<STR "(I N)>

'-=>ISIN

DEFINE ISFI STR
    MATCH2:<STR "(F I)>

'-=>ISFI

DEFINE ISNF STR
    MATCH2:<STR "(N F)>

'-=>ISNF

DEFINE ISLS STR
    MATCH2:<STR "(L S)>

'-=>ISLS

DEFINE ISPL STR
    MATCH2:<STR "(P L)>

'-=>ISPL

DEFINE ISRM STR
    MATCH2:<STR "(R M)>

'-=>ISRM
```



```
DEFINE ISUL STR
    MATCH2:<STR "(U L)>

'-=>ISUL

DEFINE ISBP STR
    MATCH2:<STR "(B P)>

'-=>ISBP

DEFINE ISBR STR
    MATCH2:<STR "(B R)>

'-=>ISBR

DEFINE ISCE STR
    MATCH2:<STR "(C E)>

'-=>ISCE

DEFINE ISSP STR
    MATCH2:<STR "(S P)>

'-=>ISSP

DEFINE ISTI STR
    MATCH2:<STR "(T I)>

'-=>ISTI


DEFINE MATCH2 (STRING (A B))
    IF OR:<NULL:STRING NULL:REST:STRING> THEN FALSE
    ELSE AND:<SAME:<1:STRING A> SAME:<2:STRING B>>

'-=>MATCH2
```

Trivial Help Functions To PROCCOM.

The following commands inspect a token, COMTYP, and return TRUE if the token matches some particular constant.

```
DEFINE UNKNOWNQ COM
    SAME:<CHAR UNKNOWN>
.
 -=>UNKNOWNQ

DEFINE BRQ COM
    SAME:<COM BR>
.
 -=>BRQ

DEFINE FILLQ COM
    SAME:<COM FI>
.
 -=>FILLQ

DEFINE NFILLQ COM
    SAME:<COM NF>
.
 -=>NFILLQ

DEFINE HEADERQ COM
    SAME:<COM HE>
.
 -=>HEADERQ

DEFINE FOOTERQ COM
    SAME:<COM FO>
.
 -=>FOOTERQ

DEFINE CENTERQ COM
    SAME:<COM CE>
.
 -=>CENTERQ

DEFINE ULQ COM
    SAME:<COM UL>
.
 -=>ULQ

DEFINE BPQ COM
    SAME:<COM BP>
.
 -=>BPQ
```



```
DEFINE INDENTQ COM
    SAME:<COM IN>
.
 -=>INDENTQ

DEFINE LSQ COM
    SAME:<COM LS>
.
 -=>LSQ

DEFINE PLQ COM
    SAME:<COM PL>
.
 -=>PLQ

DEFINE RMQ COM
    SAME:<COM RM>
.
 -=>RMQ

DEFINE SPQ COM
    SAME:<COM SP>
.
 -=>SPQ
```

Constants Introduced In Chapter 7.

```
DECLARE CTRDEF 1.
 -=>1

DECLARE ULDEF 1.
 -=>1

DECLARE INDEF 0.
 -=>0

DECLARE LSDEF 1.
 -=>1

DECLARE RMDEF 60.
 -=>60

DECLARE SPDEF 1.
 -=>1

DECLARE TIDEF 0.
 -=>0

DECLARE IDMARK "IDMARK.
 -=>IDMARK

DECLARE LL "LL.
 -=>LL

DECLARE DIR "DIR.
 -=>DIR

DECLARE LEFT "LEFT.
 -=>LEFT

DECLARE RIGHT "RIGHT.
 -=>RIGHT

DECLARE LN "LN.
 -=>LN

DECLARE PN "PN.
 -=>PN
DECLARE TI2 "TI2.
 -=>TI2

DECLARE BOL "BOL.
 -=>BOL

DECLARE PNWIDTH 5.
 -=>5
```



```
DECLARE SP "SP.
 -=>SP

DECLARE UL "UL.
 -=>UL

DECLARE FO "FO.
 -=>FO

DECLARE HE "HE.
 -=>HE

DECLARE IN "IN.
 -=>IN

DECLARE FI "FI.
 -=>FI

DECLARE NF "NF.
 -=>NF

DECLARE LS "LS
 -=>LS

DECLARE PL "PL.
 -=>PL

DECLARE RM "RM.
 -=>RM

DECLARE UNKNOWN "UNKNOWN.
 -=>UNKNOWN

DECLARE BP "BP.
 -=>BP

DECLARE BR "BR.
 -=>BR

DECLARE CE "CE.
 -=>CE

DECLARE TI "TI.
 -=>TI

DECLARE SQUOTE "SQUOTE.
 -=>SQUOTE

DECLARE DQUOTE "DQUOTE.
 -=>DQUOTE

DECLARE MARG1 3.
 -=>3
```

```
DECLARE MARG2 3.
 -=>3

DECLARE MARG3 3.
 -=>3

DECLARE MARG4 3.
 -=>3

DECLARE BO "80.
 -=>80

DECLARE PLDEF 66.
 -=>66

DECLARE HUGE 64000.
 -=>64000

DECLARE PMARK "PMARK.
 -=>PMARK

DECLARE CPARAMS
     <<CE 0><UL 0><FI TRUE><PL 66><RM 60><HE <>><FO <>>
      <IN 0><TI 0><LS 1><SP 0><BO 60><DIR RIGHT><TI2 FALSE>
      <LL 60><BOL TRUE><IDMARK FALSE><PN 0><LN HUGE>>
.
 -=>((CE 0)(UL 0)(FI TRUE)(PL 66)(RM 60)(HE ())(FO ())
     (IN 0)(TI 0)(LS 1)(SP 0)(BO 60)(DIR RIGHT)(TI2 FALSE)
     (LL 60)(BOL TRUE)(IDMARK FALSE)(PN 0)(LN HUGE))


DECLARE USCR "USCR.
 -=>USCR
```


Chapter 8

In this chapter, a macro processor is presented. The simplest function of this processor is text replacement. It allows us to define character strings as "macros". When the macro processor encounters one of these macros in an input file, the macro is replaced by a defining string of characters.

The macro processor has a built-in function, invoked when the character string, "DEFINE" is encountered, to create macro definitions. "This lets us say, for instance,

```
DEFINE (EOF, -1)
```

and thereafter have all occurrences of EOF (in the input file) replaced by '-1'." [251]

It is possible that the definition of a macro may itself be a macro. If that is the case, the replacement string for the macro is "expanded" -- replaced by a defining character string -- both at the time the macro is defined, and at any time the macro may subsequently be expanded.

To illustrate this behavior, consider the situation in which the following five character strings occur in an input file in the order presented here

```
DEFINE (X, Y)
DEFINE (Z, X)
Z
DEFINE (Y, 1)
Z.
```

The first time the character "Z" is encountered in the input, (see the third line in the example above) it is replaced by the character, "Y". The second time a "Z" is encountered, "Y" has been defined by the replacement string "1", and so "Z" expands to a "1".

Of course, if macros can expand to still other macros, there is a possibility that if unchecked, the expansion process could go on infinitely. For this reason the macro processor places a limit upon the number of macro expansions which can occur in the replacement of an individual character string. If the number of macro expansions exceeds this limit, the program aborts and returns an error message.

In addition to performing simple string replacement, this macro processor allows for the expansion of macros with arguments. Kernighan and Plauger present the example in which all occurrences of the string, "SKIPBL are replaced by the structured-Fortran statement,

"WHILE (ARRAY(X) == BLANK|ARRAY(X) == TAB) X = X+1"

"The syntax for specifying macros with arguments is an extension of what we used before:

DEFINE (name, replacement text)

defines name. This time, however, any occurrence in the replacement text of $n, where n is (a string of decimal digits), will be replaced by the nth argument when the macro is actually called. Thus

DEFINE(SKIPBL,WHILE($1($2) == BLANK|$1($2) == TAB)$2 = $2+1)

defines the SKIPBL macro." [265]

As is the case with macros without arguments, the replacement string for a macro with arguments is expanded both at the time the macro is defined, and at any time the macro is subsequently expanded.

The macro processor allows a means for delaying the expansion of a character string "so input can be treated as literal text when necessary. In our convention, any input surrounded by '[' and ']' is left absolutely alone, except that one level of '[' and ']' is stripped off." [268] Kernighan and Plauger point out that this would make possible the use of a macro 'D', written as

"DEFINE(D,[DEFINE($1,$2)])

The replacement text for D, protected by the brackets is literally DEFINE($1,$2)" so "when we say

D(A,BC)

... A is defined to be BC." [268]

As enhancements, the macro processor has other special functions besides "define". One, "IFELSE" takes four character strings as arguments. The first string is compared to the second. If they are equal, the third string is added to the output; otherwise, the fourth string is added to the output. The first and second character strings are expanded before the comparison takes place. The output string is also expanded before it is added to the output file. For example

DEFINE(COMPARE,[IFELSE($1,$2,YES,NO)])

defines COMPARE as a two argument macro returning YES if its

arguments are the same, and NO if they're not." [276] The brackets in the definition of COMPARE prevent evaluation of the IFELSE function as COMPARE is being defined.

The special function, INCR, has one argument. When the macro processor encounters some string

INCR(X)

it "converts the string 'X' to a number, adds one to it, and returns that as its replacement text (as a character string)." [276]

"The final built-in is a function to take substrings of strings.

SUBSTR(s,m,n)

produces the substring of 's' which starts at position 'm' (with origin one), of length 'n'. If 'n' is omitted or too big, the rest of the string is used, while if 'm' is out of range, the result is a null string.

SUBSTR(ABC,2,1)

is B,

SUBSTR(ABC,2)

is BC, and

SUBSTR(ABC,4)

is empty." [276]

The Code.

Execution of the macro processor begins with a call to the function MACRO, with a text file as its input. It calls MACRO1

to begin reading the input file and expanding macros as is necessary. In addition to the input file, MACRO1 has as its input a macro definition table. This table is a list of pairs; each pair contains a character string (a macro), and that macro's definition (another character string). Initially the macro definition table contains keywords and definitions for the built-in functions. Another argument to MACRO1 is a counter which keeps track of the current level of macro expansion. When MACRO1 is initially called, this counter is set at zero.

```
DEFINE MACRO INPUT
    ENDOUT:MACRO1:<INPUT 0
                  <<"(D E F I N E) <DEFTYPE>>
                   <"(I F E L S E) <IFELSTYP>>
                   <"(I N C R) <INCTYPE>>
                   <"(S U B S T R) <SUBTYPE>>>>
.
 -=>MACRO

DEFINE MACRO1 (INPUT DEFCOUNT TABLE)
    IF NULL:INPUT THEN <OK <> TABLE>
    ELSEIF GREAT:<DEFCOUNT MAXDEF>
         THEN <DEEP <> TABLE>
    ELSEIF LBRACKET:FIRST:INPUT
         THEN SPANBRAK:<GOTOBRAK:REST:INPUT DEFCOUNT TABLE>
    ELSEIF NOTALFA:FIRST:INPUT
         THEN <1   CONS          1>:<
              <#  FIRST:INPUT  #>
               MACRO1:<REST:INPUT DEFCOUNT TABLE>>
    ELSE CKTOKEN:<SEPTOKEN:INPUT DEFCOUNT TABLE>
.
 -=>MACRO1
```

MACRO1 returns as output a status indicator, an output, text-file, and the macro definition table. The function ENDOUT, called by MACRO1, inspects the status indicator; if it is OK, indicating no errors have occurred, ENDOUT returns the output file, OUTPUT. Otherwise, ENDOUT calls ERROUT to display an appropriate error message.

```
DEFINE ENDOUT (STATUS OUTPUT TABLE)
    IF OKQ:STATUS THEN OUTPUT
    ELSE ERROUT:STATUS
.
-=>ENDOUT
```

```
DEFINE ERROUT STATUS
    IF SAME:<STATUS DEEP>
        THEN "(M A C R O S-T O O-D E E P)
    ELSEIF SAME:<STATUS UNBAL>
        THEN "(U N B A L A N C E D-P A R E N S)
    ELSE "(U N B A L A N C E D-B R A C K E T S)
.
-=>ERROUT
```

MACRO1 is the real workhorse of the macro processor. It and its helpers interpret the input data and expand macros or perform built-in functions as is necessary. MACRO1 is called to process the initial input file; also, when a macro is encountered in the input, MACRO1 is called recursively to process that macro's definition. Each time MACRO1 is called to process all or part of a macro definition, the counter DEFCOUNT is incremented by 1 to indicate a new level of macro expansion. If, upon a call to MACRO1, DEFCOUNT exceeds an integer constant, MAXDEF, MACRO1 terminates, returning the symbolic constant, DEEP, as a status indicator.

If MACRO1 encounters a left bracket in its input, all data between that left bracket and a balancing right bracket are copied to the output, unchanged. The functions SPANBRAK and GOTOBRAK are called to accomplish this.

```
DEFINE LBRACKET CHAR
    SAME:<CHAR LBRAK>
.
-=>LBRACKET
```

```
DEFINE SPANBRAK ((STATUS SECTION INPUT) DEFCOUNT TABLE)
    IF ERRORQ:STATUS THEN <BRACKETS <> TABLE>
    ELSE <1    APPEND     1>:<
         <#  SECTION      #>
          MACRO1:<INPUT DEFCOUNT TABLE>>
.
-=>SPANBRAK
```

```
DEFINE GOTOBRAK INPUT
    BALDELIM:<LBRAK RBRAK 1 INPUT>
.
-=>GOTOBRAK
```

```
DEFINE BALDELIM (LDELIM RDELIM N INPUT)
    IF NULL:INPUT THEN <ERROR <> INPUT>
    ELSEIF AND:<SAME:<FIRST:INPUT RDELIM> ONEP:N>
         THEN <OK <> REST:INPUT>
    ELSEIF SAME:<FIRST:INPUT LDELIM>
         THEN <1          CONS       1>:<
              <#      FIRST:INPUT   #>
               BALDELIM:<LDELIM RDELIM ADD1:N REST:INPUT>>
    ELSEIF SAME:<FIRST:INPUT RDELIM>
         THEN <1          CONS       1>:<
              <#      FIRST:INPUT   #>
               BALDELIM:<LDELIM RDELIM SUB1:N REST:INPUT>>
    ELSE <1          CONS       1>:<
         <#     FIRST:INPUT    #>
          BALDELIM:<LDELIM RDELIM N REST:INPUT>>
.
-=>BALDELIM
```

If any other non-alphanumeric input character besides a left bracket is encountered by MACRO1 (NOTALFA:FIRST:INPUT), it is just copied to the output.

```
DEFINE NOTALFA CHAR
    NOT:MEMBER:<CHAR ALPHANUM>
.
 -=>NOTALFA
```

MACRO1 calls SEPTOKEN to strip an alphanumeric string off the beginning of the input file. CKTOKEN, also called by MACRO1 , uses LOOKUP to compare the alphanumeric string returned by SEPTOKEN to the macros in the macro definition file. If the string matches a macro in the definition file the value TRUE is returned, along with that macro's definition. It is possible that a macro's definition may be an empty list.

```
DEFINE SEPTOKEN INPUT
    IF NULL:INPUT THEN <<> INPUT>
    ELSEIF NOTALFA:FIRST:INPUT
         THEN <<> INPUT>
    ELSE <CONS          1>:<
         <FIRST:INPUT   #>
         SEPTOKEN:REST:INPUT>
.
 -=>SEPTOKEN

DEFINE CKTOKEN ((TOKEN INPUT) DEFCOUNT TABLE)
    IFTOK:<LOOKUP:<TOKEN TABLE> TOKEN DEFCOUNT TABLE INPUT>
.
 -=>CKTOKEN

DEFINE LOOKUP (TOKEN TABLE)
    IF NULL:TABLE THEN <FALSE <>>
    ELSEIF EQSTR:<TOKEN 1:FIRST:TABLE>
         THEN <TRUE 2:FIRST:TABLE>
    ELSE LOOKUP:<TOKEN REST:TABLE>
.
 -=>LOOKUP
```


The function EQSTR, which compares two character strings, was introduced in Chapter 3 .

IFTOK calls MACRO1 to process the rest of the input if LOOKUP has found no macro definition for a given alphanumeric string, or if the definition found was an empty list. Otherwise, DODEF is called to further process the macro definition found by LOOKUP.

```
DEFINE IFTOK ((STATUS DEF) TOKEN DEFCOUNT TABLE INPUT)
    IF NOT:STATUS
         THEN <1          APPEND          1>:<
              <#          TOKEN           #>
               MACRO1:<INPUT DEFCOUNT TABLE>>
    ELSEIF NULL:DEF
         THEN MACRO1:<INPUT DEFCOUNT TABLE>
    ELSE DODEF:<GETBAL:INPUT DEF DEFCOUNT TABLE>
.
 -=>IFTOK
```

If a macro is encountered, IFTOK calls GETBAL to gather its argument list. It returns a status indicator, an argument list, and the rest of its original input file. For macros with no arguments (not immediately followed by a left parenthesis), an empty list is returned as the argument list. If a left parenthesis immediately follows a macro, then IFTOK gathers all the data between the left and balancing right parenthesis.

```
DEFINE GETBAL INPUT
    IF LPARENQ:FIRST:INPUT
        THEN BALPAR:BALDELIM:<LPAR RPAR 1 REST:INPUT>
    ELSE <OK <> INPUT>
.
 -=>GETBAL

DEFINE LPARENQ CHAR
    SAME:<CHAR LPAR>
.
 -=>LPARENQ

DEFINE BALPAR (STATUS DATA INPUT)
    IF ERRORQ:STATUS THEN <UNBAL DATA INPUT>
    ELSE <OK DATA INPUT>
.
 -=>BALPAR

DEFINE RPARENQ CHAR
    SAME:<CHAR RPAR>
.
 -=>RPARENQ
```

DODEF's arguments, STATUS, PARMS, and INPUT are passed from the function GETBAL. STATUS will be OK if GETBAL has gathered data from the input between a balanced set of parentheses. STATUS is UNBAL if GETBAL has reached the end of the input file without balancing parentheses. PARMS is the data

GETBAL has found within a set of parentheses. This is the parameter list for a macro with arguments. INPUT is what remains of the input file.

The argument, DEF, is the macro definition of the most recently encountered macro in the input file. DEFCOUNT is the counter which keeps track of the current level of macro expansion, and TABLE is the macro definition file.

```
DEFINE DODEF ((STATUS PARMS INPUT) DEF DEFCOUNT TABLE)
    IF NOT:OKQ:STATUS THEN <STATUS <> TABLE>
    ELSEIF DEFTYPEQ:DEF
        THEN ADDEF:<MAKEDEF:PARMS TABLE DEFCOUNT INPUT>
    ELSEIF INCTYPEQ:DEF
        THEN DOINCR:<MACRO1:<PARMS ADD1:DEFCOUNT TABLE>
                    DEFCOUNT INPUT>
    ELSEIF SUBSTRQ:DEF
        THEN DOSUBSTR:<GETPARMS:PARMS DEFCOUNT TABLE INPUT>
    ELSEIF IFELSEQ:DEF
        THEN DOIFELSE:<GETPARMS:PARMS DEFCOUNT TABLE INPUT>
    ELSE ADDSTR:<MACRO1:<MAKSTR:<PARMS DEF> ADD1:DEFCOUNT
                        TABLE> DEFCOUNT INPUT>
    .
     -=>DODEF
```

If the most recently encountered macro was a "DEFINE" (IF DEFTYPEQ:DEF), then ADDEF is called to add a new macro to the definition file. MAKEDEF and its help function, CKSYNTAX attempt to divide the parameter list for the "DEFINE" function into two lists, the first containing an alphanumeric token, and the second containing whatever is left in the parameter list

after the token.

```
DEFINE DEFTYPEQ DEF
    TOKMATCH:<DEF DEFTYPE>
.
-=>DEFTYPEQ

DEFINE TOKMATCH (DEF TOKEN)
    AND:<SAME:<FIRST:DEF TOKEN> NULL:REST:DEF>
.
-=>TOKMATCH

DEFINE MAKEDEF PARMS
    IF NULL:PARMS THEN <<><>>
    ELSEIF NOTALFA:FIRST:PARMS THEN <<><>>
    ELSE CKSYNTAX:SEPTOKEN:PARMS
.
-=>MAKEDEF

DEFINE CKSYNTAX (NAME PARM)
    IF NULL:PARM THEN <NAME <>>
    ELSEIF COMMAQ:FIRST:PARM THEN <NAME REST:PARM>
    ELSE <<><>>
.
-=>CKSYNTAX

DEFINE ADDEF ((TOKEN DEF) TABLE DEFCOUNT INPUT)
    IF NULL:TOKEN THEN MACRO1:<INPUT DEFCOUNT TABLE>
    ELSE CKEVAL:<MACRO1:<DEF ADD1:DEFCOUNT TABLE> TOKEN
                 DEFCOUNT INPUT>
.
-=>ADDEF
```

If the parameter list has been found to be empty by MAKEDEF, or if the first character in the parameter list is not alpha-

numeric, or if the alphanumeric string beginning the parameter list is followed by a nonalphanumeric character which is not a comma, then the "DEFINE" macro is ignored. ADDEF then calls MACRO1 to process the rest of the input file (INPUT).

If none of the above conditions hold, the use of the "DEFINE" function may result in the addition of a new entry to the definition file. Uses of the "DEFINE" function which may result in the addition of a new entry to the definition table include:

DEFINE(token)

DEFINE(token,)

DEFINE(token,definition)

CKEVAL is called by ADDEF to add a new entry to the definition table. MACRO1 is called to expand the new definition before it is added to the definition table.

```
DEFINE CKEVAL ((STATUS DEF TABLE) TOKEN DEFCOUNT INPUT)
    IF OKQ:STATUS
        THEN MACRO1:<INPUT DEFCOUNT CONS:<<TOKEN DEF> TABLE>>
    ELSE <STATUS DEF TABLE>
.
-=>CKEVAL
```

The line "ELSEIF INCTYPEQ:DEF" in DODEF returns TRUE if the most recently encountered macro in the input was the "INCR" function (which has as its definition, INCTYPE). Recall that the INCR function converts its argument to an integer, adds one to that value, and converts the integer back to a character string, which is then added to the output. So,

INCR(1)

would cause a '2' to be added to the output.

DOINCR handles the "INCR" function. MACRO1 is called to expand the parameter list.

```
DEFINE INCTYPEQ DEF
    TOKMATCH:<DEF INCTYPE>
.
-=>INCTYPEQ

DEFINE DOINCR ((STATUS VAL TABLE) DEFCOUNT INPUT)
    IF NOT:OKQ:STATUS
        THEN <STATUS VAL TABLE>
    ELSE <1          APPEND          1>:<
         <#   BUMP:SEPDIGS:VAL     #>
          MACRO1:<INPUT DEFCOUNT TABLE>>
.
-=>DOINCR

DEFINE BUMP VAL
    IF NULL:VAL THEN <>
    ELSE ITOC:ADD1:CTOI:VAL
.
-=>BUMP
```

DODEF calls DOSUBSTR to handle the "SUBSTR" (substring) function.

```
DEFINE DOSUBSTR (LPARMS DEFCOUNT TABLE INPUT)
    IF OR:<NULL:LPARMS NULL:REST:LPARMS>
        THEN MACRO1:<INPUT DEFCOUNT TABLE>
    ELSE DOSBSTR1:<FIRST:LPARMS ALLNUM:REST:LPARMS
                   DEFCOUNT TABLE INPUT>
.
-=>DOSUBSTR

DEFINE SUBSTRQ DEF
    TOKMATCH:<DEF SUBTYPE>
.
-=>SUBSTRQ
```


For macros with arguments, GETPARMS creates a list out of each of the arguments in the macro's parameter list. An "argument" is a character string in a parameter list which either precedes a comma, or precedes the end of the parameter list.

```
DEFINE GETPARMS LPARMS
    ADDPARM:READTOCH:<COMMA LPARMS>
.
-=>GETPARMS

DEFINE ADDPARM (PARM LPARMS)
    IF NULL:LPARMS THEN <PARM>
    ELSE CONS:<PARM GETPARMS:LPARMS>
.
-=>ADDPARM

DEFINE READTOCH (CHAR LPARMS)
    IF NULL:LPARMS THEN <<><>>
    ELSEIF SAME:<CHAR FIRST:LPARMS> THEN <<> REST:LPARMS>
    ELSE <CONS          1>:<
         <FIRST:LPARMS  #>
          READTOCH:<CHAR REST:LPARMS>>
.
-=>READTOCH
```

If the parameter list for the "SUBSTR" function does not contain at least two arguments, the function is ignored. Otherwise, the function ALLNUM is called to verify that the second, and (possibly) the third arguments are numeric. If they aren't, the "SUBSTR" function is ignored.

```
DEFINE ALLNUM LPARMS
    IF NULL:REST:LPARMS THEN IFNUM:FIRST:LPARMS
    ELSIF NULL:IFNUM:FIRST:LPARMS
        THEN <>
    ELSE CCONS:<IFNUM:FIRST:LPARMS ALLNUM:REST:LPARMS>
.
-=>ALLNUM
```

```
DEFINE IFNUM LIST
    NUMVAL:SEPDIGS:LIST

-=>IFNUM
```

IFNUM uses help functions, NUMVAL and SEPDIGS , which were introduced in chapter 7 . If the input LIST to IFNUM consists only of digits, then the LIST of characters is converted to an integer; otherwise an empty list is returned.

DOSBSTR1 calls MACRO1 to expand the first argument to the "SUBSTR" command. DOSBSTR2 then builds a substring from the character string output by MACRO1 (STRING). For a "SUBSTR" macro with only two arguments, eg "SUBSTR(string, n)", DOSBSTR2 calls FOLLOW to build a substring which consists of "string" from its nth argument on. For a "SUBSTR" macro with three arguments, eg "SUBSTR(string,n,m), DOSBSTR2 calls OUTCHARS to build a substring consisting of the first "m" characters of the string built by FOLLOW.

```
DEFINE DOSBSTR1 (STRING LVALS DEFCOUNT TABLE INPUT)
    IF NULL:LVALS
        THEN MACRO1:<INPUT DEFCOUNT TABLE>
    ELSE DOSBSTR2:<MACRO1:<STRING ADD1:DEFCOUNT TABLE>
                  LVALS DEFCOUNT INPUT>

-=>DOSBSTR1
```



```
DEFINE DOSBSTR2 ((STATUS STRING TABLE) LVALS DEFCOUNT INPUT)
   IF NOT:OKQ:STATUS THEN <STATUS STRING TABLE>
   ELSEIF NULL:REST:LVALS
       THEN <1      APPEND                                  1>:<
            <#      FOLLOW:<FIRST:LVALS STRING>             #>
             MACRO1:<INPUT DEFCOUNT TABLE>>
   ELSE <1      APPEND                                              1>:<
        <#      OUTCHARS:<2:LVALS FOLLOW:<1:LVALS STRING>>          #>
         MACRO1:<INPUT DEFCOUNT TABLE>>

-=>DOSBSTR2

DEFINE FOLLOW (N STRING)
    IF NULL:STRING THEN <>
    ELSEIF LESS:<N 2>   THEN STRING
    ELSE FOLLOW:<SUB1:N REST:STRING>

-=>FOLLOW


DEFINE OUTCHARS (N STRING)
   IF NULL:STRING THEN <>
   ELSEIF LESS:<N 1> THEN <>
   ELSE CONS:<FIRST:STRING OUTCHARS:<SUB1:N REST:STRING>>

-=>OUTCHARS
```

DODEF calls DOIFELSE if the "IFELSE" macro is to be processed--"ELSEIF IFELSEQ:DEF." DOIFELSE ignores the "IFELSE" macro if its parameter list contains less than four arguments. The processing of the "IFELSE" macro works in the following way. First, DOIFELSE and CK1ST call MACRO1 to expand the first two arguments in the macro's parameter list.

```
DEFINE IFELSEQ DEF
    TOKMATCH:<DEF IFELSTYP>
.
 -=>IFELSEQ


DEFINE DOIFELSE (PARMS DEFCOUNT TABLE INPUT)
    IF OR:<NULL:PARMS NULL:REST:PARMS NULL:RREST:PARMS
           NULL:RRREST:PARMS>
         THEN MACRO1:<INPUT DEFCOUNT TABLE>
    ELSE CK1ST:<MACRO1:<FIRST:PARMS ADD1:DEFCOUNT TABLE>
               REST:PARMS DEFCOUNT INPUT>
.
 -=>DOIFELSE

DEFINE CK1ST ((STATUS IDATA TABLE) PARMS DEFCOUNT INPUT)
    IF NOT:OKQ:STATUS THEN <STATUS IDATA TABLE>
    ELSE CK2ND:<MACRO1:<FIRST:PARMS ADD1:DEFCOUNT TABLE>
               IDATA REST:PARMS DEFCOUNT INPUT>
.
 -=>CK1ST
```

If both the first and second arguments of the parameter list have been expanded without error, CK2ND calls EQSTR to compare the expanded, first argument to the expanded, second argument. If the result of the comparison is TRUE (the two strings are identical), MACRO1 is called to expand the third argument in the parameter list; otherwise, MACRO1 is called to expand the fourth argument in the parameter list.


```
DEFINE CK2ND ((STATUS EDATA TABLE) IDATA PARMS)
             DEFCOUNT INPUT)
    IF NOT:OKQ:STATUS THEN <STATUS EDATA TABLE>
    ELSEIF EQSTR:<IDATA EDATA>
         THEN ADDSTR:<MACRO1:<1:PARMS ADD1:DEFCOUNT TABLE>
                     DEFCOUNT INPUT>
    ELSE ADDSTR:<MACRO1:<2:PARMS ADD1:DEFCOUNT TABLE>
                DEFCOUNT INPUT>
.
 -=>CK2ND
```

ADDSTR is called to add the resulting character string of the "IFELSE" macro to the output. MACRO1 is called to process the rest of the input.

```
DEFINE ADDSTR ((STATUS STRING TABLE) DEFCOUNT INPUT)
    IF NOT:OKQ:STATUS
         THEN <STATUS STRING TABLE>
    ELSE <1   APPEND    1>:<
         <#   STRING    #>
          MACRO1:<INPUT DEFCOUNT TABLE>>
.
 -=>ADDSTR
```

DODEF calls MAKSTR to create an output string for a macro which is not one of the special functions "DEFINE", "INCR", "SUBSTR" or "IFELSE". MAKSTR takes as its input, LPARAMS, the parameter list following the occurrence of the macro in the input, and DEF, the macro's definition from the macro definition table. MAKSTR replaces every occurrence of "$n" in DEF with the nth argument in LPARAMS, adding that argument to the output.

Other character strings in DEF are just copied to the output.

```
DEFINE MAKSTR (LPARAMS DEF)
    IF NULL:DEF THEN DEF
    ELSEIF SUBQ:FIRST:DEF
         THEN SUBONE:<LPARAMS REST:DEF>
    ELSE CONS:<FIRST:DEF MAKSTR:<LPARAMS REST:DEF>>
.
-=>MAKSTR
```

MAKSTR and its helpers call SUBONE if a '$' (represented by the constant, SUB) is encountered in DEF. SUBONE isolates a string of digits following the '$' (via a call to GETDIGS), converts the string to an integer (via a call to CTOI in PROJPARM) and uses that integer--call it "n"--in obtaining the nth argument from LPARAMS (using the function PROJ, which was introduced in Chapter 2).

```
DEFINE SUBQ CHAR
    SAME:<CHAR SUB>
.
-=>SUBQ

DEFINE SUBONE (LPARAMS DEF)
   IF NULL:DEF THEN DEF
   ELSE APPEND:<PROJPARM    MAKSTR>:<
               <LPARAMS    LPARAMS>
                GETDIGS:DEF>
.
-=>SUBONE
```



```
DEFINE PROJPARM (LPARAMS LDIGS)
    PROJ:<CTOI:LDIGS LPARAMS>
.
-=>PROJPARM
```

Notice that if "$n" occurs in DEF, and there are less than "n" arguments in LPARAMS, the "$n" is ignored with nothing added to the output. "That way, if X is defined by

DEFINE(X,A$1B)

the inputs

X(+)

X(-,+)

X()

X

all produce something sensible: A+B, A-B, AB and AB respectively." [273]

Note also that if a macro has been defined as a macro with no arguments, an argument list following an occurrence of that macro in the input is ignored.

Constants Introduced in Chapter 8.

```
DECLARE DEFTYPE "DEFTYPE.
 -=>DEFTYPE

DECLARE IFELSTYP "IFELSTYP.
 -=>IFELSTYP

DECLARE INCTYPE "INCTYPE.
 -=>INCTYPE

DECLARE SUBTYPE "SUBTYPE.
 -=>SUBTYPE

DECLARE MAXDEF 6.
 -=>6

DECLARE DEEP "DEEP.
 -=>DEEP

DECLARE OK "OK.
 -=>OK

DECLARE ALPHANUM
     "(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
       0 1 2 3 4 5 6 7 8 9).
 -=>(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
     0 1 2 3 4 5 6 7 8 9)

DECLARE LPAR "LPAR.
 -=>LPAR

DECLARE RPAR "RPAR.
 -=>RPAR

DECLARE LBRAK "LBRAK.
 -=>LBRAK

DECLARE RBRAK "RBRAK.
 -=>RBRAK

DECLARE UNBAL "UNBAL.
 -=>UNBAL

DECLARE BRACKETS "BRACKETS.
 -=>BRACKETS

DECLARE COMMA "COMMA.
 -=>COMMA

DECLARE SUB "SUB.
 -=>SUB
```


Chapter 9

The programs presented in Software Tools are written in RATFOR, a variety of structured-FORTRAN. In this chapter, a program which translates RATFOR code to standard-FORTRAN code is presented.

RATFOR provides several cosmetic enhancements to FORTRAN to make the language more flexible and (possibly) more "readable." Statements may begin on any column on a line. Multiple statements may be written on one line, with each statement terminated by either a semicolon, or by the end of the line (NL). One may continue statements over two or more lines by placing commas at the end of all but the last line making up the statement. (This replaces the continuation convention for standard FORTRAN, which is not acceptable to RATFOR.) Comments may occur anywhere on a line; a comment consists of a character string beginning with a sharp (#). (This is the only acceptable syntax in RATFOR for writing comments.) Finally, literal character strings may be expressed as quoted strings in RATFOR; this replaces the Hollerith notation of standard FORTRAN.

In addition to the above mentioned cosmetic enhancements to FORTRAN, RATFOR allows the use of some control structures not available to the standard-FORTRAN user. Discussed here is the implementation of the following control structures: "IF-ELSE, WHILE, DO, BREAK, NEXT, and statement grouping with braces". [285]

The IF-ELSE Statement.

RATFOR allows the use of an IF statement with or without a matching ELSE statement. The general form of the IF without ELSE is

IF (condition) statement

The condition is any valid standard-FORTRAN boolean expression. The parenthesized condition may be expressed on more than one line of source code. The statement is any valid RATFOR statement (a formal definition of a statement is presented later in the chapter).

The standard-FORTRAN equivalent of the RATFOR, IF without ELSE is

```
        IF (.NOT. (condition)) GOTO L
            statement
    L   CONTINUE
```

"L" is a legal FORTRAN label.

The IF with ELSE has the following general form:

IF (condition) $statement_1$ ELSE $statement_2$

For the IF with ELSE, $statement_1$ is executed if condition is true; otherwise, $statement_2$ is executed. The standard FORTRAN translation of the IF with ELSE is

```
        IF (.NOT. (condition)) GOTO L
            statement1
            GOTO L1
    L   CONTINUE
            statement2
    L1  CONTINUE
```


Note that $statement_1$ and $statement_2$ in the IF with ELSE, as well as the statement in the IF without ELSE, may themselves be RATFOR IF statements. One of the responsibilities of the RATFOR-FORTRAN translator will be the translation of these embedded statements.

Also, a group of RATFOR statements enclosed within balancing left and right braces ({, }), is itself considered to be a RATFOR statement. In the case of the IF (with or without ELSE), this allows for a group of statements to be executed depending upon some condition.

The DO Statement. (BREAK and NEXT)

"The RATFOR DO is a FORTRAN DO without a label." [293]

Its general form is

DO limits statement .

The RATFOR-FORTRAN translator assumes (and does not bother to verify) that limits is an expression that would be legal in a standard-FORTRAN DO statement, e.g..

```
I = 1, 10
```

The standard-FORTRAN translation of the RATFOR DO is

```
        DO L limits
            statement
    L   CONTINUE
    L+1 CONTINUE
```

The line "L+1 CONTINUE" above is inserted in case a RATFOR BREAK statement is contained within the RATFOR DO-loop.

Execution of a BREAK statement, which consists of the key word "BREAK," causes an exit from a RATFOR loop structure. In the case of the translated DO-loop discussed above, the BREAK would be translated into

```
GOTO L+1
```

In the case of embedded loops, a BREAK causes an exit from the innermost loop in which it is contained.

Another statement, NEXT, causes re-iteration of the innermost loop in which it is contained. In the case of the translated DO-loop discussed above, a NEXT would be translated to

```
GOTO L
```

The WHILE Statement.

The WHILE statement allows for repeated iteration of a statement (which could consist of a number of statements) as long as some condition holds true. Its general form is

```
WHILE (condition) statement
```

The parenthesized condition is assumed to be of exactly the same format as the condition portion of an IF statement. The translated FORTRAN for a WHILE is

```
      CONTINUE
L     IF ( .NOT. (condition)) GOTO L+1
          statement
          GOTO L
L+1   CONTINUE
```


The first CONTINUE is output in case the original RATFOR code includes a label on the same line as the WHILE. If that is the case, the label is tacked onto the CONTINUE.

As in the DO-loop, a BREAK within the WHILE produces a "GOTO L+1," while a NEXT gives "GOTO L".

Labels and non-RATFOR Statements.

A string of digits at the beginning of an input statement, a label, "is output, beginning at column 1, and followed by enough blanks so that the next character will come out in column 7, the standard place for a FORTRAN statement to begin." [294] A statement which doesn't begin with a RATFOR key word is assumed to be a standard-FORTRAN statement. When such a statement is encountered in the input, it is just copied to the output in standard-FORTRAN format, between columns 7 and 72.

BNF Definition of RATFOR.

Kernighan and Plauger define RATFOR in Backus-Naur Form. The BNF definition of a RATFOR program follows:

```
program = statement
        or program statement
statement = IF (condition) statement
        or IF (condition) statement1 ELSE statement2
        or WHILE (condition) statement
        or DO limits statement
        or digits statement
```

or BREAK

or NEXT

or {program}

or other

A statement of type other is one which doesn't begin with a RATFOR keyword (or left brace or string of digits).

[286]

The Code.

Like the program presented in Software Tools, the translator presented here has a central, parsing routine; the routine calls a lexical scanning routine to process the beginning of each statement. Among other things, this routine returns an indicator of the type of "token" which begins the statement. Given this input, the parser calls an appropriate code generation routine. Each RATFOR statement type is represented by one of these code generation routines. The code generation routines make use of a set of output routines, which are responsible for generating standard-FORTRAN output.

Lexical Analysis:

The lexical analysis routine, LEX, calls RATTOK to isolate a "token" from the beginning of the RATFOR source file. RATTOK "breaks the input into alphanumeric strings, quoted strings, and single non-alphanumerics. It also strips out the blanks, tabs and comments that separate tokens. (Blanks can be discarded because they are not significant in FORTRAN programs.)"

[288]

The inputs to RATTOK are SOURCE, the RATFOR source input file, and LINECT, an indicator of the current line number in the source file. LINECT is used by an error message routine, SYNERR, which indicates the type and locations of syntax errors.

RATTOK returns four results, a token, a message file for reporting syntax errors, the remains of the source file, and the current value of LINECT.

```
DEFINE LEX (SOURCE LINECT)
    CKTOK:RATTOK:<SOURCE LINECT>
.
-=>LEX


DEFINE RATTOK (SOURCE LINECT)
    IF NULL:SOURCE THEN <<><><> LINECT>
    ELSEIF OR:<TABQ:FIRST:SOURCE BLANKQ:FIRST:SOURCE>
         THEN RATTOK:<REST:SOURCE LINECT>
    ELSEIF NLQ:FIRST:SOURCE
         THEN <<FIRST:SOURCE> REST:SOURCE <> ADD1:LINECT>
    ELSEIF OR:<SQUOTEQ:FIRST:SOURCE DQUOTEQ:FIRST:SOURCE>
         THEN <CONS           1     1     1>:<
              <FIRST:SOURCE   #     #     #>
               BALQUOTE:<FIRST:SOURCE REST:SOURCE LINECT>>
    ELSEIF SHARPQ:FIRST:SOURCE
         THEN <<NL> EATLINE:REST:SOURCE <> ADD1:LINECT>
    ELSEIF NOTALFA:FIRST:SOURCE
         THEN <<FIRST:SOURCE> REST:SOURCE <> LINECT>
    ELSE <1:GETALFA:SOURCE 2:GETALFA:SOURCE <> LINECT>
.
-=>RATTOK
```

When a single-quote or double-quote is encountered in the source input, RATTOK calls BALQUOTE to gather up the quoted string. BALQUOTE does not cross line boundaries to balance quotation marks. An error message is returned in the case of unbalanced quotations.

```
DEFINE BALQUOTE (TOKEN SOURCE LINECT)
    IF OR:<NULL:SOURCE NLQ:FIRST:SOURCE>
         THEN <<> SOURCE SYNERR:<QUOTE LINECT> LINECT>
    ELSEIF SAME:<TOKEN FIRST:SOURCE>
         THEN <<> REST:SOURCE <> LINECT>
    ELSE <CONS          1     1     1>:<
         <FIRST:SOURCE  #     #     #>
          BALQUOTE:<TOKEN REST:SOURCE LINECT>>
.
-=>BALQUOTE
```

The function EATLINE is called to remove the remaining characters in an input line when a sharp character is encountered (indicating a comment). The function GETALFA is used to gather an alphanumeric string from the beginning of the source file. It is very similar to the function GETWORD, which was introduced in Chapter 1.

```
DEFINE EATLINE LINE
    IF NLNULL:LINE THEN LINE
    ELSE EATLINE:REST:LINE
.
-=>EATLINE

DEFINE SHARPQ CHAR
   SAME:<CHAR SHARP>
.
-=>SHARPQ
```



```
DEFINE GETALFA SOURCE
    IF OR:<NULL:SOURCE NOTALFA:FIRST:SOURCE>
         THEN <<> SOURCE>
    ELSE <CONS          1>:<
         <FIRST:SOURCE  #>
          GETALFA:REST:SOURCE>
.
-=>GETALFA
```

Since GETALFA interprets a blank as the end of a token, RATFOR tokens should contain no embedded blanks.

The lexical analysis function, LEX, calls a help function, CKTOK, to interpret the results of RATTOK. It returns as its first result a token type, which is determined from the contents of the token returned by RATTOK. It returns EOF as the token type if the source file is empty.

```
DEFINE CKTOK (TOKEN SOURCE MSG LINECT)
    IF AND:<NULL:TOKEN NULL:SOURCE>
         THEN <EOF <> MSG SOURCE LINECT>
    ELSEIF NLQ:FIRST:TOKEN
         THEN LEX:<SOURCE LINECT>
    ELSEIF ALLNUM:TOKEN
         THEN <ALLDIG TOKEN MSG SOURCE LINECT>
    ELSEIF OR:<SEMIQ:FIRST:TOKEN LBRACEQ:FIRST:TOKEN
              RBRACEQ:FIRST:TOKEN>
         THEN <FIRST:TOKEN <> MSG SOURCE LINECT>
    ELSE SPECTOKQ:<LOOKUP:<TOKEN TOKTBL> TOKEN MSG SOURCE LINECT>
.
-=>CKTOK

DEFINE SPECTOKQ ((FOUND DEF) TOKEN MSG SOURCE LINECT)
    IF NOT:FOUND
         THEN <OTHER TOKEN MSG SOURCE LINECT>
    ELSE <FIRST:DEF <> MSG SOURCE LINECT>
.
-=>SPECTOKQ
```

Top Level Functions - - parsing.

The translator is executed by a call to the function, RATFOR. If the source file is not empty, the function RATFOR calls ENDSTAT to process the result of a call to the parsing routine, PARSE. The OUTPUT file generated by PARSE is the standard-FORTRAN code created as a result of parsing one RATFOR source statement. The message file, MSG, generated by PARSE contains any syntax error messages generated while parsing the RATFOR source statement.

```
DEFINE RATFOR SOURCE
    IF NULL:SOURCE THEN <<><>>
    ELSE ENDSTAT:PARSE:<LEX:<SOURCE 1> L23000 MINUS:1 1>
.
 -=>RATFOR

DEFINE ENDSTAT (ENDTOK OUTPUT MSG SOURCE CURLAB LOOPLAB LINECT)
    IF EOFQ:ENDTOK
        THEN <OUTPUT MSG>
    ELSEIF EOFQ:1:LEX:<SOURCE LINECT>
        THEN <OUTPUT MSG>
    ELSE <APPEND                APPEND>:<
         <OUTPUT                MSG    >
          ENDSTAT:PARSE:<LEX:<SOURCE LINECT> CURLAB MINUS:1 1>>
.
 -=>ENDSTAT
```

If the end of the input file has been reached, ENDSTAT terminates, returning the OUTPUT and MSG files. Otherwise, ENDSTAT calls itself recursively, with another call to PARSE as its argument.

The other arguments to ENDSTAT, like OUTPUT and MSG, are results of a call to PARSE. ENDTOK is a token indicating the type of statement most recently parsed (or EOF). SOURCE is what remains of the RATFOR input file. CURLAB is the value of the label most recently generated by the translator; labels are generated for the processing of IF, WHILE and DO statements. LOOPLAB is used for the processing of BREAK and NEXT statements. LINECT is the line number of the line in the input file currently being examined; it is used by the error routine, SYNERR, in indicating the location of a syntax error in the input.

PARSE takes as arguments, a list containing the results of a call to LEX, CURLAB and LOOPLAB (which have the same meaning as in ENDSTAT) and COLNO. COLNO is the column number of a line currently being output; the output routines use COLNO to create output in standard-FORTRAN format.

PARSE1, called by PARSE,does the actual parsing. Depending upon the token (TOKTYP) returned by LEX, PARSE1 calls an appropriate code generation routine.

```
DEFINE PARSE ((TOKTYP TOKEN MSG SOURCE LINECT)
              CURLAB LOOPLAB COLNO)
    <1   1    APPEND    1    1    1    1>:<
    <#   #    MSG       #    #    #    #>
     PARSE1:<TOKTYP TOKEN SOURCE LINECT CURLAB LOOPLAB COLNO>>
.
 -=>PARSE
```

```
DEFINE PARSE1 (TOKTYP TOKEN SOURCE LINECT CURLAB LOOPLAB COLNO)
    IF EOFQ:TOKTYP
         THEN <EOF <> SYNERR:<EOF LINECT> SOURCE CURLAB LOOPLAB
              LINECT>
    ELSEIF BREAKQ:TOKTYP
         THEN BRKNXT:<BRKTOK LINECT CURLAB ADD1:LOOPLAB LOOPLAB
                   SOURCE COLNO>
    ELSEIF NEXTQ:TOKTYP
         THEN BRKNXT:<NEXTOK LINECT CURLAB LOOPLAB LOOPLAB
                   SOURCE COLNO>
    ELSEIF ALLDIGQ:TOKTYP
         THEN <1   APPEND                          1    1    1    1    1>:<
              <#  1:OUTLAB:<TOKEN COLNO>    #    #    #    #    #>
               PARSE:<LEX:<SOURCE LINECT> CURLAB LOOPLAB
                   2:OUTLAB:<TOKEN COLNO>>>
    ELSEIF ELSEQ:TOKTYP
         THEN <1   1    APPEND                     1    1    1    1    1>:<
              <#   #   SYNERR:<ELSETOK LINECT>    #    #    #    #    #>
               ENDELSE:<ADD1:CURLAB
                    PARSE:<LEX:<SOURCE LINECT> CURLAB LOOPLAB
                         COLNO>>>
    ELSEIF IFTYPEQ:TOKTYP
         THEN IFCODE:<IF ADD2:CURLAB LOOPLAB COLNO
                   RATTOK:<SOURCE LINECT>>
    ELSEIF DOTYPEQ:TOKTYP
         THEN DOCODE:<ADD2:CURLAB LOOPLAB COLNO SOURCE LINECT>
    ELSEIF WHILEQ:TOKTYP
         THEN WHILCODE:<ADD2:CURLAB LOOPLAB COLNO SOURCE LINECT>
    ELSEIF LBRACEQ:TOKTYP
         THEN ENDLBRAC:<TOKTYP <> <> SOURCE CURLAB LOOPLAB LINECT>
    ELSEIF RBRACEQ:TOKTYP
         THEN <OTHER <> SYNERR:<RBRACE LINECT> SOURCE
              CURLAB LOOPLAB LINECT>
    ELSEIF OTHERQ:TOKTYP
         THEN OTHERC:<TOKEN CURLAB SOURCE LOOPLAB LINECT COLNO>
    ELSE <TOKTYP <> <> SOURCE CURLAB LOOPLAB LINECT>

-=>PARSE1
```

If an end-of-file condition exists on a call to PARSE1

"IF EOFQ:TOKTYP"

an error is indicated. End-of-file has been reached unexpectedly during the processing of a statement from the input file. A ground condition occurs, and SYNERR is called to return an

appropriate error message.

If a BREAK or NEXT statement is being processed, PARSE calls BRKNXT to generate "GOTO L" (if a NEXT) or "GOTO L+1" (if a BREAK), where "L" equals LOOPLAB.

```
DEFINE BRKNXT (TOKTYP LINECT CURLAB JUMPLAB LOOPLAB
               SOURCE COLNO)
    IF NOT:POS:JUMPLAB
         THEN <TOKTYP <> SYNERR:<TOKTYP LINECT> SOURCE
              CURLAB LOOPLAB LINECT>
    ELSE <TOKTYP OUTGO:<JUMPLAB COLNO> <> SOURCE
         CURLAB LOOPLAB LINECT>

-=>BRKNXT
```

BRKNXT calls an output routine, OUTGO to output the "GOTO" statement. If JUMPLAB is not a positive integer an error is indicated; a BREAK or NEXT has occurred which is not contained within a RATFOR loop structure.

If the token most recently examined in the input consists solely of digits (ALLDIGQ:TOKTYP), an output routine, OUTLAB, is called by PARSE1 to output a FORTRAN label.

The code for the translator is written such that upon a call to PARSE1, the token currently being examined is considered to be at the beginning of an input statement. This is why PARSE1 indicates an error if the token currently being examined is an ELSE (ELSEQ:TOKTYP). SYNERR is called to output an error message.

Even though an ELSE at the beginning of a statement is "illegal" PARSE1 calls ENDELSE to process the end of the next

input statement (gathered in a recursive call to PARSE1).
This is done to avoid a possible avalanche of error conditions.
ENDELSE calls the output routine, OUTCON, to add

L+1 CONTINUE

to the end of the most recently generated FORTRAN statement.

```
DEFINE ENDELSE (ELSELAB (TOKTYP OUTPUT MSG SOURCE CURLAB
                         LOOPLAB LINECT)
    <TOKTYP APPEND:<OUTPUT OUTCON:ELSELAB> MSG SOURCE
                   CURLAB LOOPLAB LINECT>

-=>ENDELSE
```

PARSE1 calls IFCODE to process an IF statement. IFCODE and its helpers generate the string

"IF(.NOT. (condition)) GOTO L"

from a RATFOR string

"IF (condition)."

IFCODE generates a string "IF(.NOT.(" and appends this to the output to be built by its help functions.

```
DEFINE IFCODE (IFWHILE CURLAB LOOPLAB COLNO
               (TOKEN SOURCE MSG LINECT))
   <1  APPEND2                              APPEND   1   1   1   1>:<
   <#  1:OUTTAB:COLNO                       MSG      #   #   #   #>
   <#  1:OUTSTR:<IFNOT 2:OUTTAB:COLNO>  #            #   #   #   #>
    IFARGS:<IFWHILE TOKEN SOURCE LINECT CURLAB LOOPLAB
            2:OUTSTR:<IFNOT 2:OUTTAB:COLNO>>>

-=>IFCODE
```


OUTTAB is an output routine called by IFCODE which inserts blanks into the output until the current column number for output is 7, within the range acceptable for standard-FORTRAN statements. Its definition is presented in a later discussion of output routines. OUTSTR is another output routine which just outputs a character string (in this case "IF(.NOT.(") in standard-FORTRAN format.

IFCODE's help function, IFARGS checks to see if the RATFOR "IF" is followed by a left parenthesis; if not, a syntax error results. If the "IF" is followed by a left parenthesis, GETARGS is called to gather the condition portion of the statement. GETARGS returns a syntax error if unbalanced parentheses are detected.

```
DEFINE IFARGS (IFWHILE TOKEN SOURCE LINECT CURLAB LOOPLAB COLNO)
    IF OR:<NULL:TOKEN NOT:LPARENQ:FIRST:TOKEN>
         THEN <1   1    APPEND                    1    1    1    1>:<
              <#   #  SYNERR:<LPAREN LINECT>  #    #    #    #>.
               ENDARGS:<IFWHILE SOURCE LINECT CURLAB LOOPLAB
                        COLNO>>
    ELSE PUTARGS:<GETARGS:<SOURCE LINECT 1
                           RATTOK:<SOURCE LINECT>>
                  IFWHILE COLNO CURLAB LOOPLAB>

-=>IFARGS
```

```
DEFINE GETARGS (SOURCE LINECT NLPAR
                (XTOKEN XSOURCE MSG XLINECT))
   IF OR:<NULL:XTOKEN SEMIQ:FIRST:XTOKEN LBRACEQ:FIRST:TOKEN
          RBRACEQ:FIRST:TOKEN>
        THEN <<> APPEND:<MSG SYNERR:<RPAREN LINECT>> SOURCE LINECT>
   ELSEIF AND:<RPARENQ:FIRST:XTOKEN ONEP:NLPAR>
        THEN <<FIRST:XTOKEN> MSG XSOURCE XLINECT>
   ELSEIF RPARENQ:FIRST:XTOKEN
        THEN <CONS          1    1    1>:<
             <FIRST:XTOKEN  #    #    #>
              GETARGS:<XSOURCE XLINECT SUB1:NLPAR
                       RATTOK:<XSOURCE XLINECT>>>
   ELSEIF LPARENQ:FIRST:XTOKEN
        THEN <CONS          1    1    1>:<
             <FIRST:XTOKEN  #    #    #>
             GETARGS:<XSOURCE XLINECT ADD1:NLPAR
                      RATTOK:<XSOURCE XLINECT>>
   ELSEIF NLQ:FIRST:XTOKEN
        THEN GETARGS:<XSOURCE XLINECT NLPAR
                      RATTOK:<XSOURCE XLINECT>>
   ELSE <APPEND        APPEND    1    1>:<
        <XTOKEN        MSG       #    #>
         GETARGS:<XSOURCE XLINECT NLPAR RATTOK:<XSOURCE XLINECT>>>

-=>GETARGS
```

PUTARGS appends the result built by GETARGS to the result to be built by ENDARGS. Again, the output routine, OUTSTR is used.

```
DEFINE PUTARGS ((OUTPUT MSG SOURCE LINECT)
                IFWHILE COLNO CURLAB LOOPLAB)
   <1   APPEND                  APPEND    1    1    1    1>:<
   <#  1:OUTSTR:<OUTPUT COLNO>   MSG       #    #    #    #>
    ENDARGS:<IFWHILE SOURCE LINECT CURLAB LOOPLAB
             2:OUTSTR:<OUTPUT COLNO>>>

-=>PUTARGS
```

The code executed by IFCODE and its helpers is used to process both the IF and WHILE statements. An argument to IFCODE, IFWHILE, determines the type of statement being processed. If the statement being processed is an IF (IF IFTYPEQ:IFWHILE), ENDARGS appends the string, ")GOTO L" to the output. OUTCH, called by ENDARGS, is a routine which outputs one character (in this case, a right parenthesis, RPAREN). ENDIF, with a recursive call to PARSE as one of its arguments, is called to process the end of an IF statement.

```
DEFINE ENDARGS (IFWHILE SOURCE LINECT CURLAB LOOPLAB COLNO)
   IF IFTYPEQ:IFWHILE
       THEN <1     APPEND2                                  1 1 1 1 1>:<
            <#    1:OUTCH:<RPAREN COLNO>                    # # # # #>
            <#    OUTGO:<CURLAB 2:OUTCH:<RPAREN COLNO>>     # # # # #>
             ENDIF:<CURLAB PARSE:<LEX:<SOURCE LINECT> CURLAB
                                  LOOPLAB 1>>>
   ELSE     <1  APPEND2                                      1 1 1 1 1>:<
            <# 1:OUTCH:<RPAREN COLNO>                        # # # # #>
            <# OUTGO:<ADD1:CURLAB 2:OUTCH:<RPAREN COLNO>>    # # # # #>
             ENDWHILE:<LOOPLAB CURLAB
                       PARSE:<LEX:<SOURCE LINECT> CURLAB CURLAB 1>>>

-=>ENDARGS

    DEFINE ENDIF (IFLAB (ENDTOK OUTPUT MSG SOURCE CURLAB
                         LOOPLAB LINECT))
        IF EOFQ:ENDTOK
            THEN <ENDTOK APPEND:<OUTPUT OUTCON:IFLAB> MSG
                  SOURCE CURLAB LOOPLAB LINECT>
        ELSE <1          APPEND    APPEND    1 1 1 1>:<
             <#          MSG       OUTPUT    # # # #>
              ELSETOKQ:<LEX:<SOURCE LINECT> ENDTOK SOURCE
                        LINECT IFLAB CURLAB LOOPLAB>>

    -=>ENDIF
```

If the end of the input has not been reached, ENDIF calls ELSETOKQ to see if an ELSE follows the IF statement. If an ELSE follows the IF, the string

GOTO L+1

L CONTINUE

is added to the end of the statement. ENDELSE is called to process the end of the statement following the ELSE.

If an ELSE does not follow the IF statement, the string

L CONTINUE

is added to the end of the IF statement, and a ground condition is reached.

```
DEFINE ELSETOKQ ((TOKTYP TOKEN MSG XSOURCE XLINECT)
                 ENDTOK SOURCE LINECT IFLAB CURLAB LOOPLAB)
   IF ELSEQ:TOKTYP
        THEN <1   APPEND2                        APPEND   1  1  1   1>:<
             <#  OUTGO:<ADD1:IFLAB COLNO>         MSG      #  #  #   #>
             <#  OUTCON:IFLAB                     #        #  #  #   #>
              ENDELSE:<ADD1:IFLAB
                       PARSE:<LEX:<XSOURCE XLINECT>
                              CURLAB LOOPLAB 1>>>
   ELSE <ENDTOK OUTCON:IFLAB <> SOURCE CURLAB LOOPLAB LINECT>
.
-=>ELSETOKQ
```

PARSE1 calls DOCODE to process a RATFOR DO statement. Recall that the DO statement is of the form

DO limits statement.

From this, we must generate the standard-FORTRAN

DO L limits

statement

L CONTINUE

L+1 CONTINUE

The "DO L" portion is generated by the functions DOCODE, OUTDO, and DONUM.

```
DEFINE DOCODE (CURLAB LOOPLAB COLNO SOURCE LINECT)
    <1   APPEND                    1   1   1   1   1>:<
    <#   1:OUTTAB:COLNO            #   #   #   #   #>
     OUTDO:<CURLAB LOOPLAB 2:OUTTAB:COLNO SOURCE LINECT>>
.
-=>DOCODE

DEFINE OUTDO (CURLAB LOOPLAB COLNO SOURCE LINECT)
    <1   APPEND                    1   1   1   1   1>:<
    <#   1:OUTSTR:<DOSTR COLNO>    #   #   #   #   #>
     DONUM:<CURLAB LOOPLAB 2:OUTSTR:<DOSTR COLNO>
            SOURCE LINECT>>
.
-=>OUTDO


DEFINE DONUM (CURLAB LOOPLAB COLNO SOURCE LINECT)
    <1   APPEND                    1   1   1   1   1>:<
    <# 1:OUTSTR:<ITOC:CURLAB COLNO>  #   #   #   #   #>
     DOFINISH:<EATUP:<SOURCE LINECT> CURLAB LOOPLAB COLNO>>
.
-=>DONUM
```

The limits portion is gathered up by EATUP and its helpers.

```
DEFINE EATUP (SOURCE LINECT)
    CKRPAREN:EATUP1:<RATTOK:<SOURCE LINECT> SOURCE LINECT 0>
.
-=>EATUP
```

EATUP1 does most of the work in gathering the limits.

```
DEFINE EATUP1 ((XTOKEN XSOURCE XMSG XLINECT)
               SOURCE LINECT NLPAR)
    IF OR:<NULL:XTOKEN LBRACEQ:FIRST:XTOKEN>
         THEN <<> SYNERR:<EOFBRACE LINECT> SOURCE LINECT NLPAR>
    ELSEIF OR:<SEMIQ:FIRST:XTOKEN NLQ:FIRST:XTOKEN>
         THEN <<><> XSOURCE XLINECT NLPAR>
    ELSEIF RBRACEQ:FIRST:XTOKEN
         THEN <<><> SOURCE LINECT NLPAR>
    ELSEIF COMMAQ:FIRST:XTOKEN
         THEN <1      APPEND     1     1     1>:<
              <#      XMSG       #     #     #>
               CKCONT:<RATTOK:<XSOURCE XLINECT> XSOURCE XLINECT NLPAR>>
    ELSEIF LPARENQ:FIRST:XTOKEN
         THEN <CONS           APPEND     1     1     1>:<
              <FIRST:XTOKEN   XMSG       #     #     #>
               EATUP1:<RATTOK:<XSOURCE XLINECT> XSOURCE
                      XLINECT ADD1:NLPAR>>
    ELSEIF RPARENQ:FIRST:XTOKEN
        THEN <CONS       APPEND  1    1    1>:<
             <FIRST:XTOKEN  XMSG    #    #    #>:<
              EATUP1:<RATTOK:<XSOURCE XLINECT> XSOURCE
                     XLINECT SUB1:NLPAR>>
    ELSE <APPEND     APPEND  1    1    1>:<
         <XTOKEN     XMSG    #    #    #>
          EATUP1:<RATTOK:<XSOURCE XLINECT> XSOURCE XLINECT NLPAR>>

-=>EATUP1
```

EATUP1 treats the limits as a RATFOR statement. Unlike GETARGS, EATUP1 does not cross line boundaries to balance parentheses. EATUP1 allows continuation of statements across line boundaries if a COMMA appears as the last token on an input line. When EATUP1 encounters a COMMA (IF COMMAQ:FIRST:XTOKEN), CKCONT is called to check to see if the COMMA is the last token on an input line.


```
DEFINE CKCONT ((XTOKEN XSOURCE XMSG XLINECT)
               SOURCE LINECT NLPAR)
    IF NLQ:FIRST:XTOKEN
        THEN <1      APPEND  1    1    1>:<
             <#      XMSG    #    #    #>
              EATUP1:<RATTOK:<XSOURCE XLINECT> XSOURCE XLINECT
                     NLPAR>>
    ELSE     <CONS   1    1    1    1>:<
             <COMMA  #    #    #    #>
              EATUP1:<<XTOKEN XSOURCE XMSG XLINECT>
                     SOURCE LINECT NLPAR>>

-=>CKCONT
```

CKRPAREN, called by EATUP, returns an error message if EATUP1 has encountered an unbalanced set of parentheses.

```
DEFINE CKRPAREN (OUTPUT MSG SOURCE LINECT NLPAR)
    IF ZEROP:NLPAR
         THEN <OUTPUT MSG SOURCE LINECT>
    ELSE <OUTPUT APPEND:<MSG SYNERR:<RPAREN LINECT>>
          SOURCE LINECT>

-=>CKRPAREN
```

DOFINISH, called by DONUM, appends the limits gathered by EATUP et. al. to the output. ENDDO is called to process the end of the DO statement. It is responsible for adding

```
L     CONTINUE
L+1   CONTINUE
```

to the end of the statement.

```
DEFINE DOFINISH ((OUTPUT MSG SOURCE LINECT)
                 CURLAB LOOPLAB COLNO)
    <1   APPEND                                  APPEND    1  1  1  1>:<
    <#  SNOC:<NL 1:OUTSTR:<OUTPUT COLNO>>  MSG      #  #  #  #>
    ENDDO:<LOOPLAB CURLAB PARSE:<LEX:<SOURCE LINECT>
                                 CURLAB CURLAB 1>>>
.
-=>DOFINISH


DEFINE ENDDO (OUTLOOP DOLAB (ENDTOK OUTPUT MSG SOURCE
                             CURLAB LOOPLAB LINECT))
   <ENDTOK CONCAT:<OUTPUT OUTCON:DOLAB OUTCON:ADD1:DOLAB>
    MSG SOURCE CURLAB OUTLOOP LINECT>
.
-=>ENDDO
```

PARSE1 calls WHILCODE to process a WHILE statement. The standard-FORTRAN generated for a WHILE statement is

```
        CONTINUE
L       IF(.NOT.(condition))GOTO L+1
        statement
        GOTO L
L+1     CONTINUE
```

```
DEFINE WHILCODE (CURLAB LOOPLAB COLNO SOURCE LINECT)
    <1   APPEND2                          1   1   1   1   1>:<
    <#  OUTCON:0                          #   #   #   #   #>
    <#  1:OUTSTR:<ITOC:CURLAB COLNO> #   #   #   #   #>
     IFCODE:<WHILE CURLAB LOOPLAB 2:OUTSTR:<ITOC:CURLAB COLNO>
            RATTOK:<SOURCE LINECT>>>
.
-=>WHILCODE
```


IFCODE is called by WHILCODE to output the string

IF(.NOT.(condition))GOTO L+1

ENDARGS, a help function to IFCODE, calls ENDWHILE to process the end of the WHILE statement.

```
DEFINE ENDWHILE (OUTLOOP WHILELAB (ENDTOK OUTPUT MSG
                 SOURCE CURLAB LOOPLAB LINECT>>
    <ENDTOK CONCAT:<OUTPUT OUTGO:<WHILELAB 1> OUTCON:ADD1:WHILELAB>
     MSG SOURCE CURLAB OUTLOOP LINECT>
.
-=>ENDWHILE
```

ENDWHILE adds

```
        GOTO L
L+1 CONTINUE
```

to the end of the WHILE statement.

If the first token in a statement is a left brace (IF LBRACEQ:TOKTYP), PARSE1 calls ENDLBRAC. ENDLBRAC calls RBTOKQ to see if the next token in the input is a balancing, right brace. If it is, a ground condition is met; otherwise, ENDLBRAC is called recursively with a call to PARSE as its argument.

```
DEFINE ENDLBRAC (ENDTOK OUTPUT MSG SOURCE CURLAB LOOPLAB LINECT)
    IF EOFQ:ENDTOK
        THEN <ENDTOK OUTPUT MSG SOURCE CURLAB LOOPLAB LINECT>
    ELSE <1   APPEND          APPEND    1   1   1   1>:<
         <#   OUTPUT          MSG       #   #   #   #>
          RBTOKQ:<LEX:<SOURCE LINECT> CURLAB LOOPLAB>>
.
-=>ENDLBRAC
```

```
DEFINE RBTOKQ ((TOKTYP TOKEN MSG SOURCE LINECT) CURLAB LOOPLAB)
    IF RBRACEQ:TOKTYP
        THEN <TOKTYP <> MSG SOURCE CURLAB LOOPLAB LINECT>
    ELSE ENDLBRAC:PARSE:<<TOKTYP TOKEN MSG SOURCE LINECT>
                         CURLAB LOOPLAB 1>
•
-=>RBTOKQ
```

If a token of type OTHER (a non-RATFOR token type) begins an input line, PARSE1 calls OTHERC, which with its help functions, OTHERC1 and OTHERC2, gathers up and outputs the current input line.

```
DEFINE OTHERC (TOKEN CURLAB SOURCE LOOPLAB LINECT COLNO)
    <1   APPEND              1    1    1    1    1>:<
    <# 1:OUTTAB:COLNO        #    #    #    #    #>
     OTHERC1:<TOKEN CURLAB SOURCE LOOPLAB LINECT 2:OUTTAB:COLNO>>
•
-=>OTHERC

DEFINE OTHERC1 (TOKEN CURLAB SOURCE LOOPLAB LINECT COLNO)
    <1   APPEND                   1    1    1    1    1>:<
    <#  1:OUTSTR:<TOKEN COLNO>    #    #    #    #    #>
     OTHERC2:<EATUP:<SOURCE LINECT> CURLAB LOOPLAB
            2:OUTSTR:<TOKEN COLNO>>>
•
-=>OTHERC1

DEFINE OTHERC2 ((OUTPUT MSG SOURCE LINECT)
                CURLAB LOOPLAB COLNO)
    <OTHER SNOC:<NL 1:OUTSTR:<OUTPUT COLNO>> MSG SOURCE
     CURLAB LOOPLAB LINECT>
•
-=>OTHERC2
```


Output Routines.

The output routines discussed in this section are responsible for creating valid,standard-FORTRAN output.

The function OUTCH outputs one character, CHAR. It checks to see if the current column number for output, COLNO, is greater than RMARG, the maximum possible column number for valid, standard-FORTRAN output (I have set the value of RMARG to 72). If COLNO is greater than RMARG, NEWLINE is called to continue the character on the next line, flagging a continuation by placing an asterisk (STAR) in column 6 of the next line.

```
DEFINE OUTCH (CHAR COLNO)
    IF GREAT:<COLNO RMARG>
        THEN <NEWLINE:CHAR ADD2:LMARG>
    ELSE <<CHAR> ADD1:COLNO>
•
-=>OUTCH
DEFINE NEWLINE CHAR
    CONS:<NL APPEND:<PUTCHARS:<BLANK SUB1:LMARG> <STAR CHAR>>>
•
-=>NEWLINE
DEFINE ADD2 N
    ADD1:ADD1:N
•
-=>ADD2
```

OUTTAB inserts blanks in the output until the current column number for output is equal to LMARG plus 1 , the starting column number for a valid, FORTRAN statement. LMARG is set to 6.

```
DEFINE OUTTAB COLNO
    IF GREAT:<COLNO LMARG>
        THEN <<> COLNO>
    ELSE <CONS          1>:<
         <BLANK         #>
          OUTTAB:ADD1:COLNO>
.
 -=>OUTTAB
```

OUTSTR uses OUTCH as a help function to output a character string in valid, standard-FORTRAN format. HOLLRITH is called to convert a quoted character string to a standard Hollerith form.

```
DEFINE OUTSTR (STRING COLNO)
    IF NULL:STRING THEN <<> COLNO>
    ELSEIF OR:<SQUOTEQ:FIRST:STRING
               DQUOTEQ:FIRST:STRING>
         THEN OUTSTR:<HOLLRITH:<CHARCT:REST:STRING REST:STRING>
                     COLNO>
    ELSE <APPEND                        1>:<
         <1:OUTCH:<FIRST:STRING COLNO> #>
          OUTSTR:<REST:STRING 2:OUTCH:<FIRST:STRING COLNO>>>
.
 -=>OUTSTR

DEFINE HOLLRITH (N STRING)
    APPEND:<ITOC:N CONS:<H STRING>>
.
 -=>HOLLRITH
```

OUTLAB outputs a string of digits (TOKEN) as a standard-FORTRAN label.

```
DEFINE OUTLAB (TOKEN COLNO)
    <APPEND                 1>:<
    <1:OUTSTR:<TOKEN COLNO>  #>
     OUTTAB:2:OUTSTR:<TOKEN COLNO>>
.
 -=>OUTLAB
```


OUTCON outputs a FORTRAN CONTINUE statement. If its argument, LABEL, is a positive integer, OUTCON outputs a labeled CONTINUE statement. If LABEL is zero, the CONTINUE is output without a label.

```
DEFINE OUTCON LABEL
    CONCAT:<ITOCP:LABEL 1:OUTTAB:ADD1:CHARCT:ITOCP:LABEL
            1:OUTSTR:<CONTINUE ADD1:LMARG> <NL>>
.
 -=>OUTCON

DEFINE ITOCP LABEL
    IF ZEROP:LABEL THEN <>
    ELSE ITOC:LABEL
.
 -=>ITOCP
```

OUTGO, with its helper, OUTGO1, converts its integer argument, LABEL, to a character string, and outputs a FORTRAN GOTO statement.

```
DEFINE OUTGO (LABEL COLNO)
    APPEND:<1:OUTTAB:COLNO
            OUTGO1:<OUTSTR:<GOTO 2:OUTTAB:COLNO> LABEL>>
.
 -=>OUTGO

DEFINE OUTGO1 ((STRING COLNO) LABEL)
    CONCAT:<STRING 1:OUTSTR:<ITOC:LABEL COLNO> <NL>>
.
 -=>OUTGO1
```

Syntax Errors.

The function SYNERR is called by several routines to output syntax error messages. Given an indicator of the type of error that has occurred (TOKEN), SYNERR outputs an appropriate error message, giving the line number (LINECT) in the input file at which the error occurred.

```
DEFINE SYNERR (TOKEN LINECT)
   CONS:<NL APPEND:<SYNERR1:TOKEN ITOC:LINECT>>
.
 -=>SYNERR
```

```
DEFINE SYNERR1 TOKEN
    IF EOFQ:TOKEN THEN "(U N E X P E C T E D BLANK E N D BLANK O F
                        BLANK F I L E BLANK A T BLANK L I N E BLANK)
    ELSEIF QUOTEQ:TOKEN THEN "(M I S S I N G BLANK Q U O T E BLANK
                               A T BLANK L I N E BLANK)
    ELSEIF BREAKQ:TOKEN THEN "(I L L E G A L BLANK B R E A K BLANK
                               A T BLANK L I N E BLANK)
    ELSEIF NEXTQ:TOKEN THEN "(I L L E G A L BLANK N E X T BLANK
                              A T BLANK L I N E BLANK)
    ELSEIF LPARENQ:TOKEN THEN "(M I S S I N G BLANK L E F T BLANK
                               P A R E N BLANK A T BLANK L I N E
                               BLANK)
    ELSEIF RPARENQ:TOKEN THEN "(M I S S I N G BLANK R I G H T BLANK
                               P A R E N BLANK A T BLANK L I N E
                               BLANK)
    ELSEIF ELSEQ:TOKEN THEN "(I M P R O P E R BLANK E L S E BLANK
                              A T BLANK L I N E BLANK)
    ELSEIF RBRACEQ:TOKEN THEN "(U N E X P E C T E D BLANK R I G H T
                               BLANK B R A C E BLANK A T BLANK
                               L I N E BLANK)
    ELSE "(U N E X P E C T E D BLANK E N D BLANK O F BLANK F I L E
          BLANK O R BLANK R I G H T BLANK B R A C E BLANK A T BLANK
          L I N E BLANK)
.
 -=>SYNERR1
```


Trivial Help Functions Introduced in Chapter 9.

```
DEFINE EOFQ TOKTYP
    SAME:<TOKTYP EOF>
.
 -=>EOFQ

DEFINE BREAKQ TOKTYP
    SAME:<TOKTYP BRKTOK>
.
 -=>BREAKQ

DEFINE NEXTQ TOKTYP
    SAME:<TOKTYP NEXTOK>
.
 -=>NEXTQ
.

DEFINE ALLDIGQ TOKTYP
    SAME:<TOKTYP ALLDIG>
.
 -=>ALLDIGQ

DEFINE ELSEQ TOKTYP
    SAME:<TOKTYP ELSETOK>
.
 -=>ELSEQ

DEFINE LBRACEQ TOKTYP
    SAME:<TOKTYP LBRACE>
.
 -=>LBRACEQ

DEFINE RBRACEQ TOKTYP
    SAME:<TOKTYP RBRACE>
.
 -=>RBRACEQ
```

```
DEFINE LPARENQ CHAR
   SAME:<CHAR LPAREN>
.
 -=>LPARENQ

DEFINE RPARENQ CHAR
   SAME:<CHAR RPAREN>
.
 -=>RPARENQ


DEFINE DOTYPEQ TOKTYP
    SAME:<TOKTYP DOTOK>
.
 -=>DOTYPEQ


 DEFINE WHILEQ TOKTYP
     SAME:<TOKTYP WHILE>
.
 -=>WHILEQ


 DEFINE IFTYPEQ TOKTYP
     SAME:<TOKTYP IF>
.
 -=>IFTYPEQ


 DEFINE OTHERQ TOKTYP
     SAME:<TOKTYP OTHER>
.
 -=>OTHERQ


DEFINE QUOTEQ CHAR
   SAME:<CHAR QUOTE>
.
 -=>QUOTEQ
```


Constants Introduced in Chapter 9.

```
DECLARE TOKTBL
   <<"(I F) <IF> <"(D O) <DOTOK>>
    <"(E L S E) <ELSETOK>>
    <"(W H I L E) <WHILE>>
    <"(B R E A K) <BRKTOK>>
    <"(N E X T) <NEXTOK>>>
.
 -=> (((I F) (IF)) ((D O) (DOTOK)) ((E L S E) (ELSETOK))
      ((W H I L E) (WHILE)) ((B R E A K) (BRKTOK)) ((N E X T) (NEXTOK)))



    DECLARE EOF "EOF.
     -=>EOF

    DECLARE ALLDIG "ALLDIG.
     -=>ALLDIG

    DECLARE OTHER "OTHER.
     -=>OTHER


    DECLARE L23000 23000.
     -=>23000

    DECLARE BRKTOK "BRKTOK.
     -=>BRKTOK

    DECLARE DOTOK "DOTOK
     -=>DOTOK

    DECLARE ELSETOK "ELSETOK.
     -=>ELSETOK

    DECLARE WHILE "WHILE.
     -=>WHILE

    DECLARE IF "IF.
     -=>IF

    DECLARE LBRACE "LBRACE.
     -=>LBRACE

    DECLARE RBRACE "RBRACE.
     -=>RBRACE


    DECLARE QUOTE "QUOTE.
     -=>QUOTE
```

```
DECLARE RMARG 72.
 -=>72

DECLARE LMARG 6.
 -=>6

DECLARE STAR "STAR.
 -=>STAR

DECLARE H "H.
 -=>H

DECLARE GOTO "(G O T O).
 -=> (G O T O)

DECLARE IFNOT
    "(I F LPAREN . N O T . LPAREN)
 -=> (I F LPAREN . N O T . LPAREN)

DECLARE LPAREN "LPAREN.
 -=>LPAREN

DECLARE RPAREN "RPAREN.
 -=>RPAREN

DECLARE DOSTR "(D O).
 -=> (D O)

DECLARE EOFBRACE "EOFBRACE.
 -=>EOFBRACE

DECLARE NEXTOK "NEXTOK.
 -=>NEXTOK

DECLARE ALPHANUM
    <0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N
     O P Q R S T U V W X Y Z>

-=> (0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z)

DECLARE SHARP "SHARP.
 -=>SHARP

DECLARE CONTINUE "(C O N T I N U E).
 -=> (C O N T I N U E)
```


Index of Functions and Constants.

Chapter 1--Functions.

function name	page
BLANKQ	4
CHARCT	2
COPY	1
DETAB	8
EATWORD	5
LINECT	3
NLQ	3
OR	4
OUTWORD	4
TABBLNKS	11
TABQ	4
WORDCT	4

Chapter 1--Constants. (page 12)

constant name
BLANK
NL
TAB

Chapter 2--Functions.

function name	page
ADDCHARS	22
AFTER	31
APPEND	18
BEFORE	31
BLANKOUT	17
BLANKOVR	17
BMEMBER	32
BREMBER	32
BSQ	16
CHOP	31
CKBLANKS	15
CK3	26
CKREST	23
COMPRES1	21
COMPRESS	21
CONCAT	18
CONDPICK	33
CTOI	26
DASHQ	30
DIFFER	21
DIGVAL	25
DODASH	30
ENTAB	13
ESCAPEQ	30

Chapter 2--Functions(continued).

function name	page
EXPAND	25
IFADD	31
INDEX	27
INQ	28
ITOC	23
ITOC1	23
LESSEQ	25
LCONS	22
MAKESET	30
MAPDIGIT	27
MAX	17
MEMBER	32
MINUS	25
MINUSP	25
MINUSQ	27
MOD	24
NCHARSQ	23
NEWSKIP	17
ONEP	25
OSTRIKE	16
OSTRIKE1	16
PICK	33
POWERVAL	26
PROJ	25

Chapter 2--Functions (continued).

function name	page
PUTBLNKS	25
PUTCHARS	27
PUTDEC	23
RREST	27
RRREST	27
SNOC	24
SUBTOTL	26
TCONS	32
TRIMBS	18
XCHANGE	33
XLATE	29
XLATE1	33
XOR	32
ZEROP	28

Chapter 2--Constants (page 34)

constant name	constant name
BS	NS
DASH	PLUS
DIGITS	SKIP
ESCAPE	
MINUS	
NCHARS	

Chapter 3--functions.

function name	page
ADDFILE	46
ALLNEW	65
AND	36
APPEND	42
ARCHIVE	58
BADCOM	66
CANT	42
CKANS	47
CKNAMES	60
CKRESULT	46
COMPARE	37
CREATE	56
DELETEQ	58
DOCOMPAR	37
DOCOMPQ	37
DUPMESS	60
DUPNAME	60
EQSTR	39
ERRCOM	66
ERRMODE	58
ERRORQ	42
EXTRACTQ	69
FCOMPARE	42
FCONCAT	50

Chapter 3--Functions(continued)

function name	page
FCOPY	50
FDISPLAY	67
FDISP1	67
FETCHONE	50
FEXPAND	43
FMATCH	56
FOUNDQ	62
FPRETTY	52
FPRETTY1	52
FPRETTY2	52
GETCOM	66
GETCOM1	68
GETLINE	38
GETNAME	65
GETNAMES	60
GETSTR	44
GETWORD	44
GRABONE	62
HEAD	53
INCLUDE	43
INCLUDE1	43
INCLUDEQ	45
ISDIF	35
LDIFFER	35

Chapter 3--Functions (continued)

function name	page
LISTQ	68
LOCDIFF	35
MAKECOPY	55
MAKEFILE	65
MAKHDR	63
MAKUPDIF	53
MODECHEK	56
MTCHMODE	39
NEWFILE	65
NFOUND	67
ONEDISP	67
OPEN	39
OUTWORD	44
PHEAD	69
PRETTY	51
PRETTY1	51
PUTFILE	61
READWRTQ	39
SKIP	53
SUBFILE	55
SUM	52
TLIST	69
TOPPAGE	53

Chapter 3--Functions (continued).

function name	page
TRIM	44
TRYOPEN	66
UP2CREATE	64
UPDATE	58
UPDATE2	61
UPDATEQ	58

Chapter 3--Constants. (pp 70,71)

constant name	constant name
ARCHCOMS	TABLE
BOTTOM	UPDATE
DEEP	WRITE
DELETE	
ERROR	
EXTRACT	
LIST	
MARG1	
MARG2	
OK	
PAGESIZE	
PWIDTH	
READ	
READWRIT	

Chapter 4--Functions.

function name	page
ADBLNKS	91
ADCHNKS	90
ADDLINE	81
ALASCII	77
ALCHRVAL	76
ALFA	87
ALLROT	85
ALUNROT	89
ASCII	77
BUBBLE	72
CEQ	75
CHRVAL	76
CKADD	88
CKNULL	86
CKPAIR	84
CLEANLIN	88
CLESS	75
CLESSEQ	75
CONVERT	76
EQLINE	84
EXCHANGE	74
EXCHANG1	74
FILBLNKS	90

Chapter 4--Functions (continued)

function name	page
FOLDQ	87
GETSOME	82
GTEXT	81
GTEXT1	81
KWIC	85
LASCII	77
LINECONV	76
LINELEN	81
LLESS	75
LOLINE	83
LORDER	83
LQUICK	79
LSHELL	79
LTEQGT	79
MERGE	83
MERGEGP	82
MRGCHNKS	82
NEXCHNK	90
ORDER	73
POS	91
QUICK	78
QUICKRNS	81
REMEOLNS	88

Chapter 4--Functions (continued)

function name	page
REVERT	77
ROTATE	87
ROTBLNKS	87
ROTLINE	85
ROTLINE1	87
ROTWORD	87
SEPLINES	76
SETLEFT	91
SETRIGHT	89
SHELL	73
SHELL1	73
SHELL2	73
SHELL3	73.
SINKSMAL	72
SNOC	87
SORTINT	78
SORTX	80
SWITCH	74
TCONS1	83
TRIM	86
UNIQUE	84
UNROT	89

Chapter 4--Constants. (page 92)

constant name
ALPHANUM
FLIMIT
FOLD
HALFLINE
MAXCHARS
MAXFILES
TOPALFA

Chapter 5--Functions.

function name	page
ALCHANGE	102
AMATCH	98
AMATCHX	98
ANYQ	100
BOLQ	97
CHANGE	101
CHANGE1	101
CKREST	99
CLOSMACH	99
CLOSQ	100
DITTOQ	102
EOLNQ	100
FILSET	95
FIND	95
FINDX	96
GETLINE	96
MAKESUB	102
MATCH	97
MATCH1	97
NCCLQ	102
OMATCH	100
OMATCHAT	100
OMATCHL	101

Chapter 5--Constants (page 103)

constant name
ANY
BOL
CLOS
DITTO
EOLN
EXSETS
NCCL

Chapter 6--Functions.

function name	page
ADDEND	138
APPENDQ	167
ATEND	153
BACKSCAN	126
BAKWARDQ	166
BEHINDQ	147
BFIND	163
BMATCH	163
CANT	138
CARRANGE	154
CHANGEQ	167
CHECKG	156
CHECKP	153
CKAPPEND	164
CKEND	149
CKERR	160
CKFINAL	134
CKLSHIFT	125
CKRSHIFT	127
COMMAQ	130
CURBEGIN	133
CURLNQ	166
DELETEQ	167
DIGITQ	166

Chapter 6--Functions (continued)

function name	page
DIGVAL	123
DOAPPEND	151
DOCHANGE	154
DOCOM	144
DODELETE	155
DOENTER	142
DOGLOB	161
DOINSERT	151
DOPRINC	152
DOPRINT	155
DOPRINTF	143
DOREAD	150
DOSINGLE	141
DOSUBST	156
DOWRITE	146
EDAPPEND	151
EDARANGE	150
EDCHANGE	154
EDCREATE	148
EDINSERT	151
EDIT	139
EDIT1	140
EDREAD	150

Chapter 6--Functions (continued)

function name	page
EDWRITE	147
ENDQ	166
ENTERQ	142
ERRORQ	138
EXEC	144
FINALSET	132
FINDEND	124
FINDLINE	124
FINDX	126
FORESCAN	125
FORWARDQ	166
GCKERR	165
GETDIGS	123
GETFIRST	132
GETFSTAD	134
GETLAST	134
GETLIST	120
GETLIST1	120
GETMARK	163
GETNO	152
GETONE	130
GETSPOT	121
GETTARG	140
GLOBDEF	161

function name	page
GLOBALQ	140
GLOMARK	162
GVAL	140
IFFIND1	134
IFFIND2	134
IFOKGO	164
INORDER	124
INSERTQ	167
ISLAT	142
LASTQ	166
LENGTH	148
LMEMBER	147
LOCMRKER	133
LSHIFT	125
MARKRANG	162
MINUSMOV	129
MKGLOB	162
ONEADD	130
ONECHANG	158
PDEFAULT	149
PLUSMOVE	128
PLUSMOV1	128
PLUSQ	128
PRCGLOBS	163

function name	page
PRINC	152
PRINCQ	167
PRINTFNQ	143
PRINTOUT	155
PRINTQ	167
PROC1COM	164
PROCFRST	122
PROCREST	127
PUTCUR	132
PUTFIRST	133
PUTMARKS	162
PUTNO	152
QUITQ	139
RAC	127
RDC	127
READANS	150
READCQ	167
REVCONC2	133
REV2	126
REVERSE	126
RSHIFT	127
SARRANGE	158
SEMIQ	130

Chapter 6--Functions (continued).

function name	page
SETANS	149
SETARGS	144
SETGRANG	162
SETMINUS	129
SETSUB	159
SETUPC	131
SETUPS	136
SETUPS1	136
SGLOBALQ	156
SUBST	157
SUBST1	157
SUBST2	157
SUBST3	158
SUBSTQ	167
TEXTED	138
TEXTED1	138
WRITECQ	146
WTCOUNT	148
XOR	163

Chapter 6--Constants (pp 168,169).

constant name	constant name
APPEND	READC
BACK	SEMI
CHANGE	SUBST
COMMA	WRITEC
CURLN	
DELETE	
END	
ENTER	
ERROR	
FIRSTAD	
FORE	
GLOVAL	
GMARK	
INSERT	
LAST	
MINUS	
NOGLOB	
PDWIDTH	
PLUS	
PR	
PRINC	
PRINT	
PRINTF	

Chapter 7--Functions.

function name	page
AD1WORD	210
ADDBLNK	204
ADWORDS	200
ADDLINE	176
ALLBLNKS	198
APPEND2	206
APINDENT	207
BIGSPACE	186
BOLQ	211
BOTTOM	211
BRQ	215
BLDLINE	181
BPQ	215
CAPPEND2	201
CCONS	188
CENTER	198
CENTERQ	215
CEVAL	211
CHANGDIR	203
CKBNDS	190
CKFILL	197
CKLINE	180
COMQ	180
CONDREV	204

Chapter 7--Functions (continued)

function name	page
DIRQ	210
DOCOM	184
DQUOTEQ	191
EATBLNKS	185
EATWORD	185
ENDPUT	207
FILLQ	215
FILVAL	212
FOOTER	211
FOOTERQ	215
FORMAT	175
FORMAT1	175
GCOMTYP	185
GET	210
GETCOM	185
GETVAL	187
HEADER	211
HEADERQ	215
HEADQ	205
IDMARKQ	212
INCLL	197
INDENTP	211
INDENTQ	216

Chapter 7--Functions (continued).

function name	page
ISBR	214
ISBP	214
ISCE	214
ISFI	213
ISFO	213
ISHE	213
ISIN	213
ISLS	213
ISNF	213
ISPL	213
ISRM	213
ISSP	214
ISTI	214
ISUL	214
LASTPAGE	209
LEFTQ	203
LINO	211
LLVAL	212
LOCVALS	183
LSPACE	207
LSQ	216
LSVAL	211
MATCH2	214

Chapter 7--Functions (continued).

function name	page
MAXSPACE	193
NEWVAL	189
NFILLQ	215
NLNULL	182
NUMVAL	187
ONEWORD	202
OTEXT	195
OWIDTH	201
PAGENO	210
PARSECOM	185
PFOOT	207
PFOOT1	208
PHEAD	206
PLQ	216
PLVAL	212
PMARKQ	206
PROCCOM	194
PUT	202
PUTHEAD	205
PUTLINE	178
PUTTL	206
PUTWORDS	198
REMIDMK	196

Chapter 7--Functions (continued)

function name	page
REPVAL	183
RIGHTQ	204
RMQ	216
RMVAL	212
SEPDIGS	187
SETBOL	179
SETBOTTM	192
SETCTR	197
SETIDMK	196
SETIVAL1	209
SETLEN	199
SETLL	210
SETNUM	288
SETNUMTI	194
SETPAG	209
SETPL	192
SETTIVAL	209
SETVAL	189
SKIPBLKS	203
SLIPBLKS	204
SPQ	216
SQUOTEQ	191
SPACEP	211

Chapter 7--Functions (continued).

function name	page
SPREAD	202
SPREAD1	204
STRIP	191
SUBSPACE	180
SUBUNDER	183
TIVAL	211
TRIMLINE	185
ULQ	215
ULVAL	212
UNKNOWNQ	215
UPLINPAG	208
ZIPTI	196

Chapter 7--Constants (pp217-219)

constant name	constant name
BO	DQUOTE
BOL	FI
BR	FO
BP	HE
CE	HUGE
CPARAMS	IDMARK
CTRDEF	IN
DIR	INDEF

Chapter 7--Constants (continued).

constant name	constant name
LEFT	TIDEF
LL	UL
LN	ULDEF
LS	UNKNOWN
LSDEF	USCR
MARG1	
MARG2	
MARG3	
MARG4	
NF	
PL	
PLDEF	
PMARK	
PN	
PNWIDTH	
RIGHT	
RM	
RMDEF	
SP	
SPDEF	
SQUOTE	
TI	
TI2	

Chapter 8--Functions.

function name	page
ADDEF	231
ADDPARM	234
ADDSTR	238
ALLNUM	234
BALDELIM	226
BALPAR	229
BUMP	233
CK1ST	237
CK2ND	238
CKEVAL	232
CKSYNTAX	231
CKTOKEN	227
DEFTYPEQ	231
DODEF	230
DOIFELSE	237
DOINCR	233
DOSBSTR1	235
DOSBSTR2	236
DOSUBSTR	233
ENDOUT	225
ERROUT	225
FOLLOW	236
GETBAL	229

Chapter 8--Functions (continued).

function name	page
GETPARMS	234
GOTOBRAK	226
IFELSEQ	237
IFNUM	235
IFTOK	228
INCTYPEQ	233
LBRACKET	226
LOOKUP	227
LPARENQ	229
MACRO	224
MACRO1	224
MAKEDEF	231
MAKSTR	239
NOTALFA	227
OUTCHARS	236
PROJPARM	240
RBRACKET	226
READTOCH	234
RPARENQ	229
SEPTOKEN	227
SPANBRAK	226
SUBONE	239
SUBQ	239

Chapter 8--Functions (continued)

function name	page
SUBSTRQ	233
TOKMATCH	231

Chapter 8--Constants. (p 241)

constant name
ALPHANUM
BRACKETS
COMMA
DEEP
DEFTYPE
IFELSTYP
LBRAK
LPAR
MAXDEF
OK
RBRAK
RPAR
SUB
UNBAL

Chapter 9--Functions.

function name	page
ADD2	266
ALLDIGQ	270
BALQUOTE	249
BREAKQ	270
BRKNXT	254
CKCONT	262
CKRPAREN	262
CKTOK	250
DOCODE	260
DOFINISH	263
DONUM	260
DOTYPEQ	271
EATLINE	249
EATUP	260
EATUP1	261
ELSEQ	270
ELSETOKQ	259
ENDARGS	258
ENDDO	263
ENDELSE	255
ENDIF	258
ENDLBRAC	264
ENDSTAT	251

Chapter 9--Functions (continued).

function name	page
ENDWHILE	264
EOFQ	270
GETALFA	250
GETARGS	257
HOLLRITH	267
IFARGS	256
IFCODE	255
IFTYPEQ	271
ITOCP	268
LBRACEQ	270
LEX	248
LPARENQ	271
NEWLINE	266
NEXTQ	270
OTHERC	265
OTHERC1	265
OTHERC2	265
OTHERQ	271
OUTCH	266
OUTCON	268
OUTDO	260
OUTGO	268
OUTGO1	268
OUTLAB	267

Chapter 9--Functions (continued)

function name	page
OUTSTR	267
OUTTAB	267
PARSE	252
PARSE1	253
PUTARGS	257
QUOTEQ	272
RATFOR	251
RATTOK	248
RBRACEQ	270
RBTOKQ	265
RPARENQ	271
SHARPQ	249
SPECTOKQ	250
SYNERR	269
SYNERR1	269
WHILCODE	263
WHILEQ	271

Chapter 9--Constants (pp272,273).

constant name	constant name
ALLDIG	STAR
ALPHANUM	TOKTBL
BRKTOK	WHILE
CONTINUE	
DOSTR	
DOTOK	
ELSETOK	
EOF	
EOFBRACE	
GOTO	
H	
IF	
IFNOT	
L23000	
LBRACE	
LMARG	
LPAREN	
NEXTOK	
OTHER	
QUOTE	
RBRACE	
RMARG	
RPAREN	
SHARP	

References

MC -- J. McCarthy, P. J. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, MA (1962).

FW -- D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. IEEE Trans. Comput., C-27 (4) (1978) 289-296.

SJ -- S. D. Johnson. An Interpretive Model for a Language Based on Suspended Construction. M. S. Thesis, Indiana University (1977).

KP* -- B. W. Kernighan and P. J. Plauger. Software Tools, Addison-Wesley, Reading, MA (1976).

FW1 -- D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In S. Michelson and R. Milner (eds.), Automata, Languages and Programming, Edinburgh University Press, Edinburgh (1976), 257-284.

FW2 -- D. P. Friedman and D. S. Wise. Functional combination. J. Computer Languages 3, (1) (1978), 31-35.

---

* References to Software Tools are indicated by page number enclosed in square brackets.