Paul McJones

SERIES 60 (LEVEL 68)

MULTICS EMACS EXTENSION WRITERS' GUIDE

SUBJECT

Guide for Programmers Writing Extensions and Terminal Control Modules (CTL) in the LISP Programming Language for the Emacs Text Editor

SPECIAL INSTRUCTIONS

This manual presupposes thorough familiarity with the Emacs text editor, which is described in the *Emacs Text Editor Users' Guide*. Extensions and CTLs can be written by those without programming experience, but familiarity with some programming language is valuable. Experience with Lisp is useful, but not necessary.

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

CJ52-00

January 1980

Honeywell

PREFACE

This manual describes how to write user extensions to the Multics Emacs editor. The reader should be thoroughly familiar with the Emacs editor, proficient in its use, and acquainted with its visible organization. The Emacs Text Editor Users' Guide, Order No. CH27, provides this necessary information. The methods for writing terminal control modules (CTLs) to support additional terminal types are also described here.

Programming knowledge is not necessary to write extensions successfully, although it is helpful. Section 1 is a short introduction to extension writing. Section 2 provides a short course in Lisp, the programming language used for writing extensions, and the language in which the Emacs editor itself is written. Basically, the extension writer only needs to learn enough about Lisp to be able to imitate examples.

Section 3 shows, by example, how to write extensions. It includes the functions and forms most likely to be needed by the extension writer. Section 4 describes LDEBUG mode, the Emacs mode for debugging the Lisp code used in extensions. Finally, Section 5 demonstrates how to write a CTL to support a new terminal type. Again, the CTL writer uses existing CTLs to learn to write his own.

This manual contains sufficient information to effectively write and debug Multics Emacs extensions. However, it is not intended to be a reference document for either Lisp, in general, or Multics MacLisp. Reference documentation for MacLisp is available from:

MIT Information Processing Center
Publications Office
60 Vassar Street
Cambridge, MA 02139

CONTENTS

CONTENTS (cont)

SECTION 1

INTRODUCTION

An editor extension is a user-provided capability, which is added to the editor to extend its power. It is different from a macro, which is simply a collection of editor requests gathered up and (perhaps) given a name. Extensions are programs; they are written in the language of the Multics Emacs environment. An extension is a body of code that augments the editor's capability, but does not embed or require knowledge of how data in the editor is stored or manipulated. In this sense, all of the word, sentence, paragraph, and Lisp-list requests, and the various "modes" (e.g., PL/I mode) are extensions.

The person who wishes to add to his Emacs environment any powerful or sophisticated capability must learn to write extensions. The keyboard macro facility (^X(, ^X)) is not intended for such usage. This manual explains how to write extensions.

One of the guiding design principles in the Emacs editor was that the creation of editor extensions, either by the editor implementors or end users, should be in a programming language of established elegance and power. Lisp was the language chosen. This primer gives you a starting point for writing Lisp code to run as editor extensions in the Emacs environment. If you have some knowledge of Lisp already, it will be of value. However, it is assumed in this manual that the reader has no familiarity with Lisp, but does, perhaps, with PL/I or BASIC.

For examples of extension coding, the extension writer's ultimate reference material will be the Emacs source. The Emacs mail system (RMAIL), FORTRAN and PL/I modes, and the code for the word, sentence, and paragraph requests (along with most of the other code in the Emacs module e_macops_.lisp) are standard examples of extension code. Techniques, styles, and subtleties difficult to convey in print may be gleaned by careful study of this code.

SECTION 2

AN INTRODUCTION TO LISP

Lisp programs are built of functions, which are similar to procedures or subroutines in other languages, although more akin to PL/I and ALGOL functions. You write a Lisp program by creating a file full of function definitions. A function definition specifies the name of a function, and what it does. Here is a sample function definition:

```
(defun addandmult (a b c)          ;This is a comment
       (* (- a b)
          (+ a b c)))
```

This defines a function named addandmult that takes three arguments, called a, b, and c. The addandmult function computes the result of multiplying the difference of a and b by the sum of a, b, and c, and returns that number as a result, or value. The semicolon on the first line above begins a comment; comments throughout the examples provide some additional information about the code.

Here is another function definition:

```
(defun squareprint (arg)
       (print "The square of")
       (print arg)
       (print "is")
       (print (* arg arg))
       5)
```

This function prints the message "The square of", prints the value of its argument, prints the word "is", and prints the value of the square of its argument. In addition, it returns the value 5. The function "squareprint" has side effects: it causes output on the terminal. It also returns a value, the number 5. Note that all Lisp functions produce a value; only some produce side effects. The first function defined returns the product of those numbers as a value; the second returns 5.

If you look at squareprint, you see that it consists of several statements, the "print statements" that print things. These statements are called forms, and they are, in fact, calls to other functions, in this case the builtin print function. In the form:

```
(print "The square of")
```

the string "The square of" is being passed as an argument to the print function. Like all functions, print returns a value, which is not used in this case. The side effect of printing something does occur. In the form:

```
(+ a b c)
```

you are invoking the "+" function, which is also builtin. The values of the parameter variables a, b, and c are passed to it as arguments. It returns a value, which is the requested sum, and produces no side effects.

There are five forms in the function-definition for squareprint:

```
(print "The square of")
(print arg)
(print "is")
(print (* arg arg))
5
```

Forms immediately inside a function definition are executed sequentially, like statements in other programming languages. The value produced by the last form is the one the function itself returns. What does it mean to "execute" a 5? Execute is not exactly the right term; what really happens is that these forms are evaluated. This means that a value is produced from them. Evaluating a 5 produces the number 5; evaluating the form:

```
(+ a b c)
```

calls the "+" function with the appropriate arguments, and produces whatever value the "+" function returns. The value produced by the "print" function is something that is not interesting, but a value is produced.

Numbers, like 5, and strings, like "The square of", are said to evaluate to themselves. Things between parentheses, like:

```
(+ a b c)
(print "The square of")
```

are calls to functions, which are evaluated by calling the function indicated, and producing the value it returns.

Function calls have the syntax:

```
(FUNCTIONNAME ARGFORM1 ARGFORM2 ARGFORM3 ... ARGFORMn)
```

where FUNCTIONNAME is the name of the function to call and the ARGFORMs are themselves forms, which are evaluated to produce the arguments to give to the function. Thus, to evaluate (i.e., "execute" and find the value returned) a form like:

```
(+    (*  a  b)
      15
      c)
```

- evaluate the <u>inner form</u> (* a b) to produce a value
- evaluate the 15 to produce 15 (remember, numbers and strings evaluate to themselves)
- evaluate the <u>variable</u> c to produce its value
- pass these three values on to the "+" function, and return what it returns.

The newlines are ignored.

Thus, forms are either numbers like 5, strings like "is", variables like b, or function calls like (* a b).

Variables are much like variables in other languages. A variable has a value, which is called its <u>binding</u>. At this stage, assume that this value must be a string or a number. When a function is invoked, the parameter variables (like a, b, and c above) of the function acquire the arguments of the function call as bindings. Evaluating a variable produces its binding as a value. For instance, if someplace in a function you evaluate the form:

```
(addandmult 2 (+ 3 2) 6)
```

a, b, and c will have the bindings 2, 5, and 6 while the forms in the definition of <u>addandmult</u> are being evaluated. This is not unlike the subroutine parameter mechanism in other languages. It is different insofar as it specifies what <u>value</u> a variable has during "subroutine" execution. In PL/I or FORTRAN, a parameter is associated with a variable in the calling program, not a value, during subroutine execution.

There are parameter variables, as used above, temporary variables, described below, and global variables. Regardless of the kind of variable, they all have bindings (values), and evaluation of the variable produces that value.

To summarize:

1. Lisp programs are built of functions.

2. Function definitions consist of the word "defun", the function's name, a parameter list, and a number of forms, which are to be sequentially evaluated at function call time, with a pair of parentheses around the whole thing.

3. The value of the last form in a function is the value returned by that function.

4. Forms can be strings, numbers, variables, or calls to functions. Forms are evaluated to produce values, which are passed between functions as arguments and results.

5. Strings and numbers evaluate to themselves.

6. Variables evaluate to the datum to which they are bound, which, for a parameter, is the corresponding argument to the containing function.

7. Function calls contain the name of a function to call and forms that are evaluated to produce the arguments to the function. Function calls may produce side effects. Like any form, when a function call is evaluated, it produces a value.

PREDICATES

Programming languages need conditional execution. In order to control conditional execution, you need things upon which to base a decision. Two data objects in the Lisp world correspond to truth and falsity, for the purposes of parts of the Lisp system that deal with conditions. A set of functions called predicates return these objects as values. For instance, a function called ">", invoked as:

```
    (> 4 6)
```

returns the indicator of falsity, and when invoked as:

```
    (> 4 1)
```

returns the indicator of truth. Predicates work just like other builtin and nonbuiltin functions, like print, addandmult, squareprint, and +. They take arguments, and produce a result. In the case of predicates, however, the result is not a string or a number, but an indication of truth or falsity. The result of a predicate can be used by the if special form (see below) to control the execution of a function.

The following are some of the most useful Lisp predicates. In all of these examples, A1, A2, S1, O1, etc., stand for forms, which means they can be 12, (+ 6 q), (myfun 33 (- a b)), etc. "A1 is a number," below, means that A1 is some form which evaluates to a number, such as 3, (+ 6 2), or x49, if x49's value is indeed a number.

Predicates for Numbers

A1 and A2 are numbers:

Predicate	Example	Returns TRUTH if ..., otherwise falsity.
=	(= A1 A2)	A1 and A2 are the same number.
>	(> A1 A2)	A1 is a bigger number than A2.
<	(< A1 A2)	A1 is a smaller number than A2.

Predicates for Strings

S1 and S2 are strings:

samepnamep
(samepnamep S1 S2)
S1 and S2 are strings of identical content, i.e., the "same string". This is the standard way to see if two strings are the same, as in (samepnamep test "-hold")

alphalessp
(alphalessp S1 S2)
S1 collates before S2 alphabetically, e.g., (alphalessp "Able" "Baker") returns truth, but (alphalessp "Zeke" "Joe") does not.

Predicates for Any Objects

O1 is some object, of perhaps unknown type (objects are discussed later):

eq	(eq O1 O2)	O1 and O2 are the same symbol or the same cons.
fixp	(fixp O1)	O1 is a number, as opposed to some other kind of object.
stringp	(stringp O1)	O1 is a string, as opposed to anything else.
symbolp	(symbolp O1)	O1 is a symbol, as opposed to anything else.

null	(null 01)	01 is not only a symbol, but the important and critical symbol named "nil".

LISP SPECIAL FORMS

A number of special forms in Lisp do not go by the simple rules given above. You have already seen one. The function-defining form, which begins with the word "defun", is not simply a function call with forms to produce the function's arguments. By all rights, a form like:

```
(defun square (x)
       (* x x))
```

should evaluate, in order, to produce arguments for "defun":

1. A variable named "square".
2. The form, (x), calling a function named "x" with no arguments.
3. The form, (* x x), multiplying the value of a variable named "x" by itself,

This form should then pass these three values on to the defun function. This, however, is not what actually happens. Evaluating the defun form causes a function named square to be defined, whose parameter list and "body" are as given. Defun is a special form, and when Lisp sees "defun" as the function name in a form, it acts in a special way. In this case, Lisp defines a function built out of this form itself. The above is not a call to defun with arguments. It may seem unusual, but you must have at least one such special form in order to have an operative Lisp system.

The if Special Form

A special form in the Multics Emacs Lisp environment, called if, controls conditional evaluation. An example of its use:

```
(defun which-is-greater (first second)
       (if (> first second)
           (print "The first one is the greater.")
           else
           (if (> second first)
               (print "The second one is greater")
               else
               (print "They are equal")))
```

The syntax of _if_ is as follows:

```
     (if <PREDICATE>
          <THEN-FORM-1>
          <THEN-FORM-2>
              ...
          <THEN-FORM-m>
        else
          <ELSE-FORM-1>
          <ELSE-FORM-2>
              ...
          <ELSE-FORM-n>    )
```

Any number, including none, of THEN-FORMs can be supplied. Similarly, any number, including none, of the ELSE-FORMs can be given. If there are no ELSE-FORMs, then the keyword "else" may be omitted, too.

Note that all the forms in the _if_ are _not_ sequentially evaluated; the word _else_ is not even intended to be a form. If all of the forms inside the _if_ were evaluated, it would be useless, for evaluation would not be conditional. That is why _if_ is a special form; there are special rules about how forms inside it are to be evaluated. The rule for all nonspecial forms is the same: you evaluate all the subforms sequentially to produce the arguments to the function. Each special form has its own rules.

The _if_ special form evaluates the PREDICATE: if it results in truth, the THEN-FORMs are sequentially evaluated, and the value of the last one is returned as the value of the _if_. Otherwise, the ELSE-FORMs are evaluated sequentially, and the value of the last returned. If there are none, the symbol nil (see below) is returned, but is useless in these cases.

There are two global variables in Lisp, called "t" and "nil", whose bindings are always the truth and falsity indicators, respectively. Thus:

```
     (if t
          (print "Truth")
        else
          (print "Not so truth"))
```

when evaluated, always prints "Truth".

The setq Special Form

Variables acquire values by being parameters, and acquiring values at function call time. In addition, variable values can be changed by the special form setq:

```
(defun adder-of-one (x)
       (print "The value of x is")
       (print x)
       ("And the value of x plus one is")
       (setq x (+ x 1))
       (print x))
```

A setq form has the word "setq", the name of a variable, and an inside form. The inside form is evaluated, and that value assigned to the variable. It is like an assignment statement in other languages.

The do-forever and stop-doing Special Forms

The construct for looping in the Emacs Lisp environment is also a special form, called do-forever:

```
(do-forever
    (print "Yay Multics")
    (print "scitluM yaY"))
```

When evaluated, it prints these two sayings forever. The way you stop doing in a do-forever is to evaluate the stop-doing special form:

```
(defun print-n-times (n)
       (do-forever
         (if (= n 0)(stop-doing))
         (print "foo")
         (setq n (- n 1))))
```

This function, given a number as an argument, prints "foo" that many times. The "=" builtin function/predicate compares its two arguments, which must be numbers, and returns truth or falsity depending on whether or not they are numerically equal. The arguments to = are not n and 0, but rather, the numbers that are the bindings of n and 0. The number which is the binding of n is different each time around the loop; that is the point of the program. It is setq that changes the value of n each time around, as do-forever executes the loop. A do-forever form generally returns something useless (nil), unless you exit by saying (return 5) or (return nil), or (return a). In the latter case, the value of the variable a is returned.

The let Special Form

You can acquire temporary variables via the let special form:

```
(defun sumtimesdif (x y)
   (let ((sum (+ x y))
         (dif (- x y)))
        (print "Sum times difference is ")
        (print (* sum dif))
        (print "Sum squared is")
        (print (* sum sum))))
```

This function has two temporary variables, sum and dif, which are initialized to the values of (+ x y) and (- x y). The general syntax of let is:

```
(let ((VAR1 VAL1)
      (VAR2 VAL2)
      ...........
      (VARn VALn))
     <FORM1>
     <FORM2>
     .......
     <FORMm>)
```

The temporary variables VAR1...VARn exist only within the let. They get the initial values of VAL1-VALn, which are forms that will be evaluated. All the VALs are evaluated before any of their values are assigned to the VARs. Then, with all these temporary variables set up and initialized, each FORMi is evaluated sequentially, and the value of the last FORMi is returned by let.

The prog and go Special Forms

Another, less useful way of acquiring temporary variables is via the special form prog. Forms inside a prog are evaluated sequentially, like forms in a function definition. However, the first form in a prog is not really a form at all, but a list of temporary variables used in the prog, such as "(a b c)". That is why prog is a special form. The value returned by prog is usually useless, unless (return...) is used to return something meaningful.

Inside a prog, you can put labels, to use for go-to's:

```
(defun bar2 (x y)
       (prog ()                    ;note the empty variable list
            (if (< x y)(go lab1))
            (print "X is not less than Y")
            (return nil)           ;return "false" indication
        lab1
            (print "so be it ")
            (return t)))           ;return "true" indication
```

In the special form <u>go</u>, its <u>operand</u> (not argument) is a <u>label</u> to which to go, i.e., continue sequential evaluation of forms in the prog. Labels are rarely needed, due to the powerful <u>if</u> and <u>do-forever</u> constructs.

<u>The or And and Special Forms, And not</u>

There are special forms for or-ing and and-ing predicate results: they are special because they stop evaluating their operands (from which arguments are produced) when they "know" their answer for certain:

```
    (if (and (not (= x 0))
             (> (// 10 x) 5))
        (print "Quotient too large."))
```

The <u>not</u> function inverts truth and falsity. The double slash indicates division, because slash is the escape character in Lisp.

The <u>and</u> does not attempt to evaluate the second form within it if the first produces falsity. This prevents an error that would result if an attempt were made to divide by zero. Sequential execution and stopping at an intermediate result are defined and useful features here, as opposed to the logical operators of, say, PL/I.

<u>The progn and prog2 Special Forms</u>

Two more special forms are progn and prog2. To force sequential execution of forms and return the value of the last, use progn. For instance:

```
     (if (and (> x 3)
              (progn (print "Oh dear this is getting serious")
                     (> y 5))
              (print "Fatal difficulty")))
```

In the above, progn returns the value of its last form. Thus, the <u>and</u> tests whether x is greater than 3, and y is greater than 5, before the "print" of "Fatal difficulty" is evaluated. The printing of "Oh dear..." occurs as part of the evaluation of the progn, but the and sees only the second value in the progn. The progn is used to force evaluation of the print form.

A prog2 is just like progn, except that it returns its second argument, evaluated, rather than its last. It must have at least two arguments. It is useful for saving some value that is subsequently going to be destroyed. The following form, when evaluated, interchanges the values of x and y:

```
(setq x (prog2 0               ; this zero is evaluated to 0,
                               ; and its value thrown away.
               y               ; the value of y is obtained here,
                               ; and remembered as it is here.
               (setq y x)))    ; x is evaluated, and that value
                               ; assigned  to y. The value of
                               ; setq form is that value.
```

In the above, however, the value of prog2 is that value of y as it was before it was assigned into y, and now the outer setq assigns that to x.

SYMBOLS

Another type of data object in Lisp is called the symbol. Symbols are named data objects kept in a registry of symbols, by Lisp. For current purposes, there is only one symbol of any name. Symbols are used in Emacs to represent buffer names, and various quantities associated with buffers. Lisp uses symbols to keep track of functions, and internally to keep track of global variables.

To use a symbol in a program, give the name of the symbol preceded by the ASCII quote character, '. For instance, the form:

```
(setq x 'Brunhilde)
```

assigns the symbol named Brunhilde to x. Note that this is different from:

```
(setq x "Brunhilde")
```

which assigns the string Brunhilde to x, and from:

```
(setq x Brunhilde)
```

which assigns the value of the variable Brunhilde to x.

LISP LISTS

The final Lisp data type of importance in writing extensions is the cons (for construct), and the larger data type built out of it, the list. A cons is a block that relates to two (usually other) objects in the environment, known as its car and its cdr. The function cons, given two objects, produces a new cons, whose car and cdr, respectively, are the two objects given. For instance, if the variable x has a value of the string "Brunhilde", as above, then:

```
(cons 7 x)
```

produces a cons whose car is the number 7 and whose cdr is the string "Brunhilde", returning it as a value. The functions car and cdr can be used to obtain the car and cdr of a cons. If you set the variable c to the result of the form (cons 7 x) above, then:

```
(car c)
```

produces the number 7 as a value.

Usually, you make larger and larger structures out of conses, by setting up conses whose car and cdr are more conses, and so forth, until you have a large enough structure to represent all the values you need. The resulting construction serves the same purpose as a PL/I structure: its various parts have meaning assigned by the programmer.

The most common construction of conses is the list. A list is defined as a chain of conses, each of which has the next one in the chain as its cdr, except the last one, which has the symbol "nil" as its cdr. A list built in this way of n conses is called a list of n elements, the elements being the n objects that are the cars of the conses. The cons at the head of the list is identified as being "the list": its car is the first element in the list, its cdr is the cons whose car is the second element of the list, and so forth. To construct a list of the numbers 2, 4, 5, and 7, in that order, and set the variable b to it, you would need:

```
(setq b (cons 2 (cons 4 (cons 5 (cons 7 nil)))))
```

(Note that the variable "nil" is peculiar insofar as its value is always the symbol "nil", thus you need not say 'nil.)

A function that simplifies the writing of such forms, for constructing lists, builds lists directly and accepts any number of arguments. It produces the same result as the type of construction shown above. It is called "list":

```
(setq b (list 2 4 5 7))
```

To get the third element of the list, once this form is evaluated, you could evaluate the form:

```
(car (cdr (cdr b)))
```

(i.e., the car of the cons that is the cdr of the cons that is the cdr of the cons that is the value of b). Again, there are Lisp functions to simplify such constructions. The above form is equivalent to:

```
(caddr b)
```

In general, for up to 4 cars and cdrs deep, total, functions like cadr, cdar, caddr, cadar, and so forth, are provided (up through caaaar and cddddr). The first four elements of a list are gotten by car, cadr, caddr, and cadddr (it is a good exercise to work that through and verify why this is the case).

When lists are printed out by Lisp, they are represented as a pair of parentheses around the printed representations of all of the elements, in sequence, separated by spaces. Thus, if Lisp printed out the list that was b's value above, it would appear:

```
(2 4 5 7)
```

A cons whose cdr is the symbol nil can always be viewed as a list of one item, and is so printed out by Lisp, unless it is in the process of printing a larger list of which the cons at issue is a chain-link. A cons whose cdr is neither nil nor another cons is printed with a dot preceding the cdr. Thus:

```
(cons 'a 'b)  =>  (a . b)
(cons 'a nil) =>  (a)              ;a list of one element
(cons 'a (cons 'b 'c)) => (a b . c)
(cons 'a (cons 'b nil)) => (a b)   ;list of two elements
(cons 'a (cons (cons 'b 'c)(cons 'd nil)))
         => (a (b . c) d)          ;list of three elements
```

Lists can be put into programs, by quoting them, as symbols are quoted:

```
(setq b1 '(this is (a list)(of lists)))
```

Two functions are provided to redefine the car or cdr of an existing cons. They can be very dangerous if misused, especially if they alter a list as in the form above, which is written into a program as a constant. The rplaca function (replace car) and the rplacd function (replace cdr) each take two arguments. The first is the cons that is to be altered, and the second is the new car or new cdr, respectively. The returned value is the cons itself.

SECTION 3

WRITING EMACS EXTENSIONS

The starting point for writing extensions is building functions out of those provided in the Emacs Lisp environment, and hooking them up to keys. The Emacs set-key and set-permanent-key extended requests can connect keys to Lisp functions that you provide, as well as to the standard requests and keyboard macros.

Many simple and useful extensions are just groups of Emacs requests strung together. For instance, to go to the beginning of a line, delete all whitespace there, go to the end of the line, do the same, and then return to the beginning of the line, you could type:

```
     ^A ESC \ ^E ESC \ ^A
```

Alternatively, you could write a function, called shave-line here, to do the same:

```
(defun shave-line ()          ;keystroke functions take no args.
       (go-to-beginning-of-line)
       (delete-white-sides)
       (go-to-end-of-line)
       (delete-white-sides)
       (go-to-beginning-of-line))
```

Write this function into a file. When in Emacs, type ESC X loadfile PATHNAME CR, to load it in as code. Then hook it up, perhaps by typing:

```
     ESC X set-key ^XA shave-line CR
```

Thereafter, hitting ^XA causes the chosen sequence of actions to happen.

To use conditionals and variables, you might, for example, want a function that goes to the beginning of a line and deletes all words that start with "foo" from the beginning of the line.

```
    (%include e-macros)

    (defun foodeleter ()
           (go-to-beginning-of-line)
           (do-forever
              (if (looking-at "foo")
                  (delete-word)
                  (delete-white-sides)
                else (stop-doing))))
```

The (%include e-macros) must be at the beginning of any file that uses the Emacs environment Lisp macros. The e-macros.incl.lisp file should be in your "translator" search path in order to do any Emacs extension development work.

What this function does in essence is type ^A, and as long as the first three characters on the line are "foo", does an ESC D, followed by ESC \ to remove the whitespace after the word. When the first three characters are no longer "foo", it returns. The "looking-at" is an Emacs predicate (to be described in detail below) that tests whether a given string is to the right of the current "cursor". For this function and any others that you write, you could set a key as described above (^XA for shave-line).

The code for the foodeleter makes no mention of printing, output, or displays because it does not need to. The screen or printing terminal is managed automatically by the Emacs redisplay. The display need never be thought about in coding Emacs extensions.

USING EMACS REQUESTS IN EXTENSION CODING

Many of the Emacs requests can and should be used in coding extensions, for example, go-to-end-of-line, forward-char, go-to-beginning-of-buffer, delete-word and skip-over-indentation. Some requests, however, should not be used in extension code. For example, if you want to search for some string, you do not want to invoke string-search (^S), since that prompts the user in the minibuffer for a search string. The following table lists some important keystroke requests whose command names you should not use and gives alternative functions to use.

KEY	DO NOT USE	USE INSTEAD
^N	next-line-command	next-line

The next-line-command function is unnecessarily expensive in considering screen position, and handles numeric arguments. The next-line function always goes to the beginning of the next line.

^P	prev-line-command	prev-line	Same reasons as above. The prev-line function always goes to the previous line.
^K	kill-lines	kill-to-end-of-line delete-char (at eol)	The kill-lines function is complex, has many cases, and handles numeric arguments.
^S	string-search	forward-search	The forward-search function takes a string as a Lisp argument, does not prompt, moves the cursor if the search succeeds, and returns truth or falsity to indicate result.
^R	reverse-string-search	reverse-search	Same as ^S.
^X^R	read-file	read-in-file	The read-in-file function takes a Lisp argument for pathname, does not prompt.
^X^W	write-file	write-out-file	Same as ^X^R.
^W	wipe-region	wipe-point-mark	Use local marks, see below.
ESC /	regexp-search-command	regexp-search	Same issues as ^S. Takes a Lisp argument, no slashes. Returns falsity if not found or moves cursor to after, and returns mark to before, matched string. Be careful to release this mark (see below).
^XB	select-buffer	go-to-or-create-buffer	Takes an argument, does not prompt.
^X^F	find-file	find-file-subr	Takes an argument, does not prompt.

Requests that accept a positive numeric argument as meaning repeat that number of times, e.g., ^B, ^D, ^F, ESC B, ESC D, ESC F, #, ESC #, etc., are acceptable in extensions; they do not inspect their arguments. They are invoked multiple times by the Emacs listener if appropriate. Requests whose names include the word "command" (other than ^G, command-quit) are usually not intended to be used in code.

The value of a numeric argument, e.g., 5 in ESC 5 ^B, is available as the binding of the global variable "numarg"; if no numeric argument is given, this variable is set to the symbol "nil" (not to be confused with the global variable nil, whose binding is the symbol nil), which is the representation of falsity. The defcom facility, discussed later, can be used to advantage as well.

The normal printing characters are bound to the self-insert function, which inserts the last physical character typed at the current point in the buffer. This is clearly unusable from code, if your desire is to insert text into the buffer. For this purpose, the Emacs environment provides the insert-string function, whose argument is a string to be inserted into the buffer at the cursor. As in typing in text manually, the cursor is left after the inserted text:

```
(defun make-a-point ()
       (go-to-beginning-of-line)
       (insert-string "CASE IN POINT:   "))
```

This make-a-point function, when invoked, goes to the beginning of the line, and inserts the string "CASE IN POINT: " in the buffer. The cursor is left after the inserted string.

As used here, phrases like, "the cursor is moved around" or "a string is inserted" in a function, do not imply that the user watching the screen can see all these things happen. No action on the screen occurs until the entire function has finished running, at which time the screen is updated all at once, showing the cumulative effect of what has happened, regardless of how it happened.

MARKS AND THEIR MANAGEMENT

Like the cursor, a mark is a conceptual pointer to the position between two characters in the current buffer. Marks remain between these two characters regardless of other insertions or deletions in the same buffer, even on the same line as the mark. Marks are valuable because regions of text in the buffer are specified as the extent between the current conceptual cursor, (the point), and a given mark. Marks are a type of data object in the Emacs Lisp environment, like strings, numbers, and symbols. The value of any variable can be made to be a mark. The value of several variables might even be the same mark. The words "the-mark" used in Emacs descriptions designate one mark that is the value of a global variable that many supplied functions know about. Emacs functions use many temporary marks.

The set-mark, release-mark and wipe-point-mark Functions

The set-mark function creates a new mark, which points to the current point in the current buffer. It stays around, and is updated by the editor, any time text is inserted or deleted in this buffer. This is expensive, so you must take care to discard, or release marks when you are done using them. This is done by giving them to the release-mark function. An example of a function which deletes three words and everything between them follows:

```
(defun delete-three-words ()
       (let ((temp-mark (set-mark)))     ;make a mark in
                                         ;a temp var.
             (do-times 3 (forward-word)) ;3 words forward
             (wipe-point-mark temp-mark) ;wipe out the stuff
                                         ;between point and
                                         ;where point was.
             (release-mark temp-mark)))
```

The variable temp-mark is set to a mark representing the point at the time delete-three-words is entered. The "do-times" is a special form that repeats the evaluation of one or more forms a given number of times. Its syntax is:

```
    (do-times <HOWMANY> <FORM1> <FORM2> .. <FORMn>)
```

The wipe-point-mark is a function that, given a mark, takes all the text between point at the time it is invoked and that mark (i.e., point at the time that mark was created) and deletes it from the buffer. It is, however, pushed on to the kill ring, so that ^Y can be used to retrieve it. If you do not want it pushed onto the kill ring, use:

```
     (without-saving (wipe-point-mark temp-mark))
```

instead of:

```
     (wipe-point-mark temp-mark)
```

and no saving occurs. After the computation, the mark is freed, (for better performance).

The with-mark Special Form

The sequence of setting a mark, using it, and releasing it is so common that a special construct in the Emacs Lisp environment is provided that takes care of all of this, including the creation of a temporary variable, so no prog or let is needed. It is called with-mark. The delete-three-words function, rewritten to use it, looks like this:

```
(defun delete-three-words ()
       (with-mark m   ;m is usually used for the name of a mark
                  (do-times 3 (forward-word))
                  (wipe-point-mark m)))
```

The syntax of the with-mark construct is:

```
     (with-mark <MARKNAME>
        <FORM1>
        <FORM2>
          ...
        <FORMn> )
```

It means: "Where I am now, call that <MARKNAME>. Evaluate the forms <FORM1> to <FORMn>, sequentially, returning the value of the last one as a value. Before returning anything, however, free the mark I made."

Marks allow you to return easily to where you were at the time you started something. The following function truncates a line longer than 50 print positions, and handles backspaces and tabs properly:

```
(defun trunc-50 ()
     (with-mark m                      ;remember where you started
       (go-to-end-of-line)
       (if (> (cur-hpos) 50.)          ;dot is for decimal
                                       ;default is octal
           (go-to-hpos 50.)
           (kill-to-end-of-line)) ;what ^K does at not e.o.l.
       (go-to-mark m)))                ;return to where you were
```

A function that tells you the horizontal position (on a dprint, not on the screen) of the current point is <u>cur-hpos</u> (the left margin is considered to be 0). It takes no arguments. The function go-to-hpos moves point to a position on the current line whose horizontal position is its argument, or the end of the line, if the line is shorter than that.

The "(go-to-mark m)" above tells the editor to move the current point in this buffer to the point where it was at the time the mark, to which the variable m is bound, was created.

The save-excursion Special Form

Although moving the editor's point to previously saved marks is extremely common, just using the mark mechanism to remember where you were before some excursion and get back there is so common that a special mechanism is provided just for this: it is called save-excursion, and it deals with all the issues of temporary variables and releasing the mark when done. The sample function trunc-50 recoded to use it looks like this:

```
     (defun trunc-50 ()
            (save-excursion
               (go-to-end-of-line)
               (if (> (cur-hpos) 50.)
                   (go-to-hpos 50.)
                   (kill-to-end-of-line))))
```

The save-excursion special form does the following: remembers where you are, via a mark saved in an internal variable, evaluates all of the forms within the save-excursion, and returns as a value the value of the last one. Before returning anything however, it moves the editor point back to where it was when the save-excursion was first entered, and releases the mark used to remember this place.

If point were at print position 75. at the time trunc-50 was called, it winds up at position 50, even though the mark to which it wants to return points to what was at position 75. No error is indicated, or has occurred. Marks remain even if characters to the right or left of them are deleted.

CLEANUP HANDLERS

You may have wondered, in the previous section, what happens if an extension encounters an error while executing, and never gets to release a mark it has set. When errors occur (for example, moving past the end of the buffer), Emacs aborts execution of request functions, returns to its listener, and beeps (as when a ^G is performed).

The unwind-protect Special Form

Since the releasing of marks is important, a facility like a cleanup-handler is needed to make sure that marks get released when code is aborted. There is such a facility in Lisp that is useful for many other things, too: save-excursion returns the cursor to the point at which it found it if aborted through; save-excursion-buffer returns to the buffer where it found the editor if aborted through; all the mark-handling forms release their mark, and so forth. These Emacs-environment primitives use the cleanup-handler facility internally, so you need not worry about cleanup-handlers if you use them. However, occasionally (see the code for columnating the Emacs wall chart, for example) you must use cleanup-handlers explicitly. The Lisp form unwind-protect is the primitive cleanup-handler. Its syntax is:

```
(unwind-protect
     <SUBJECTFORM>
     <CLEANUPFORM1>
     <CLEANUPFORM2>
     ...
     <CLEANUPFORMn>)
```

The <SUBJECTFORM> is evaluated, and then <CLEANUPFORM1> to <CLEANUPFORMn> (any number of cleanup forms are permissible), and the value of the <SUBJECTFORM> returned. So far, unwind-protect is much like prog2 or progn. The difference, however, is that <CLEANUPFORM1> to <CLEANUPFORMn> are executed even if the execution of <SUBJECTFORM> fails and aborts. Similarly, the cleanup forms are executed even if things like a return from a prog inside the <SUBJECTFORM> causes its execution to terminate prematurely.

Thus, the cleanup forms are executed after every termination of the <SUBJECTFORM>, whether normal or abnormal. The following use of unwind-protect (which could be done in simpler ways, but is here for illustrative purposes) performs "complex-function", and returns the cursor to the beginning of the buffer, even if "complex-function" explodes:

```
(unwind-protect
     (complex-function)
     (go-to-beginning-of-buffer))
```

If you want more than one <SUBJECTFORM>, you should use progn to encompass them, and make your <SUBJECTFORM> this progn.

Unlike Multics/PL/I cleanup handlers, unwind-protect cleanup forms are executed upon normal termination of the subject form, too.

USEFUL PREDICATES

The following predicates in the Emacs environment are basic to all extension-writing; they are used to test various hypotheses about point, marks, and the buffer:

(eolp)
: End of line predicate. True if point is at end of a text line right before the newline character.

(bolp)
: Beginning of line predicate. True if point is at the start of a text line, either before the first character of the buffer, or after a newline.

(firstlinep)
: First line predicate. True if point is on the first text line of the buffer.

(lastlinep)
: Last line predicate. True if on last buffer line.

(at-beginning-of-buffer)
: True if point is right before the first character in the buffer.

(at-end-of-buffer)
: True if point is right before the newline on the last line of the buffer. You cannot go past it.

(looking-at <STRING-VALUE>)
: True if <STRING-VALUE> appears in the buffer immediately to the right of point. Restriction: <STRING-VALUE> can not contain a newline character, except as its last character.

(at-white-char)
: True if the character to the right of point is a space, newline, or tab.

(point>markp <MARK>)
: True if the current point is further in the buffer than the position defined by <MARK>. This is expensive, and should not be used casually in loops.

(mark-reached <MARK>)
: True if the current point is up to or beyond <MARK> in the buffer. Intended for use in controlling character-by-character loops; it expects that point starts to the left of <MARK> and moves toward it. The function (order-mark-last <MARK>) can be used to switch point and mark if needed at the start of such loops. Does not terminate unless executed with mark and point on same line.

(mark-at-current-point-p <MARK>)
: True if the mark <MARK> represents the same position as the current point.

(mark-on-current-line-p <MARK>)
: True if the mark <MARK> represents a position on the same line as the current point.

(mark-same-line-p <MARK1> <MARK2>)
True if two marks that are arguments represent positions on the same line.

(line-is-blank)
True if current line is all blanks or empty.

(empty-buffer-p <BUFFER-SYMBOL>)
True if the buffer identified by <BUFFER-SYMBOL> is empty, i.e., contains exactly one line with only a newline character in it. The form (empty-buffer-p current-buffer) may be used to test the emptiness of the current buffer. See below for a discussion of buffer symbols.

This function that ltrims all the lines in the buffer demonstrates the use of these predicates:

```
(defun ltrim-all-lines ()
       (save-excursion                      ;be polite, restore point
          (go-to-beginning-of-buffer)
          (do-forever                       ;loop on lines thru buffer
           (do-forever                     ;loop thru chars on line
            (if (eolp)(stop-doing))         ;stop at eol.
            (if (at-white-char)(delete-char)  ;do the work
              else (stop-doing)))           ;non-white char, next line
           (if (lastlinep)(stop-doing));quit when did last line
           (next-line))))                   ;leaves you at b.o.l.
```

WHITESPACE MANAGEMENT

Neatly formatted editor output and displays, as well as program and document formatting, require good whitespace management. The following functions exist to deal with whitespace:

skip-over-whitespace
Takes no arguments. Moves point forward over all tabs, blanks, and newlines until a non-white character or the end of the buffer is reached.

skip-back-whitespace
Takes no arguments. Moves point backward over all tabs, newlines, and blanks until the character to the left of point is none of these, or the beginning of the buffer is reached.

skip-to-whitespace
Moves forward until character to right of point is tab, blank, or newline. Since last character in buffer must be a newline, there is no special end condition.

skip-back-to-whitespace
Moves backward until the character to the right of point is a tab, blank, or newline, or the beginning of the buffer is reached.

delete-white-sides
Deletes leading or trailing blanks from anything, or deletes space between words.

skip-over-whitespace-in-line
Same as skip-over-whitespace, but stops before the newline character at the end of the line (i.e., stops at the end of the line) if it gets that far.

skip-back-whitespace-in-line
Same as skip-back-whitespace, but does not proceed backward beyond the beginning of the line.

You often need to generate whitespace to reach a given horizontal position (column), for tabbing and page layouts. The function whitespace-to-hpos performs this service; it generates tabs and spaces as appropriate, moving point until the horizontal position that is its argument is reached. The following function moves all lines in the buffer seven spaces over, regardless of their original indentation, with the right amount of tabs and spaces:

```
(defun move-over-7 ()
       (save-excursion
        (go-to-beginning-of-buffer)      ;all do-for-all-lines
        (do-forever                      ;start like this.
                (skip-over-indentation)  ;This is ESC M
                (let ((hpos (cur-hpos)))
                               ;let hpos be the curr. pos.
                     (delete-white-sides)
                               ;close up all original space
                     (whitespace-to-hpos (+ hpos 7)))
                                         ;make just enough
                (if (lastlinep)(stop-doing))
                (next-line))))
```

A related need is to space to a given position, leaving a single space if you are already there or beyond. This is useful for producing columnar output where overlength fields must be separated (as ^X^B does in its local display). The whitespace-to-hpos does not do this; it stops if it is far enough. However, format-to-col takes a single argument, a horizontal position to be spaced to. If the current point is already that far, it inserts a space. .

EXTRACTING TEXT FROM THE BUFFER

The function point-mark-to-string gets a Lisp string whose value is the string of characters between point and the mark that is its argument. To demonstrate, a function that finds a vertical bar (¦) on a line, deletes it, and swaps the two line-halves around it is defined below. For instance, the line:

```
     An Indian with a zebra ¦ never trips in the snow
```

comes out:

```
     never trips in the snowAn Indian with a zebra
```

The function is:

```
(defun swap-around-bar ()
       (go-to-beginning-of-line)
       (if (not (forward-search-in-line "¦"))      ;check for one
           (display-error "Hey, there is no ""¦"" "))
       (rubout-char)                                ;what # does
       (with-mark m                       ;m in middle of line
               (go-to-end-of-line)
               (let ((temp (point-mark-to-string m))) ;get
                                                      ;middle
                                                      ;to end
                    (without-saving (wipe-point-mark m))
                    (go-to-beginning-of-line)
                    (insert-string temp))))         ;put in text
```

The forward-search-in-line is just like forward-search, except that it indicates failure if it cannot find its search string in the current line. If the vertical bar is not found, display-error lets you know and does a command-quit, (^G), which stops the execution of this function at once and returns to Emacs command level (see below). This is useful by itself to search for some string only in a given line. There is also a reverse-search-in-line, and a regexp-search-in-line, which are similar in their relation to ^R and ESC /.

TALKING TO THE USER

You cannot use the Lisp I/O system to print out messages and/or query the user. The Emacs redisplay manages the screen itself, entirely. Thus, you can not use "print", or "read", or other Lisp functions that you may be familiar with.

A function called minibuffer-print prints all the messages that Emacs outputs in the minibuffer screen area. It takes any number of arguments, which must be strings. The function decimal-rep is provided to convert numbers into strings for inserting them in the buffer or handing them to display-error. The following function counts the number of As in the current line:

```
(defun a-counter ()
    (let ((n 0))                                          ;initial count
         (save-excursion                                  ;why not?
          (go-to-beginning-of-line)
          (do-forever
           (if (not (forward-search-in-line "A"))
               (minibuffer-print "Found " (decimal-rep n) "As.")
               (stop-doing))
           (setq n (+ 1 n))))))          ;count them.
```

The forward-search-in-line moves to the right of what it finds (like ^S), so that it does not find the same occurrence the next time.

To prompt the user for input, via the minibuffer, use the function minibuf-response. It takes two arguments. The first is the prompting string. The second should be specified by the value of one of the global variables, ESC or NL, which are bound to special symbols known to minibuf-response. If the value of ESC is used, minibuffer input terminates on an ESC. If the value of NL is used, (NL, <u>not</u> CR), minibuffer input terminates on a carriage return. Thus:

```
(minibuf-response "Type new division name: " NL)
```

returns the user's response to this question when he terminates it with a carriage return. The value of minibuf-response is a Lisp string. The carriage return does not appear in it, nor does the prompt.

To display an error message in the minibuffer, and then abort execution of an extension, i.e., execute a command-quit (^G), use display-error. The display-error is like minibuffer-print, except that it does not return, but aborts to Emacs top level immediately after printing its error message in the minibuffer. Like minibuffer-print, it takes any number of string arguments.

Message Printing Functions

Messages printed by minibuffer-print are suppressed during keyboard macro execution, just as search strings are not displayed, and other gratuitous messages are suppressed. The following set of functions describes the repertoire of message-printing:

display-error
: Prints a message in the minibuffer and aborts to editor top level. It is intended for use in error message printing.

display-error-noabort
: Prints a message in the minibuffer and continues execution. This function is intended for reporting nonfatal errors such as "User not accepting messages...".

minibuffer-print
: Prints a message in the minibuffer, but not during macro execution. This function is intended for use by extensions that print messages in the normal process of their execution, such as the line count from ^X=. For this function, as well as the others below, in multiline minibuffer situations, an appropriate line is chosen based upon availability of empty lines and several other criteria.

minibuffer-print-noclear
: Prints a message in the minibuffer (not during macro execution), but does not erase the previous contents. Output is appended to the last minibuffer line used.

display-com-error
: Prints a message in the minibuffer and aborts to editor top level. Its first argument is a Multics standard error code. Its remaining arguments are character strings or symbols. See "Multics Error Table" below for the technique used to get error_table_ values into your program.

display-com-error-noabort
: Prints a message in the minibuffer and continues execution. Its first argument is a Multics standard error code.

minibuffer-clear
: Clears out the last minibuffer line that was written, except during macro execution. This function should be used to clear out minibuffers written in by minibuffer-print and minibuffer-print-noclear at the end of subsystem invocation.

display-error-remark

Identical to display-error-noabort, except that the particular minibuffer line on which this remark is printed becomes the next one overwritten for any minibuffer remark or output. This function should be used for transient remarks (such as "Writing", "Modified", etc.), that you wish to remove from the screen as soon as possible.

VARIABLES

Many groups of Emacs requests need global variables to communicate among themselves and the functions they call. A global variable is a Lisp variable that is not the parameter of any particular function; its value can be accessed or set by any function. Some of the global variables in Emacs are highly user-visible, for example, "fill-column", which contains the column number of the fill column as set by ^XF, and used by the filling requests and fill mode. Similarly, the character string that is the comment prefix is the binding of the global variable "comment-prefix". Extensions often need global variables to communicate among their parts.

Normally, global variables in Lisp are accessed just like other variables, i.e., those that are parameters of functions or _prog_ or _let_ variables. For instance, a function to set the fill column to 30 if it is over 40, might contain the code:

```
(if (> fill-column 40.)(setq fill-column 30.))
```

When a global variable is used in your program, say one named "my-global", the "declaration"

```
(declare (special my-global))
```

must appear in the program before its first use, to tell the compiler about this "special" variable (the Lisp term for a global variable). The e-macros include file declares many of the provided global variables, which you need not declare.

The global variable situation is complicated by the fact that editing activity is usually local to each buffer. That is, if a set of global variables contains a set of values about what is being edited, it usually pertains to what is going on in only one editor buffer. If you switch to a different buffer, and use the same editor facility, you do not want to use or change the values of those global variables that pertained to activity in the other buffer. At first, this would seem to make global variables unusable, because all functions would have to keep track of what buffer they are talking about before using any global variables, and therefore maintain several sets of them.

Fortunately, it is a lot easier than that. The buffer-switcher in Emacs saves and restores values of global variables as buffers are switched, if you tell it what variables you want so saved and restored, when the buffer you are operating in is exited and reentered, respectively. Such a variable is called a per-buffer variable, and the act of telling the buffer-switcher about it, thereby associating its current value with this buffer, is called registering it. Once a variable has been registered in a given buffer, the functions that use it can assume that its value will be what it last was in that buffer whenever the editor enters that buffer. Another term for a per-buffer variable is a local variable. The following two primitives exist for registering local variables; there are no primitives for setting or retrieving their values, because the whole point of this mechanism is to allow them to be accessed as normal Lisp variables.

register-local-variable
: Called with one argument, the symbol whose name is the name of the local variable you wish to register. Registers it in the current buffer, if not already registered there, and the variable initially inherits its "global value". If registered, its value is left alone. If it has no global value, it acquires the symbol "nil" as its value if this is its first registration in this buffer.

establish-local-var
: Just like register-local-variable, but takes a second argument, a default value to be initially assigned to the variable the first time it is registered in this buffer, if it has no global value.

The global value of a per-buffer variable is the value it has in buffers in which it is not registered. It is this value that is set if you set this variable while in a buffer in which it is not registered. A local variable "inherits" its global value when it is first registered in a given buffer. For variables that have no global value (i.e., were never assigned one), establish-local-var can be used to provide default initialization.

EXAMPLE OF LOCAL VARIABLES

Three functions that maintain a "problem count" in a given buffer are started up by typing ESC X monitor-problems CR. Once started, use ^XP to count a problem, and ^XR to report the number of problems noted:

```
(defun monitor-problems ()                ;command-level function
       (set-key '^XP 'note-a-problem)     ;set the keys needed,
       (set-key '^XR 'report-problems)    ;only in this buffer
       (establish-local-var 'problem-count 0))    ;register the
                    ;local var, initial value 0 here.

(defun note-a-problem ()                  ;executed on ^XP
      (setq problem-count (+ 1 problem-count)))   ;Increment the
                                                  ;variable

(defun report-problems ()                         ;on ^XR
       (minibuffer-print "There have been "
                         (decimal-rep problem-count)
                         " problems in this buffer."))
```

By calling establish-local-var on the symbol "problem-count", the programmer here has ensured that the problem-counts in each buffer in which he counts problems will be maintained separately.

REGISTERED VARIABLES

The following per-buffer variables are automatically registered by the editor. Their values can be inspected or set in extension code.

buffer-modified-flag
: Contains t or nil, indicating that this buffer has or has not been modified since last read in or written. Set automatically by the editor. Modification of a buffer executed within the special form:

```
(without-modifying <form1><form2>...<formn>)
```

does not set this flag.

read-only-flag
: Contains t or nil indicating whether or not this is read-only buffer. The editor does not set this flag; it is set only by extensions. An attempt to modify the text in this buffer produces an error and a quit to editor command level if this flag is on and buffer-modified-flag is off (nil). The buffer can be modified, however, by functions executed from within extension code within a "(without-modifying ...)".

fpathname
: Contains the full Multics pathname associated with this buffer by the last file read or written into/out of it, or by find-file. It is nil if there is none. Changing it from extension code modifies or "forgets" the pathname as you set it.

der-wahrer-mark
: Contains the mark associated with the user-visible mark that ^X^X and other related requests see. Is nil if the user set no mark in this buffer. Do not set this variable; call set-the-mark to do so.

current-buffer-mode
: Contains the major mode in effect in this buffer. The value is a symbol. To state that a major mode of your construction is in effect in a buffer, simply set this variable.

comment-column
: Contains the comment column, measured from 0.

comment-prefix
: Contains the string, which can be a null string, that is the comment prefix.

tab-equivalent
: Contains the number of spaces for a tab. Initialized to 10., the Multics standard, this can be set either in code or by ESC ESC to edit code from other operating systems. The redisplay obeys this variable too, but not in two-window mode.

buffer-minor-modes
: Contains the Lisp list of symbols representing the minor modes in effect in this buffer.

LARGE SCALE OUTPUT

Output of multiline information, or information longer than about 60 characters, should not be done via minibuffer printing, but via the _local-display_, or _printout_ facility. This is the facility with which buffer listings, global searches, apropos, and other requests display their output. On display terminals, it displays lines at the top of the screen, asking for "MORE?" as each screen fills up, pausing for the next Emacs request at the end of the display, and restoring the screen. On printing terminals, the data is simply printed line by line, with no "MORE?" processing or pausing at the end. The local display facility is an integral part of the Emacs redisplay.

Three functions used in generating local displays are:

init-local-displays
: Is called with no arguments to start a local display. It sets up the necessary redisplay mechanism, initializing it to the top of the screen.

local-display-generator
: This function is called with a string, whose last character must be a newline, and displays it as the next line (or lines, if continuation lines are required) of local output. If you do not have a newline at the end of your string, calling local-display-generator-nnl instead provides one automatically. There must be no embedded newlines in strings for local output. A null string causes an empty line.

end-local-displays
: Finishes a local display, restoring the screen. Causes the next redisplay to be suppressed, so the local display remains visible on the screen.

The sequence of calls:

```
(init-local-displays)
(local-display-generator{-nnl} ...)     ;perhaps many
                                        ;times
(end-local-displays)
```

correctly produces a local display.

The best way to generate a well-formatted local display is to set up a temporary buffer (see "Manipulating Buffers" below), build some text in it, and display its content, in part or in whole, as a local display. Three functions are provided to facilitate this:

local-display-current-line
: Does a local-display-generator on the current editor line in this buffer.

display-buffer-as-printout
: Does an init-local-displays, and displays all lines of the current buffer as local output. It does not do an end-local-displays; you have to do that yourself, hopefully <u>after</u> you have gotten out of your temporary buffer and cleaned up whatever else you had to.

view-region-as-lines
: Displays the entire point-to-user-visible-mark as local display, making all the necessary calls, including end-local-displays.

While in a function that has a local display in progress, you must never call the redisplay (see "Calling the Redisplay" below), or call minibuf-response or any other function that causes redisplay, for that instantaneously restores the screen contents to the windows on display, obliterating the local display in progress.

The following function locally displays all lines in the buffer that contain the string "defun":

```
(defun look-for-defuns()          ;use ESC X look-for-defuns CR
       (save-excursion            ;remember where you are.
          (go-to-beginning-of-buffer)
          (init-local-displays)   ;set up for printout.
          (do-forever             ;loop the buffer
            (if (forward-search-in-line "defun") ;look for
                                                 ;"defun"
                (local-display-current-line))    ;cause printout
                                                 ;of it
            (if (lastlinep)(stop-doing)) ;check for EOB.
            (next-line)))                ;Go to start of
                                         ;next line
       (end-local-displays))             ;wait for user, and
                                         ;next request
```

A special form, display-as-printout, is available. It generates a new buffer, executes your contained forms, displays the whole buffer as local display, destroys the buffer, and returns. Its syntax is:

```
(display-as-printout
     <FORM1>
     <FORM2>
       .
       .
       .
     <FORMn>
```

MANIPULATING BUFFERS

Often, the easiest way to do string processing in the editor environment, i.e., handle strings, catenating, searching, etc., is to use the primitives of the editor itself, since it is a string-processing language. To do this, temporary buffers are necessary. To create a buffer, you should use the primitive go-to-or-create-buffer (what ^XB uses), which goes to a buffer associated with the symbol you give it as an argument.

Most symbols are kept in a registry: this registry is called the obarray, and there is only one symbol of any given name in it. A symbol registered in the obarray is said to be interned. Only one interned symbol named "joe" exists, but you can create many uninterned symbols named "joe". If you refer to a symbol named "joe" in a program, however, by saying "'joe", you always get the interned one.

A major feature of symbols in Lisp is that they can be given properties, arbitrary user-defined attributes. These attributes are catalogued "in" the symbol via indicators, symbols that indicate what property you want. The Lisp functions "putprop" and "get" store and retrieve properties.

```
(putprop 'Fred 'blue 'eyes) ;Gives the interned symbol
                            ;named "Fred" an "eyes"
                            ;property of "blue".

(get 'Fred 'eyes)        ;retrieves the property under the
                         ;indicator "eyes", and thus returns
                         ;the interned symbol "blue".
```

In Emacs, symbols represent buffers. All of the information associated with a buffer is catalogued as properties of some symbol whose name is the name of the buffer. Thus, it is possible to have two buffers of the same name, which would imply that of the symbols representing them, only one is interned. The ^XB request always uses the interned symbol of the name given; that is why you can ^XB back to an existing buffer instead of creating a new one each time.

Creating a Temporary Buffer

To create a temporary buffer, you must first create an uninterned symbol, to make sure that you are not going to switch to a buffer that is already real. To do this, you give a string to be used in naming the symbol to the Lisp cliche:

```
(maknam (explodec  "A string"))
```

The explodec blows the string apart into a Lisp list of characters; the maknam builds a symbol out of it. The value of this form is the new symbol. You can then go to a (guaranteed) new buffer of that name, i.e.,

```
(go-to-or-create-buffer (maknam (explodec "A string")))
```

and the global variable "current-buffer" will have that symbol as its value. A temporary buffer is one that is destroyed automatically by the editor upon switching out of it. To make a buffer temporary, all you have to do is give the symbol that represents it (the "buffer symbol") a "temporary-buffer" property of the symbol "t". This can be done by the Lisp form:

```
     (putprop current-buffer t 'temporary-buffer)
```

(The variable "t" is always bound to the symbol "t"). Once this has been done, you must be careful not to switch out of this buffer until you are done with it. If your code involves manipulating many buffers, some of them temporary, you must give the temporary buffers their temporary-buffer properties at the end of your manipulations.

A better way to do this is via the set-buffer-self-destruct function. Calling this function upon the buffer-symbol, as below:

```
     (set-buffer-self-destruct current-buffer)
```

schedules the buffer for deletion as soon as the buffer is exited. Using this, you find out sooner if you mistype this function name than if you mistype the temporary buffer property.

When a new buffer is created, it contains one line, which consists of a linefeed only. There are no truly empty buffers in Emacs. The predicate empty-buffer-p can be applied to a buffer symbol to determine if that buffer is in this state. When buffers are switched, all information related to the old buffer is stored as properties of the buffer symbol: this includes not only the local variables registered in that buffer, but the location of point, the user-visible (and all other) marks, etc. Thus, when buffers are switched back and forth, the cursor retains its position in each buffer (as can be seen while editing), although the redisplay might choose to display a screen differently after visiting another buffer and coming back.

Some applications require making a nontemporary buffer, putting some text in it, and going back there on occasion. Therefore, you might want to go into a nontemporary buffer of an interned buffer symbol:

```
     (go-to-or-create-buffer 'name-and-address-buffer)
```

or perhaps keep a global (not per-buffer) variable that you set once to an uninterned symbol:

```
     (setq name-and-address-keep-track
             (maknam (explodec "Name and Address Buffer")))
```

and switch into it by saying:

```
     (go-to-or-create-buffer name-and-address-keep-track)
```

The function buffer-kill can be called with a buffer symbol to destroy a buffer. The function destroy-contents-of-buffer (of no arguments) can be called to reduce the current buffer to a single "empty" line.

Variable for Buffer Manipulation

The following two variables are relevant to buffer manipulation:

current-buffer
: The value of this variable is the buffer symbol of the current buffer. Do not change it, or incorrect operation results. Use go-to-or-create-buffer.

previous-buffer
: The value of this variable is the buffer symbol of the last buffer, which is returned to when ^XB CR is typed. It is acceptable to setq this variable.

The go-to-or-create-buffer function accepts a buffer-name of "" as meaning go to that previous buffer.

The save-excursion-buffer Special Form

The special form save-excursion-buffer is invaluable when writing functions that switch buffers. It provides for remembering which buffer you were in, and switching back to it when you are done. It also saves and restores the state of "previous-buffer". The save-excursion-buffer is like save-excursion; it executes its contained forms while pushing the buffer-state of the editor on an internal stack, and returns the value of the last form within it.

The following program, when invoked after typing somebody's name (say you hook it up to a key), follows it with his title in parentheses. Assume the file >udd>FamNam>personnel_data looks like this:

```
     Washington, G. =Lumberjack
     Duck, D. =Pessimist
     Nietzsche, F. =Existentialist
     Mouse, M. =Optimist
     Eisenhower, D. D. =Golfer
```

```
(defun insert-person-title ()
       (let ((name (save-excursion              ;save guy's point
                     (skip-back-whitespace)     ;get to end of word
                     (with-mark m               ;m = end of word
                        (backward-word)         ;go to beg. of wd.
                        (catenate (point-mark-to-string m)
                                  ",")))))
                                  ;return the word with a "," after it.
             (insert-string                     ;insert
              (catenate " ("                    ;open paren and sp
                (save-excursion-buffer          ;save the old buff
                 (go-to-or-create-buffer 'name-position-records)
                                                ;go to stuff
                 (if (empty-buffer-p current-buffer) ;read it
                                                   ;once
                     (read-in-file ">udd>FamNam>personnel_data"))
                 (go-to-beginning-of-buffer)  ;set up for search
                 (do-forever                  ;scan lines
                  (if (looking-at name)       ;Is point at
                                              ;"name,"?
                      (forward-search "=")    ;look for the =.
                      (return (with-mark n    ;get to the end.
                                 (go-to-end-of-line)
                                 (point-mark-to-string n))))
                  (if (lastlinep)(return "???"))  ;couldn't
                                                  ;find him
                  (next-line))                    ;move on
                 ") ")))))
```

This function picks out the name you just typed by skipping back over whitespace, and picking up all between there and the start of the previous (current) word. It then inserts, between parentheses, the portion of that line of the data file that contains the sought name at its front after the equals sign. The buffer name-position-records is read into once, and contains the data file thereafter.

The initial save-excursion remembers the user's point location while the word is collected. The save-excursion-buffer remembers what buffer and where in it all its modes, local variables, etc., are, while you operate in the data file buffer.

The function catenate is a valuable one in the context of Emacs; it takes any number of strings (or symbols, whose print-name will be used), builds a string by catenating them first-to-last, and returns it.

Another useful function in this context is apply-catenate, which takes as an argument a list of any number of strings or symbols and builds a string by catenating the strings and names of the symbols, first to last.

CALLING THE REDISPLAY

The Emacs redisplay decides what lines of the current buffer should be shown on the screen, determines how to modify the current screen to show the contents of those lines, and updates the screen in an optimal manner. It is called by the editor whenever there is no more input available. It is very simple to call. It takes no arguments, i.e., you just say:

```
(redisplay)
```

The redisplay does not know or care by what means the buffer was modified; if you delete several words with ESC D, ^D, or ^W, it is all the same to the redisplay, and it acts similarly in updating the screen. Normally, the extension writer need not be concerned at all about the redisplay. A major feature of Emacs is that only the total effect of a complex manipulation is displayed, not every small operation that the manipulation used to achieve its effect.

In some situations, however, it is advantageous to call the redisplay explicitly from extension code. One example is a function that takes a tremendous amount of computer time and might wish to update the screen every so often as it finishes some major section. You do not tell the redisplay what to display or how to display it; it displays some excerpt of the current buffer that contains the current line, and shows the cursor where the current point is. If you call it during a buffer excursion, i.e., while in some special buffer in a function, it displays that buffer around its "point". As soon as that function returns to editor command level, the screen is overwritten with the original buffer's lines. Thus, calling redisplay is _not_ to be considered a substitute for local displays.

The most common need for calling redisplay is in functions that add text (or change text) on a line, and move to another line. For example, the electric semicolon of electric PL/I mode adds a semicolon to the current line and moves to the next. On a printing terminal, the user would never see the semicolon unless special action were taken. The text in the buffer would indeed be right, but by the time the next redisplay occurred (the electric semicolon request returned), the editor would be off that line, and thus would display the next line, where the electric semicolon request left it. While this is correct, the printing terminal user looking at his type-in would, with some validity, complain that "all the semicolons seem to be missing". Thus, the electric PL/I semicolon request calls the redisplay immediately after it executes "(insert-string ";")".

The following is a function for a "card-numbering FORTRAN mode", which when invoked (perhaps hook it up to CR) puts a sequence number in column 72 (71 from 0) and goes to column 7 of the next line. It must call the redisplay so that, on a printing terminal, the card numbers get shown:

```
(defun fortran-next-line ()
       (whitespace-to-hpos 71.)              ;go to col 72.
       (insert-string (decimal-rep cardno)) ;cardno is a local
                                            ;buffer var
       (setq cardno (+ 1 cardno))           ;up the next
                                            ;card number
       (redisplay)                          ;let printing
                                            ;user see.
       (new-line)                           ;get to
                                            ;next line
       (whitespace-to-hpos 6.))             ;6 rel = card col 7.
```

Another commonly called redisplay function is <u>full-redisplay</u>, of no arguments, which clears and rewrites the screen, as with ^L.

<u>EIS TABLES</u>

The Emacs environment provides a facility for utilizing the sophisticated 68/80 processor instructions for scanning for characters in, or not in, a particular set of characters. These operations correspond to the PL/I "search" and "verify" builtins. The word requests operate using these facilities.

A set of characters is represented by a <u>charscan table</u>, a compound Lisp object occupying about 200 words of storage. You can get a charscan table by giving a set of characters, as a string, to the function charscan-table. It returns a charscan table representing that set of characters:

```
     (setq number-verify-table (charscan-table "0123456789+-"))
```

<u>Functions Using the Charscan Table</u>

Given such a table, there are a set of functions that can be called to utilize it to search for characters in or out of that set, backward, forward, whole buffer, or only one line. All the following functions take one argument, a charscan table representing a set of characters (called S here). They return nil (falsity) if they hit the end of the buffer or line (as appropriate) without finding what they are looking for. If they succeed, they move point and return a truth indication. If they fail, they do not move point.

search-for-first-charset-line
: Scans current line forward from point. Success is stopping to the left of a character in S.

search-for-first-not-charset-line
: Same as above, but success is stopping to the left of a character <u>not</u> in S.

search-back-first-charset-line
: Scans current line backward from point. Success is stopping to the right of a character in S.

search-back-first-not-charset-line
: Same as search-back-first-charset-line, but success is stopping to the right of a character <u>not</u> in S.

search-charset-forward
: Scans the buffer from point to the end of the buffer. Success is stopping to the left of a character in S.

search-charset-backward
: Scans the buffer backward from point to the beginning of the buffer. Success is stopping to the right of a character in S.

search-not-charset-forward
: Scans the buffer forward from point to the end. Success is stopping to the left of a character not in S.

search-not-charset-backward
: Scans the buffer backward from point to the beginning of the buffer. Success is stopping to the right of a character not in S.

The following function finds the first nonnumeric character on the line it is invoked on:

```
(defun find-first-non-numeric ()
       (establish-local-var numscan-table nil)  ;does
                                                ;var exist
       (if (not numscan-table)  ;if nil, i.e., not init yet,
           (setq numscan-table (charscan-table "0123456789")))
       (go-to-beginning-of-line)
       (if (not (search-for-first-not-charset-line
                                  numscan-table))
           (minibuffer-print "Line is O.K. ")))   ;failure
                                                  ;is all are
                                                  ;in charset
```

OPTIONS

The Emacs option mechanism provides for user-settable variables in the Lisp environment. The only difference between an "option" and any other global Lisp variable in the editor (basic or extended) is that the options are listed at the user-visible level by typing ESC X opt list CR, and can be set or interrogated via the opt request. The option mechanism also provides for checking that numeric variables stay numeric, and that variables restricted to "t" or "nil" as values stay restricted to those values.

Thus, options can control per-buffer or truly global variables; the option mechanism imposes no restraints upon the dynamic scope of the variables managed by it. The option mechanism also provides for a default global value of variables it manages.

A global variable is registered with the option mechanism by invoking the function register-option upon the Lisp symbol that represents (has the name of) that variable, and its default global value. If that value is a number, the option mechanism restricts the variable's value to numbers; if it is one of t or nil, the option mechanism restricts its values to t or nil (which you indicate as "on" or "off").

The choice of whether a variable should be made an official option or not depends upon whether or not you want the user to see it when an "opt list" is done, and whether finer control than that provided by the option mechanism over the values assigned to it is necessary. It is acceptable to register an option the first time some code is executed; only then does it appear in the option list. It is usual to have forms invoking register-option at "top-level" in a file full of code, i.e., outside of any function. Such code is executed when the code is brought into the editor environment.

The following code registers an option describing default paragraph indentation, and shows a function that creates a new paragraph (that should probably be hooked up to a key). Like all Lisp global variables, options must be declared "special" for the Lisp compiler (see "Compilation" below):

```
(declare (special paragraph-indentation))     ;for compiler.

(register-option 'paragraph-indentation 10.)  ;default is ten

(defun new-paragraph ()
       (new-line)                              ;two new-lines
       (new-line)
       (whitespace-to-hpos paragraph-indentation)) ;tab out
```

By issuing the request:

```
ESC X opt paragraph-indentation 5 CR
```

You can set the amount of indentation inserted by new-paragraph to 5.

NAME SCOPE ISSUES

All of the functions and variables in the Lisp environment are accessible to all functions running in it. At times, this can be a problem. When adding your own extensions to the editor environment, nothing prevents you from choosing a name for one of your functions that happens to be the name of some internal (or user-visible) function in Emacs. Occasionally, there may be reason to do this deliberately, e.g., writing your own version of next-line to do something special. This is dangerous, and not recommended.

In general, you want to make sure that none of your functions or variables conflict with those of the editor. The best way to do this is to choose some set of names that cannot possibly conflict. To achieve this, use capital letters anywhere (such as initial capitals) or use underscores or double hyphens in your names. No Emacs or Lisp system functions have leading capitals or trailing underscores. There are a few Lisp system functions with embedded underscores, but other than make_atom, it does not hurt if you accidentally redefine them. The Lisp compiler also warns you if you attempt to redefine a system function. No functions in Emacs contain underscores in their names.

Another technique that has been used is the use of double hyphens.

Another way to avoid name scope conflicts is to prefix all of your names in a given package with some prefix indicative of the facility that you are trying to implement. For instance, if you are implementing a SNOBOL edit mode, you might name your functions "snobol-find-match-string", "snobol-get-branch-target", etc. The same holds true for global variable names. This is the standard, recommended, and most mnemonic way.

You can also be reasonably certain that names constructed somewhat whimsically (e.g.,"Johns-special-tsplp-hack", "find-third-foo", etc.) will not conflict.

MODES

The major and minor mode mechanism of Emacs is a way for the user to switch in and out of large sets of key-bindings and column settings, and to be informed of this via the mode line.

Major Modes

A major mode involves a large body of optional code (e.g., PL/I mode), sets up for editing code written in a particular language, or sets up buffer for some highly specialized task where very common keys (e.g., CR) do nonobvious things (e.g., the Message mode buffers of the Emacs message facility). Minor modes generally involve the way that whitespace or delimiters are interpreted, e.g., fill mode and speedtype mode.

A major mode is set up by a user-visible function called "XXX-mode", where XXX is the name of the mode. This "mode function" establishes key-bindings (using set-key), and sets columns (e.g., fill-column, comment-column) and prefixes as necessary. The mode function establishes the mode by setting the per-buffer-variable "current-buffer-mode" to a symbol whose name indicates the mode. The name of the symbol appears in the mode line when the redisplay is invoked while in this buffer. The following function sets up a major mode for editing FORTRAN programs:

```
(defun fortran-mode ()  ;the mode function
       (setq current-buffer-mode 'FORTRAN)      ;symbol
                                                ;for mode
       (setq fill-column 70.)                   ;set columns
       (setq fill-prefix "      ")             ;six spaces on CR
       (set-key 'CR 'fortran-new-line)          ;set up CR key
       (setq comment-column 0)
       (setq comment-prefix "C "))              ;that begins cmts
```

The function fortran-new-line is assumed to be one that does something appropriate, such as numbering cards. The use of the function set-key implies that this key binding (of the carriage return key) is local to this buffer, and will be reverted when this buffer is exited.

Minor Modes

Minor modes are less straightforward. Minor modes such as speedtype and fill mode have different actions associated with the keys they affect (for instance, all the punctuation keys), and the minor modes have to have detailed and specialized interaction between themselves. There is no way to generalize the interactions between the minor modes; no completely adequate solution to this problem has been developed.

Minor modes are asserted and turned off in a given buffer by calling the functions "assert-minor-mode" and "negate-minor-mode" while in that buffer, with an interned symbol that identifies the mode (and appears in the mode line). A per-buffer variable called buffer-minor-modes has as a value a Lisp list of all the symbols identifying the minor modes in effect in this buffer. The Lisp predicate memq can be used to test whether a given interned symbol is a member of a list, and thus, whether a given minor mode is in effect in the current buffer:

```
     (memq 'fill buffer-minor-modes)
```

returns a truth indication if fill mode is in effect in this buffer; otherwise, it returns "nil" (false). Functions implementing the actions of keys in minor modes should check in this way to see what other minor modes are in effect, and what they ought do in that case.

The global variable fill-mode-delimiters is bound to a Lisp list of keys that act as punctuation in many minor modes. By use of the Lisp function mapc, all punctuation can be set to trigger a given action. The mapc function takes two arguments, a function and a Lisp list; the function is called upon each element of the list:

```
(defun no-punc-mode-word-on-a-line-mode-on ()  ;mode function
     (mapc 'word-on-a-line-setter fill-mode-delimiters) ;set
                                                        ;keys
     (assert-minor-mode 'word-on-a-line))  ;get in mode line

(defun word-on-a-line-setter (key)  ;key is the key
       (set-key key 'word-on-a-line-responder)) ;set these keys

(defun word-on-a-line-responder ()  ;key function
       (delete-white-sides)  ;get rid of whitespace
       (self-insert)  ;insert the typed character
       (new-line))    ;start a new line.
```

This set of functions establishes a minor mode in which each word goes on a separate line as it is typed.

CHARACTER DISPATCHING

Several special forms and functions facilitate the making of decisions based upon the identity of the character to the right (or left) of the current point. All of these functions and forms accept either of two ways of describing characters: either a single-character string (e.g., "."), or a symbol whose name is that character (e.g., 'a, as it would appear in a program). The first kind, is called the "string form", and the second kind, "character objects".

The function curchar, of no arguments, returns the character to the right of the current point as a character object (this is done for storage efficiency; character objects are unique, while strings require allocation). You can test for two character objects being the same unique object (or any two objects, in general) via the Lisp predicate eq:

```
     (if (eq (curchar) 'a)
         (display-error "You are looking at an ""a"".""))
```

You could do this with the looking-at predicate described above, but for single characters, looking-at is a lot less efficient, in both time and storage.

You cannot use eq to test if two strings have the same characters in them; Lisp strings are not uniquely defined in the same way that symbols are uniquely defined via the obarray. Use samepnamep instead.

In order to facilitate the use of special characters (tabs, linefeeds, spaces, quotes, etc.) in this way, several global variables have values of the character objects for these characters:

ESC	ASCII ESC, Ascii 033.
CRET	ASCII carriage return (Ascii 015)
NL	ASCII newline (linefeed), Ascii 012.
SPACE	ASCII blank, Ascii 040.
TAB	ASCII tab, Ascii 011.
BACKSPACE	ASCII backspace, Ascii 010.
DOUBLEQUOTE	", Ascii 042.
SLASH	/, Ascii 057, hard to type in Lisp code.

A (eq (curchar) NL) is equivalent to (eolp).

A special form to test if the current (to the right of point) character is a given character is called if-at:

```
(if-at "&" (display-error "You can't have an ampersand here "))
```

Its syntax is the same as if, i.e., it has one, none, or many "then" and/or "else" clauses, separated by the keyword "else" if there are any else clauses. However, instead of a predicate, if-at takes either a single-character string or a character object to be compared to the current character. If the current character is that character, the then forms are evaluated, etc. The if-at converts the character string to a character object at Lisp compile time, if necessary. The specification of the character must be a form that evaluates to the character of interest (e.g., "a", 'a, variable-bound-to-an-a):

```
(if-at TAB (delete-char)
           (whitespace-to-hpos next-field))   ;tab to next field.
```

The exact effect (and actual implementation) of if-at is as though it were shorthand for:

```
     (if (eq (curchar) ....) ..... ..... ..... )
```

Similarly, a function called lefthand-char is like curchar except that it returns the character to the _left_ of the current point; if the current point is at the beginning of the buffer, it returns a character object for a newline (which is almost always what you want). Similarly, an if-back-at special form exists, whose syntax and semantics are identical to if-at, except that it deals with the character to the _left_ of the current point.

Two special forms for dispatching on the current (lefthand or righthand) character are called dispatch-on-current-char and dispatch-on-lefthand-char they dispatch upon the character to the _right_ and the _left_ of the current point, respectively:

```
(declare (special parentable))  ;global variable
(setq parentable nil)  ;done when code is
                       ;loaded into editor

(defun count-parens-in-buffer ()
       (if (not parentable)  ;if not initialized
           (setq parentable (charscan-table "()")))  ;init it
       (let ((leftcount 0)(rightcount 0))  ;init the counts
            (save-excursion                ;be nice
             (go-to-beginning-of-buffer)
             (do-forever
              (if (not (search-charset-forward parentable))
                                              ;look for ( or )
                  (stop-doing))  ;exit the do
              (dispatch-on-current-char  ;see which
               ( "("  (setq leftcount (+ 1 leftcount)))
               ( ")"  (setq rightcount (+ 1 rightcount))))))
           (minibuffer-print (decimal-rep leftcount) " opens, "
                             (decimal-rep rightcount)
                                          "closes.")))
```

The general syntax of dispatch-on-current-char and dispatch-on-lefthand-char is as follows:

```
(dispatch-on-current-char
  (CH1                <CH1-form1>
                      <CH1-form2>
                      ...........
                      <CH1-formn>)
  (CH2                <CH2-form1>
                      <CH2-form2>
                      ...........
                      <CH2-formn2>)
...............................
  (CHk                <CHk-form1>
                      <CHk-form2>
                      ...........
                      <CHk-formnk>)
  (else               <else-form1>
                      <else-form2>
                      ............
                      <else-formn>))
```

CHi can be any form that evaluates to a single-character string or to a character object. When the current character (left or right as appropriate) matches a CHi, all of the <CHi-form> in that clause are evaluated sequentially, and the value of the last returned as the value of the dispatch-on-current-char (nil is returned if there are no <CHi-form>). If no CHi matches, the else clause is evaluated as though it were a matching clause. The else clause is optional; if omitted, and no CHi matches, nil is returned.

PROGRAM DEVELOPMENT

The editor itself provides many powerful tools for developing extension code and testing it while editing it. The following is a typical scenario in the development of an extension.

You decide to write an extension. You sit down and think about it, and decide to code it. You enter Emacs. You do a ^X^F on the shaver.lisp file to go into a new buffer with a proper file name and select Lisp major mode (assuming that you have the option for find-file-set-modes "on"). Then type the form:

```
(%include e-macros)
```

at the top of your file; this is necessary to compile it (see "Compilation" below), or to use the loadit request, described below. The file e-macros.incl.lisp should be in the "translator" search rules for your process. For efficiency, put a link to it in the directory in which you do Emacs extension development. Now begin to type in a function:

```
     (%include e-macros)

     (defun shave-line ()
          (go-to-beginning-of-line)
```

At this point, to type the next line, lining it up with the last Lisp form, use the indent-to-lisp request, which is on ESC CR in Lisp mode, and the next form automatically indents properly:

```
     (delete-white-space)   ;wrong name given deliberately here
```

When typing in Lisp in general, ESC CR (in Lisp mode) indents you on the next line the right amount. So, continue with:

```
     (go-to-end-of-line)
     (delete-white-space)
```

Now you are looking at the buffer with the code for "shave-line". To try it, load the code in the buffer into the editor. ESC ^Z in Lisp mode does this. Immediately, you get the message:

```
     Unbalanced parentheses.
```

This means that there were not enough close parentheses somewhere: Emacs could not find the boundaries of the Lisp form. Fix the program problem. You are on the last line, so just type the close parenthesis:

```
     (delete-white-space))
```

Now do the ESC ^Z again. The cursor returns to the function you are trying to edit. To see if it works, invoke it from Lisp:

```
     ESC ESC shave-line CR
```

ESC ESC puts parentheses around what you type, evaluates it, and types out the Lisp value so returned. However, you find the message:

```
     lisp: undefined function: delete-white-space
```

printed in the minibuffer, with the terminal bell rung, so you must have the wrong function name. Since you know it is on a key, type:

```
     ESC X apropos white CR
```

and learn about delete-white-sides. Now go to the first line that has the bad function name, do an @ to clear the line, ESC ^I to line up to retype the form, and:

```
     (delete-white-sides)
```

Fix the other bad line, too, and again type:

```
ESC ESC shave-line CR
```

Surprisingly, it still says:

```
lisp: undefined function: delete-white-space
```

as though you had not changed anything. Indeed, fixing it in the buffer is not good enough. You must reload it into the editor environment; use ESC ^Z again. Now try it again:

```
ESC ESC shave-line CR
```

and immediately your function on the screen changes appearance; all the whitespace on the ends of the last line of the function disappears. It works, but its appearance is messy. This is a problem with editing what you are testing: it must either be innocuous, i.e., do something harmless, or you must be prepared to reconstruct damage your function does, or switch to a test buffer before running it.

Fix your function, and you are almost done. Although it exists in an editor buffer, and in the editor Lisp environment, you must remember to write it out:

```
^X^S
```

writes it out to shaver.lisp as you set up for initially. Now you have an operative Lisp program that you can use again. If, in a future invocation of the editor, you need to use it, you type:

```
ESC X loadfile <path>shaver.lisp CR
```

and get it into the environment. There are two problems with this, however:

1. Whoever loadfiles must have e-macros.incl.lisp in his translator search rules.

2. The code is executed interpretively by the Lisp interpreter in the Lisp subsystem; Emacs is compiled Lisp, and compiled Lisp runs up to 100 times faster than interpreted Lisp and has fewer problems.

Thus, the file shaver.lisp should be compiled. Then, the compiled object segment can be loaded into the editor with:

```
ESC X loadfile <path>shaver
```

See below for a description of how to compile Lisp programs.

Coding Problems

Some other problems are of immediate interest to the extension writer. It is possible, and fairly common, to write loops that do not terminate, or that generate infinite garbage. If you invoke your request, and the cursor never leaves the minibuffer, and ^G seems to have no effect, you are in a loop. Hit QUIT, and use the program_interrupt (pi) Multics command to reenter Emacs. If you are singularly unfortunate, you get:

```
lisp: (nointerrupt t) mode, unable to accept interrupt
```

in which case you are stuck in the process of generating infinite garbage. In this case, you must release, and your editing session is lost. If you are more fortunate, you will get your screen back, with the cursor at the place your function left it. Often, by looking at exactly where it left it, you can get a good idea of what kind of thing was giving your program a hard time.

If you get messages from Multics that tend to indicate that there is no more room in your process directory, you are probably generating an infinite number of lines, i.e., an infinite buffer.

Another thing that can happen is you might expose some bug, or what you believe to be a bug, in Emacs, or worse yet, Multics Lisp. Use the trouble report forwarding mechanism to describe what you encountered and why you think it is a bug.

You can also destroy the editor environment by bad coding. This is particularly true in running compiled code that was not checked out interpretively (i.e., via ESC ^Z). Storing into "nil" is one common way to do this. If the entire editor seems broken, and the redisplay does not even show the screen, this is what you have done. Quit and release and start all over again.

A function called debug-e is called as:

```
ESC X debug-e CR
```

It sets "(*rset t)" mode and other Lisp debugging aids, and unsnaps all "lisp links". It also reverts to native Maclisp QUIT/pi handling. To use this, however, you must be familiar with the debugging features of Multics Maclisp.

To get the value of a global variable to be printed out, say, fill-column, type:

```
ESC ESC progn fill-column CR
```

Be careful, for values typed out are in octal.

A Lisp code debugging facility within Emacs, called LDEBUG, or Lisp Debug mode, allows for the setting of breakpoints, dialogue with Lisp within Emacs, tracing, and so forth. See Section 4.

COMPILATION

All production Multics Lisp programs are compiled. This results in a tremendous performance improvement, both for the user and the system. Compiled Lisp programs are executed directly by the Multics processor; interpreted programs are interpreted by the Lisp interpreter. Emacs is compiled Lisp.

The Lisp compiler is a Multics program that can be invoked from command level. It has the names lcp and lisp_compiler. To compile a program named myfuns.lisp, you say:

```
     lcp myfuns
```

to Multics, and you get an object program named "myfuns", which can be loadfiled, in the working directory.

The compiler diagnoses Lisp syntax errors. It warns you of implied special variables (if you did not declare a variable special, and it is not a local variable in the function in which it was referenced, you probably made a mistake. All global variables should be declared for this reason; e-macros declares the provided ones.)

At the end of compilation, the compiler prints out the names of functions referenced in the code but not defined in the file. This is normal; however, you should inspect the list it prints out to see if any are ones that you thought you defined; if so, you have a problem. Check also for ones that are obvious typing errors.

While editing a large extension program, you may wish to load only the function that you are looking at on the screen into the editor environment. The function compile-function, on ESC ^C in Lisp mode, compiles the function you are looking at (whose start is found by ESC ^A from where you are now) out of a temporary segment in the process directory, loads the object segment, and displays the compiler diagnostics via local printout. It should be used with care by any except experienced Emacs extension coders. When using it, remember to write out your changes, and recompile your whole program, because a program incrementally debugged in this mode gives the impression that it is working properly when it is only doing so in the current editor environment.

DOCUMENTING REQUESTS

The automatic documentation system (apropos, ESC ?) provides customized Emacs request documentation. Documentation for supplied requests is kept in a special file in the Emacs environment directory. You can provide documentation for your own requests by placing a string, which is that documentation, as the "documentation" property of the symbol that is the request being documented. For instance, if the symbol remove-every-other-word has the documentation property of:

"Removes every other word from the sentence in which the cursor appears."

this information is displayed by ESC ? when used on some key set to remove-every-other-word, or by:

```
     ESC X describe remove-every-other-word CR
```

Documentation properties are assigned most conveniently via the Lisp special form "defprop", whose general syntax is:

```
     (defprop SYMBOL WHAT PROPERTY)
```

This assigns the symbol (SYMBOL) a property (PROPERTY) of WHAT. The defprop is a special form because the actual symbols appearing in the form are used; they are not variables, as in "(+ a b c)". Thus,

```
     (defprop Joe Fred father)
```

gives the symbol "Joe" a "father" property of "Fred". (The "defprop" is a special-form way of doing the same thing as the "putprop" function, but it is a special form because its "arguments" are not forms to be evaluated to produce symbols whose properties are to be dealt with, but the symbols themselves). To use defprop to establish Emacs request documentation, place forms like:

```
     (defprop remove-every-other-word
     "Removes every other word from the current sentence. Will
     not work on sentences ending in ""?"". For indented
     sentences, use $$remove-other-word-from-indented-sentences$.
     $$$ is a powerful, dangerous, command."
                   documentation)
```

Note several things about the documentation string:

1. It does not need to end in a newline, and can contain newlines.

2. Quotes (") inside of it must be doubled.

3. The string "$$$" will be replaced by the key being asked about (e.g., "ESC ^Z" or "ESC X remove-every-other-word") at the time the documentation is displayed.

4. The keys used to invoke other requests can be referenced by stating two dollar signs, the name of the request, and one dollar sign. Thus, $$go-to-end-of-line$ appears as ^E in most environments; the point of this and the previous paragraph is to make documentation expansion independent of a user's key-bindings.

The entire documentation string is "filled" (ESC Q) after all command-name substitutions are made; thus, the placement of newlines in the documentation string is ignored. Two consecutive newlines, however, are preserved, and thus, lines can be set off for examples, etc., by surrounding them with blank lines.

It is slightly more efficient, but clearly less readable, to place the defprop documenting a request before the defun defining the request itself. The defcom facility can also be used to document requests; see "Defining Requests With defcom" below.

WINDOW MANAGEMENT

Although buffers appear in windows on request, and are switched between automatically by the redisplay when you switch windows with ^X^O, ^X4, etc., there are times when you may want to take advantage of multiple windows explicitly. Good examples in supplied code are RMAIL reply mode and the comout-command (^X^E).

Most of the extensions of interest are ones in which the extension writer wants to place some information in a buffer, or else prepare some buffer to have information placed in it (e.g., RMAIL reply) and then display that information in a window. Usually, all that is required is to "go to" that buffer (e.g., with go-to-buffer or go-to-or-create-buffer). The redisplay "finds" the editor in that buffer at the time of the next redisplay, and replaces the contents of the selected window on the screen. Such requests are called autophanic (self-showing). Examples are ^XB (select-buffer) and ^X^F (find-file).

However, some requests set up buffers in some window other than the current window, usually for multi-window operations such as mail reply, so as not to disturb the contents of the current window. They are called heterophanic (other-showing). The standard examples are dired-examine, mail reply, and comout-command (^X^E). All the examples given are sub-requests of larger, autophanic requests.

Heterophanic buffer behavior is provided by the function find-buffer-in-window. It takes as an argument a buffer-symbol (Lisp symbol representing a buffer). That buffer is created if it does not now exist, and is gone to, as if go-to-buffer had been used. If Emacs is in single-window mode, the effect is the same as that of go-to-or-create-buffer. In two-window mode, that buffer is put on display as follows:

- If it is already on display in some window, it is left there.
- If it is not, it is put on display in some other window, one in which the cursor is not, and the cursor moves to that window, as if a ^X^O had been done. The least-recently used window is chosen.

Thus, on printing terminals and in single-window mode, the effect of find-buffer-in-window is indistinguishable from that of go-to-or-create-buffer. In multi-window mode, it is equivalent to go-to-or-create-buffer, displaying that buffer in another window.

You must not use find-buffer-in-window to place a buffer on the screen once you have already gone to it; if you think of find-buffer-in-window as a kind of go-to-or-create-buffer, you will find no need for doing so.

An extension must establish multiple windows if it needs them; no current Emacs code requires multiple windows, although the facilities mentioned above are more useful when already in it.

Most extensions that place an auxiliary buffer on display via find-buffer-in-window provide some request to return to the "main" buffer (e.g., the RMAIL Incoming Message buffer, the buffer from which ^X^E was issued, etc.). If you enter a buffer via find-buffer-in-window, you should probably return to the buffer from whence you came via find-buffer-in-window as well; the effect of this is to restore not only the original buffer, but also the original window. Thus, save-excursion-buffer <u>cannot</u> be used effectively to return from buffers entered via find-buffer-in-window; an attempt to use save-excursion-buffer results in both windows' showing the same buffer, since the selected window (i.e., the cursor-bearing window) is changed and a new buffer selection means a new buffer in that window.

The ^X^Q key sequence should be used to exit auxiliary buffers used by extensions to return to their main buffer, and usually switch windows as well, if the multiple-window strategy outlined above is used.

Pop-up window mode, in essence, makes all requests heterophanic. Requests or subrequests that are naturally heterophanic need not worry about pop-up window mode, because find-buffer-in-window takes the appropriate action in either pop-up or non-pop-up mode. However, if proper heterophanic behavior under pop-up windows is desired, naturally autophanic requests and subrequests must call a window-management primitive to obtain heterophanic behavior in pop-up window mode. This primitive is called select-buffer-window. It takes two arguments, a buffer-symbol, and a "key" that gives pop-up window management a preferred window size.

In non-pop-up window mode, select-buffer-window is equivalent to go-to-or-create-buffer, and the key is ignored. In pop-up mode, it is equivalent to find-buffer-in-window, with the key suggesting the new window size.

The following values for the key argument to select-buffer-window are accepted. They specify the window size in pop-up mode if the window does not exist already:

any number
: That many lines.

'cursize
: Make a choice based on the current number of lines in the buffer.

nil
: Chooses some reasonable fraction of the screen.

'cursize-not-empty
Same as nil if the buffer is empty; same as 'cursize if it is not. For example, ^X^F uses this, because you can type into a new buffer.

'default-cursize
If this buffer has never been displayed before, makes a choice based on the number of lines. Otherwise, uses the same size was chosen last time.

The find-buffer-in-window can not be used to display the "current buffer" heterophanically. If you attempt to do this:

```
(find-buffer-in-window current-buffer)
```

you find it appearing in both the old and new windows, for the window manager finds that you were in this buffer in the current window (a truth) before you went to another one (you had to go to another one, as per heterophanic behavior), and indicates that the current buffer is to be displayed in the old window as well, for that was the last buffer you were in in that window. To avoid this, use select-buffer-find-window (of two arguments, the buffer and a key as for select-buffer-window) if heterophanic display of the current buffer is needed:

```
(select-buffer-find-window current-buffer nil)
```

This is rare, since you seldom go to a buffer and then want to find-buffer-in-window it; in Emacs, only ^X^E does this.

Since all things using these features are moderately sophisticated, only an outline of an extension using them is given here. It is a typical sub-subsystem (e.g., dired) that sets itself up in an autophanic buffer display, with specific key bindings, etc., and has a heterophanic subdisplay by which it displays a "menu" in addition to the main display:

```
(defun unusual-mode ()          ;Setup function for this mode
       ...........
       (go-to-or-create-buffer (maknam
                               (explodec "Unusual buffer")))
       (set-key 'ESC-^S 'unusual-mode-show-menu)
       (select-buffer-window current-buffer nil)
       (register-local-var 'unusual-mode-buffer-to-return-to)
       ...........)

 ............
(declare (special unusual-mode-buffer-to-return-to));for compiler
(defun unusual-mode-show menu ()
       (setq unusual-mode-buffer-to-return-to current-buffer)
                                                   ;save buffer
       (find-buffer-in-window 'Unusual-Menu)  ;Display menu
       (set-key 'r 'unusual-mode-select-item) ;Set key bindings
       (set-key '^X^Q 'unusual-mode-menu-return)
       (insert-string "Unusual menu delicacies")  ;Fill it up
       ;; Will not actually be displayed until request finishes.
       .........
       (go-to-beginning-of-buffer)
       (setq current-buffer-mode 'Unusual/ Menu
             buffer-modified-flag nil read-only-flag t)
       .........)

(defun unusual-mode-menu-return ()
       (find-buffer-in-window unusual-mode-buffer-to-return-to))
       ;;Return to calling buffer.
```

The following are several primitives available to deal with windows by window number. The topmost window on the screen is window number 1; the next one down, if any, is number 2, etc., (the minibuffer and mode line do not count as windows). The selected window is the one in which the cursor currently appears.

selected-window
: This _variable_ contains the number of the currently selected window. Do _not_ attempt to setq it to select a window; use select-window instead.

nuwindows
: This _variable_ contains the number of windows on the screen; do _not_ attempt to setq it to create or delete windows; use delete-window and the ^X2 and ^X3 functions to do these things.

select-window
: This _function_ (of one argument, a window number) selects that window (as ^X4 with an argument does).

delete-window
: This function (of one argument, a window number) removes that window from the screen, distributing its space to the other windows.

buffer-on-display-in-window
: This predicate function (of one argument, a buffer-symbol) returns truth if the specified buffer is on display in some window on the screen. If used as a function, i.e., the value returned is inspected, the returned value the window number in which the specified buffer is on display (if it is not on display, the symbol "nil", representing falsity, is returned).

window-info
: This function (of one argument, a window number) returns information about that window. The information is in the form of a piece of Lisp list structure, which can be interpreted by the Lisp list destructuring functions; assuming that "info" has the result of window-info, the following forms return the information as follows:

(caar info)	=>	The top line-number on the screen of the window. The topmost is 0.
(cdar info)	=>	The number of lines in the window.
(caddr info)	=>	The buffer-symbol of the buffer on display in the window.
(cadddr info)	=>	A string duplicating the contents of the "cursor line" of the window, including its newline character. The cursor line of a buffer is that line where the cursor is (if it is in the selected window) or would be if that window became selected (e.g., with ^XO).

window-adjust-upper
: A function of two arguments, the first a window number, and the second a signed number of lines to move its upper divider-line down (negative is up).

window-adjust-lower
: Same as window-adjust-upper, but deals with lower divider line.

WRITING SEARCHES

Several functions aid in providing search-type requests. These functions prompt for the search string, provide default search strings, and announce search failure in a standardized way. All supplied Emacs searches use them.

get-search-string
: Takes one argument, the prompt. The prompt should contain the word "search". The get-search-string prompts the user for a search string, which the user must terminate with a CR, and returns it as a string. If the user gives a null string, the last search string is used and echoed. The last search string is set to the returned string for the next defaulting.

search-failure-annunciator
: Causes the "Search Fails." message to appear in the minibuffer, and a command-quit (^G) to be performed. This aborts any keyboard macro collection or execution in progress.

When writing a search-type request, you should provide two interfaces, a "command", which calls the above two primitives, and a "search primitive", also called by the "command". The search primitive should return t (truth) if the search succeeds, leaving point at the proper place, as the search defines. If the search fails, the primitive must return nil (falsity), and leave point where it was when the primitive was invoked.

A simple implementation of a wraparound search, below, first looks from point to the end of the buffer for the search string. If that fails, it goes to the top and searches again. It is not optimal because it needlessly scans farther than the original point when starting from the top. Using point>markp and searching a line at a time would be very expensive, due to point>markp's expense. Searching a line at a time using forward-search-in-line and mark-on-current-linep would be acceptable, but more complex than this example need be. For a search that is probably going to be used only as a user interface (i.e., not internally), this implementation is adequately efficient. Recall that with-mark releases its mark and returns 't last value.

Here is the internal primitive for wraparound search:

```
(defun wraparound-search-primitive (string)
       (with-mark m          ;Remember starting point
           (if (forward-search string);Look to end of buffer
               t                      ;Return truth
               else
               (go-to-beginning-of-buffer)
               (if (forward-search string)   ;Look from top
                   t
                   else
                   (go-to-mark m) ;Return to orig. place
                   nil)))))       ;Return falsity
           ;; with-mark and this function
           ;; return the value of the outer "if"
```

The request for calling the primitive:

```
(defun wraparound-search ()
       (if (not (wraparound-search-primitive
                  (get-search-string "Wraparound Search: ")))
           (search-failure-annunciator)))
```

The wraparound-search request should have some key bound to it if this type of search is to be made available from the keyboard.

CALLING MULTICS COMMANDS

In some extensions, especially those like DIRED that manipulate the Multics environment, you must call Multics commands, or execute Multics command lines.

Multics command lines are strings submitted to cu_$cp for execution. This is the Multics agency to which the "e" requests of the Multics edm and qedx editors, the ".." requests of read_mail, send_mail, and debug, and other subsystems submit command lines. The two primitives for executing Multics command lines are:

e_cline
: Takes one argument, a string, which is passed to cu_$cp for execution. No reattachment of output takes place. If the command line produces output, it messes up the screen. This should only be used when no output is anticipated, and should be used then in preference to comout-get-output, since it is much faster.

comout-get-output
Takes any number of arguments, which may be strings or symbols, and catenates them with one space between them to form a Multics command line, facilitating things like:

```
(comout-get-output 'delete this-seg '-bf)
```

Reattaches user_output and error_output during the execution, rerouting them to a process directory file. When the command execution completes, the contents of the current buffer are obliterated (!) and the temporary file read in to it. This is the primitive that comout-command (^X^E) uses; e_cline_ is used by comout-get-output internally.

These primitives set up a condition handler that catches all abnormal Multics signals and aborts to a second Multics command level with a message if one occurs. However, requests for input by these command lines cannot at this time be dealt with well. In the case of e_cline_, the user gets the query in raw teletype modes, and has to answer it in raw, nonedited teletype modes. In the case of comout-get-output, the query never appears, having been routed to the temporary segment, and the user's process hangs since the user, having never seen the query, does not know to respond.

MULTICS ERROR TABLE

To get the value of standard Multics error codes, from error_table_, into a program to see if a given Multics interface has in fact returned it, the function "error_table_" (with underscores, not hyphens) is used. Its single argument is a symbol, whose name is the name of the error_table_ entry whose value is sought, and the returned result is that value, or 1 if it is not a valid entry.

The error_table_ function optimizes finding the same name over and over again, so you need not go through machinations to save an error_table_ value computed by these means. An example of the use of error_table_ follows:

```
(let ((status-result (hcs_$get_user_effmode dir entry "")))
      (if (not (= (cadr status-result) 0) ;the return code
          (if (= (cadr status-result)
                      (error_table_ 'incorrect_access))
              (display-error-noabort "Warning: not checking
                      access")
              else
              (display-com-error (cadr status-result) dir ">"
                      entry))))
```

Defining Requests With defcom

The defcom (for define-command) facility simplifies the definition of Lisp functions to be used as Emacs requests. Defcom cooperates with the Emacs command reader to provide prompting and defaulting of unspecified arguments, range-checking of numeric arguments, automatic repetition for numeric arguments, cross-connecting symmetrical functions via negative arguments, and other features.

Defcom is a relatively new facility in the Emacs extension environment; not all of Emacs' internal code has been converted to use it. Perusing the Emacs source, you will find examples of defcom's use intermixed with older examples using defun to define request functions.

Defcom should only be used for defining functions actually to be used as Emacs requests; internal and auxiliary functions to be used by these functions should still be defined with defun. Emacs requests defined with defun will work, but those defined with defcom produce better diagnostics and offer more features. Defcom is a technique whereby the necessary defuns are generated automatically, so functions defined with defcom can be called from other functions, as well.

To define a function with defcom, use defcom instead of defun, and supply no Lisp argument list:

```
(defcom one-word-from-beginning
        (go-to-beginning-of-buffer)
        (forward-word))
```

This is the simplest form of defcom; optional features are supplied by placing, between the function name and the function code, various keywords, all of which begin with the "&" character, and some of which take optional arguments, expressed as lists.

The most common optional specification is &numeric-argument, (or &na), which specifies what to do with a supplied numeric argument. The keyword &numeric-argument must be followed by a list of specifications, which must include one of the following major processing types:

&reject
: Any numeric argument is rejected as invalid. No other specifications are valid in this case. This is the default if &numeric-argument is not given.

&ignore
: A numeric argument is ignored.

&repeat
    If the argument is positive, the request is repeated that many times.

&pass
    The value of the Lisp variable "numarg" is set, as in nondefcom requests

In addition to the major processing type, optional bounds can be specified by the keywords &upper-bound (&ub) or &lower-bound (&lb). These, in turn, must be followed by either an integer representing the bound, or the keyword &eval followed by an expression to evaluate at the time command execution is attempted, which then produces a value (such an expression is called an "&eval expression".) Here are some examples of &numeric-argument specifications:

```
&numeric-argument (&pass)

&numeric-argument
          (&repeat &lower-bound 1
                &upper-bound &eval (+ max-foos 2))

&numeric-argument
          (&pass &upper-bound 15.)
```

A request defined with defcom may elect to receive Lisp arguments, values that are to be prompted for or supplied as extended request arguments. They can be provided automatically, and prompted for, by the Emacs command reader, and supplied as Lisp arguments to the request function. Instead of a normal Lisp argument list, the keyword &arguments (or &args or &a) are followed by a list of argument specifications, one for each Lisp argument to be supplied.

Each argument specification consists of the Lisp name of the argument, i.e., the name of the variable to be referred to inside the function, and any number of argument qualifiers, separated by spaces. Each argument qualifier can consist of several tokens, as necessary. Argument qualifiers specify the prompts, defaults, etc., for an argument. An argument specification may also be given as the name of the variable alone, as opposed to a list of it and qualifiers. In this case, it is equivalent to having its own name as a prompt for its value.

When a defcom-defined request is invoked as an extended request, (i.e., via ESC X), the Emacs command reader checks the type and number of request arguments supplied and necessary, and prompts for those not supplied, or defaults them as specified.

When a defcom-defined request that has arguments is invoked from a key, it is as if it were invoked as an extended request with no request arguments given, and all are either prompted for or defaulted.

The valid argument qualifiers are

&string
&symbol
&integer
: Specifies how the argument, when read by ESC X or prompted for, is to be converted before being passed. Only one of these is valid in a given argument specification, and &string (i.e., no conversion) is the default.

&default
: Must be followed by either a string, symbol, or integer, as consistent with the expected data type for this argument, or an &eval expression. Specifies the default value to be used if this argument is not supplied, or a null response is given to a prompt for this argument, if any.

&prompt
: Specifies the prompt for this argument, if not supplied via ESC X. Prompts are put to the user before defaults are evaluated or used; a null string causes the &default value to be used. An &prompt is followed by a prompt string (in quotes), or an &eval expression, and one of the two optional keywords NL or ESC, specifiying the prompt terminator (NL is the default).

&rest-as-list
: Valid only for the last argument. Causes this variable to be given, as a value, a list of all of the remaining supplied arguments. If &rest-as-list is used, the caller of this function from Lisp (including start-ups written by not-Lisp-conscious users) must know that the number and organization of Lisp arguments is different from the apparent argument array given to ESC X.

&rest-as-string
: Valid only for the last argument; causes all remaining arguments to be supplied as a single string to the function, as they appeared to ESC X, with spaces and so forth included. Same cautions as for &rest-as-list apply.

A function definition that accepts three arguments follows:

```
(defcom replace-n-times
        &arguments
        ((oldstring &string  &default &eval
                                     (get-search-string "Old: "))
         (newstring &string &prompt "New String: " NL)
         (count &integer &prompt "How many times? " NL
                                     &default 1))

        (do-times count
          (if (not (forward-search oldstring))
                                     (search-failure-annunciator))
          (do-times (stringlength oldstring)(rubout-char))
          (insert-string newstring)))
```

It can be invoked as:

```
     ESC X replace-n-times Washington Lincoln 2 CR
```

or:

```
     ESC X replace-n-times CR
```

in which case all arguments are prompted for, or:

```
   set-perm-key  ^Z9 replace-n-times
```

followed by striking ^Z9 at some time, prompts for all arguments, too. This function is defined so that it can be called from Lisp as:

```
     (replace-n-times "this" "that" 17)
```

or whatever, i.e., it is a Lisp function of three arguments.

When defcom-defined requests are reexecuted by ^C, they are repeated with identical arguments. This is what makes search-repetition by ^C work.

In addition to numeric arguments and request arguments, defcom can be used to specify prologues, documentation, and negative functions of request functions. Documentation is specified by the keyword &documentation (or &doc) followed by a documentation string subject to the same rules as given above under "Documenting Requests". Prologues are functions or code to be executed before any arguments are prompted for, perhaps to check for valid circumstances for calling this request. Negative functions are functions or code to be executed if the request is given a negative numeric argument: the negative function is given the negative numeric argument made positive. Negative functions are specified by the keyword &negative-function (&nf), followed by the name of the appropriate function, or forms, terminated by &end-code. Prologues are specified by the keyword &prologue, and the name of a prologue function or an &eval expression. The following is an example of the use of some of these features:

```
(defcom forward-topic
        &doc "Goes forward one or more topics. See also
                  $$backward-topic$."
        &numeric-argument (&repeat)
        &negative-function backward-topic
        (with-mark m
           (forward-search "Topic::"...........
```

SECTION 4

LDEBUG MODE

Emacs LDEBUG mode (Lisp Debug) provides an interactive Lisp environment designed for the debugging of Emacs extension code. Facilities are provided for tracing the Lisp stack, breakpointing code, and interacting with the native MacLisp trace facility. LDEBUG mode is specifically optimized for multiple-window interaction.

LDEBUG BUFFERS

The heart of the LDEBUG mode facilities is the LDEBUG buffer. The buffer named LDEBUG, when created by ldebug mode (either in response to a breakpoint's being executed, a trapped Lisp error, or the explicit "ldebug" extended request), evaluates any Lisp form typed into it when carriage return is struck after it. The form must be on one line; an error occurs if the form has syntactic errors (e.g., miscounted parentheses). The result of the evaluation is placed in the LDEBUG buffer on the next line, following the sign "=>", which indicates the result of such an evaluation. The Lisp variable "*" is set to the result of each successive evaluation, as at raw Lisp top level; this may be used to reference the last printed result.

Random Lisp forms such as "(+ 2 3)" or "current-buffer" can be typed at LDEBUG buffers, and the resulting buffer contents will in effect be a dialogue of an interaction with Lisp. Such buffers are often dprintable for later perusal. The values of variables can be set by evaluating the normal Lisp setq form, e.g., (setq var (+ foo 27)). As lines are placed into the LDEBUG buffer by the LDEBUG facility, the window (if any) containing it scrolls, if necessary.

Lisp values "printed" into the LDEBUG buffer are by default limited in length to ten and depth to six. The values of the option variables "ldebug-prinlength" and "ldebug-prinlevel" can be set to alter these defaults. The default input and output radices are both 8: these can be altered as the option variables "ldebug-ibase" and "ldebug-base".

Most Emacs requests can be used in LDEBUG buffers; they are in Lisp Debug mode, which is an extension of ordinary Lisp mode, with requests differing as detailed below.

EMACS AND LISP DEBUG MODE

The ldebug (ESC X ldebug CR) extended request can be invoked at any time, in the usual way Emacs extended requests are invoked. It places Emacs in the LDEBUG buffer as described above, and also sets up a system of Lisp error handlers "under" a new invocation of the Emacs request loop. Should any Lisp error occur while these handlers exist, the LDEBUG buffer is entered, placed on display if not already on display, the terminal's bell is beeped, and the Lisp error message is entered in the LDEBUG buffer. You are then at a "second (or greater) level" of LDEBUG, similar to being at Multics command level when an error occurs. The level number is part of the message entered in the LDEBUG buffer.

Recursive (level greater than 1) LDEBUG buffers can be released (aborting all executing code between the LDEBUG level being released and the previous level) via the ESC G (ldebug-return-to-emacs-top-level) request, the analogue of the Multics release command. It beeps and types "$g" in the LDEBUG buffer. The value of the variable ldebug-level tells the current level of LDEBUG buffers.

ESC P (for proceed) is the analogue of the Multics start command; more about its meaning for each different type of entry to an LDEBUG buffer is described below. In general, it restores the buffer and window from which the LDEBUG buffer entered.

ERROR TRAP ENTRIES TO LDEBUG

When an error trap entry to the LDEBUG buffer has occured, the Lisp stack can be traced via the ESC T (ldebug-trace-stack) request, and the value of variables can be inspected simply by typing their names (since they are Lisp forms) to the LDEBUG buffer. For this to work most effectively, at least one level of LDEBUG should be in the stack before the error is encountered.

A value can be returned to the Lisp error handler by typing it on a line, and instead of ending the line with carriage return (which would evaluate and "print" the result), ending it with ESC P. Lisp error handlers often want a list of the value to replace some erroneous value. For instance, in the following dialogue, an LDEBUG trap was entered because of the unbound variable "stuff": the programmer returned the symbol "value-i-wanted" as the intended value of the unbound variable:

```
     (myfun huff stuff)

     Lisp breakpoint unbnd-vrbl at level 1 in buffer LDEBUG:
     lisp: undefined atomic symbol   stuff

     ('value-i-wanted)$p
```

All correctable Lisp error breakpoints accept a <u>retry value</u> to be used to retry the failing operation; the undefined function breakpoint ("undf-fnctn") also accepts a list of a new value, in this case a function to be used instead.

The "$p" is always printed by ESC P, to remind the user of the $p which is used in raw Multics MacLisp to restart breaks. ESC P can also be used alone on a line (i.e., no value to be returned preceding it) to restart a break and let Lisp's default action occur.

ESC G can be used as usual to release a level of errors to the next lower LDEBUG level; ^G (command-quit) does not release past LDEBUG levels.

<u>CODE BREAKPOINTS</u>

Breakpoints can be set in interpreted extension code being debugged by typing ESC & in a Lisp Mode buffer with the cursor at the point in some function being debugged where you would like this break set. The LDEBUG mechanism creates this breakpoint by putting a call to a tracing function ("%%") in the code in the buffer, and evaluating the function definition it is looking at. This break code is left in the function to let you know that it is there: it includes a break number (they are assigned sequentially) to which this breakpoint can be referred by requests yet to be described.

You should be in at least one level of LDEBUG buffers before setting a break: thus, you should have said "ESC X ldebug CR" some time before setting breaks.

Having set a break, you can run the code being debugged. When the breakpoint is entered, the LDEBUG buffer is entered at a new higher level. A message of the form:

```
     Break 4 in function testfun
```

is put in the buffer, and the LDEBUG buffer is put on display. As in all LDEBUG buffers, arbitrary forms can be evaluated (including inspecting variables), and ESC T can be used to trace the Lisp stack. Again, ESC G releases a level of LDEBUG buffers.

ESC P is used to restart code breakpoints as well. A given breakpoint can be set for some number of proceeds (i.e., "3" means proceed, and proceed this breakpoint the next two times it is encountered automatically) by giving that number as a numeric argument to ESC P (i.e., ESC 3 ESC P). A message indicating the number of proceeds is inserted in the LDEBUG buffer. ESC P should be used alone on a line (i.e., no retry value) when restarting code (or trace) breaks.

When in a code break, ESC R (ldebug-reset-break) resets the current breakpoint, before restarting or releasing. The break code is removed from the function definition (visibly, if it is on display), and the function definition is reevaluated. ESC R with a numeric argument can be used to reset a break by number.

In an LDEBUG buffer, ESC L (ldebug-list-breaks) lists all the known code breakpoints: their numbers, the function in which each break appears, the buffer that function appears in, and the status of each break.

The source for the current breakpoint can be shown by issuing the request ESC S (ldebug-show-bkpt-source). It is placed in an available window (if in multiple window or pop-up-window mode), and the cursor is moved to the break code. Use ^XO to get back, or, in one-window mode, ^XB CR.

During function breakpointing, to determine where the editor was (i.e., what was the current buffer, and where was the current point) at the time the breakpoint was encountered use ESC ^S (ldebug-display-where-editor-was). It selects the appropriate buffer, moving the cursor to the point in it where the current point was when the breakpoint was taken. If the buffer is already on display in some window (or pop-up windows are being used), that window is selected, and ^XO returns you to the LDEBUG buffer for further probing or restarting. In one-window mode, the correct buffer is switched to, and ^XB gets you back. If the current point is moved by you explicitly (i.e., via normal Emacs requests) while visiting the buffer where the breakpoint was taken, it has its new position when the breakpoint is restarted. This is analogous to setting a variable before restarting with usual Multics debugging.

Using two or three windows to contain the LDEBUG buffer, the breakpoint source (function being debugged), and the buffer the functions being debugged are working on, is highly effective.

FUNCTION TRACING WITH LDEBUG

The standard MacLisp trace package can be used while in Emacs; extensibility features of the former allow LDEBUG to take control of the trace output and breakpointing provided by it.

All the facilities of the standard trace package can be used, by invoking trace from ESC ESC minibuffers. The trace package allows tracing of entries and exits to functions, arguments, and return values, and breakpoints when functions are entered. Some sample forms to trace the function testfun are given here: these are in Lisp syntax, and can be typed as such to LDEBUG mode. When typed to an ESC ESC minibuffer, the outer set of parentheses should not be supplied.

(trace testfun)
: Traces the input arguments and returns value of testfun each time it is invoked.

(trace (testfun break (< x 3)))
: Traces input and returns value of testfun, enters a breakpoint when entered and x (x can be an argument to testfun) is less than 3.

(trace (testfun break t))
: Same, but enters a breakpoint at every entry to testfun.

(trace (testfun entry (a b) exit (c)))
: Traces input arguments and returns value. Also prints out the values of a and b when testfun is entered and the value of c when it is exited.

The general syntax of trace invocations is (brackets indicate optional clauses, and angle brackets are syntactic variables):

```
     (trace <fnname-or-clause-1>  ... <fnname-or-clause-n>)
```

where <fnname-or-clause> is either a function name to be traced for input arguments only and return value, or:

```
     (<fnname> [break <break-condition>] [entry (<entry-vals>)]
                                         [exit (<exit-vals>)])
```

When a function is traced within Emacs (it is not recommended to trace internal Lisp or Emacs primitives, and no part of the redisplay should be traced in this way), trace output for entry and exit tracings are placed (and scrolled) directly into the LDEBUG buffer if it is on display; if it is not on display, this output is put in the LDEBUG buffer, and locally displayed as it is produced. The line of dashes and asterisks of local displays is not produced, as it cannot be known when the end of trace output has been reached. Thus, traced functions invoked from the minibuffer may often leave the cursor in the minibuffer awaiting clearing of the local display via linefeed or ^L.

Trace output generally looks like:

```
(3 enter testfun (3 5 (a . b)) /¦/¦ (4 5))
```

The indentation level gives the depth in currently active traced functions. The "3" is the recursion depth of the given function (e.g., testfun) being traced. The "enter" is the type of trace (enter vs. exit), (3 5 (a . b)) is the list of arguments (in this case, three arguments). The /¦/¦ sets off the entry values and exit values optionally selectable by the entry and exit keywords in the trace-invoking form. Exit traces look like:

```
(3 exit testfun 17)
```

If trace is used to set an entry breakpoint, the LDEBUG buffer is trapped to at the time the traced function is entered, in a way very much like a Lisp error break to LDEBUG. A message such as:

```
Entry breakpoint to function testfun
```

is printed into the LDEBUG buffer, and the terminal beeped. As with LDEBUG code breaks, ESC G releases, ESC P restarts, ESC R resets, and ESC ^S shows where the editor was at the time the break was taken. When in entry breakpoints to interpreted functions, the arguments can be inspected by name. ESC T can trace the Lisp stack, but unless *rset t mode was in effect (setting up an LDEBUG level does this automatically), trace information may not be present.

It is not necessary to have invoked ldebug before invoking trace in Emacs; LDEBUG is invoked automatically if an attempt is made to use trace in Emacs. If some critical mechanism is being debugged and normal trace handling (i.e., breakpointing/tracing to user_i/o from Lisp, not the Emacs handling just described) is necessary, the variables trace-printer and trace-break-fun should be made unbound (e.g., ESC ESC makunbound 'trace-printer) before the first reference to trace in a given invocation of Emacs.

SECTION 5

WRITING EMACS TERMINAL CONTROL MODULES (CTLS)

Support of video (and printing) terminals in Emacs is accomplished via terminal-dependent modules known as CTLs (from the typical name, e.g., "super58ctl" for a "super58" terminal). There are about two dozen supplied CTLs. Emacs attempts to locate an appropriate CTL in this directory at the time it is entered, based upon the terminal type maintained by Multics, and optional Emacs control arguments.

To support a type of terminal not supported by a supplied CTL, you must write a new CTL. A CTL is written as a Lisp source program, name TTYTYPEctl.lisp, where TTYTYPE is the name of the terminal type to be supported. If this terminal type is in your site's Terminal Type File (TTF), the name chosen should appear the same as it appears in the TTF, except that the name of the CTL should be all lowercase (Emacs lowercases terminal types when looking for CTLs).

CTLs are usually written by example from supplied CTLs. Personnel with no knowledge of Lisp at all have achieved this successfully. Once the CTL is written, it must be compiled before it can be used. Compilation is performed via the Lisp compiler, lcp. A typical command line to compile a CTL is:

```
lcp super58ctl
```

This produces an object segment, super58ctl, which, when debugged, can be installed in the "ctls" directory. To use a CTL being debugged, invoke emacs with the -ttp control argument and the full pathname of the <u>compiled</u> CTL:

```
emacs -ttp >udd>Support>Jones>emacs_development>super58ctl
```

Three control arguments for setting the terminal type are recognized by Emacs when given as the first command line argument. These are:

emacs -terminal_type STR or emacs -ttp STR
: where STR is your terminal type. The value of STR can be any recognized editor terminal type or the pathname of a control file to load. The terminal type given by STR is set permanently. Changing your Multics terminal type does not affect Emacs' memory of this STR.

emacs -reset
: Emacs forgets any characteristics of the terminal set by the -ttp option. Instead, Emacs checks the Multics terminal type, as is the normal case.

emacs -query
: Emacs queries the user for the terminal type without checking the Multics terminal type first. The answer you give may be any STR accepted by the -ttp option.

Once a CTL has been debugged, it should be installed (using normal Multics online installation conventions) in the ctls directory. Added names and links can be used to support many TTF terminal types via one CTL. When Emacs is given a terminal type (either from Multics Communication System or the -ttp control argument) for which it cannot find a CTL in the ctls directory, it lists the known terminal types by listing the primary names of all segments and links in the ctls directory, stripped of the ctl suffix. The choice between added names and links (or added names to links) should be made based upon whether Emacs should list a given name in this context.

The most effective method of writing a new CTL is to take one that was written for a similar terminal and modify it. Almost all of the extant CTLs were written in this way. The sources are Lisp source segments, generally one or two printed pages long. Good starting points are:

- vip7200ctl.lisp, typical of terminals that do not have the ability to insert or delete lines or characters.

- vip7800ctl.lisp, typical of terminals that do have these abilities. The two facilities are independent, either one, both, or neither may be present, although use of terminals without insert/delete lines at less than 300 baud may be found to be unacceptable.

The interfaces (function definitions) in a CTL are standardized. They have the same names in all CTLs. The Emacs screen manager calls these interfaces anonymously after the appropriate CTL has been loaded. The interface DCTL-init is called at Emacs start up time; it has the responsibility of setting various flags, and initializing the terminal. It should contain the statements:

```
(setq idel-lines-availablep t)
      if the terminal can insert/delete lines

(setq idel-lines-availablep nil)
      if it cannot

(setq idel-chars-availablep t)
      if the terminal can insert/delete/ characters

(setq idel-chars-availablep nil)
      if it cannot

(setq screenheight N.)
      where N is the number of lines on the screen (note
      the dot after the N).

(setq screenlinelen M.)
      Where M is one less the number of characters in a
      line on this terminal. Again, note the dot.

(setq tty-type 'TYPENAME)
      Where TYPENAME is a word like "super58" that
      identifies the terminal type.
```

At the time DCTL-init is invoked, the variable ospeed is set to the speed of the communications line in characters per second. This can be used to perform padding calculations. This value is usually computed from the line speed maintained by the Multics Communication System. The -line_speed control argument can be used to specify terminal speed for users logged in via the ARPANET.

Also before DCTL-init is invoked, the variable given-tty-type is set to the name by which the CTL was loaded with the "ctl" suffix stripped. This variable can be used in DCTL-init (and elsewhere) to enable and use different features of a terminal dependent on the name used to reference that terminal. To ensure that given-tty-type is different for various versions of a terminal, give the additional varieties of the terminal as links in the ctls segment. For example, there are links for vt100wctl and vt100wsctl to vt100ctl. These links allow the VT100 CTL to distinguish between various screen widths and heights by using the value of given-tty-type. The "eq" predicate (i.e., (eq given-tty-type 'dd4000)) can be used to check the value of this variable. The variable tty-type should be set by the CTL to a generic terminal type, e.g., vt100 for all varieties of VT100, as opposed to the type given in given-tty-type.

The following functions are available to the CTL writer:

- Rtyo takes one argument, a number (fixnum), and outputs that number as ASCII data. For example, (Rtyo 141) outputs an "a", and (Rtyo 33) outputs an ESC.

- Rprinc takes one argument, a character string, and outputs it. For example, (Rprinc "]I") outputs a right bracket and an I.

Both of these functions buffer their output until the Emacs screen manager dumps this buffer. This is always done at the end of any redisplay at all, and after DCTL-init is called.

The CTL writer must maintain the values of the special (global) variables X and Y relative to a zero origin screen position where the cursor was left. In return, you get to inspect these variables to do positioning optimization.

The CTL writer must provide the following interfaces to be called by the Emacs screen manager:

- DCTL-init (no arguments). Must set the flags listed above, initialize the terminal (if necessary), clear the terminal screen, and leave the cursor at position (0, 0) (home).

- DCTL-position-cursor (two arguments, a new X position and a new Y position). Move the terminal's cursor to the given position. Position 0, 0 is defined as the upper left hand corner of the screen. This function must check the variables X and Y, and output no characters if the cursor is known to be already at the desired position. Otherwise, it must use the values of X and Y to determine what type of motion is necessary, output characters to move the cursor, and update X and Y to the input parameters (the delay of the buffered output is not an issue).

Typically, DCTL-position-cursor determines which is the optimal movement based upon the relative positions of the cursor and the desired position. For terminals that have many forms of cursor movement, some combination of backspaces, linefeeds, and carriage returns may be adequate to effect some forms of cursor movement. Sometimes the sequences generated by the arrow buttons on the terminal may be used for relative positioning. Just about all terminals include some form of absolute positioning. The choice of optimal cursor positioning should be based upon which will output the fewest characters to effect the desired move. See hp2645ctl.lisp for an example of a very well optimized cursor positioner.

One useful trick in the writing of DCTL-position-cursor is the use of recursion. See adds980ctl.lisp for an example. If you choose to use terminal tabs, then your DCTL-init must set them, and you must take care not to clear them. No supplied CTLs (other than the extremely special-case printing terminal controller) use tabs.

- DCTL-display-char-string (one argument, a character string to be displayed). Must output this character string to the terminal at the current assumed cursor position. The string is guaranteed to contain no control or other nonprinting characters, and each character in it is guaranteed to take up only one print position. Be careful to update cursor position after printing the string; the lisp function stringlength may be used to ascertain the length/printing length of the string.

- DCTL-kill-line (no arguments). Clear the line from the current assumed cursor position to the end of the line, and leave the cursor at that original assumed position. Most video terminals have a clear-to-end-of-line feature; it should be used here if available. Some terminals do not, yet this function must be provided anyway. The machinations for simulating clear to end of line are somewhat involved; see adm3actl.lisp for an example of clearing to end of line by overwriting with blanks. Performance of this technique at 300 baud is generally completely unacceptable, rendering such terminals unfit for use with Emacs at that speed.

- DCTL-clear-rest-of-screen (no arguments). Clear the screen from the current assumed cursor position to the end. Leave the cursor where it was supplied. Some terminals have a "clear whole screen" function, but not clear to end of screen. Currently, you can use the clear whole screen function. If your terminal does not even have a clear-whole-screen function, it is probably not worth using with Emacs. If you choose to use tabs in cursor positioning, be wary of clearing them via this function.

Those are all the required functions. Some terminals require control sequences to change modes between normal Multics operation and operation within Emacs. (For example, a terminal might be switched between line-at-a-time transmission and character-at-a-time transmission.) Yet other terminals might use features during the operation of Emacs that should be disabled/reset when using Multics. (For example, the Digital Equipment Corporation VT100 uses "scroll" regions to simulate insert/delete lines. However, if a scroll region exists, it makes parts of the screen unusable when using Multics.) It is possible and quite common to switch between Multics and Emacs by using the ATTN key and the program_interrupt command. In such cases, the terminal is in the wrong mode at various times. If the terminal for which you are writing a CTL exhibits this behavior, you should add the following statements to DCTL-init:

(setq DCTL-prologue-availablep t)
: to specify that certain functions must be performed each time Emacs is entered from Multics.

(setq DCTL-epilogue-availablep t)
: to specify that certain functions must be performed each time Multics is entered from Emacs.

In addition, you must then supply the following two functions:

- DCTL-prologue (no arguments). Perform any operations that are required when Emacs is entered from Multics. This function is invoked immediately after DCTL-init is called and after Emacs is reentered after a QUIT via either the Multics program_interrupt or start commands.

- DCTL-epilogue (no arguments). Perform any operations that are required when Multics is to be entered from Emacs. This function is invoked immediately before Emacs is exited when the ^X^C (quit-the-editor) request is invoked, and immediately before Emacs is suspended when the ^Z^Z (quit) request is invoked or the ATTN key is hit on the terminal.

If you have stated that insert/delete lines is available, via setting the flag idel-lines-availablep to t, you must supply the following two functions. If you set this flag to nil, you need not write these functions:

- DCTL-insert-lines (one argument, a number of lines to be inserted). Open up the given number of lines on the screen. There are that many blank lines (created by DCTL-delete-lines) at the bottom of the screen at the time this function is invoked. The cursor is at position 0 of some line at the time DCTL-insert-lines is invoked. It must push the contents of that line down the supplied number of lines, leaving the cursor in the same place, and the line the cursor is on and the n-1 succeeding lines blank.

- DCTL-delete-lines (one argument, a number of lines to be deleted). Delete from the screen the supplied number of lines, starting with the line the cursor is on and proceeding downward. The cursor is to be left in the same place it was given. That many blank lines are assumed to be pulled up on the bottom of the screen.

If the flag idel-chars-availablep is set to t, indicating that insertion and deletion of characters is available, the following two functions must be supplied:

- DCTL-insert-char-string (one argument, a character string to be inserted at the current assumed cursor position). Insert the character string supplied at the current cursor position. Push to the right all characters at, and to the right of, the current cursor position. There are only blanks on the screen in the region being pushed off. Leave the cursor (and so update) after the last character of the inserted string.

- DCTL-delete-chars (one argument, the number of characters to be deleted). Physically delete from the screen the supplied number of characters, starting with the character at the cursor and on to the right. Move all characters to the right of these characters that many positions to the left. That many blanks are assumed to be moved in from the right edge. Leave the cursor where it was supplied.

Writing a CTL usually involves editing an existing one, trying it, modifying it, and iterating until it is solid. You use the -ttp control argument many times to switch back and forth between printing terminal mode and the new CTL when logged in from the terminal on which the CTL is being developed. For terminals with insert/delete features, it may be convenient to debug the CTL first without these features (claim they are not there in the DCTL-init), and add them later. Similarly, you are encouraged to write a better DCTL-position-cursors once you have one that works at all, for the convenience of editing the CTL with Emacs substantially reduces the effort of improving it.

For some terminals, padding may be necessary for some operations at some or all line speeds. If terminal behavior appears random, or garbage is left on the screen after a ^L or ^K, this may be the problem. Check the manual for your terminal about padding requirements. It may be convenient to define a function called DCTL-pad, which takes a number of microseconds or milliseconds as an argument, and issues enough pad characters to perform this padding. (Rtyo 0) or (Rtyo 177) are common, but check your terminal manual for what your terminal expects; (Rtyo 0) generally works. The variable ospeed gives the line speed in characters per second, for use in such calculations. Getting the padding right may involve quite a bit of tinkering on some terminals; one proven method in cases where padding is felt to be a problem is to specify a very large amount of padding (e.g., a second) and cut it down until it works. See dd40Q0ctl.lisp for an example of terminal padding.

The Lisp special forms cond and do are used heavily in CTLs. Since Emacs environment macros (do-forever, if, etc) should not be used in CTLs, the native Lisp forms are necessary. Here are the descriptions of cond and do:

```
(cond ((= this that) (thing1)(thing2))
      ((> a b)(second)(third) 27)
      ((< c 15)(other))
      (t (best 5)(chance)))
```

means:

> "If this equals that, call thing1 of no arguments, then call thing2 of no arguments, and return as the value of the cond the value returned by thing2. Otherwise, if a is greater than b, call "second" with no arguments, then call "third", and return 27 as a value. Yet otherwise, if c is less than 15 (all numbers octal), return the value obtained by applying "other" to no arguments. If none of the above are true, call "best" with an argument of 5, and then return the value obtained by calling "chance" with no arguments."

The cond special form is much like PL/I's

```
if ( .... ) then do;
     .....
end;
else if ( .... ) then do;
     .....
end;
else if ( .... ) then do;
     .....
end;
else do;
     .....
end;
```

The format of Lisp "do" used in CTLs to iterate is:

```
(do VARIABLE INITIAL-VALUE REPEAT-VALUE TEST form1 form2
    form3.. )
```

It is equivalent to PL/I's:

```
do VARIABLE = INITIAL-VALUE
   repeat REPEAT-VALUE
   while (^ TEST);

   form1;form2; ...
end;
```

which, itself, is equivalent to:

```
     VARIABLE = INITIAL-VALUE;
l:   if TEST then go to e;
     form1;form2; ...
     VARIABLE = REPEAT-VALUE;
     go to l;
e:   ;
```

The variable VARIABLE is locally defined inside the do. It may be used in the forms inside the do, in the "end test" TEST, and in the repeat value REPEAT-VALUE.

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 68) MULTICS EMACS EXTENSION WRITERS' GUIDE	ORDER NO.	CJ52-00
		DATED	JANUARY 1980

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ____________________ DATE ____________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

CUT ALONG LINE

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell