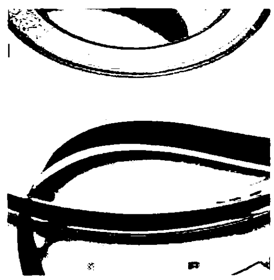
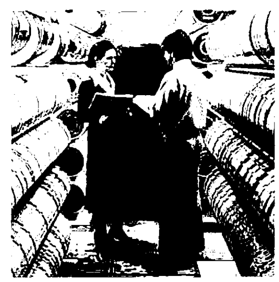
Prime Computer, Inc.

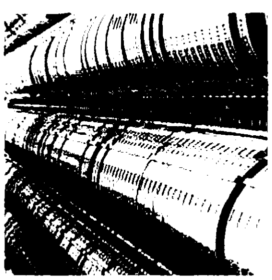
DOC5025-2LA

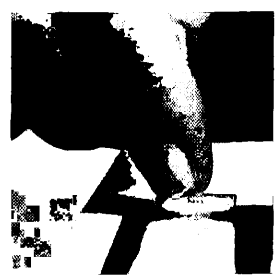
EMACS Extension Writing Guide

Revision 19.4








EMACS Extension Writing Guide

Second Edition

by

Barry M. Kingsbury

and

John Xenakis

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.4 (Rev. 19.4).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

CREDITS

Editor	Herbert Korn
Technical Support	Peter Neilson
Project Support	Phil Fulchino
Production Support	Judy Gordon
Document Preparation	Mary Mixon

PRINTING HISTORY — EMACS EXTENSION WRITING GUIDE

Edition	Date	Number	Software Release
First Edition	March 1982	IDR5025	18.3
Second Edition	January 1986	DOC5025-2LA	19.4

In document numbers, L indicates loose-leaf.

CUSTOMER SUPPORT CENTER

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)
1-800-343-2320 (within other states)
1-800-541-8888 (within Alaska)
1-800-651-1313 (within Hawaii)

HOW TO ORDER TECHNICAL DOCUMENTS

Follow the instructions below to obtain a catalog, price list, and information on placing orders.

United States Only

Call Prime Telemarketing, toll free, at 800-343-2533, Monday through Friday, 8:30 a.m. to 8:00 p.m. (EST)

International

Contact your local Prime subsidiary or distributor.

Contents

3 ELEMENTARY PEEL PROGRAMMING

4 LOGIC AND LOOPING

About This Book

This book shows you how to write commands and functions that extend the capabilities of the EMACS editor. If you are not familiar with the concept of a programmable editor, you may not appreciate the full capabilities that you now have at your fingertips. You no longer have just a text editor. Instead, you now have a tool for creating editors. This means that you can restructure the way EMACS looks, what it does, and how it acts on text.

If you are not familiar with the EMACS editor, you should read the EMACS Reference Guide (IDR5026) and its three update packages (PTU2600-105, PTU2600-107, and UPD5026-31A) before reading this book. This book assumes familiarity with the EMACS commands discussed in the EMACS Reference Guide. If you are not familiar with these commands, much of the information presented here will not make sense.

The language used to write EMACS commands and functions is called the Prime EMACS Extension Language (PEEL). Although PEEL is designed to operate on textual information, it contains everything necessary to write complete functions. Because these functions are only usable within EMACS, they serve to extend the capabilites of the editor.

As EMACS exists, it has many commands. Why is there a need to write more commands? Even though EMACS has most of the commands you might want, there are many situations where you want to perform actions based on particular information in the text. In other cases, you may want to adapt commands to some special organization in a text file. In addition, you can use the extension language to put together commands that talk to the PRIMOS® operating system.

The best way to learn how PEEL works is to examine real programs that do real operations. Consequently, the majority of the examples in this book are taken from the library functions in EMACS*>EXTENSIONS. In addition, you should look at the code in these libraries to see how the elements of a PEEL program fit together. Another important purpose of reading the programs in EMACS* is that you will gain an appreciation of the kinds of actions that must be performed in a string processing program.

This book assumes that you are an experienced programmer. While PEEL concepts are explained, no attempt is made to explain programming. If you know the LISP language, you already know the structure of PEEL. PL/I and PASCAL programmers will also be familiar with many of the concepts presented here.

Even though PEEL is a relatively new and unfamiliar language, experience has shown that PEEL rivals BASIC in its simplicity.

HOW TO USE THIS BOOK

The book is divided into two parts. The first part, Chapters 1 through 9, is a discussion of how to write extensions. The second part, Appendixes A and B, is a reference list of PEEL statements.

Chapters 1 and 2 introduce the PEEL language and discuss PEEL's strengths and limitations, the EMACS environment, compilation, and the binding of functions into the environment. The last topic is presented in the context of keyboard macros, which were discussed in the <u>EMACS Reference Guide</u>.

Chapters 3 and 4 discuss arithmetic functions, such as multiplication and division, and control statements, such as if, do, and select.

Chapter 5 puts together the information presented in Chapters 1 through 4 so that you can begin creating functions and commands. Other topics discussed are the command environment, data typing, transferring information between programs, and recursion.

Chapters 6 and 7 discuss the two forms of I/O available in EMACS: file (buffer) I/O and interactive I/O. Chapter 6 discusses how a program communicates to a user and how a user communicates back to a function. Chapter 7 looks at how information is altered, removed, and modified in a buffer. Of particular importance is the second half of Chapter 7, which explains cursors.

Chapters 8 and 9 discuss two advanced topics. Chapter 8 examines modes, which are a way of changing command definitions on a temporary basis. Chapter 9 looks at the information that EMACS keeps track of while it is executing.

Appendix A is an alphabetical listing of all built-in functions and commands of EMACS, as well as the majority of functions and commands contained in the libraries in EMACS*>EXTENSIONS.

Appendix B is a cross-reference listing of the information presented in Appendix A. Commands and functions are grouped by what they do in order to help you find the information that you need.

1 Introduction

The EMACS text editor provides a large and powerful command structure that can meet the needs of a wide range of users. Beginners can quickly learn a subset of commands that will help them perform most of the tasks needed to edit a file. More experienced users can draw upon the entire command vocabulary and use the full power of EMACS.

Whenever you use a text editor, you are performing the actions that someone thought would be appropriate for what you have to do. Consequently, the editor may lack features that would make editing sessions easier for you. In addition, you can only manipulate the kinds of things the designer thought should be manipulated and only in the way the designer intended. As discussed in the EMACS Reference Guide, the EMACS editor also *appears* to have these limitations. However, the information contained in this book shows you how to customize EMACS so that it meets your expectations and your needs.

FINDING INFORMATION

As you will see, the programming language used to write new commands is very rich. Because the programming language, PEEL, has been designed to manipulate textual information, its statements are tailored to the kinds of things found in text. For example, statements exist for manipulating words, sentences, and paragraphs.

Because PEEL is a large language, it usually offers a variety of ways to do the same thing. (All the PEEL statements are listed in Appendix A.) However, you never have to guess what the best way to program an operation is. EMACS has three commands to help you find the information you need.

Command	Command Name	Function
{CTRL-_} A	Apropos	Lists all commands related to an operation.
{CTRL-_} C	Explain	Lists what a keystroke does.
{CTRL-_} D	Describe	Lists textual information about commands and PEEL statements.

Let's look at how you would use these commands.

Example 1: Apropos

Suppose you want to move the cursor forward and you want to see which commands perform forward movement. You would type {CTRL-_} A followed by the word "forward". EMACS will list all commands that have the word "forward" in it.

Example 2: Explain

Suppose you want to write a command that, among other things, moves point (the current cursor) forward one character. As you know, the keystroke {CTRL-F} moves point forward. Typing {CTRL-_} C, then typing {CTRL-F} tells you that the internal name for that keystroke is forward_char.

Example 3: Describe

Suppose that after EMACS lists a command, you want to obtain more information. If you type {CTRL-_} D followed by the command name, EMACS prints the information you need.

The describe command, {CTRL-_} D, also lists the built-in functions of EMACS. For example, it lists the forward_search function, which is not available at command level. As you learn to program in PEEL you will find that describe is one of your best helpers. To learn more about describe, type ? while in describe.

THE RELATIONSHIP BETWEEN PEEL, EMACS, AND YOUR COMMANDS

In a traditional programming environment, the end result is a unit of code that can be invoked from PRIMOS. However, the commands that you write in PEEL can only be executed within EMACS. This is because all the commands you write are dependent on PEEL statements that only exist within the EMACS environment. Also, EMACS is an interpreter, not a compiler. Consequently, its code is never designed to be self-contained.

The PEEL statements are, in many respects, similar to statements in other languages. For example, here is the statement for moving forward a character:

```
(forward_char)
```

Notice the parentheses and the statement text. The way PEEL uses parentheses is identical to the way LISP uses them. If you have not programmed in LISP, you may find that entering parentheses before and after statements is awkward. For example, the following is PEEL's equivalent of a do loop:

```
(do_n_times count
     (forward_char))
```

Notice that this statement ends with two parentheses. (This example moves the current cursor forward the number of positions indicated by count.) Thus you can understand why, as statements become nested within statements, it becomes easy to lose track of which opening parenthesis belongs to which closing parenthesis.

EMACS can help you out of this difficulty. Type the following command:

```
{ESC} X lisp_on
```

This invokes the LISP programming mode. The most important command in LISP mode redefines the closing parenthesis so that EMACS momentarily jumps to the corresponding open parenthesis. This shows you if you have entered parentheses correctly.

One way to understand how EMACS and the commands you write relate is to think of EMACS as if it were the main module in a FORTRAN or PL/I program. Then all the functions and commands that you write can be thought of as being subroutines of this EMACS main module.

After you write a command and bring it into the EMACS environment, as is discussed in the next chapter, that command is indistinguishable from the fundamental commands supplied with EMACS.

HOW TO FIND OUT WHAT A STATEMENT DOES

Whenever you are in EMACS, you have the full capability of PEEL available at all times. As is described in Chapter 2, you can tell EMACS to interpret a command or file. The contents of the file or the command are then available. In addition, you can have EMACS directly execute PEEL statements at any time, causing the specified action to be performed immediately. You can use this fact to help see what a function does. To invoke the programming language, type {ESC}{ESC}. EMACS will respond with the prompt PL: (for programming language). You can now type a statement (or series of statements). When you type a carriage return, EMACS executes the PEEL statements.

As an example, suppose you want to write a command that, in part, moves the current cursor, or point, back to a previous space. EMACS contains the statement "skip_back_to_white". The question you might have is where does this leave point? Is it left so that the white space is immediately after point or before point? To find out, type {ESC}{ESC} while in the middle of a file, then type:

```
(skip_back_to_white)
```

In this way, if you are unsure about something, you can find out.

LISP

PEEL not only shares the format of LISP, it contains many elements of the LISP language. However, it is not a fully developed LISP language. For the most part PEEL merely contains the basic primitives necessary for creating, examining, and taking apart lists. (These statements are listed in Appendix B.) If you do not know LISP, you will not be able to use these functions. Fortunately, for the most part you can get by very easily without these functions. For example, more than 95% of all functions in the EMACS libraries make no use of these basic list statements.

If you have heard anything about LISP, you know that it is a language designed for processing lists. Because most programmers are more familiar with arrays, PEEL lets you use arrays in the same manner as traditional programming languages. Thus, with PEEL, there is little need to use lists if you do not want to.

In other words, you can write all the extensions you want without knowing anything about LISP.

If you want to learn more about LISP, read:

Winston, Patrick Henry and Horn, Berthold Klaus Paul. LISP, second edition. Reading, Mass.: Addison-Wesley Publishing Company, 1981, 1984.

2
Creating, Transforming, and Binding Extensions

This chapter discusses the following:

- Creating a macro
- Converting a macro into a PEEL source code extension
- Using PEEL source code
  - Saving source code in a file
  - Creating PEEL source libraries
- Binding an extension to a key
  - Key path conventions
  - Control characters in key paths
  - Letters in a key path
  - Key binding within local buffer only
  - Adding key binding to a PEEL source library
- EMACS fast load (EFASL) files
  - File naming conventions
- Loading extensions (library management)

The easiest way to create an extension, aside from having someone else write it for you, is to have EMACS write it. The extensions that EMACS can write are the keyboard macros discussed in the EMACS Reference Guide. This chapter reviews the steps involved in creating a keyboard macro, demonstrates the steps involved in transforming a keyboard macro into an extension, points out some of the limitations of this method, and shows how an extension is bound into EMACS. This binding procedure is the same one you use when writing your own extensions.

Note

The section titled LOADING EXTENSIONS: LIBRARY MANAGEMENT in this chapter tells you how to create and manage libraries of extensions. It also explains the best way to install new commands.

CREATING A KEYBOARD MACRO

Let's begin by looking at the simplest way to use EMACS's macro capability. Suppose, for example, you want to perform the same set of keystrokes over and over. You can tell EMACS to save that set of keystrokes and then have them executed again and again by means of a single EMACS command.

Suppose you wish to move the cursor to the right three positions, then insert the period character (.) into the text buffer. You would type the following:

```
{CTRL-F} {CTRL-F} {CTRL-F} .
```

Now suppose you wish to do this over and over again. You tell EMACS to "learn" this sequence of keystrokes and then to repeat the sequence whenever you desire.

To tell EMACS to "learn" a sequence of keystrokes, type the following:

```
{CTRL-X} (
```

This command tells EMACS to begin learning. From that point on, any keystrokes you type will be remembered for future use. EMACS will continue remembering keystrokes until you type the following:

```
{CTRL-X} )
```

This command tells EMACS that the keystroke sequence definition is completed.

To combine the example above (moving the cursor right three positions and inserting a dot) with this method of remembering sequences of keystrokes, you would type the following:

```
{CTRL-X} ( {CTRL-F} {CTRL-F} {CTRL-F} . {CTRL-X} )
```

The four keystrokes between the {CTRL-X} (and {CTRL-X}) are saved, ready for reexecution whenever you request it.

The way you request it is by typing the following:

```
{CTRL-X} E
```

This command tells EMACS to reexecute whatever keystrokes were saved by the commands {CTRL-X} (... {CTRL-X}).

You can also supply a numeric argument in order to specify repeated execution of the saved keystrokes. For example:

```
{ESC} 5 {CTRL-X} E
```

This command specifies that the saved keystrokes are to be executed five times.

Note that this method of saving and reexecuting keystrokes works with only one set of keystrokes at a time. If you use {CTRL-X} (and {CTRL-X}) again, the new keystroke definition will overwrite the previous one.

If you are in the habit of using fill mode, you should turn that mode off in any buffer in which you are creating macros, so that words at the end of one line will not spill over into the next line. Therefore, before creating a macro, type the following:

```
{ESC} X fill_off
```

CONVERTING A MACRO INTO A PEEL SOURCE CODE EXTENSION

The method of defining a macro we have just described can be used to define only one macro at a time; if you define a new macro, you erase the definition of the old macro.

Obviously, you need a method for defining and saving many macro definitions. To do this, you must save the macro definition in terms of PEEL source code. EMACS provides a simple method for converting a keystroke sequence macro into PEEL source code. However, before using this method, you must create a new EMACS buffer into which you will place the source code. The easiest way to do this is to create a new file using the following command:

    {CTRL-X} {CTRL-F}

This command prompts you for the name of a file. You should choose a filename of the form "name.EM". As we will see, the ".EM" suffix is the standard one for PEEL source code files. (Of course, you can create a new buffer without creating a new file. To do that, use the {CTRL-X} B command.)

Once you are within the buffer for the file you are creating, you are ready to load into that buffer the source code for the keystroke sequence macro you have created. To do so, type the following command:

    {ESC} X expand_macro

EMACS prompts you for the name of the macro, converts the macro into PEEL source code, and places the source code in the new buffer. For the name of the macro, you should choose a name that will be easy to remember, since you will be using that name in the future when you wish to invoke the macro.

For Standard User Interface (SUI), you may press the Command key instead of using {ESC} X.

Suppose you have used the keystroke sequence described previously, and have given it the name "tx". Then the following source code will appear in the buffer:

```
(defcom tx
  (do_n_times (numeric_argument 1)
    (forward_char)
    (forward_char)
    (forward_char)
    (self_insert \.)
))
```

While it is not yet necessary for you to understand this source code, the following points will give you the idea of what it is doing:

- The basic format is:

  (defcom name (action))

  defcom is the keyword which defines a new command; <u>name</u> is the name of the command you are defining (tx in the example above); and <u>action</u> is the action to be performed when the command is invoked.

- The first line of the action portion of the command is:

  (do_n_times (numeric_argument 1)

  <u>Notes</u>

  1. This line contains an unmatched left parenthesis -- the matching right parenthesis is on the last line of the PEEL source code.

  2. As we shall see in a later chapter, there is a way to supply a numeric argument when invoking the tx command you have just defined. (Recall that the discussion of the {CTRL-X} E command, earlier in this chapter, shows how to supply a numeric argument to specify repeated execution.) The line above uses that numeric argument, if supplied, to control the number of times that the specified action will be performed.

  The line above has two parts. The first, "do_n_times", defines a PEEL loop. The action following it is to be performed the number of times specified by the second part of the line, "(numeric_argument 1)".

  This is a PEEL function that has the value of the numeric argument (if any) specified with the tx command when it is invoked. Thus, if you invoke tx with no argument, the action you have specified is executed exactly once.

- Each of the next three lines contains the following code:

  (forward_char)

This is a PEEL function, corresponding to the keystroke {CTRL-F}, which moves the current cursor (point) ahead one character in your buffer.

- Next comes the line:

  ```
  (self_insert \.)
  ```

  This function inserts a period at the current point. It is the PEEL equivalent of typing a period at your keyboard to insert that character into your buffer.

- The last line contains two right parentheses. These parentheses match the two unmatched left parentheses in the first two lines of the source code.

The meaning of the PEEL source code will be explained in greater detail in later sections. For now, you should be satisfied with the gist of how the source code works.

Once you have inserted the source code into the buffer, you should write the buffer out to the file you specified. You do this by means of the following command sequence:

```
{CTRL-X} {CTRL-S}
```

This command sequence, which corresponds to SAVE FILE in the Standard User Interface, saves the buffer in a file defined by the current pathname.

USING PEEL SOURCE CODE

If you have been following our example, you have now done the following:

1. Used {CTRL-X} (and {CTRL-X}) to define a keystroke sequence
2. Used {CTRL-X} {CTRL-F} to create a new buffer and file
3. Used {ESC} X expand_macro to convert the macro you have defined into PEEL source code
4. Used {CTRL-X} {CTRL-S} to save the PEEL source code in the file

However, none of these steps make the new tx command available to you. Of course, until you define a new macro using {CTRL-X} (and {CTRL-X}), you can continue to use {CTRL-X} E to execute the macro again.

However, we are talking about a way to make the command known to EMACS so that you don't have to worry about overwriting it.

Making the New Command Available to EMACS

The method requires two steps. First, type the following:

    {ESC} X pl

This command tells EMACS to use the PEEL source code in the current buffer to make any command definitions a part of EMACS for the current EMACS session. (The letters "pl" stand for "programming language".) Thus, after executing this command, tx is available as an EMACS command for the current EMACS session.

Then, to use the command, just type:

    {ESC} X tx

For the Standard User Interface, type Command tx.

Now anytime you use this command during your editing session, EMACS will move the cursor right three characters and insert a period.

Saving Source Code in a File

The pl command we have just been discussing causes EMACS to compile and execute the PEEL source code in your current buffer. In the example above, the current buffer contains PEEL source code for a defcom, or command definition, of the command named tx. Therefore, compiling and executing this source code makes the tx command available to EMACS. However, the command is lost when you exit EMACS.

If you plan to use the same commands in session after session, you should save the source code in a file, and then compile and execute that file at the beginning of each session. To save the code, use {CTRL-X} {CTRL-S} as discussed above, or use {CTRL-X} {CTRL-W} to create a new file.

To compile and execute the source code in a file, you use the following command:

    {ESC} X load_pl_source

After you type this command, EMACS prompts you for a pathname of a PL source file. After you type the pathname, EMACS finds the file, then compiles and executes it. At that point, your saved command is available.

It may interest you to know that the command we have just discussed loads the source code into a buffer named ".pl". If you use the list_buffers command, {CTRL-X} {CTRL-B}, you will see this buffer listed.

Creating PEEL Source Libraries

If desired, you may create a file containing the PEEL source code for many commands similar to the ones illustrated above. Each time you use the expand_macro command, EMACS inserts your latest command definition into the current buffer. Thus, you can insert as many of these commands into the same buffer as you wish, and store them all in the same file.

If you start your next EMACS session with the load_pl_source command, all the commands you have defined in the source file you specify will be available to you in that session.

BINDING AN EXTENSION TO A KEY

If you have been trying out the example so far, then you have created a source file defining one or more new command names. To invoke one of these commands, you must use {ESC} X followed by the name of the command.

Frequently, it is more convenient if you redefine some of your keyboard command keys to correspond to the commands you have defined. For example, suppose you want the backslash key (\) to correspond to the tx command. To do so, type the following:

```
{ESC} X set_permanent_key
```

EMACS prompts you for a key path. In reply you press the backslash key, as follows:

```
Key Path: \
```

Next, EMACS prompts you for the command name. In reply, you type the command name tx, as follows:

    Command Name: tx

This sequence of commands causes EMACS to "bind" the backslash key to the tx command. Thus, from this point on, whenever you press the backslash key, EMACS automatically executes the tx command.

Note

> In binding a key to a command, you are doing something that is fairly common within EMACS. For example, the {CTRL-F} key is automatically bound to the forward_char command whenever you invoke EMACS. That is why {CTRL-F} moves the cursor one character forward. However, you can almost always overwrite an existing binding by binding the same key sequence to a new function.

Key Path Conventions

The last example bound a single key, \, to the tx command. You also may bind entire strings of keys to given commands. This allows you to put keystrokes together in a variety of ways to define as many commands as you wish.

For example, suppose you invoked the set_permanent_key command as described above, with the same command name, tx, but specified the key path as follows:

    Key Path: #%

You would be specifying that these two keystrokes together, #%, are needed to invoke the tx command. Similarly, you could bind such keystroke combinations as #+ or #$ to other commands that you have defined.

Any such string of keystrokes bound to a command is called a key path. You may have up to and including 10 keystrokes in a key path. Although any sequence of keystrokes may be used, a sequence beginning with control or escape characters is recommended to avoid losing the normal meaning of other characters.

> **Caution**
>
> In the examples above it would not be correct to bind the pound sign (#) alone to a function, because that would interfere with its use in the key path pairs #%, #+ and #$. Similarly, the bindings of the prefix keys {ESC} and {CTRL-X} should never be changed, because that would prevent many of the standard EMACS commands from working.

Control Characters in Key Paths

To specify control characters in a key path, you must use special symbols. (Note that {ESC} is a control character, equivalent to {CTRL-[}.) These special symbols are as follows:

^x	for {CTRL-x}
^[	for {ESC}
~~	for ~
~cx	for {CTRL-x}

The first and last lines of this table are for specifying the control code corresponding to any letter on the keyboard. For example, the character {CTRL-A} may be specified either as ^A or ~cA. (A few terminals do not have a ^ symbol. Most of these terminals have an up arrow key (↑) instead.)

> **Caution**
>
> Many terminals have both the ^ key and the up arrow key. If both are present, be careful to distinguish between these two keys.

For further information on specifying control characters, see the Escape Sequences entry in Appendix A. See also the quote_command function in Appendix A. This function helps you bind a command to a function key without knowing the keypath of the function key. However, you must still change the resulting sequence in your code to reflect the conventions in the preceding table.

Letter in a Key Path

When a key path contains a letter, EMACS distinguishes between uppercase and lowercase letters. For example, the key path ^[A creates a binding to {ESC} A. It does not create a binding to {ESC} a. If you wish to create both bindings, you must invoke the set_permanent_key command twice, specifying the key path ^[A the first time and ^[a the second time.

Key Binding Within Local Buffer Only

EMACS permits you to define a key binding that is valid only within the buffer that was active when you invoked the key binding command. To perform this kind of binding, use the set_key command rather than the set_permanent_key command. Except for that change, everything we have discussed is the same.

The set_key command is generally not very useful. In most cases, you should use set_permanent_key.

Adding Key Binding to a PEEL Source Library

Previously, you learned how to create a source file containing PEEL command definitions, and then how to load, compile, and execute that command at the beginning of each EMACS session so that the commands defined become available during your session. You now must learn how to add to that source file so that your key bindings also are automatically defined.

To bind a key path to a command, insert the following command sequence at the end of your PEEL source file:

```
(set_permanent_key "key path" "command")
```

You may add as many of these lines as you need to bind different key paths to different commands.

For example, to specify that the key path {ESC} A is to be bound to the command tx, you would type the following at the end of your PEEL source file:

```
(set_permanent_key "^[A" "tx")
```

In future EMACS sessions, when you use the load_pl_source command to load your PEEL source, the keystroke bindings you have specified will be automatically defined and available for the sessions.

Remember that you may use the set_key command as well as the set_permanent_key command in your source file. However, set_key commands normally would be completely useless because the keystroke bindings would be available only in the .pl buffer containing the source file.

EMACS FAST LOAD (EFASL) FILES

We have been discussing how to create a source file of PEEL commands and how to load that source file when you start a new EMACS session. Now we are going to take that process one step further.

If you had a large source library containing many command definitions, it would take a long time to load and compile that source file each time you began an EMACS session. Therefore, EMACS provides a way for you to save that file in a compressed form, so that it can be loaded much more quickly when you begin a new session.

The compressed command file is called an EMACS fast load file, and it always has a .EFASL suffix.

Suppose that your PEEL source file has the name MYPEEL.EM. (As stated earlier in this chapter, it is standard practice that a PEEL source file should have a .EM suffix. You will see why this is important in a few paragraphs.)

First, you should load the source file into an EMACS buffer with the find_file command, as follows:

```
{CTRL-X} {CTRL-F} MYPEEL.EM
```

Notice that this command does not compile and execute the source file; it simply loads it into a buffer. Of course, you could now compile and execute it with the {ESC} X pl command. Instead we are going to compress and save the source file by typing the following command:

```
{ESC} X dump_file
```

EMACS compresses the source file and stores the result into the file MYPEEL.EFASL.

Although EMACS has now compressed the source file, the commands defined in the file are still not available to EMACS. To make them available, you must load the partially compiled file with the following command:

```
{ESC} X load_compiled
```

This command prompts you for a file name. Respond by typing MYPEEL, so that the minibuffer at the bottom of your screen reads as follows:

```
Fasdump file name: MYPEEL
```

(It is an error to type MYPEEL.EFASL.) If the file is not located in your current directory, you can type an entire path name, without the .EFASL suffix.

The dump_file command, which compresses a source file and dumps the result to a file, performs what is known as a fasdump operation, while the load_compiled command performs a fasload operation.

File Naming Conventions

When you use the dump_file command, the name of the file created by EMACS depends upon the name of your source file in your current buffer and whether that source file name has a .EM suffix.

- If the source file has a .EM suffix, then EMACS simply removes that suffix and replaces it with the .EFASL suffix. For example, as we have just seen, if the source file name is MYPEEL.EM, then the dump file name is MYPEEL.EFASL.
- If the source file name does not have a .EM suffix, EMACS simply adds the .EFASL suffix. For example, if the source file name is MYPEEL, then the dump file name would be MYPEEL.EFASL.

Note that in all the commands that we have discussed, you never actually type the .EFASL suffix. In all cases, EMACS automatically supplies it.

LOADING EXTENSIONS: LIBRARY MANAGEMENT

Once you have created a fasload file, you can cause EMACS to load that file automatically at the beginning of each EMACS session. For example, to load the file MYPEEL.EFASL, you would type the following:

```
EMACS filename -ulib MYPEEL
```

If you prefer, you can shorten this command line by defining a new abbreviation using the PRIMOS ABBREV command, as follows:

```
ABBREV -AC MYEMACS EMACS %1% -ULIB MYPEEL
```

Once you have created the abbreviation, you can invoke EMACS and cause it to load the fasload file simply by typing the following:

```
MYEMACS [filename]
```

The ABBREV feature is explained in the Prime User's Guide.

To load several library files, you can create a top level library file which contains commands to load all the other library files. For example, suppose your PEEL source for your top level library file is the following:

```
(load_compiled "file 1")
(load_compiled "file 2")
(load_compiled "file 3")
```

Then when you load the top level library file, it will in turn load file 1, file 2, and file 3.

SUMMARY OF STEPS FOR CREATING EXTENSIONS

1. Create PEEL source code by:

   - Expanding a macro, as explained in this chapter
   - Writing PEEL source code directly, as explained in the rest of this book

   Be sure that EMACS is in no-fill mode. If necessary, enter {ESC} X fill_off.

   To associate PEEL commands with keys, add the set_permanent_key command at the end of the code for each command.

   To use the commands in the same EMACS session, type {ESC} X pl once and then, for each command, {ESC} X command_name.

2. Save the PEEL code in either uncompiled or compiled format.

   - To save as uncompiled PEEL source code, use a write_file or save_file command in a file with a name of your choice.

- To save as compiled PEEL code (the preferred way), use {ESC} X dump_file to save a file of compiled PEEL code. Before you use dump_file, the name of your buffer or file of source code should end in .EM, so that dump_file can create a corresponding filename with an .EFASL (EMACS fastload) suffix.

3. For your next EMACS session, load the code in one of the following ways:

   - If the code is uncompiled source code, use {ESC} X load_pl_source.

   - If the code is compiled, do a "fast load" with either of the following methods:

     - Use {ESC} X load_compiled after you invoke EMACS or

     - Use -ulib filename to load the code when you invoke EMACS or

   In either case, use only the root of the filename without the .EFASL suffix.

4. To invoke any command you have loaded, use {ESC} X command_name. If the set_permanent_key command is included in the source code, you can invoke the associated command by pressing the appropriate key or combination of keys.

WHAT IS NEXT

In this chapter, you learned how to make customized EMACS commands from macros. You can save these personalized commands and reload them at every EMACS session.

The rest of this book shows you how to create your own commands directly in PEEL. However, the methods of saving and reloading these commands are the same ones you learned in this chapter.

3
Elementary PEEL Programming

This chapter discusses the following:

- PEEL as a programming language
- Elements of list processing
- Assignments
- Arithmetic operations
- Arrays
- Functions that manipulate lists

PEEL AS A PROGRAMMING LANGUAGE

The preceding chapters have introduced you to some of the simpler ways you can use PEEL. In this chapter, we are going to discuss PEEL as a programming language, show you how to create and execute programs in that language, and then how to compile and execute those programs.

Also in this chapter we will look at the mathematical operations that can be performed using PEEL. It may surprise you that PEEL has arithmetic or mathematical operations, because PEEL is an extension to an editor, and might be expected to be strictly a string processing language. However, as with any programming language, PEEL programs use counters for such things as loops and comparisons, and arithmetic operations are needed for these.

You should not think of PEEL either as a string processing language or a numbers processing language, although it does process both numbers and strings. Rather, you should classify PEEL as a list processing language. In fact, PEEL is based on LISP, the list processing language invented by John McCarthy in the late 1950's. LISP has been used for many years to program artificial intelligence applications. It is used in EMACS as the extension language, PEEL, which enables EMACS to become a very intelligent editor indeed.

ELEMENTS OF LIST PROCESSING

This section discusses the following:

- Atoms
- Integer and string atoms
- Lists
- Nested lists
- Symbolic expressions (S-expressions) and forms
- How to execute PEEL programs
- Output from PEEL programs
- The value of an S-expression
- Evaluating S-expressions: effects and side-effects

Atoms

The fundamental objects in PEEL are atoms. You specify an atom by typing its name, which normally consists of letters, digits, or any printing character except a blank, a left parenthisis, or a right parenthesis. Often, the name of an atom is an ordinary word, such as "counter".

Integer and String Atoms

As we shall see, an atom can have as a value any of a wide variety of data types, including numeric or string values. You specify numeric or string atoms directly by typing the value of the atom.

For example, you may specify an integer atom by typing one or more decimal integers, optionally preceded by a plus sign (+) or a minus sign (-). The following specify integer atoms:

```
234
+234
-234
```

You may specify a string atom by typing any printable ASCII characters enclosed in double quotation marks. The following specify string atoms:

```
"a b c"
"This is a sentence."
"a(b"
```

Lists

A list is a group of atoms enclosed in parentheses. For example:

```
(forward_char)
```

This is a list containing the single atom forward_char.

Another example is:

```
(setq x 5)
```

This is a list containing three atoms: setq, x, and the integer atom 5.

It is possible for a list to contain no atoms whatsoever. Such a list is called the null list, and is written as follows:

```
()
```

Each of the atoms in a list is said to be a member of the list. For example:

```
(setq x 5)
```

This list has three members: setq, x, and 5.

Nested Lists

It is of fundamental importance that a member of a list can be another list. This expands our definition of list, since now a list can contain more than just atoms. Consider the following example:

```
(setq x ( + 2 3 ))
```

This example illustrates a list as a member of another list. The outer list contains three members, but only two of these members, setq and x, are atoms. The third member is the inner list (+ 2 3). The inner list, for its part, contains three atoms: +, 2, and 3.

Symbolic Expressions (S-expressions) and Forms

A symbolic expression, abbreviated s-expression, is either an atom or a list.

A form consists of one or more s-expressions.

When you write the source code for a PEEL program, you are really writing a form, consisting of s-expressions.

EVALUATION OF S-EXPRESSIONS

Now let us look at some actual examples of the PEEL programming language in action. If you have a terminal available to you, you can try out these examples as you read.

How to Execute PEEL Programs

Let us review the major ways we have already learned to execute PEEL programs.

Suppose, for example, you are editing in EMACS, and you have text displayed on your screen. If in the middle of your edit you type {ESC} {ESC} (that is, you press the "escape" key twice), at the bottom of your screen, in the minibuffer, EMACS displays the prompt PL: to indicate that it is ready to accept a PEEL statement.

If you now type (forward_char), what you have typed is an s-expression, a list containing the single atom forward_char. If you then press RETURN, EMACS automatically executes what you have typed as a PEEL statement. You see the result on your screen: the cursor, indicating the location of the EMACS point, will have moved to the next character in the buffer.

This is the simplest way to execute a PEEL program, but obviously you can use it for only the very shortest PEEL programs, usually programs consisting of a single short s-expression.

To execute a more complex PEEL program, you can type the source program into an EMACS buffer and then tell EMACS to execute the contents of that buffer as a PEEL program. Following is a simple example of how you do that.

1. Start up a new EMACS buffer by typing the following EMACS command:

   ```
   {CTRL-X} B
   ```

2. Type the following text into the buffer:

   ```
   (forward_char)
   ```

3. Move the cursor in your buffer so that it is on the character "f", and type the EMACS command:

   ```
   {ESC} X pl
   ```

This command tells EMACS to interpret everything in your current buffer as a PEEL source program, and to compile and execute it. After EMACS completes execution, you will see the result immediately on your screen: the cursor will have moved forward one character, to the letter "o" following "f".

In the examples that follow, if the example is simple, you can use either method described above to execute the program on your terminal. However, as soon as the example gets a little more complicated, be sure to use the second method, typing the source into a regular text buffer and then executing the contents of the buffer by means of {ESC} X pl.

Output From PEEL Programs

Following is a PEEL statement that can be executed by either of the methods described above.

```
(+ 2 3)
```

As we will see in the next few pages, this statement directs PEEL to add the values of 2 and 3. Because the statement does not tell PEEL what to do with the sum, PEEL throws the sum away.

To print out the value of this s-expression, use the print function, as follows:

```
(print (+ 2 3))
```

This statement tells PEEL to compute the sum of 2 and 3 and print out the result. When you execute this statement, PEEL prints the sum at the top line of your edit buffer, overwriting whatever was already displayed there. It is important to understand that the sum is not actually stored into the text buffer; rather, the sum is simply displayed on your screen. If you now type {CTRL-G}, EMACS refreshes the screen, the sum that you just computed is removed, and the original text from the text buffer is again displayed.

The Value of an S-Expression

Every s-expression, whether an atom or a list, has a value. Whenever you execute a PEEL program, EMACS determines the value of each s-expression as it goes along.

For example, consider the example we used above:

```
(+ 2 3)
```

As stated above, the value of this s-expression is 5, which PEEL computes by adding together 2 and 3.

Let us take a closer look at how PEEL evaluates this expression, starting with the two numeric atoms 2 and 3. Since each of these is a numeric atom, it automatically has a value. The values, of course, are 2 and 3, respectively. When PEEL evaluates the s-expression illustrated above, it evaluates these two atoms first, before doing anything else.

Functions in S-Expressions: PEEL does not evaluate the + atom in the same way it evaluates the other two atoms. The difference is that, by convention, the first atom in a list is always considered to be a function name. An atom appearing at the beginning of a list is a function that requires arguments before it can be evaluated, while atoms appearing later in the list are evaluated intrinsically.

In the example we are considering, + is an atom which, when used as the first atom in a list, is a function that adds together its arguments.

The format of a function reference in PEEL is as follows:

```
(functionname arg1 arg2 ...)
```

That is, the function name appears first in the list and the arguments are the other members of the list.

The * atom is similar to the + atom, except that it multiplies its arguments rather than adding them. For example:

```
(* 2 3)
```

This s-expression uses the * function and returns the value 6.

There can be as many as eight arguments. As we shall see, most functions require a specific number of arguments, but, as it happens, + and * are functions that can have a variable number of arguments. For example:

```
(+ 1 2 3 4 5)
```

This is a function reference with + as the function name and 1, 2, 3, 4 and 5 as five separate arguments. PEEL evaluates this function by adding together the values of the five arguments. The result is 15. To print the result, you would use the following statement:

```
(print (+ 1 2 3 4 5))
```

Functions With No Arguments: Some functions take no arguments. It may surprise you to know that we have already seen several examples of this. Consider the following:

```
(forward_char)
```

This is a list containing a single atom, forward_char. When PEEL evaluates this list, it considers the first (and only) atom in the list, forward_char, to be a function name. Since there are no other members of the list, PEEL invokes the forward_char function with no arguments. PEEL responds by moving the point in your current buffer forward one position. On the other hand, you can invoke this same function with a numeric argument, as follows:

```
(forward_char 5)
```

In this case, PEEL invokes the same function with a single numeric argument, whose value is 5. The result is that EMACS moves the point in your current buffer forward five characters.

If you have been following this discussion, you may be puzzled at this point, because we have been discussing forward_char as a function without telling you what the value of the function is. We will explain this later, but for now just think of forward_char as a function that does something, but does not compute a value that you care about.

Functions With Expressions as Arguments: The arguments of a function need not be atoms. Consider the following example:

```
(+ 2 (* 3 4))
```

Here we see the function +, again with two arguments. The first argument is atom 2, but the second argument is another list, (* 3 4). To evaluate this list, PEEL multiplies 3 and 4 together to get the product 12. Thus, evaluation of (+ 2 (* 3 4)) is equivalent to evaluation of (+ 2 12), for which PEEL computes the value 14.

You can build up extremely complex s-expressions. For example, consider the following:

```
(* (+ 2 3) (+ 4 5))
```

In this example, we see that * is a function with two arguments, and that each of those arguments itself requires a function computation. The value of the first argument is computed by adding together 2 and 3 to get 5, and the value of the second argument is 4 plus 5, or 9. The value of the entire s-expression is computed by multiplying 5 by 9 to get 45.

These examples illustrate the following general rules about evaluation of s-expressions:

- PEEL evaluates s-expressions from the inside out. That is, it evaluates the arguments of a function before evaluating the function. (More precisely, the first item in the list is evaluated, and if found to be a function name, then the arguments are evaluated before the function is applied to the arguments.)

- PEEL evaluates the arguments of a function from left to right.

Note

We will see later that it is even possible for the first member of a list, the member we have been calling the function name, to be an s-expression. In that case, PEEL evaluates that s-expression first, before evaluating any of the arguments.

Evaluating S-Expressions: Effect and Side-effect

When an s-expression is evaluated, the effect of that evaluation is the value of the s-expression. For example, the effect of (+ 2 3) is the value 5.

Sometimes an s-expression also has a side-effect. A side-effect is an action that PEEL or EMACS takes in addition to evaluating the expression. The simple example of +, just above, has no side-effect. However, the s-expression (forward_char 5) has the side-effect of moving the point in your text buffer forward five characters.

Every s-expression has an effect, or value, but not all s-expressions have side-effects.

What then, you may wonder, is the value of (forward_char 5)? You can easily discover that for yourself by simply executing the following statement by either of the two methods we have discussed:

```
(print (forward_char 5))
```

If you try this, you will discover the following:

- PEEL prints () at the top of your terminal screen. This is the null list, which we discussed previously in this chapter, and it is the effect or value returned by the forward_char function.

- EMACS moves the point in your buffer forward five characters. This is the side-effect of the forward-char function.

Thus we see that forward-char is an example of a function whose value is not numeric but is a list. As we shall see, a function may return any of a wide variety of data types, including numbers, lists, strings, and so forth.

ASSIGNMENTS

In most high-level programming languages, the = sign is used to indicate assignment of a value to a variable. For example, in many languages the following statement assigns to the variable X the numeric value 5:

```
X = 5
```

In PEEL, assignments are performed by executing s-expressions involving PEEL functions. This section presents the following:

- The setq function
- The quote (') operator
- The eval function
- PEEL Data Types

The Setq Function

The setq function can be used to assign a value to a variable. For example:

```
(setq x 5)
```

This statement assigns to the atom x the value 5. This is the PEEL equivalent of the assignment statement previously shown using the = sign. Notice that assignment is a side-effect of the setq function.

Once an atom has been assigned a value, you can use that atom in any s-expression, and PEEL will use the assigned value when it executes the s-expression. For example, once the preceding statement has been executed, then the statement (print (+ x 12)) prints the value 17, computed by adding 5 (the value of x) to 12.

The Quote (') Operator

Normally when an atom appears in an s-expression, PEEL evaluates that atom and replaces it with its value. You may suppress the evaluation by using the quote operator. The quote operator consists of single quotation mark (') placed before the atom to be quoted.

Consider the following:

```
(setq v x)

(setq w 'x)
```

The first line assigns to the variable v the value of the atom x, or 5. The second line assigns to the atom w the atom x itself, not its value.

To verify this for yourself, execute the preceding setq statements. Then try each of the following print statements:

```
(print x)

(print v)
```

As you see, PEEL prints the value 5. Now execute the following print statement:

```
(print w)
```

This time, you will notice, PEEL prints the name of the assigned atom, x.

Eval Function

The eval function is, in a sense, the inverse of the quote operator. Normally, when an atom is used in an s-expression, PEEL performs one evaluation of that atom. As we have seen, the quote operator suppresses that one evaluation.

The eval function causes PEEL to perform a second evaluation, that is, an evaluation of the result computed from the first evaluation. To understand this, consider this example:

```
(setq x 5)

(setq w 'x)
```

As we have seen, the value of w is not 5, but is rather the atom x. Suppose you now attempt to execute this statement:

```
(print (+ w 3))
```

You might think that PEEL prints the value 8. However, remember that the value of w is the atom x, not the integer 5, so in fact PEEL prints an error message telling you that it is illegal to add the value of w (the atom x) to something else.

Now suppose you attempt to execute this statement:

```
(print (+ (eval w) 3))
```

How does PEEL execute this statement? First, it evaluates the atom w to get the atom x. However, since w is used as an argument to the EVAL function, PEEL does an additional evaluation, computing the value of x as 5. In this case, PEEL does print the value 8, computed by adding 5 to 3.

PEEL Data Types

Similar to most high-level languages, PEEL supports a variety of data types. We will discuss them in detail in a later chapter, but for now let us look at the most commonly used data types.

You may use the setq function to assign a value of almost any data type to a variable. We have already seen two different data types used in assignments, the integer data type and the atom data type. As we have seen, the statement (setq x 5) assigns to the atom x the integer value 5, while the statement (setq w 'x) assigns to w the atom x. This illustrates the use of the atom data type as opposed to the integer data type.

The string data type may also be used in assignments. Consider, for example, the statement:

```
(setq str "This is a string.")
```

This statement assigns to the atom str the string shown. You may print out this string by using the statement:

```
(print str)
```

Perhaps most interesting is the list data type. That is, you may assign to a variable a value which is equal to a list of other atoms. To do this, you usually need the quote operator ('). For example, consider the following:

```
(setq lv '(+ 2 3))
```

Here the ' operator applies to the entire list that begins with the following left parenthesis. Therefore, PEEL does not evaluate this list, but simply assigns the list to the atom lv. Therefore, lv does not have the integer (data type) value 5, but rather has the list (data type) value (+ 2 3). You can see this for yourself by executing the following statement:

```
(print lv)
```

Notice that this statement prints out the actual list as its value.

As before, you can force an additional evaluation to take place by using the eval function. Consider, for example, the following statement:

```
(print (eval lv))
```

This statement causes PEEL to actually evaluate the list assigned to lv, and to print its value, 5.

ARITHMETIC OPERATIONS

All numeric variables in PEEL are integer variables. Therefore, you cannot use PEEL to manipulate floating point or fractional values. Furthermore, all mathematical operations are performed using integer arithmetic.

The following functions are used to perform arithmetic operations:

- Addition and multiplication
- Subtraction and negation
- Incrementing and decrementing by one
- Division and modulo
- Forms and nested statements

Addition and Multiplication

As we have already seen, + performs addition and * performs multiplication. For example, (+ 2 3) has the value 5, while (* 2 3) computes the value 6. Neither of these operators has a side-effect.

The + function can take from one to eight arguments. All arguments must be numeric. If there is only one argument, + and * simply return the value of this one argument. If there is more than one argument, + returns the sum of all the arguments, and * returns the product of all the arguments. Here are some examples:

```
(+ 23)

(* 23)
```

Each computes the value 23.

```
(+ 1 2 3 4 5)
```

This s-expression computes the value 15.

```
(* 1 2 3 4 5)
```

This s-expression computes the value 120.

As with all functions, + and * can have as their arguments variables and s-expressions. However, all arguments to these and other arithmetic operators must have numeric values. For example, consider the following:

```
(setq length 20)
(setq width 5)
(setq area (* length width))
(setq perimeter (* 2 (+ length width)))
```

The first two lines of this example set the variables length and width to the integer values 20 and 5, respectively. The third line is the PEEL representation of the ordinary formula area = length * width, and it assigns to the variable area the value 100. The last s-expression is the PEEL representation of the ordinary formula perimeter = 2 * (length + width), and it assigns to the variable perimeter the value 50.

Subtraction and Negation

The - function is used for both subtraction and negation. When it has one argument, it performs the negation operation, and when it has two arguments, it performs the subtraction operation.

Here is an example:

```
(setq a 5)

(setq b 8)

(setq c (- a))

(setq d (- b a))
```

The third line of this example negates the value of a and assigns the result, -5, to c. The fourth line subtracts the value of a from the value of b, and assigns the result, 3, to the variable d.

Incrementing or Decrementing by One

Adding 1 to, or subtracting 1 from, an integer is a fairly common operation in computer programs. Thus, PEEL provides two operators for convenience.

The 1+ function takes one argument and returns the value obtained by adding 1 to that argument. For example:

```
(setq counter (1+ counter))
```

This adds 1 to the value of counter and assigns the result back to counter, thereby incrementing the counter by 1.

The 1- function takes one argument and returns the value obtained by subtracting 1 from that argument. For example:

```
(setq counter (1- counter))
```

This subtracts 1 from the value of counter and assigns the result back to counter, thereby decrementing the counter by 1.

Although these two operators are often convenient, they do not really provide any additional functional capability to PEEL. As you can see, the examples above are equivalent, respectively, to the following examples:

```
(setq counter (+ counter 1))

(setq counter (- counter 1))
```

Division and Modulo

As previously stated, PEEL performs only integer arithmetic. This may be inconvenient when you wish to divide two values and obtain a noninteger value. To make this easier for you, PEEL provides functions that compute either the quotient or the remainder of a division.

The division (/) and modulo functions complement each other. Each of these functions takes two arguments. The / function returns the quotient obtained by dividing the first argument by the second. For example:

```
(/ numerator denominator)
```

This returns the value of the quotient obtained when the numerator is divided by the denominator.

The modulo function returns the remainder obtained when the first argument is divided by the second. For example:

```
(modulo numerator denominator)
```

This returns the remainder obtained when the numerator is divided by the denominator.

In most cases, the quantities you will be dividing are positive integers. When PEEL divides two integers using /, it <u>truncates</u> the result by throwing away any fractional part. For example:

```
(/ 44 5)
```

This returns the value 8. The actual value of 44 divided by 5 is 8.8, and PEEL truncates this result and returns 8.

On the other hand, consider the following expression:

```
(modulo 44 5)
```

This returns the remainder from the same division, which is 4.

The above discussion tells what happens when both the numerator and the denominator are positive. The following rules describe what happens when one or both are negative:

- The / function always truncates toward 0. Therefore, (/ -44 5) and (/ 44 -5) always return the value -8.

- If the value returned by modulo is non-zero, then its sign is always the sign of the denominator. For example:

  ```
  (modulo -44 5)

  (modulo 44 -5)
  ```

These return the values 1 and -1, respectively.

Forms and Nested Statements

Because PEEL statements can be nested to any depth, you have the power to make your PEEL programs as unintelligible as you wish.

For example, consider the following statement which converts a temperature value from celsius to fahrenheit:

```
(setq fahrenheit (+ (/ (* (setq celsius 20) 9) 5) 32))
```

This example contains several nesting levels, and may be difficult to read. You can simplify it as follows:

```
(setq celsius 20)
(setq celsius (* celsius 9))
(setq celsius (/ celsius 5))
(setq fahrenheit (+ celsius 32))
```

You may find this way of writing the program is not nested enough. Ultimately, how you write a statement is a matter of personal taste.

In informal terminology, an s-expression that computes one simple operation is often called a statement. A nested series of statements is called a form. Using this terminology, the first fahrenheit example above is defined as one form that contains five statements.

ARRAYS

Most high-level languages have the capability of storing data values in arrays. Usually the high-level language provides a statement such as DIMENSION or DECLARE, which you use to specify that a certain variable is to be an array. When such declarative statements are used, you specify at the time your program is compiled that a certain variable is to be an array of a certain data type and a certain size.

Unlike other high-level languages, PEEL does not use declarative statements to define arrays. As with other PEEL capabilities, array creation and manipulation is performed in PEEL by invoking predefined functions.

This section discusses the following topics:

- Creating an array
- Assigning values to an array
- Filling all or a portion of an array
- Referencing an array
- Arrays of other data types
- Arrays with multiple names
- Releasing storage allocated to an array
- Other array functions

Creating an Array

You use the make_array function to create an array. PEEL creates the array as a side-effect of the function invocation. The format for using this function is as follows:

```
(make_array type number)
```

Here type is an atom, a quoted name, specifying the data type of the elements of the array, and number is the number of elements in the array.

For example, consider the following:

```
(make_array 'integer 5)
```

This function creates an integer array containing five elements. As we shall see later, you may reference the individual elements by means of subscript index values, which range from 0 through 4.

Unfortunately, the preceding example by itself is useless. The reason is that, as in the case of other PEEL statements, unless you assign the result to a variable, the result is simply thrown away. To do this, you use the setq function. The full format for use of make_array when you wish to use the created array is as follows:

```
(setq variable (make_array type number))
```

Here, <u>variable</u> is the name of the variable that you wish to use in referencing the array. (You can think of it as the name of the array.) For example, consider the following statement:

```
(setq boxes (make_array 'integer 5))
```

This creates an array called boxes that contains five individual integer values, numbered 0 through 4.

<u>Assigning Values to an Array</u>

You use the aset function to assign a value to an individual element of an array. The format is as follows:

```
(aset value variable index)
```

Here, <u>value</u> is the value to be assigned to the array element, <u>variable</u> is the name of the array, and <u>index</u> is the index of the element in the array. The value must have the same data type as the data type of the elements of the array. In addition, the index must have an integer value greater than or equal to 0 and less than the second argument of the make_array function invocation that created the array.

For example, consider the array boxes, which was created earlier with the statement (setq boxes (make_array 'integer 5)). For this array you may use a statement in the following format to assign a value to an individual element of that array:

```
(aset value boxes index)
```

Here, value is integer value less than 268435456 or 2**28, and index is an integer value between 0 and 4. For example:

```
(aset 243 boxes 2)
```

This statement assigns the value 243 to the element of boxes with the index 2. (Note that this is the third element of the array boxes, since the first element has the index 0.)

Filling All or a Portion of an Array

You may use the fill_array function to assign the same value either to all elements of an array or a contiguous portion of the array.

To fill an entire array, use the statement in the following format:

```
(fill_array variable value)
```

Here, variable is the name of the array, and value is any value whose data type equals the data type of the elements of the array. PEEL assigns the value to every element of the specified array.

For example:

```
(fill_array boxes 100)
```

This statement assigns the value 100 to each of the five elements of the array boxes, which we previously defined.

If desired, you may specify two additional arguments to fill_array to assign a range of index values for a contiguous set of elements. For example:

```
(fill_array boxes 100 2 4)
```

This statement assigns the value 100 to the three elements of the array boxes with indices 2, 3, and 4. The first two elements, with indices 0 and 1, are left unchanged.

The format of this use of fill_array is as follows:

```
(fill_array variable value index1 index2)
```

Here, index1 is an integer greater than or equal to 0 and less than the size of the array, and index2 is an integer greater than or equal to index1 and less than the size of the array. PEEL assigns the specified value to all elements of the array whose index value is greater than or equal to index1 and less than or equal to index2.

Referencing an Array

To recover the value of an individual element of an array, use the aref function. The format is as follows:

```
(aref variable index)
```

Here, variable and index are the same as for the aset function. The value returned by this function reference is the value stored in that array element. Suppose, for example, you have filled an array by using the following statements:

```
(fill_array boxes 200 0 2)
(fill_array boxes 100 3 4)
```

Then the statement (print (aref boxes 1)) prints the value 200, and the statement (print (aref boxes 3)) prints the value 100.

Arrays of Other Data Types

We will discuss PEEL data types in detail in a later chapter, but for now let's look ahead and see how to define arrays of other data types.

Consider the following example:

```
(setq messages (make_array 'string 30))
```

PEEL creates an array called messages containing thirty elements, each of which has the data type string.

You may use the aset, aref, and fill_array functions just as you did before, provided that you use string values, where before you used integer values. For example:

```
(fill_array messages "This is a message.")
```

This statement assigns to each of the thirty elements of the array messages the string "This is a message.". Now let's change one element of the array:

```
(aset "new message." messages 20)
```

This statement assigns the string "new message" to the element of the array messages with index 20. Now let's change another element of the array:

```
(aset (aref messages 21) messages 20)
```

This fetches the value of the element of messages with index 20 and assigns that value to the element of messages with index 21. In other high-level languages this string assignment might be written as:

```
messages(21) = messages(20)
```

Now consider the following examples:

```
(setq laundry_lists (make_array 'list 12))
(setq in_trouble (make_array 'Boolean 100))
```

When PEEL executes these statements, it creates two arrays, a twelve-element array called laundry_lists, each of whose elements is a list, and a 100-element array called in_trouble, each of whose elements is Boolean. In a later chapter, we will discuss the list and Boolean data types, and describe how they are used.

Arrays With Multiple Names

You may use the setq function to bind a given array to more than one name. For example, given the array boxes previously defined, you could use the following statement:

```
(setq rectangles boxes)
```

This statement gives this same array a second name, rectangles.

It is important for you to understand that this statement is not an array assignment in the sense that array assignment is used in other high-level languages. In those languages, an array assignment is performed when there are two arrays, occupying separate storage areas, and the value of each element from the first array is assigned to the corresponding element in the second array.

In the example we are considering here, we are discussing a single array occupying one storage area. The setq statement above gives two different names to that single array. (More precisely, the statement creates a new atom, rectangles, whose value is the same as boxes in the sense that they both reference the same location in storage.) For example:

```
(aset 874 boxes 0)
```

This statement assigns the value 874 to the first element of the array boxes, and therefore also to the first element of the array rectangles. As a result, the statement (print (aref rectangles 0)) prints the value 874.

Releasing Storage Allocated to an Array

If you use make_array to create an array with a large number of elements, then a great deal of memory will be allocated by PEEL. For example:

```
(setq huge (make_array 'integer 20000))
```

This statement creates a 20,000 element array named huge. If you wish to release the storage occupied by that array, the easiest way is to assign a new value to huge. For example:

```
(setq huge 0)
```

This assigns the integer value 0 to huge, and releases the storage previously occupied by the array huge.

If you have created an array and given it multiple names, then you must reassign each of the multiple names in order to release the storage occupied by the array.

Other Array Functions

The array_dimension function: The array_dimension function takes one argument, which must be an array name, and returns the number of elements in that array, as defined in the make_array function. For example:

```
(setq boxes (make_array 'integer 5))
(print (array_dimension boxes))
```

These statements cause the value 5 to be printed.

The array_type function: The array_type function takes one argument, which must be the name of an array, and returns as an atom the data type of the array specified in the make_array function. For example, given the dimension of the array boxes above, consider the following statement:

```
(print (array_type boxes))
```

This would print the atom integer.

As a slightly more esoteric example, suppose you wish to create a new array with the same data type and size as an existing array. You could use the array_dimension and array_type functions in the make_array function, as follows:

```
(setq squares (make_array (array_type boxes)
    (array_dimension boxes)))
```

In this example, the first argument of make_array is the data type of the array boxes, and the second argument is the dimension size of the array boxes. As a result, a completely new array, squares, is created with the same data type and the same dimension size as the array boxes.

FUNCTIONS THAT MANIPULATE LISTS

Because PEEL is a variant of the LISP (list processing) language, and because lists are such an important part of the language, it is not surprising that PEEL has special functions that allow you to manipulate lists themselves. This section discusses functions that:

- Reference lists
- Form lists
- Pull apart lists

Referencing Lists

Most of the programming examples we have discussed manipulate either numbers or strings. Furthermore, the variables in most cases have had either integers or strings as their values. Now let us look at some situations where a variable has a list as its value.

As we have seen, you usually use the ' operator if you wish to assign a list to a variable. For example:

```
(setq listval '(a b c))
```

This statement assigns to listval the list shown. If you now use (print listval), PEEL prints out the value (a b c). Note that, in the setq statement, if the quote operator (') had been omitted, then PEEL would have attempted to evaluate the list (a b c) as a function call to the function a with arguments b and c, and would have assigned the result of that function call to listval. By including the quote operator, we indicate to PEEL that no such evaluation should take place. The result is that listval has as its value the list itself.

The Reverse and Length Functions: The reverse and length functions provide simple manipulations on values with the list data type. The length function takes one argument, which must be a list, and returns the number of elements in that list. Consider the following example, after listval has been defined as above:

```
(print (length listval))
```

The statement prints the value 3, the number of elements in the list which has been assigned to listval.

The reverse function takes one argument, which must be a list, and returns another list equal to the first list but with the elements reversed. For example:

```
(setq revlist (reverse listval))
```

This assigns to revlist the reverse of the list assigned to listval. The result is that revlist has as the value (c b a).

These functions may be applied to null lists, with more or less obvious results. For example:

```
(setq nullist '())
```

This statement assigns the null list to the variable nullist.

```
(length nullist)
```

This returns the value 0, the number of elements in the null list.

```
(reverse nullist)
```

This returns the null list.

As further illustration of these functions, let's consider the case where one or the elements of a list is itself another list. For example:

```
(setq nest '(a b (c d e) f))
```

This statement assigns to the variable nest a list containing four elements. The third of these elements is itself another list. When you apply the reverse or length functions to this list value, PEEL simply treats the third element as simply another element, no different from the other three elements that are atoms. Therefore, the function (length nest) returns the value 4, and the function (reverse nest) returns the value (f (c d e) b a). In both cases, the element (c d e) is treated the same as the other elements.

Forming Lists

If you have programmed in high-level languages with several data types, then you know there are special operations that are applied to variables of each data type. For example, the operations applied most often to variables with integer data types are addition, subtraction, multiplication, and division. The operations performed most commonly on string values are concatenation and substring. Concatenation puts strings together and substring pulls strings apart.

Similarly, there are special operations performed on lists. These operations can be thought of as similar to the operations performed on strings, in that they put lists together and pull lists apart. Let's start with the operations that form lists by putting smaller lists together.

The List Function: The list function takes one or more arguments and returns a list as follows:

- The number of elements in the returned list equals the number of arguments.

- Each element of the returned list equals the value of the corresponding argument.

Consider the following example:

```
(setq x 23)
(setq str "This is a test.")
(setq y (list x str))
```

In this example, the reference to the list function has two arguments. This reference returns a list containing two elements, each equal to the value of its respective argument. As a result, the variable y is assigned the following value:

```
(23 "This is a test.")
```

The Append Function: The append built-in function performs a more complex list operation. It takes one or more arguments, and each of its arguments must be a list. It returns a list value as follows:

- The number of elements in the returned list is equal to the sum of all the elements in all of the arguments.

- The elements of the returned list comprise all of the elements in all of the argument lists.

For example, consider the following:

```
(setq listval '(a b c))
(setq newval '(x y))
```

These statements assign list values to both the variables listval and newval. Suppose that once these assignments have been made, we execute the following statement:

```
(append listval newval)
```

This returns a list value containing five elements, (a b c x y). You can use the value returned by the append function in the way that you use other functions. For example:

```
(setq longlist (append listval newval))
```

This statement assigns to longlist this list containing five elements. The result is the same as if you had executed the following statement:

```
(setq longlist '(a b c x y))
```

The Cons Function: The cons function adds a new element to the front of an existing list. Consider, for example, the following:

```
(setq listval '(a b c ))
(setq list2 (cons 'd listval))
```

These statements assign to list2 the list obtained by adding the element d to the front of the value of listval. The result is that list2 equals (d a b c).

Pulling Apart Lists

The car and cdr functions can be used to pull lists apart. Suppose we define qv as follows

```
(setq qv '(a b c d e f))
```

We know that we can use the length function to determine how many elements there are in qv, but we do not yet have any functions to allow us to determine what those elements are.

The Car Function: The car function takes a single argument, a list, and returns as its value the first element in that list. Consider, for example:

```
(car qv)
```

Given the definition of qv above, this function reference returns the atom a.

By using car in conjunction with the reverse function, you can obtain the last element of a list. For example:

```
(car (reverse qv))
```

This statement returns the atom f. This results from reversing the list qv, to obtain (f e d c b a), and then applying car to that list to get the first element, which is the atom f.

The Cdr Function: The cdr function takes one argument, a list, and returns a list defined as follows:

- If the argument is the null list, then cdr returns the null list.
- If the argument is a list containing one or more elements, then the list returned contains all the elements of the argument list except the first element.

For example, given the definition of qv above, consider the following:

```
(cdr qv)
```

This function reference returns the list (b c d e f), which is obtained by removing the first element, a, from the argument list.

By using car and cdr in various combination, you can obtain any element in the list. For example:

```
(car (cdr qv))
```

This statement returns the second element of the list qv, b. This works because: first, (cdr qv) returns the list (b c d e f); and second, car applied to that list returns its first element, b, which is the second element of the original argument list.

Similarly, consider the following statement:

```
(car (cdr (cdr qv)))
```

This returns the third element, c, of the list qv.

By combining the car and cdr functions, which pull lists apart, with the functions previously described, which put lists together, you can perform very complex list operations. For example, given the list qv defined above, suppose you wish to form a new list that is the same as qv except that the second element is removed. You can do so by using the following expression:

```
(cons (car qv)(cdr (cdr qv)))
```

In this example, (car qv) returns a, and (cdr (cdr qv)) returns (c d e f). Applying cons to these two arguments yields the result (a c d e f), which is the original argument with the second element missing.

4

Logic and Looping

Although the subjects of the preceding chapters have been a necessary preliminary to our discussion, they really have not shown how to write programs. This chapter begins the discussion of PEEL statements that are necessary to write programs.

PEEL has great similarities to other programming languages in the area of program logic. Specifically, there are two categories of command statements: looping statements and decision statements.

PEEL offers two types of looping statements:

Statement	Definition
do_n_times	Repeat an action a specified number of times.
do_forever	Repeat an action until something happens that causes a stop_doing statement to be executed.

PEEL also offers three types of decision-making statements:

Statement	Definition
if ... else	Choose one action or another, based on a Boolean value.

select — Choose an action based on any one of a number of listed conditions.

dispatch — Choose an action based on the text at the current cursor.

These five statements will be discussed at some length in this chapter, together with Boolean and relational operators and the PEEL statements that return Boolean values.

As you will see, these control statements are very simple and straightforward. The only thing you should keep in mind is that the control structure does not contain a GO TO statement. Consequently, you should plan to write structured programs.

LOOPING

The two statements, or forms, that control looping are do_n_times and do_forever. (Recall that a form, defined in Chapter 3, consists of a number of s-expressions. The do_n_times form, for example, contains the function name do_n_times, an integer iteration count, and any number of statements or forms.) The do_n_times form tells EMACS to perform the statements the number of times indicated between the do_n_times and the parenthesis concluding the form. After the do_n_times count is exhausted, execution continues to the statement or form following the closing parenthesis.

The do_forever form tells PEEL to go into an infinite loop, executing each statement over and over again until something, we hope, tells it to stop. (The do_forever statement is analogous to a DO UNTIL in PL/I.) The statement that stops the looping is stop_doing.

You can also use the stop_doing statement in a do_n_times form. In this case the stop_doing acts as an early exit. The following three examples illustrate the do_n_times form.

Example 1:

```
(setq counter 0)
(setq accumulator 1)
(do_n_times 5
     (setq counter (1+ counter))
     (setq accumulator (* accumulator counter)))
(print accumulator)
```

This function computes five factorial. A more general function is:

Example 2:

```
(setq counter 0)
(setq accumulator 1)
(setq loop_counter (prompt_for_integer "Type a number" 1))
(do_n_times loop_counter
     (setq counter (1+ counter))
     (setq accumulator (* accumulator counter)))
(print accumulator)
```

The purpose of the prompt function is to print a message at the bottom of the screen, then return what is typed at the terminal as its value. The print string "Type a number" is followed by the default value of 1.

If you wanted to insure that the value of this function does not exceed 300,000, you would do the following:

Example 3:

```
(do_n_times loop_counter
     (if (> accumulator 300000)
         (stop_doing))
     (setq counter (1+ counter))
     (setq accumulator (* accumulator counter)))
```

While the _if_ statement will be discussed in greater depth later in this chapter, its use in PEEL is identical to its use in other programming languages. The _if_ statement in this example means that if the value of accumulator is greater than 300,000, EMACS should execute the stop_doing statement.

The following example illustrates the do_forever form.

Example 4:

```
(do_forever
     (if (line_is_blank)
         (delete_char)
     else
         (stop_doing)))
(do_forever
     (prev_line)
     (if (line_is_blank)
         (delete_char)
      else
         (stop_doing)))
```

These two do_forever forms delete blank lines that may surround point. The first form goes forward from point. Notice that the delete_char command brings the following line to point; consequently, no movement command is necessary. The second form requires a movement command because the delete_char always results in a nonblank line at point.

Looping With Arrays

Looping is commonly used with arrays, which were defined in the previous chapter. For example, consider the following array statement:

```
(setq boxes (make_array 'integer 5))
```

Execution of this statement creates an array called boxes with five individual integer values, with indices 0 through 4. We have already discussed how to set every element of an array to a value, as in the following statement:

```
(fill_array boxes 100)
```

The same result can be accomplished by using the do_n_times form, as in the following:

```
(setq counter 0)
(do_n_times 5)
    (aset 100 boxes counter)
    (setq counter (1+ counter)))
```

As a more complex example involving arrays and loops, consider the following:

```
(setq counter 0)
(aset 1 boxes 0)
(do_n_times 4
    (aset (* (aref boxes counter) 2) boxes (1+ counter))
    (setq counter (1+ counter)))
(print (aref boxes 4))
```

The second line of this example assigns the value 1 to the first element of the array boxes, the one with index 0. The next three lines comprise a loop that sets subsequent elements of boxes to twice the preceding element. The last line prints the value of the last element of boxes, the one with index 4. The value printed is 16.

DECISION STATEMENTS

The If Form

Although the if form was used before, it will now be discussed in detail.

The structure of the if form is:

```
(if (condition)
    (statements/forms)
 else
    (statements/forms))
```

Condition must ultimately have the effect of returning a true or false; that is, it must return a Boolean value. Only when condition is true does EMACS execute the statements immediately following the if. When condition is false, EMACS executes the statements following the else. Note that the else is not within its own set of parentheses. The reason is that the else is part of the if form.

Here is an example of a simple if form:

```
(if (= counter 3)
     (print "Counter equals 3")
 else
     (print "Counter does not equal 3")
```

As in other programming languages, the condition following the if can be complicated.

```
(setq yes_no (prompt "Do you want to continue"))
(if (| (= yes_no "YES") (= yes_no "yes"))
     (do_something)
 else
     (do_something_else))
```

In these statements, | means "or". This form (where do_something is not a real statement) says that if yes_no equals "YES" or "yes", it should perform the actions indicated by do_something. Otherwise, it should "do_something_else".

The if statements can test either user-defined conditions or PEEL statements. In particular, PEEL contains a number of statements that monitor the state of the current buffer.

Buffer Conditions

So far, you have seen very little interaction between information in a buffer and the extension language. In order to write meaningful extensions, EMACS must be able to give you information about a variety of things. Below is a list of what EMACS can tell you. Remember that "point" refers to the current cursor position.

Statement	Definition
at_white_char	Returns true if point is at a space.
beginning_of_buffer_p	Returns true if point is at the beginning of the buffer.
beginning_of_line_p	Returns true if point is at the beginning of a line.
empty_buffer_p	Returns true if the buffer is empty.
end_of_buffer_p	Returns true if point is at the end of the buffer.
end_of_line_p	Returns true if point is at the end of a line.
first_line_p	Returns true if point is anywhere on the first line of a buffer.
last_line_p	Returns true if point is anywhere on the last line of the buffer.
line_is_blank	Returns true if point is at a blank line. A blank line is a line which is either just a carriage return or spaces and a carriage return.
looking_at "string"	Returns true if <u>string</u> is immediately to the right of <u>point</u>.

Because (if (looking_at "string") ...) is very common, the two have been combined into the form:

```
(if_at "string" ....
 else ... )
```

Let's see how these statements are used. The following example shows how to find the end of a paragraph of text. For this example, an end_of_paragraph is defined as one of the following:

1. A line beginning with a period (that is, a RUNOFF command)

2. A blank line

3. The end of the buffer

The example checks for each condition.

```
(begin_line)
(next_line)
(do_forever
    (if_at "."
        (stop_doing))
    (if (line_is_blank)
        (stop_doing))
    (if (last_line_p)
        (move_bottom)
        (stop_doing))
    (next_line))
```

The only new things here are the begin_line and next_line statements. These, and other movement commands, are discussed in Chapter 8. However, if you are impatient to get started, you can find out what many of the movement commands are by typing {CTRL-_} C followed by the keystroke that invokes a movement. For example:

```
{CTRL-_} C {CTRL-A}
```

This tells you that the name of the function that goes to the beginning of a line is begin_line.

Action Commands

The previous section talked about PEEL statements that provide information about the placement of point. A second category of conditions indicates that an operation has been successful. You have already seen one example, namely:

```
(if (forward_search "the") ... )
```

This statement returns true if and only if EMACS finds a "the" between point and the end of the buffer. Therefore, any statements after the condition are executed only if a "the" is found.

In general, if a statement can fail, it will return a Boolean that indicates that the operation has failed. Appendix A lists what is returned for every command.

Boolean and Relational Operators

Throughout this chapter, you have seen examples of Boolean operators. The Boolean operators are:

Operator	Function
&	Performs logical "and" on its arguments. If all arguments are true, & returns true.
|	Performs logical "or" on its arguments. If one or more arguments are true, | returns true.
^	Negates (or inverts) a Boolean value. That is, if the argument is true, this function returns false. If the argument is false, this returns true.

For example, if an operation will be performed when a line is not blank, you would have the form:

```
(if (^ (line_is_blank)) (do_something))
```

The relational operators compare elements and report on their relationship. The PEEL relational operators are:

Operator	Function
<	Returns true if the first argument is less than the second argument.
<=	Returns true if the first argument is less than or equal to the second argument.
=	Returns true if the first argument is equal to the second argument.

>=	Returns true if the first argument is greater than or equal to the second argument.
>	Returns true if the first argument is greater than the second argument.
^=	Returns true if the first argument does not equal the second argument.

The Select Statement

When creating an extension, you often want the program to choose from a number of alternatives. One way to do this is by having a long series of nested if statements. For example:

```
(if (= counter 1) (do_something)
 else
    (if (= counter 2) (do_something_else)
     else
        (if (= counter 3) (do_a_third_thing)
     ...
         )))
```

Fairly soon, you are nested so far down that you do not know where you are. Moreover, keeping track of all the parentheses can be next to impossible.

The PEEL statement that gets you out of this bind is select. Using the above example, you might type:

```
(select counter
      1     (do_something)
      2     (do_something_else)
      3     (do_a_third_thing)
                 ...
 otherwise
            (do_what_you_have_to))
```

This form tells PEEL to choose an action based on the value of a variable (in this case, counter). If no action can be chosen because counter does not have a value specified in the list, do the action indicated by otherwise. The select statement, then, is just another incarnation of the standard programming language construct of a CASE statement.

The full structure of this form is:

```
(select expression
     constants-1 actions-1
     constants-2 actions-2
       ...         ...
 otherwise
     other-actions)
```

The structure augments the above example by showing that two other things are possible: 1) that the argument list can have multiple entries, and 2) that more than one statement can follow a choice.

This structure is a little hard to read (especially since the statement is fairly trivial). The following example better illustrates what the select statement does. If you have used the settab command (located in library EMACS*>EXTENSIONS>SOURCES>TAB1.EM), you will recall that you are told to type one of several characters. The following example, while modified slightly, is taken from that function:

```
(defun move ()
    (setq movement (prompt "Type a space, T, b, f, or q"))
    (select movement
           " "      (delete_char)
                    (insert " ")
           "t"
           "T"      (insert "T")
                    (delete_char)
           "f"
           "F"      (forward_char)
           "b"
           "B"      (if (^ (beginning_of_line_p))
                        (back_char))
           "q"
           "Q"      (return)
           otherwise
                    (info_message "unknown response")
                    (sleep_for_n_milliseconds 1000)))
```

As you can see, this is a very simple function. First of all, a prompt is displayed. The keystroke the user types is assigned to the variable called movement. If it is a space, the actions following the space are executed. Likewise, if the user types either f or F, the action indicated there is executed, and so on. Finally, if what is typed is not in the list, a message is displayed (using the info_message built-in function). So that the message is displayed long enough to be seen, EMACS is told to go to sleep for 1 second. In this function, return means exit from the routine.

The Dispatch Statement

In a similar manner, you can use the dispatch statement to have EMACS choose from a number of choices based on what text follows point. The structure of this form is:

```
(dispatch
     strings-1 actions-1
     strings-2 actions-2
        ...        ...
 otherwise
     other-actions)
```

In this form, string is the text that follows point. The way this form works is identical to the select form. The only difference is that there is no variable assignment phrase. Instead, EMACS checks to see if the text following point matches one of the strings listed in the strings list.

5
Writing Extensions

Previous chapters have given you numeric examples and functions that you can use to write EMACS extensions. This chapter pulls together all the underlying rules for writing EMACS extensions by describing the programming environment for doing so.

THE EMACS/PEEL ENVIRONMENT

Although PEEL is a high-level language, the method you use to write programs in PEEL is quite different from the method you use to write programs in most other languages.

When you write a program in other high-level languages, you usually use the following procedure:

- Use an editor (such as EMACS) to create an ASCII source file for the program you wish to execute.
- Execute the compiler for the high-level language to transform the source file into an object file.
- Execute the linking loader to produce an executable binary file.
- Execute the resulting executable file.

The PEEL environment is much more dynamic than the environment just described for other high-level languages. As you already know, you can create a PEEL source file in EMACS. You can then execute the program by typing {ESC} {ESC} pl without having to perform separate compilation or linking steps, and, in fact, without ever having to leave EMACS itself. The result is that the PEEL programs you write become a part of the EMACS environment, making it easy to extend the power of EMACS and tailor it to your specific needs.

Especially important to the PEEL language are the defcom and defun functions. When one of these functions is executed, it has the important side-effect of permanently adding a new command or function, respectively, to the EMACS environment you are using. The result is you can use the command or function that you have defined at any point after that in your EMACS session.

THE DEFCOM FUNCTION

You use the defcom function to define a new command. For example, in Chapter 2, you saw the following defcom example:

```
(defcom tx
  (do_n_times (numeric_argument 1)
    (forward_char)
    (forward_char)
    (forward_char)
    (self_insert \.)
))
```

When you execute this function, PEEL defines a new command called tx to perform the specified action. This definition of tx illustrates the simplest form of the defcom function, which is as follows:

```
(defcom name (action))
```

Before describing more complex uses of defcom, let's make it clear what the definition of tx in the above example does.

In the example, defcom defines tx to be a command that moves the current cursor (point) in your source file ahead three characters, and then inserts a period at the new position of point in your text buffer.

The defcom action also contains a loop specifying that the operation just described may be repeated several times, depending upon the value of the following:

```
(numeric_argument 1)
```

This function reference appears because, once the defcom has been executed, it is possible to invoke tx with a numeric argument. The numeric_argument function returns the value of that numeric argument. The "1" specifies the default value to be used in case tx was invoked with no numeric argument at all.

Inserting a Documentation Line

When you use {CTRL-_} to obtain information about a command, normally EMACS provides you with information just about system commands. When you add your own PEEL commands to the EMACS environment, you may also wish at the same time to make command descriptions available with the help facility. You can do this by means of the &doc option to defcom. For example, here is how you would change the definition of tx previously given:

```
(defcom tx
  &doc "Move cursor 3 chars, insert dot"
  (do_n_times (numeric_argument 1)
    (forward_char)
    (forward_char)
    (forward_char)
    (self_insert \.)
))
```

This example is exactly like the preceding one, except that the second line has been inserted. This line specifies the character string to be used as documentation for this command when it is requested.

If you now use {ESC} X pl, EMACS does two things: 1) it adds the tx command to its command environment, and 2) it adds the specified documentation to its help facility. You can see this for yourself by typing {CTRL-_} A, and then, in response to the "Apropos:" prompt, typing the words "insert dot". EMACS will respond by listing the tx command with the documentation line, "Move cursor 3 chars, insert dot", that you specified.

ARGUMENT HANDLING WITH DEFCOM

Let us now look at some variations of the tx command example we have already given. In these variations, we wish to focus on some of the different ways to handle arguments to a defcom command.

In fact, PEEL provides several different methods for handling arguments. We have already seen how to use the numeric_argument function, which returns as its value the value of the numeric argument supplied with the function. (Recall that to invoke the tx command with a numeric argument you use {ESC} n {ESC} X tx.) However, PEEL also provides other methods for handling the numeric argument, and some of these methods are more convenient in certain circumstances.

Using Numeric Argument as a Repeat Factor

In the tx command example we just gave, we used the numeric argument simply for the purpose of specifying how many times the action specified by the command was to be performed. There is a way of doing that automatically, as illustrated in the following example:

```
(defcom txb
  &doc "Move cursor 3 chars, insert dot"
  &na (&repeat)
    (forward_char)
    (forward_char)
    (forward_char)
    (self_insert \.)
)
```

When you execute this defcom, PEEL defines txb as a command that does exactly the same thing as tx, but specifies it in a slightly different way. The difference has to do with the following line:

```
&na (&repeat)
```

In this form, &na tells PEEL that it is to process a numeric argument to the txb command in a special way, and &repeat tells PEEL what that special way is: as a repeat factor. As a result of this line, PEEL uses any numeric argument you specify with the txb command as a repeat character, and it repeats the body of the command that number of times.

Notice that when you use this method to specify the handling of a numeric argument, you need not specify any default value, because the default value is always 1.

Passing the Numeric Argument to a Variable

You may also specify that PEEL pass the numeric argument to a variable whose name you specify. For example, suppose we redefine the tx command again as follows:

```
(defcom txc
  &doc "Move cursor <arg> chars, insert dot"
  &na (&pass curmov &default 1)
    (do_n_times curmov (forward_char)
    (self_insert \.)
)
```

This command works a little differently from the previous examples. Notice that the third line of the definition is:

```
&na (&pass curmov &default 1)
```

This line specifies that when there is a numeric argument with the txc command, PEEL is to pass the value of that numeric argument to the variable curmov, which you have specified. Moreover, if no numeric argument is provided when txc is invoked, the default value 1 is to be assigned to curmov.

Once you have assigned the value of the numeric argument to a variable, you can of course use that variable in any way you like. In the definition of txc above, we have used it to specify the number of times that the forward_char function is executed before the dot is inserted. Therefore, the txc command works a little differently than either txb or tx, because the cursor is not automatically moved three times.

Specification for &na

Any defcom definition can contain a line beginning with &na to specify how a numeric argument is to be handled by the command. The following are the three possible formats for use of &na:

```
&na (&repeat)
&na (&pass name &default value)
&na (&ignore)
```

The first format, using &repeat, tells PEEL that a numeric argument specifies the number of times that action is to be performed. If no argument is specified when the command it invoked, then the action is performed once.

The second format, using &pass, specifies that a numeric argument is to be assigned to the variable with the specified name, and that if no numeric argument is given, then the specified value is to be assigned to the variable name.

Note

The variable named in the &pass option is treated as a "local variable" by PEEL. This means any value assigned to this variable is valid only within the action performed by that command. If you have elsewhere used a variable with the same name, then that variable is unaffected. We will discuss local variables in greater detail later in this chapter.

The third format, using &ignore, specifies that any numeric argument supplied with the defcom command is to be ignored.

Prompting for an Argument Value

All the methods for handling numeric arguments that we have described so far use the {ESC} n convention for supplying the numeric argument when the command is invoked. Following is a totally separate method for supplying a numeric argument to a command.

When the command is invoked, the command prompts the user for the value of the information needed, and allows the user to type whatever value is desired. To understand how this works, suppose we rewrite the definition of txc as follows:

```
(defcom txd
  &doc "Move cursor specified # chars, insert dot"
  &args (
    (curmov &prompt "Type amount of cursor movement"
            &default 3 &integer)
        )
   (do_n_times curmov (forward_char)
    (self_insert \.)
)
```

Notice that lines three through six of this definition contain an option beginning with &args. This option specifies a great deal of information, as follows:

- &args specifies that when the command is invoked it will, as part of its action, request information to be typed at the terminal.

- curmov is the name of a variable into which PEEL stores whatever value is typed at the terminal when the command is invoked. You may use your own choice for the variable name. (As with the &pass option of &na, the variable specified here is a local variable.)
- &prompt and the following text specify the text that PEEL will display when the command is invoked in order to prompt the user for the desired information.
- &default 3 specifies that after the prompt is displayed, if the user types {RETURN} without typing a number, then a default value of 3 will be assigned to the variable curmov.
- &integer specifies that the value to be typed in response to the prompt must be an integer value.

When the command txd is invoked, PEEL displays the following line at the bottom of your screen:

```
Type amount of cursor movement:
```

The user may then type an integer value that PEEL receives and assigns to the variable curmov. Then, the txd function uses this value to determine the number of times to invoke the forward_char function.

Specifications for &args

As you can see, &args allows you to specify an argument by an entirely different method from &na. You may use both &na and &args in the same defcom definition. Although a command defined by defcom may use only one numeric argument of the type described by &na, it may use any number of arguments with &args. Furthermore, the latter arguments may be integer or string data types.

The format of &args is as follows:

```
&args ( (spec1) (spec2)...)
```

That is, &args is followed by a parenthesized list of one or more specifications, and each of those specifications is itself enclosed in parentheses. The format of a parenthesized specification is as follows:

```
(name &prompt string
              &default value
              type)
```

This specification contains the following information:

- Name is the name of a variable to which the value typed at the terminal is to be assigned.

- String is the text of the prompt string that PEEL will use to prompt for the argument value.

- Value is the default value to be assigned to the variable in case the user responds to the prompt by hitting {RETURN} without typing any value.

- Type is the data type of the value that the user must type at the terminal in response to the prompt. Note that the default value specified by value must have a corresponding data type. The data type must be one of the following: &integer, &string, or &symbol; the corresponding data types for value are integer, string, and atom, respectively.

FORMAT OF DEFCOM

The full format of the defcom function is:

```
(defcom command_name
    &doc  documentation_string
    &na   (&repeat)
          (&pass name [&default value])
          (&ignore)
    &args ((name &prompt string
                 &default value
                 &string | &symbol | &integer )
                  ... )
    &chararg
    body
    ... )
```

The elements have the following meaning:

defcom — Is the name of the special function that defines a command, establishing the command's name, its documentation sequence, and argument acceptance mode.

&doc — Documents what the command does. The text of the documentation string will appear in such help facility texts as apropos and explain_key.

&na — Tells EMACS that this defcom accepts a numeric argument for the command. It also specifies how the argument will be handled. The following are keywords used in the &na clause:

- &repeat — Tells EMACS how many times it should repeat the body of the defcom code.
- &pass — Tells EMACS the numeric argument is to be transmitted to a named variable. If the &default option is specified, it gives the default value to be used if no argument is typed when the command is invoked.
- &ignore — Tells defcom to ignore numeric arguments.

&args — Declares a name that will receive text typed at the terminal. The following are keywords used in the &args clause:

- &prompt — Displays the given string in the minibuffer. The reply to the prompt becomes the value of the argument.
- &string — Says that the argument will be a string data type.
- &symbol — Says that the argument will be a symbol data type.
- &integer — Says that the argument will be an integer data type.
- &default — Defines the default value for an argument. This value will be used if a carriage return is typed in response to &prompt.

&chararg — Tells EMACS to save the character (the keypath) used to invoke a command. (This is needed for macros that wish to use the command argument for further processing.)

Note

As currently implemented, PEEL does not require that &chararg be used. The character_argument function can obtain the keypath information regardless of whether &chararg was specified or not.

Further Examples

Here are a few defcom examples that show how its syntax is put together.

Example 1:

```
(defcom mark_end_of_word
    &doc "places a mark at the end of the current word"
    (mark)
    (forward_word)
    (exchange_mark))
```

This simple macro executes three statements that place a mark at the end of the current word, and then return point to where it was before the command was invoked.

Example 2:

```
(defcom set_right_margin
    &doc "Sets the right margin for word wrapping"
    &args ((foobar &prompt "Type right margin value"
                   &default 70
                   &integer))
    (setq fill_column foobar))
```

The variable fill_column is a global variable used by the EMACS word wrapping routines. EMACS uses this variable to determine what the maximum right margin should be. Thus, you can affect how EMACS does word wrapping simply by changing the value of this variable. The set_right_margin command can do that easily.

In this command definition, &args defines an argument called foobar that is used in the command as an intermediate variable that is eventually assigned to fill_column. This works as follows:

- The value of foobar is the number typed in response to the prompt.
- If a carriage return is typed, the default value is 70.

- foobar is an integer variable.

Once the value of foobar has been established, the defcom can do its "work" by setting the variable fill_column to the value stored in foobar.

If a user types a reply that is nonnumeric, EMACS will print an error message indicating that a conversion error has taken place.

Example 3:

```
(defcom backward_para
     &doc "Move backward a paragraph"
     &na (&pass count &default 1)
     (if (> count 0)
          (backward_paraf count)
      else
          (forward_paraf (- count))))
```

This command is used to call one of two functions. The &na accepts a numeric argument, if one is typed, and assigns it to a variable called count. If no argument is typed, the default value is 1. If the value of count is positive, the backward_paraf function is called. However, if count is negative (meaning that the user wants to go in the opposite direction) forward_paraf is called.

THE DEFUN FUNCTION

Just as the defcom function can be used to define commands that are invoked from the keyboard, the defun function can be used to define functions that can be invoked from other PEEL programs. In other programming languages, similar capabilities are provided by functions, subroutines, and procedures.

You have actually been using functions all along in your PEEL programs. For example:

```
(forward_char 5)
```

This function invokes the forward_char function with the argument 5. The function moves the cursor forward five characters. As another example:

```
(+ 2 3)
```

This function invokes the + function with arguments 2 and 3. This function returns the value, 5, computed by adding together all the arguments.

You will learn how to define your own functions. In order to define a function you must specify several pieces of information, including the following:

- The name of the function. In the examples we have just been considering, the names of the functions were forward_char and +.

- Information about the arguments to the function. For example, you can specify the number of arguments, and whether some of the arguments are optional. (Recall that the + function can have from one to eight arguments. This means that the first argument is required, and the other seven arguments are optional. Also, recall that the single argument to the forward_char function is also optional, with a default value of 1 if you do not specify it.)

  In addition, you may specify what the data type of each of the arguments is to be. For example, you can specify that the argument must be an integer, or must be a string, or may be of any PEEL data type.

- What action the function is to take. For example, you can specify that when the function is invoked, it is to manipulate the text in your buffer in certain ways, or it is to prompt you for certain values, or it is to make certain computations. In fact, you can use any other PEEL functions to specify the action to be taken, even functions that you yourself have defined in other defun functions.

- What value, if any, the function is to return. If you specify no return value, then the function automatically returns a null list.

Format of a Simple Defun

The simplest format of a defun is as follows:

```
(defun name (argument_list)
   action
    )
```

This format consists of the following elements:

- Defun is the name of the function that defines a user-defined function. When the defun is executed, the new function becomes defined.

- Name is the name of the function being defined.
- Argument_list is the specification for the arguments and variables to be used with the function. This specification will be described in greater detail below.
- Action specifies what action the new function is to take when it is invoked.

Let us look at some examples.

Function Without Arguments

Consider the following defcom example:

```
(defun up_char ()
  (mark)
  (forward_char)
  (uppercase_region))
```

This defun defines a new function, called up_char. The purpose of this function is to convert the character in your buffer at point to uppercase. As you can see from the definition, the name of the function is up_char, there are no argument specifications, and the action proceeds as follows:

```
(mark)
```

This marks the current position in your text buffer.

```
(forward_char)
```

This moves the point ahead one character, thus defining a region consisting of that single character.

```
(uppercase_region)
```

This converts that single character to uppercase.

Function With a Single Argument

In order to use defun, you define a function that has one argument, using the following format:

```
(defun name ((argname type))
   action
   )
```

In this format, argname is the name of the argument that you are defining, and type is the data type of the argument, usually integer or string.

For example, we can increase the usefulness of the up_char function previously defined by letting it take an argument equal to the number of characters to be converted to uppercase. Consider the following:

```
; up_chars converts the specified # chars to upper case
(defun up_chars ((count integer))
    (mark)
    (forward_char count)
    (uppercase_region))
```

In this example, the function up_chars is defined as having a single argument, called count, which is of the integer data type. The action for up_chars is the same as the action for the previously defined function up_char, except for the following line:

```
(forward_char count)
```

This line moves the point ahead by the number of characters specified in the argument, rather than just one character. The result is that the marked region becomes the number of characters specified by the argument, and that entire region is converted to uppercase. For example:

```
(up_chars 8)
```

This invocation of the function defines a region consisting of the eight characters following point and converts them to uppercase.

Functions With Several Arguments

The following is the format of defun for a function with two or more arguments:

```
(defun name
   ((argname1 type1) (argname2 type2)...)
  action
  )
```

As you can see, you simply specify the names of each of the arguments, and the corresponding data types.

Function With a Single Optional Argument

We have previously defined the up_chars function as a function that takes a single integer argument, and converts that number of characters to uppercase. Let us now change that example so that the argument is optional and, if not specified, defaults to 1.

The format ot the defun that defines such a function is as follows:

```
(defun name (&optional (argname type))
   action
   )
```

The only difference between this format and the format with a single required argument is the insertion of &optional at the beginning of the argument list.

Following this format, here is the new definition of the up_chars function:

```
(defun up_chars (&optional (count integer))
    (mark)
    (if (null count) (setq count 1))
    (forward_char count)
    (uppercase_region))
```

The first line of this definition now specifies that the single argument, count, is optional. Notice, however, there is an extra line in the action portion of the definition:

```
(if (null count) (setq count 1))
```

This PEEL statement specifies what is to happen if the function up_chars is invoked with no argument. In such a case, PEEL gives the argument variable count a value equal to the null list; namely that if count equals the null list, it should be set to the integer value 1.

Functions With Some Required and Some Optional Arguments

The format of the defun which defines a function with several arguments, some of which are optional, is the same as defun with several required arguments, with the following exception: you insert the key word &optional between the last required argument and the first optional argument in the last argument list. Note that, as you would expect, PEEL requires that all optional arguments follow all required arguments in the argument list.

Local Variables

Suppose you define a function which contains the following as part of its action specification:

```
(setq direction -1)
```

This is an assignment statement that assigns the integer value -1 to the PEEL variable direction. You can then use that variable in other statements in your function definition. In this respect, PEEL is no different from other high-level languages where values are assigned to variables.

The problem is the following: suppose you are using many PEEL functions and, by coincidence, one of those other functions also uses a variable named direction. In that situation, you may have a serious but subtle programming bug, because one function would be changing the value of a variable used by another function.

The solution to this problem is to use local variables. When you define a local variable in a function, you are specifying: 1) that local variable may be used only within the function being defined; and 2) if a variable by the same name is used in a different PEEL function, PEEL is to treat that as a completely different variable, just as if it had a different name.

The format of a defun function definition with local variables is as follows.

```
(defun name ( args
          &local (varl typl) (var2 type2)...)
   action
   )
```

The elements of this definition are as follows:

- defun and name are, respectively, the function defining keyword and the name of the function you are defining, as previously described.

- args is the argument list as previously described. There may be no arguments or one or more arguments, and the keyword &optional may appear to specify that the arguments that follow are optional.

- &local is the keyword specifying that the variable names that follow are local variables, not arguments.

- (varl typel) is the name of the first or only local variable and its data type.

- (var2 typ2)... represents other local variables, if any, and their data types.

- action is as previously described.

A function definition in that format may use any of the variables defined as local variables in the &local clause without fear of disturbing the values of variables in other functions that happen to have the same names.

For example, here is a fairly sophisticated example of a function that centers the line on which point lies:

```
(defun center_linef (&optional (count integer)
                     &local (line string)
                            (direction integer))
    (if (< count 0)
        (setq direction -1)
        (setq count (- count))
     else
        (setq direction 1))
    (do_n_times count
         (setq line (current_line))
         (begin_line)
         (kill_line)
         (whitespace_to_hpos (/ (- 80 (string_length line)) 2))
         (insert line)
         (if (< direction 0)
             (prev_line)
          else
             (next_line))))
```

Since this example contains a number of new concepts and functions, let's go through it step by step and see how it works:

- The function, which is named center-linef, takes one optional argument, called count, and has two local variables, called line and direction.

  The purpose of this function is to center one or more text lines on the page. The optional argument count specifies the number of consecutive lines to be centered. The argument count may be either positive or negative, to indicate that lines following or preceding point, respectively, are to be centered.

  line is a local variable with the string data type, and direction is a local variable with the integer data type. As we shall see, the variable line will be assigned the text of the line in the text buffer being centered, and the variable direction will be used as a flag to indicate whether the argument count is positive or negative.

- The if clause tests whether count is positive or negative, and sets the variable direction accordingly. In addition, if count is negative, then it is reassigned so that it is positive.

- The do_n_times function defines a loop to be executed once for each line being centered, as indicated by the value of count.

- The statement (setq line (current_line)) uses the local variable line. The function current_line is a standard PEEL function that returns a string value equal to the text in the line on which point lies. Therefore, the setq statement assigns the text of the current line to the local string variable line.

- The next two statements move point to the beginning of the line and delete that line.

- The next statement is more complicated than the previous ones, because it uses several different functions in a nested form. Let us look at it one step at a time.

  The function (string_length line) uses the standard PEEL function string_length to determine the number of characters in the line being centered.

Next, the - function is used to subtract this number of characters from 80. Note that we are assuming that you wish to center your line in a field 80 characters wide. If this assumption is incorrect, you would have to use a different number or expression here.

Next, the / function divides the quantity computed so far by 2. This equals the number of characters of white space that will appear on either side of the centered line.

Finally, the whitespace_to_hpos function, which takes as its argument the quotient from the division by 2, inserts blank characters so that point is positioned precisely to center the reinserted text.

- Next, (insert line) reinserts the text (stored in the string variable line) of the line being centered.
- Finally, the if clause moves point either to the previous line or to the next line, depending upon whether the original value of the argument count was negative or positive.

Note that this example illustrates the following interesting point about local variables: from the point of view of the action taken by the function, there is no real difference between an argument and a variable. Either can be assigned a value, and either can be used in any statement in the action specified for the function. The only difference is the obvious one: an argument may be assigned an initial value by the argument list when the function is invoked, while a local variable cannot.

Functions That Return Values

All the functions that we have defined so far perform some sort of action but do not return a value. (In LISP terminology, this means the functions have a side-effect but no effect. More strictly speaking, the effect, or value, of each of these functions is a null list.)

Now let's see how you define a function that returns a value you specify. The format is as follows:

```
(defun name (...
          &returns type ...)
   action
   return value))
```

The explanation of this format is as follows:

- Defun name is as previously described.

- The list in parentheses following the name contains different types of information, such as specifications for arguments and local variables, as we have already seen. Now we are adding a new piece of information in the form:

  ```
  &returns type
  ```

  This specifies the data type that this function will return. In place of type, you should specify integer or string or other data type that the function might return.

- The function action concludes as follows:

  ```
  (return value))
  ```

  When the function you are defining is invoked, the return function will terminate the action and return, as the value of the function, the value you specified as the argument to return. The value may be a constant or expression, but its data type must be the same as the data type you specified in the &returns option.

As an example, let's define a function that counts the number of spaces to the left and to the right of point, and returns that count as the value of the function. Here is the definition of the function.

```
; count_spaces takes no arguments and returns a count of
; the number of spaces around point
(defun count_spaces
    (&returns integer  &local (count integer))
    (save_excursion
    (do_forever          ; move back over spaces
        (if (looked_at " ") (back_char)
         else (stop_doing)))
    (setq count 0)       ; init space counter
    (do_forever          ; move forward over spaces
        (if (looking_at " ")
         (forward_char) (setq count (1+ count))
         else (stop_doing))))
    (return count))
```

This example defines a function called count_spaces. This function works as follows:

- The list in parentheses following the function name count_spaces indicates that the function takes no arguments, returns an integer value, and uses one local integer variable called count.

- In many of the examples we have seen, the action specified by the function moved point in the text buffer and left it in an unpredictable place. Often this is inconvenient, and you would prefer that a function leave point where it was when the function was first invoked. One method of doing this is to use the following:

  ```
  (save_excursion action)
  ```

  When you use the save_excursion function, as we have done in count_spaces, PEEL remembers where the point is, performs the action you specify, and then returns the point to wherever it was before the action occurred.

- The purpose of the first do_forever loop is to move the point to the left until a character other than a space is found. The (looked_at " ") function determines whether the character preceding the point is a space or not, and returns a Boolean value to indicate that. The (back_char) function moves the point back one character if that character is a blank, and the (stop_doing) function terminates the do_forever loop if the character was not a blank.

- The first setq initializes the counter variable count to 0. When we are finished, count will contain the number of spaces we are counting.

- The next do_forever loop moves the point forward, counting spaces, until a character other than a space is found. The (looking_at " ") function returns a truth value indicating whether or not the character at point is a blank. If it is a blank, then (forward_char) moves the point one character ahead, and (setq count (1+ count)) increases the counter by one to indicate we have passed over one space.

- Finally, (return count) terminates the function invocation, returning the value of count to the caller.

Another Example of &returns

Consider the following example:

```
(defun move (&local (movement string)
             &returns Boolean)
   (setq movement (prompt "Type a space, t, b, f, or q"))
   (select movement
           " "     (delete_char)
                   (insert " ")
                   (return true)
           "t"
           "T"     (insert "T")
                   (delete_char)
                   (return true)
           "f"
           "F"     (forward_char)
                   (return true)
           "b"
           "B"     (if (^ (beginning_of_line_p))
                       (back_char)
                   (return true)
           "q"
           "Q"     (return false)
           otherwise
                   (info_message "unknown response")
                   (sleep_for_n_milliseconds 1000)
                   (return true)))
```

This defun defines a function called move that moves point or modifies the text buffer depending upon the character typed in at a prompt. This example uses some of the interactive features of PEEL that we will be studying in a later chapter. The function works as follows:

- The list in parentheses following the function name indicates that the function has no arguments, uses one local string variable called movement, and returns a Boolean value.
- The statement (prompt "type a space, t, b, f, or q") types the specified character string in the minibuffer at the bottom of your display screen, and then waits for your to type a character in response.
- The setq statement assigns the character typed in response to the prompt to the string variable called movement.
- The select statement chooses an action to be performed, based on the value of the variable movement.

- If the character typed was a space, the function replaces the character at point with a space. It does this by deleting the character at point and returning a space. The function returns a true Boolean value to indicate the operation is successful.
- If the character typed was "t" or "T", then the function replaces the character at point with "T".
- For "f" or "F", the function moves point ahead one character. In case of "b" or "B", the function moves point back one character, although it leaves point unchanged if point is already on the first character of the line.
- For "q" or "Q", the function returns a false Boolean value.
- Finally, in the case of any other input value, the function displays the message "unknown response" in your minibuffer, leaves the display for 1,000 milliseconds (one second), and then returns with a true Boolean value.

As you can see, the only case where this function returns a false Boolean value is when, in response to the prompt, the user types "q" or "Q" for quit.

How would you use a function like move that we have just described? You would probably want to call it over and over again until q is typed. For example, consider the following:

```
(do_forever
    (if (^ (move)) (stop_doing)))
```

As you can see, this form calls the move routine in an infinite loop, and terminates only when q is typed.

FORMAT OF DEFUN

The full format of the defun function is as follows:

```
(defun name ((argument1 type1) ...
             &optional ...
             &rest ...
             &quote ...
             &eval ...
             &returns type
             &local (variable1 type2) ... )
      statements_of_defun_program
        ...   )
```

The words in the above structure have the following meaning:

defun	Builds a function with the specified name using the given argument list and body.
argument	Declares the name to be used for a parameter passed to this defun from the defun or defcom that calls it.
type	Declares the data type of argument. Data types are discussed later in this chapter.
&optional	Says that all arguments past this point are not required. If they are not there, they are set to NIL.
&rest	Tells EMACS that it should take the rest of the arguments and put them into a list, as for example: &rest (r list)
&quote	Tells EMACS that it should only bind all following atoms, not evaluate them. To resume evaluation of atoms, use the &eval argument.
&eval	Shuts off the &quote argument used in defuns so that all following arguments are evaluated.
&returns	Specifies the data type of the information returned to the calling routine. (This is explained in depth later in this chapter.)
&local	Says there are no further arguments in a defun argument list, and that the remaining items of information are variables that will have only a scope of the current function.

COMBINING DEFCOM COMMANDS WITH DEFUN FUNCTIONS

We have now seen how to use defcom to define a command and defun to define a function. It is common practice for PEEL programmers to set up the defcom definition so that it does no work other than to call a function defined by defun. This is very convenient because it puts all the command logic into a function that can then also be called from other places, if desired.

For example, here is a command that indents one line of your text buffer the same as your previous line.

```
(defcom indent_relative
    &doc "Lines up text or tabs text over more"
    &na (&pass count &default 0)
    (indent_relativef count))
(defun indent_relativef ((count integer)
                         &local (indentation integer))
    (prev_line)
    (begin_line)
    (skip_over_white)                   ; Finds column indent of
    (setq indentation (cur_hpos))       ;   previous line, saves it
    (next_line)                         ; Goes to beginning of next
    (begin_line)                        ;    line
    (white_delete)                      ; Deletes space, if there
    (whitespace_to_hpos indentation)    ; Indents line
    (if (< count 0)                     ; Only positive values for
        (setq count (- count)))         ;    count are used
    (if (> count 1)                     ; Indents the number of
        (do_n_times (1- count)          ;    additional tabs if
            (type_tab))))               ;    there is an argument
```

As you can see, the defcom defines a function called indent_relative. The action specified for the command does nothing more than pass the numeric argument to a function called indent_relativef. The &doc line of the command provides user documentation containing text for the help facilities apropos and explain_key. The &na line specifies that the numeric argument, whose default value is 0, is to be assigned to the variable count. The last line invokes the function indent_relativef with the argument count.

The defun defines a function called indent_relativef that works as follows:

- The list in parentheses following the function name indicates that the function has one integer argument called count and one integer local variable called indentation.

- When the function is invoked, the prev_line and begin_line functions move point to the beginning of the preceding line.

- The skip_over_white function moves point to the first nonblank character on that line. This is the method used to determine how far the preceding line is indented.

- The cur_hpos function returns an integer value equal to the horizontal position of point. In effect, this counts the number of leading spaces on that line. The setq statement assigns that value to the variable indentation.

- The next_line and begin_line functions move point to the beginning of the next line, the line on which point originally lay when the function indent_relativef was invoked.

- The white_delete function deletes any leading spaces that already appear at the beginning of that line. The whitespace_to_hpos function inserts as many blanks as are specified by the variable indentation. Since indentation equals the number of blanks on the preceding line, this statement aligns the current line with the preceding line.

- The if statement sets count equal to its absolute value.

- The last if statement indents by additional tab amounts if count is greater than one.

In the above example, notice the extensive use of the semicolon (;). This character is used to delimit comments. It tells EMACS to disregard any text from it to the end of the line. You may use comments anywhere they are needed.

PEEL DATA TYPES

In this and preceding chapters, we have illustrated some of the more commonly used PEEL data types. The ones we have illustrated that are most like data types in other high-level languages are integer and string data types. However, we also have discussed data types that are unique to PEEL, such as the list data type.

Now let us summarize the PEEL data types.

The following five data types exist in some programming languages:

integer	Can contain a positive or negative whole number
string	Can contain a string; that is, a collection of characters
character	Can contain one character
Boolean	Can contain a true or false value
array	Can contain multiple occurrences of a data type

The following three data types are used in LISP-style programming.

atom	Can contain the basic structural unit of a program
list	Defines a variable that will contain a list
function	Defines a variable that can contain a function

The following two data types are unique to PEEL.

cursor	Can contain a value that indicates a place in a buffer
dispatch_table	Can contain an array-like variable that holds entries that tell EMACS what functions it should call when a keystroke invokes a function

The data type "any" can have any one of the above data types assigned to it.

Finally, handler and window are internal data types used by EMACS. A handler is a special kind of function, while a window is the part of a screen used to display a buffer. For example, you have two windows when you split the screen.

Because many PEEL statements only operate on certain data types, it is sometimes necessary to check to see what kind of data is being acted upon. The way you check a variable's data type is with the typef function. For example:

```
(typef variable)
```

This statement returns a number between 1 and 14, as follows:

1	any	8	list
2	Boolean	9	cursor
3	character	11	dispatch_table
4	integer	12	handler
5	string	14	window
6	atom	15	array
7	function		

All of these have symbolic definitions to aid comparisons. For example, Type.integer is an atom that has a value of 4 while Type.function has a value of 7. This means that you can use mnemonics when writing code. For example:

```
(if (= (typef atom) Type.integer) ...
```

Notice that the word "Type" begins with a capital letter.

GLOBAL AND LOCAL VARIABLES

Generally speaking, a global variable is one whose value can be referenced in and changed by any PEEL command or function, while a local variable is one whose value can be referenced in or changed by only the command or function in which the variable is used. In this section, we will examine how to use global and local variables in PEEL programs.

Global Variables

When a variable always has meaning, it is referred to as a global variable. Here are two examples used in the current EMACS libraries:

token_chars	Globally establishes a set of 63 characters those forming valid tokens. (The characters are A-Z, a-z, 0-9, and the underscore character.)
whitespace	Globally establishes " " (the space character) as the definition of blank space. Sometimes whitespace is changed to include newline or other characters.

These variables allow values to be used in a variety of contexts. For example, token_chars is used by all functions that act upon words. For the purposes of forward_word, or delete_word, a word is a token, and a token is any string made solely of token characters. The variable whitespace is used by such functions as delete_white_left and skip_over_white.

The value of a global variable remains after a function has finished executing. Thus, whitespace always remains available to every function that needs it. If, however, one function changes the value of whitespace, all other functions using it thereafter will use the changed value.

How to Set Global Variables: A global variable is established by assigning a value to a variable. For example:

```
(setq rightmost_column 70)
```

This establishes a variable called rightmost_column and equates the name rightmost_column to the value 70. Stated in another way, all that you do is make up a name and assign something to it.

Limitations of Global Variables: Many books on structured design and structured analysis decry the use of common blocks and common storage because they contain things that are, in effect, global variables. The reason they are not too well liked is as follows:

> Suppose you have three subroutines, called A, B, and C, and A calls B then C. Assume B modifies a global variable in anticipation for some action of C. Obviously, this will work fine. However, what happens if a new routine, D, is written and it modifies the same value and is called between B and C? This means that the application programmer must study B and C carefully, learning how they share global variables, to avoid accidentally interfering with any of them. If subroutines B and C had been constructed to pass information only through argument lists, avoiding global variables, the application programmer would not need to be concerned about global variables at all. Stated in a different way, global variables mean that routines have less control over their data and that they cannot be considered "black boxes".

Sometimes, as with token_chars and whitespace, the variable pertains to the entire environment. In this case, and probably only in this case, a global variable should be used.

Local Variables

In most cases, it is convenient to define variables in which you can store values for a limited time. By this we mean variables that are used only in the command or function you are defining, and that do not affect variables in other commands or functions, even when those variables have the same names.

Such variables are called local variables, and we have discussed them earlier in this chapter. We have identified three ways to define local variables, all illustrated earlier in this chapter:

- In a defcom, the variable specified with the &pass option of the &na clause is local to the defcom in which the clause appears.
- In a defun, all the arguments to the function you are defining are local variables.
- Also in a defun, local variables may be explicitly specified by means of the &local option.

If your command or function uses a variable that is not specified as local by any of the means just described, then PEEL considers it to be a global variable, and you may have conflict with a global variable of the same name used in a different command or function.

Properties of a Variable

Any PEEL variable has three properties:

- The name of the variable
- The value of the variable
- The attributes of the variable

There are two kinds of attributes:

- The data type of the variable, which indicates what type of value (integer, string, list, any, and so forth) can be stored as the value of the variable
- The scope of the variable, the portion of your PEEL program in which you may reference or change the value of the variable.

In the remainder of this chapter, we will discuss saving and reinstating the properties of a variable. Keep in mind that we will be discussing all of the properties just described.

Scope of a Variable

As we have noted, the scope of a variable is that portion of your PEEL program in which you may reference or change the value of the variable. There are two basic kinds of scope, local and global.

If a variable is specified local to a defcom command or a defun function, using one of the methods described above, then the scope of the variable is local. In addition, the value of the variable may be referenced or changed only by the statements within the command or function in which the specification occurs. If a local or global variable with the same name is used in a different PEEL command or function, then it is treated as a completely different variable (with a different value and attributes) just as if it had a different name.

If a variable is used without any specification that it is local, then PEEL considers it to be a global variable. In this case, the scope of the variable is your entire PEEL program, with the following exception: any command or function that specifies a local variable with the same name as the global variable cannot reference the value of the global variable. Therefore, the statements of that command or function are not included in the scope of the global variable.

Separation of Functions

In some procedural languages, like PL/I, it is possible for one procedure to be imbedded inside another procedure. This usually means that the imbedded procedure "inherits" the local variables of the procedure in which it is imbedded.

This is not the case in PEEL. All functions and commands are completely external to one another, and it is not possible for one function or command to be imbedded in another function or command. Therefore, no inheriting of variables ever takes place in PEEL.

Allocation and Freeing of Local Variables

Whenever a command or function is invoked during execution of your PEEL program, PEEL automatically allocates space for any local variables specified within the defcom or defun for that command or function. This space holds all of the function's properties, including its name, its value, and its attributes. During execution of that command or function, whenever that variable name is referenced, it always refers to the local variable whose space has just been allocated.

When the command or function has completed, any space allocated for the name, value, and attributes of local variables is freed. In particular, any value assigned to local variables is lost permanently.

RECURSION

Many high-level languages permit you to define recursive functions and procedures, and PEEL is no exception. Let us see how you can use them.

Recursive Definition of Factorial

In a previous chapter we showed you how to define a simple PEEL program that computes the factorial function. Now we are going to show a new PEEL function that also computes factorial, but which does so in a recursive manner. This example will therefore allow us to illustrate recursive functions in PEEL. The new PEEL function is based on the so called recursive definition of the factorial function. This definition is as follows:

```
0! = 1
if n>0, then n! = n * (n-1)!
```

This is a two part definition. The first line gives you the value of 0 factorial, namely 1. The second line gives you a rule for computing your factorial where n is greater than 1: namely, it tells you to multiply n by the value of (n-1) factorial.

If you have never seen this definition before, it may appear to be a circular definition, that is, that it defines factorial in terms of factorial. Actually that is not true. The definition tells us what the value of 0 factorial is, and, for positive integers, it tells us what the value of factorial is in terms of the factorial of smaller integers. Thus, the factorial of a given number is never defined in terms of itself. For example, how would we use the above definition to compute the value of 3 factorial? Using the second line of the definition, we would get the following:

```
3!  = 3 * 2!
```

This does not give the value of 3 factorial, but it does tell us how to compute the value of 3 factorial if we know the value of 2 factorial. Applying the second line of the definition again, we get:

```
3! = 3 * 2!
   = 3 * (2 * 1!)
   = 6 * 1!
```

This reduces the problem to computing the value of 1 factorial. Applying the definition again gives us:

```
3! = 6 * 1!
   = 6 * (1 *0!)
   = 6 * 0!
```

This gives us the value of 3 factorial in terms of the value of 0 factorial.

However, now we can apply the first line of the recursive definition of factorial which tells us what the value of 0 factorial is. This gives us:

```
3! = 6 * 0!
   = 6 * 1
   = 6
```

Thus, we know the value of 3 factorial is 6.

Similarly, the recursive definition of factorial given above can be used to compute the factorial function for any nonnegative integer.

Factorial Command and Function

Here are a PEEL command and a PEEL function that compute factorial using the recursive definition we have just given:

```
(defcom compute_a_factorial
     &doc "This function computes factorials"
     (print (factorial (prompt_for_integer "Type an integer" 1))))

(defun factorial ((n integer)
                  &local (temp integer)
                  &returns integer)
    (if (= n 1) (return 1))
    (setq temp (* n (factorial (1- n))))
    (return temp))
```

This example creates a defcom and a defun. The function of the defcom is to prompt the user for an integer and pass it to the factorial function. The factorial function does the work and ultimately returns a number that is printed by the defcom.

Look carefully at what happens in the defun. Assume that you are starting by asking for 10!. In the first invocation, _n_ equals 10. Therefore, the function sets _temp_ to the value of 10 times the number returned by the second invocation of factorial. However, factorial is now invoked with a value of 9, meaning that 10! now equals 10 times 9!.

The second time factorial is invoked, _n_ equals 9. The same procedure is gone through again, so that 9! is interpreted as 9 times 8!. This continues until _n_ is equal to 1. Only at this time does EMACS actually begin returning values. In all cases, the very first value returned is 1. However, this is returned to the factorial that invoked it, and the multiplication of 2 times 1 is performed. This value is returned, then it is used in the multiplication of 3 times 2, and so on.

Words tend to get in the way when talking about recursion. The following illustrates the steps taken.

```
(* 10 (factorial 9))
(*  9 (factorial 8))
(*  8 (factorial 7))

     ...

(*  2 (factorial 1))
(*  2 1)
(*  3 2)
(*  4 6)

     ...

(* 10 362880)
```

As you can see, two things are happening. The first is that the factorial is unwound until EMACS gets a factorial that can be computed. In this case 1!. At this point, EMACS traces its path back until it reaches the starting point.

Also, at each invocation of factorial, new variables for n and temp are created. When this happens, the old values are saved.

6
Interactive I/O

In traditional programming languages, I/O means taking information and writing it to a file in a variety of forms. Information can be blocked or unblocked. There also can be many different kinds of data; for example, characters, floating-point numbers, integers, double-precision numbers, complex numbers, and the like. This information can be written in streams or it can be written as records.

In most cases, the information in the file is structured. For example, in PL/I or COBOL, the data might be organized something like:

```
1   EMPLOYEE.
    2   NAME.
        3   LAST-NAME.
        3   FIRST-NAME.
        3   MIDDLE-INITIAL.
    2   ADDRESS.
        3   NUMBER-AND-STREET.
        3   CITY.
        3   STATE.
        3   ZIP.
```

The file, then, consists of information that conforms to a structure. While the information differs from occurrence to occurrence, the structure of the information does not change. Moreover, the data types of each data item can be different.

In EMACS, the information is completely unstructured and the data in the files must be ASCII data. This means that primitives for structured input and output do not exist.

Where EMACS has similarities to traditional programming languages is in the existence of two kinds of I/O: file (buffer) I/O and interactive I/O. The subject of this chapter is interactive I/O. That is, this chapter describes how a user talks to EMACS and how EMACS talks back to the user. Buffer I/O is discussed in the next chapter.

When EMACS displays a file or buffer, it divides the screen into two parts. The first part is the top 21 lines of the screen. This is where text appears. The next line is the status line and the remaining two lines are the minibuffer. It is here that you type responses to prompts.

THE MINIBUFFER

When typing a command that requires a response (such as query_replace), EMACS prints a message in the minibuffer that tells you what to type. It can then read your response and use that response to perform a function. EMACS also uses this area for printing messages that tell you what to do. In other cases, this area is used for printing error messages.

How to Write to the Minibuffer

The following statement writes a message to the minibuffer:

```
(info_message text)
```

text can either be characters delimited by double quotation marks or a string variable.

Two problems can occur when using info_message. The first is that something else may come along and write over the message before a user has a chance to read it. For example, two messages are produced by an extension within a short period of time. The way around this problem is to use the following statement right after the info_message:

```
(sleep_for_n_milliseconds integer)
```

This puts EMACS to sleep for the number of milliseconds specified. In this way, you can guarantee that the first message will not be overwritten too quickly.

The second problem is that the text printed by info_message can stay around too long. For example, in the describe function, a message is printed in the minibuffer. When EMACS returns the user to the place where describe was invoked, the message should disappear. The way you get the message to disappear is by writing a null info_message to the screen:

```
(info_message "")
```

This statement overwrites the old message with a blank line, which, in effect, removes the message.

Prompting

When creating an interactive program in a language such as COBOL, you have to write one statement that prints a prompt. To receive data from the user, you have to write a second statement that accepts information from the terminal. Because it does not make a lot of sense for a program to want to accept data without telling the user what should be typed, EMACS contains several functions whose purpose is to print a message and accept the information typed at the terminal. These functions are:

Function	Action
prompt	Displays a message and returns what the user types as a string.
prompt_for_string	Displays a message and returns what the user types as a string. This function lets you specify a default value for the string.
prompt_for_integer	Displays a message and returns what the user types as an integer. This function lets you specify a default value for the integer.
prompt_for_symbol	Displays a message and returns what the user types as an atom. This function lets you specify a default type for the atom.

Here are two examples:

Example 1:

```
(setq foo (prompt "What is your name"))
```

This form would print "What is your name" in the minibuffer. After you type your name, your name would be assigned to the string variable foo. Notice that the following is not correct if you intend foo to be used as an integer variable:

```
(setq foo (prompt "What is the right margin"))
```

The reason this does not work is that EMACS assumes that the value of foo is to be a string where you want it to be a number. What would work is shown in Example 2.

Example 2:

```
(setq foo (prompt_for_integer "What is the right margin" 70))
```

This tells EMACS that the response is an integer. Notice the number 70. This is the value that will be used if someone types a carriage return.

For all prompting functions, the side-effect is the printing of the message and the effect is returning what was typed in response to the prompt.

Read Functions

Often, you want to read a character from the screen and return this value. For example, the settab and describe functions both use their own readers and handle what is typed in their own ways. The two functions that can be used for this are:

Statement	Action
assure_character	Returns the next character typed by the user and inserts it into the buffer. This waits for a user to type the character and returns the character.

read_character	Reads a character from the terminal. It returns the resulting character but does not insert it into the buffer.

Once again, here is the move routine from the tab package. This time, however, you will see how the reader is actually implemented.

```
(defun move (&local (movement string)
            &returns Boolean)
    (info_message "Type a space, t, b, f, h, r, ?, or q")
    (setq movement (char_to_string (read_character)))
    (select movement
            " "      (delete_char)
                     (insert " ")

                  ...

           otherwise
                     (info_message "unknown response")
                     (sleep_for_n_milliseconds 1000)
                     (return true)))
```

The one new thing here is a conversion statement that converts the character data returned by read_character into a string.

Conversion Routines

It often occurs that data is in one form and you need it to be in another, as the last example illustrated. EMACS contains the following conversion statements:

Statement	Action
char_to_string	Converts a character to a string.
integer_to_string	Converts an integer into a string.
string_to_integer	Converts a string into an integer.
CtoI	Converts a character to an integer between 0 and 255.
ItoC	Converts an integer between 0 and 255 into a character.
ItoP	Converts an integer between 0 and 127 into a Prime character with the high-order bit on.

PtoI	Converts a Prime character, which has the high-order bit on, into an integer in the range 0 through 127.
high_bit_off	Turns off the high-order bit in every character of a string.
high_bit_on	Turns on the high-order bit in every character of a string.

WRITING OVER THE TEXT AREA

Many times, you want to display some text that is too long to fit on one line of the screen. In this case, what you will want to do is overlay the text area with message text. An example of this is the {CTRL-X} {CTRL-B} command that lists buffers.

EMACS contains four commands for overwriting the text area of the screen. In all cases, the text printed is written on top of existing text solely because it needs a place to be displayed. This text never becomes part of the text in the buffer.

Statement	Action
print	Prints information on the screen. EMACS terminates this information with a carriage return. If you have more than a screen of data to be printed, EMACS prints 20 lines, stops, then prompts for a space to continue.
prinl	This is the same as print except that no carriage return is printed.
init_local_displays	Clears the screen before printing begins. This command does not pause after printing a screen of data. This means that information could be lost. After an init_local_displays, the user must type a {CTRL-L} to clear the screen.
local_display_generator	Begins printing text at point without clearing the screen. Otherwise, this is the same as init_local_displays.

In many cases, you really do not want overprinting. Instead, it is easier to create a buffer, insert the text into that buffer, and then bring the user to that buffer. This will be discussed in the next chapter.

DEBUGGING PEEL PROGRAMS

If you are attempting to debug a large PEEL program, and you are confused by PEEL's occasionally somewhat obscure error messages, there is a convenient debugging facility you can use.

This facility is based on the fact that you may insert any of the prompt functions into your program at any point. When your PEEL program reaches that point, it displays the prompt string and waits for your reply, as you know.

In response to any such prompt, you may type:

```
{ESC} {ESC}
```

At that point, PEEL displays a new prompt, "2PL:". You may then type any PEEL command.

Alternatively, in response to your debugging prompt, you may type

```
[{ESC}n] {ESC} X
```

At that point, PEEL displays a new prompt, "2Command:". You may then type any EMACS command.

The power of this capability lies in the fact that you may stop your program at any point with a prompt, and then interrogate variables, change the value of variables, or even invoke other functions, in order to determine why your PEEL program is not working properly.

7
Manipulating Text in Buffers

In a buffer, there exist many things you can deal with as "atomic" entities. For example, at times you might want a character to be the basic unit; at other times, a word; at still others, an arbitrary region of text. This becomes more complicated when a context is applied to these atomic entities. For example, in text, a sentence terminates with a period, question mark, or exclamation point. In PL/I, a statement must end with a semicolon. In FORTRAN, the statement terminator is the end of a noncontinued line.

In EMACS, excluding the libraries, the following entities exist:

- Character
- Whitespace
- Words
- Lines
- Regions
- Buffers

The library file EMACS*>EXTENSIONS>SOURCES>TEXT.EM adds the following:

- Clauses

- Sentences
- Paragraphs

These entities are not sensitive to different programming environments. (Changing the definition on a temporary basis is the function of modes. See Chapter 9.)

The whitespace entity is exactly what it sounds like. It is an atom that contains the space character. (This is a value set by the user libraries in EMACS*. If you do not use the libraries, the whitespace atom includes the newline character.)

CHARACTERS

A character is the simplest element to deal with. The character-oriented statements are:

Statement	Action
back_char	Moves point back one character.
forward_char	Moves point forward one character.
rubout_char	Removes the character preceding point.
delete_char	Removes the character following point.
twiddle	Inverts the position of the two characters preceding point.
current_char	Returns the current character.

WHITESPACE

Because there are few files that do not contain blanks, PEEL has a number of statements that make dealing with these blanks much easier. As mentioned previously, whitespace can have one of two system-defined definitions. In most cases, you will want it to be equal to just a space (which is how it is set in the libraries). However, if you need the system-defined definition, you will have to do this yourself. The old value has been saved in the variable init_whitespace. Therefore, to restore it, all you need do is add the following statement to any extension:

```
(setq whitespace init_whitespace)
```

(The init_whitespace variable has two characters: the space and the newline character.) The interesting thing about the whitespace variable is that you can put anything you want into it and then have the EMACS whitespace functions treat what you put into it as whitespace.

The following statements manipulate whitespace or blanks:

Statement	Action
skip_back_over_white	Moves point backward so that it is looking at the first character it finds that is not in whitespace.
skip_over_white	Same as skip_back_over_white except that it moves forward.
skip_back_to_white	Moves point backward so that it is looking at the first character it finds in whitespace.
skip_to_white	Same as skip_back_to_white except that it moves forward.
trim	Removes spaces from the beginning and the end of a string.
delete_white_left	Deletes backward from point until it reaches a character not in the whitespace atom.
delete_white_right	Deletes forward from point until it reaches a character not in the whitespace atom.
delete_white_sides	Combines both delete_white_left and delete_white_right into one atom.
tab	Inserts blanks to the system-defined tab-stops, which are located every five spaces. Note that these tabs cannot be changed. This function is not bound to {CTRL-I} when you use the TAB library.
type_tab	Inserts spaces to user-defined tab positions if point is located after the last tab stop on the line.
insert_tab	Inserts blanks from point to the next tab stop.

The last two functions are in the TAB library.

WORDS

In EMACS, a word is defined as an alphanumeric string that is delimited by a separator. The separators are all punctuation marks, a carriage return, a space character, or a hyphen. However, an underscore is not a separator. The basic word operations are:

Statement	Action
forward_word	Moves point forward one word.
backward_word	Moves point backward one word.
delete_word	Kills the word in front of the cursor.
rubout_word	Kills the word behind the word.

LINES

The way that information is presented on the screen is in lines. This is what most people think of when they look at a screen of information. Consequently, EMACS contains a number of functions for manipulating lines, moving from line to line, and checking line status. The following list presents line commands.

Statement	Action
next_line	Moves point down one line to the first character on the next line. If point is on the last line of the file, moves point to the beginning of the line. It also returns a Boolean that indicates if the operation was successful.
next_line_command	The same as next_line except that it tries to retain horizontal position and will move down from the last line. This command does not return a Boolean.
prev_line	Moves point up a line. This statement does not retain horizontal position; that is, point becomes the first character on the line. It returns a Boolean that indicates if the operation was successful.
prev_line_command	Same as prev_line except that it retains horizontal position.

begin_line	Moves point to the beginning of the current line.
end_line	Moves point to the end of the current line.
goto_line	Goes to a specific line in the file; for example, (goto_line 15).
kill_line	Kills text from point to the end of the line. If point is at the end of a line, this command kills just the carriage return.
cr	Inserts a carriage return.
open_line	Inserts a carriage return after point.
current_line	Returns the text on the current line.
rest_of_line	Returns the text from point to the end of the line.
stem_of_line	Returns the text from the beginning of the line to point.
first_line_p	Returns true if point is anywhere on the first line in a buffer.
beginning_of_line_p	Returns true if point is at the beginning of a line.
end_of_line_p	Returns true if point is at the end of a line.
last_line_p	Returns true if point is anywhere on the last line in a buffer.

CLAUSES

The following statements treat clauses as atomic entities. An end-of-clause is defined as a terminator followed by a space or end-of-line. A clause terminator is any one of the following characters:

```
. ! ? , ; : ( ) { } [ ]
```

A beginning-of-clause is defined as the text following an end-of-clause.

Statement	Action
backward_clause	Moves point to the beginning of a clause.
forward_clause	Moves point to the end of a clause.
forward_kill_clause	Kills from point to the end of a clause. This places the killed text onto the kill ring.
backward_kill_clause	Kills from the beginning of a clause to point. This places the killed text onto the kill ring.

SENTENCES

The following statements treat sentences as atomic entities. An end-of-sentence is defined as being a terminator (. ! ?) followed by a space or end-of-line. A beginning-of-sentence is defined as the text following an end-of-sentence.

Statement	Action
backward_sentence	Moves point to the beginning of a sentence.
forward_sentence	Moves point to the end of a sentence.
forward_kill_sentence	Kills from point to the end of a sentence. This places the killed text onto the kill ring.
backward_kill_sentence	Kills from the beginning of a sentence to point. This places the killed text onto the kill ring.

REGIONS

A region is an arbitrary area that is specified by the user or under program control. Its main function is copying text to a kill buffer. After it is in a kill buffer, it can be inserted back into a buffer.

The region statements are:

Statement	Action
mark	Places a pointer that identifies one end of the region.
copy_region	Places the text between mark and point onto the kill ring.
kill_region	Places the text between mark and point onto the kill ring. It also removes the text from the buffer.
delete_region	Kills the text between mark and point. This statement does not put the text onto the kill ring.
append_to_buf	Takes a region and puts it at the end of a named buffer. If this command is given an argument, it does not kill the region first.
append_to_file	Same as append_to_buf, but it writes the region into a file.
prepend_to_buf	Takes a region and puts it at the beginning of a named buffer. If this command is given an argument, it does not kill the region first.
prepend_to_file	Same as prepend_to_buf, but it writes the region into a file.

The last four commands are contained in the file BUFFER.EM.

CURSORS

As discussed in the EMACS Reference Guide, a mark lets you create an arbitrary region of text. However, there is little that you can do with the text. What is needed is a way to assign a region to a variable and manipulate it in some manner. The PEEL functions that allow these kinds of interactions all use cursors, which are the subject of this section.

A cursor is an entity that indicates a place in a buffer. Specifically, it contains the following information:

1. The name of the buffer

2. The line number

3. The horizontal position (the column) in a line

Like a mark, a cursor has the property that it is bound to a place in a file. If something occurs that would change this position, the value of the cursor changes. This means that the cursor always points to the same place. Suppose we have the following command:

```
(defcom foo
    (with_cursor start
        (prev_line 5)
        (open_line)
        (insert "Foobar")
        (go_to_cursor start)))
```

This function creates a cursor called start, moves up five lines, opens up the line, and inserts the string "Foobar" into the text. Finally, the function returns point to the place indicated by start. However, the value of start has changed because of the insertion. A slight modification to this command makes it print out the value of the cursor. The function now reads:

```
(defcom foo
    (with_cursor start
        (print start start)
        (prev_line 5)
        (open_line)
        (insert "Foobar")
        (go_to_cursor start)
        (print start start)))
```

The difference between the two functions is the print statements. (The second option to the print statement tells EMACS at what cursor position it should print the text. In this way, EMACS is inserting text into the buffer.) When executed, this function might print the following:

```
[CURSOR c07 340,1]
[CURSOR c07 342,1]
```

<u>Note</u>

> When EMACS prints something in square brackets, the quantity within the brackets has more than one attribute to be described. In this case, EMACS is telling you that it is printing information about a cursor, and the three properties of the cursor are c07, which is the buffer name, 340, which is the line number, and 1, which is the column number.

As indicated above, the value changes so that the text pointed to by the cursor remains the same, although its location changes.

The basic difference between a cursor and a mark is that a mark becomes a permanent placemarker into the file, as long as it is not pushed off the ring of marks. As point moves, all the marks in the ring must be updated. Although this is also true of cursors, they do not go into the ring and are not permanent. This means that when you are done with them, they leave and EMACS does not incur any additional overhead.

Like all EMACS entities, a cursor can be assigned to a variable. The following two statements are used:

Statement	Action
current_cursor	Returns the value of point.
copy_cursor	Returns a copy of a cursor.

Notice the difference between the following two statements:

```
(setq foo current_cursor)
(setq bar (copy_cursor current_cursor))
```

The first statement equates the variable <u>foo</u> with the current cursor. This means that as point moves, the value of <u>foo</u> will change. (Notice that current_cursor is not within parentheses.) The second statement equates <u>bar</u> with the value of current_cursor. However, <u>bar</u> equals the value of current_cursor when the assignment is made. As point moves, the value of current_cursor changes; however, the value of <u>bar</u> will not change.

After point has changed, you can return to the cursor position with the following command:

```
(go_to_cursor named-cursor)
```

For example:

```
(go_to_cursor bar)
```

The operation of saving the present position and then returning at a later time is so common that PEEL contains a special form that combines saving and restoring position into one operation. That is, the following statement appears over and over again in extensions:

```
(setq start_position (copy_cursor current_cursor))
...
(go_to_cursor start_position)
```

To save you a little effort, the EMACS save_excursion special form does both of these operations. It is used as follows:

```
(save_excursion
     (do_something_that_moves_cursor))
```

This form indicates that when the save_excursion form completes, EMACS should go back to the position point was at when the form began. The one peculiarity is that when the form ends, the cursor is sometimes left in the middle of the screen. Chapter 8 shows how to add another step, using the window_info command, to ensure that the position of point does not change.

Now that different ways of assigning cursors have been discussed, the next step is to look at how to work with text contained in a buffer. Basically, there are only three operations that can be done:

- Copying text in the buffer
- Inserting text into the buffer
- Deleting text in the buffer

You have already seen these operations in a different context. You are now ready to use cursors to perform these operations.

Copying Text

It often occurs that an extension needs to extract information from text contained in a buffer, and then modify it in some way. The statement that extracts text from a buffer is point_cursor_to_string. Its syntax is:

```
(point_cursor_to_string cursor)
```

As was discussed earlier, cursors can be associated with variables in two ways: using the copy_cursor statement or using the with_cursor statement. It does not matter which of the two you use, as the following examples illustrate:

Example 1:

```
(with_cursor start
     (if (forward_search "foo")
         (setq text_string (point_cursor_to_string start))))
```

This form performs the following operations:

1. Establishes a cursor named start. This placemarker is the position of point when the form begins execution.

2. Moves point to the first character after the string foo.

3. Copies into the variable called text_string all the text from the position marked by start up to point.

The above function is identical to:

```
(setq start (copy_cursor current_cursor))
(if (forward_search "foo")
    (setq text_string (point_cursor_to_string start)))
```

The only difference between the two is that in the first example start only has context within the form. In the second, start has meaning for the remainder of the extension.

Example 2:

This example is taken from the view_kill_ring function.

```
(defun view_kill_ringf (&local (counter integer)
                               (kill_array array))
    (setq kill_array (make_array 'string 11))
                                          ;all kill buf names in array
    (setq counter 1)
    (select_buf ".buffers")               ;this is where buffer names
                                          ;are stored
    (save_excursion
      (move_top)
      (do_forever
         (if (forward_search ".kill.")          ;look for kill bufs
             (begin_line)
             (with_cursor start
                  (forward_search " ")          ; put it into array
                  (aset (point_cursor_to_string
                                 start) kill_array counter)
                  (setq counter (1+ counter)))
         else
              (stop_doing))))

       ...
```

This segment of the view_kill_ring function goes to the EMACS internal buffer (named .buffers) and then locates all buffers that have the string .kill. in it. Then, using the with_cursor form, it extracts the buffer name and puts it into the kill_array variable.

A second useful function is range_to_string. Its syntax is:

```
(range_to_string cursor_1 cursor_2)
```

The only real difference between this statement and point_cursor_to_string is that the region is bounded by two named cursors rather than by a cursor and point.

Deleting Text From a Buffer

The statement used to delete text from a buffer is:

```
(delete_point_cursor cursor)
```

You use this function in the same manner as point_cursor_to_string.

Inserting Text Into the Buffer

The statement used for inserting text into the buffer is insert. For example, in the view_kill_ring function, the user is shown the contents of the kill buffers. A prompt line is printed that gives the user (among other choices) the option of saving the contents of a kill_buffer. The code for this is:

```
(if (= response "s")
    (move_top)
    (with_cursor start
        (move_bottom)
        (setq save_text (point_cursor_to_string start))))
```

The yank_kill_text function, compared to the view_kill_ring function, is simplicity itself. Here it is in its entirety:

```
(defcom yank_kill_text
     &doc "Inserts text saved by view_kill_ring"
     (insert save_text))
```

The insert commands simply tell EMACS to place the argument into the buffer at point.

Another useful function is self_insert. Its function is to insert one character into the buffer. It comes in particularly handy when you want to insert more than one identical character. For example:

```
(setq indentation (prompt_for_integer "What is the indentation" 0))
(self_insert " " indentation)
```

The first statement prompts for a number. The second uses this number to tell EMACS how many spaces to insert. For example, if you had answered 10, EMACS would insert 10 spaces.

Working With Text Contained in Variables

EMACS contains a variety of primitives that will assist you in working with text copied into variables. Some of these are unique to EMACS; others have great similarity to PL/I built-in functions.

Statement	Action
catenate	Joins two strings together.
substr	Is the PL/I substr function. That is, this function either returns or sets parts of a string to the indicated value.
index	Is the PL/I index function. That is, this function finds where an indicated string begins in a second string.
translate	Is the PL/I translate function. That is, this transforms the indicated characters to the character shown.
remove_charset	Removes specified characters from string.
downcase	Transforms a string into all lowercase.
upcase	Transforms a string into all uppercase.

Here are some examples using these functions.

Example 1: catenate

```
(setq temp (/ (* (line_number current_cursor) 100) num_lines))
(setq message (catenate "-- "
                        (integer_to_string temp)
                        " % -- " message))
```

The first statement performs a computation that creates an integer number that is where point is in the file expressed as a percentage of the total length of the file. The second catenates three items together and assigns the result to the variable called message. Notice in particular that the integer_to_string function was used to convert an integer value to a string. If this conversion were not made, EMACS would have given you an error message.

Example 2: substr

```
(setq answer (substr (prompt "Do you wish to continue") 1 1))
```

This form prompts the user and returns the answer. The substr function then extracts the first character of the returned reply. Finally, this character is assigned to answer. Here is a slightly more complicated version of the same function.

```
(setq yes_no (substr (prompt "Continue") 1 1))
(if (| (= yes_no "y") (= yes_no "Y"))
    (do_something))
```

In this example, the first character of the reply is saved. This character is then compared against both "y" and "Y". The two results are logically OR'd together to see what should be done.

The way the above action is handled in many of the libraries is as follows:

```
(defun yesno ((query string)
               &returns Boolean
               &local (reply string))
     (setq reply (downcase (prompt query)))
     (do_forever
        (select reply
          "yes" "y" "ok" "true"
               (return true)
          "no" "n" "false"
               (return false))
        (setq reply
           (downcase (prompt (catenate query " (Yes or No)"))))))
```

This allows greater flexibility in handling user responses.

Example 3: index

```
(setq pos (index name ".EM"))
(if (> pos 0)
    (setq name (substr name 1 (- pos 1))))
```

In this example, the variable called name has been assigned some value. The index function then looks for the string ".EM" in the variable. If it is contained in name, pos is set to the position where the .EM would be. Otherwise, pos is set to 0. The next statement performs an action based on the value of pos. In this case, the substr function returns the text from the beginning of the variable to the character right before .EM.

Example 4: downcase

```
(defun lowercasef (&local (word string))
     (with_cursor here
          (forward_word)
          (setq word (downcase (point_cursor_to_string here)))
          (delete_point_cursor here))
     (insert word))
```

This function transforms the word following point to lowercase. It begins by setting a temporary cursor called here. It then moves point forward a word. The text between point and here is returned by point_cursor_to_string and then transformed by downcase. The text between point and here is then deleted and the converted text is inserted.

8
Modes

One problem that occurs when writing commands and systems of commands is that they become permanent parts of the EMACS environment. In many instances, what is wanted is a temporary way to redefine the meaning of EMACS commands and then restore EMACS back to the way it was before these commands were used. This ability in EMACS is called a mode.

EMACS, as it is shipped, contains a variety of modes. Let us look at one so that you can gain an appreciation of what a mode can do and is capable of doing. The explore mode consists of a variety of functions, each of which performs an action that this mode's designer thought essential. Here is a modification of the wallpaper command output that shows what the mode definitions are:

```
^X-U    explore_pop          EXPLORE: Pop from explore sublevel.
^X-u    explore_pop          EXPLORE: Pop from explore sublevel.
A       explore_attributes   EXPLORE: Show attributes.
?,H     explore_help         EXPLORE: Describe explore.
C,N     explore_create       EXPLORE: Copy a file.
D,G     explore_dive         EXPLORE: Dive from an explore directory.
K       explore_delete       EXPLORE: Delete a file.
R       explore_rename       EXPLORE: Change a file's name.
S       explore_spool        EXPLORE: Spool a file.
U       explore_pop          EXPLORE: Pop from explore sublevel.
```

This listing indicates that certain keys have been reassigned. For example, the letter K now calls a command called explore_delete. This means that while EMACS is in EXPLORE mode, you cannot use the letter K to insert a capital letter K. In a similar manner, the other elements have been redefined. The advantage of this is that the mode designer has created an interface that is uniquely tailored to the actions being performed.

The following example illustrates transforming the begin_line function so that it is useful in COBOL. In COBOL, you would want to write a command that keeps the user out of columns one through six.

```
(defcom cobol_begin_line
    (if (^ (go_to_hpos 7))
        (end_line)
        (whitespace_to_hpos 7)))
```

As may be obvious, the manner in which you write commands for a mode is identical to the way you would write commands in fundamental EMACS. This simple routine first tries to go to column 7. If it is unsuccessful, the routine adds spaces so that point is at column 7.

The problem is how to bind this to a key on a temporary basis. This is the subject of this chapter.

MODES DEFINED

The actions that occur when you type a keystroke that invokes a binding are not simple. EMACS does not simply take the keybinding and invoke the command. Instead, EMACS goes through a series of table invocations and lookups until it finds these commands. The keystrokes that you type tell EMACS what tables to use. (These tables are called dispatch tables.) When EMACS is first invoked, the following dispatch tables exist:

main	The main character dispatch mode
x	The dispatch table for the {CTRL-X} prefix in the main dispatch table
esc	The dispatch table for ESCAPE in the main dispatch table
mb_mode	The dispatch table for minibuffers
reader	The dispatch table used by the keyboard reader This is used, for example, to define {CTRL-_} as help_on_tap. If the function returns a string or character value, the result is returned in place of the character actually read. Otherwise, the reader will read another character from the keyboard.

When you define a mode, you are actually creating a new dispatch table. When the mode is invoked, it overlays the mode dispatch table on top of the existing tables. When you type a keystroke command, EMACS first searches the mode dispatch table. If it finds the keystroke there, it executes the mode definition of the command. If the function is not found there, it falls through to the "lower" table and checks to see if the command exists there. This means that the definitions in the latest dispatch tables take precedence over commands in previous dispatch tables. Chapter 9 tells more about getting information from dispatch tables.

There is nothing to prevent you from using more than one mode at a time. That is, you could be in a state where LISP, overlay, and fill modes are all operating at one time. However, when more than one mode is in force, the order in which they are declared is significant. For example, in overlay mode, the space character wipes out the character following point and then inserts a space. The definition of the space character in fill mode is different. This means that if overlay mode was the most recent mode, the fill will not work. However, if fill was the most recent mode, the space character would not overlay the character in front of point.

Prime does not recommend that you use the EMACS commands that directly invoke modes. Instead, you should use the functions turn_mode_on and turn_mode_off. Here is how they are used:

```
(defcom lisp_on
    &doc "Set lisp mode"
    (turn_mode_on (find_mode 'lisp) first))


(defcom lisp_off
    &doc "Turns off lisp mode"
    (turn_mode_off (find_mode 'lisp)))
```

Here are the functions:

```
(defun turn_mode_on ((mode dispatch)
                     &optional &quote (side atom)
                     &local (modes list))
    (turn_mode_off mode)
    (setq modes (buffer_info modes))
    (if (eq side 'first)
        (buffer_info modes (cons mode modes))
     else
        (buffer_info modes (append modes (list mode)))))
```

```
(defun turn_mode_off ((mode dispatch)
                      &local (modes list))
    (do_forever
        (setq modes (buffer_info modes))
        (if (not (member mode modes)) (return))
        (buffer_info modes (remove mode modes))))
```

The first part of this example shows the code for turning a mode on. The function of the find_mode statement is to return a dispatch table that is usable for the mode. The function of the turn_mode_off statement within turn_mode_on is to insure that the mode only exists once in a buffer. The buffer_info statements insert the dispatch table into the buffer mode list. Finally, you can also state the order in which the modes are inserted. If you specify first, as the example illustrates, the mode just inserted becomes the first one in the search path through the dispatch tables. Otherwise, the mode just entered becomes the last one in the search list.

In a similar manner, the turn_mode_off function removes a dispatch table from a buffer's mode list.

BINDING MODE FUNCTIONS AND COMMANDS

The way you create a keybinding for a mode is nearly identical to the way you do it for other bindings. The only difference is that you also have to specify the mode in which the binding will take effect. The statement that you use is set_mode_key. Here are two examples:

```
(set_mode_key "lisp" ")" "close_paren")

(set_mode_key "lisp" "^[^F" "balfor")
```

As you can see, this statement takes three arguments. The first is the name of the mode, the second is the keybinding, and the third is the name of the function or command.

After these definitions are created, EMACS will place them into a dispatch table, ready for use when a mode is invoked.

A different way of setting mode keys is illustrated in the overlay library, and is described in the dispatch_info discussion in Chapter 9.

9
Information Commands

EMACS performs and keeps track of a great number of things at the same time. For example, it retains information about every file, buffer, and window that it is using. You may find out the state of any of these things at any time by using the built-in functions described in this chapter. While this may be useful, what is important is that EMACS knows what kinds of properties an entity can have. Using these same functions, you can set these values to something you want. For example, you may want to present a buffer that is read-only so that a user cannot modify it.

A second category of information commands are those that give information about names and states, such as date or file_name.

BUFFER_INFO

The buffer_info command either gets or sets information about a buffer. It takes the form:

```
(buffer_info any optional-any)
```

The arguments to this command have the following meaning:

any	Either an atom, string, or user atom or user string that indicates the variable or property to be examined or set.
optional-any	The new value for the property or variable. These properties are defined below.

The buffer_info command is used to set and/or access buffer values. When the value is changed, it returns the previous value. The first argument is the name of the value to be accessed. The second argument is optional. If it is used, this second argument is the value to assign. Note that some values, such as the buffer name, cannot be modified.

The following values (properties) may be accessed:

Value	Meaning
name	The name of the buffer. This is read-only.
default_file	The pathname of the default file associated with the buffer.
modified	If this is true, the buffer is considered modified. It is indicated by an asterisk on the status line. The unmodify command sets this to false. Specifically, typing the unmodify command is the same as typing: (buffer_info modified false)
modes	Lists the modes associated with this buffer. Note that the order of the modes in the list is significant! Although you can use this attribute to set the modes, Prime strongly recommends that you use the turn_mode_on and turn_mode_off functions.
read_only	Prevents accidental modification of a buffer. For example: (buffer_info read_only true) This tells EMACS not to let a user enter any commands that will modify the buffer.

changed_ok	If this is true, the user is allowed to quit the editor even if this buffer has been changed. That is, modification status does not affect {CTRL-X}{CTRL-C}.
dont_show	If this is true, the buffer is suppressed in the {CTRL-X}{CTRL-B} listing. This command is most useful when creating temporary buffers that you do not want a user to know about. For example, the describe command uses two temporary buffers called .DESCRIBE and .DESCRIBE.EMACS. Neither of these will ever appear in the buffer listing.
two_dimensional	Controls whether next_line and prev_line move strictly vertically, and whether forward_char moves to the next line when reaching the end of the current line. This is normally set and unset with the 2don and 2doff commands.
fill_column	Can be set for word-wrapping and related packages. It is used by the fill-mode software. It is not used by the wrapping software.
mark	The current mark position in the buffer.
top_cursor	The beginning of buffer pointer.
bottom_cursor	The end of buffer pointer.
user	Used for extended values, as in:

```
(buffer_info (user comment_column) 40)
```

This statement associates a variable called comment_column with the buffer and assigns to it a value of 40. This value could be used as follows:

```
(if (< (cur_hpos)
       (buffer_info (user comment_column)))
    (whitespace_to_hpos
       (buffer_info (user comment_column))))
```

This form tells EMACS to check the current position and compare it to the user-defined comment_column. If the current position is less than the comment_column, EMACS adds spaces until the line is the proper length.

DISPATCH_INFO

The dispatch_info statement returns information on a mode or dispatch table. It returns the old value of the property. It takes the form:

```
(dispatch_info dispatch_table any-1 any-2)
```

Here any-1 is the key indicating what property to get/set. The optional any-2 argument is the new value.

The dispatch_info function can be used to set an item in the dispatch table so that a function is bound to a key. A mode value is a dispatch table that is found using the find_mode function. The argument to find_mode is either an atom or a string. The returned value is the mode value. The following modes are predefined:

Mode	Definition
main	The main character dispatch mode.
x	The dispatch table for the {CTRL-X} prefix in the main dispatch table.
esc	The dispatch table for ESCAPE in the main dispatch table.
mb_mode	The dispatch table for minibuffers.
reader	The dispatch table used by the keyboard reader. It is used, for example, to define {CTRL-_} as help_on_tap. If the function returns a string or character value, the result is returned in place of the character actually read. Otherwise, the reader will read another character from the keyboard.

The dispatch_info function can be used to interrogate and modify dispatch tables. The first argument is a dispatch table (that is, a mode). The second argument is either the atom "name" (in which case the name is returned), or it is a character, string, or integer value identifying the entry to be interrogated and/or modified.

If there is a third argument, it is the new object to be placed into the dispatch table. In all cases, the old value is returned.

FILE_INFO

The file_info command returns information about a file. It returns the old value of the property. It takes the form:

```
(file_info string property any)
```

Here, string is the pathname of the file, property is an atom explained below, and the any argument (which is optional) is the new value of the property.

The values for property are:

Value	Meaning
path_name	Returns the absolute pathname of the file. For example: `(file_info (file_name cur_cursor) path_name)` This returns the current pathname.
entry_name	Returns the entry name of the file.
directory_name	Returns the name of the directory that contains the file.
type	Returns the type of the file as "none", "file", "directory", "segdir", or "unknown".
dumped	Returns true if and only if the file has been dumped.
exists	Returns true if and only if the file exists. This is one of the most useful functions. For example, here is the mod_write_file extension:

```
(defcom mod_write_file
   &doc "Write specified file"
   &args ((place &prompt "Write file" &string))
   (if (= place "")
       (setq place (file_name cur_cursor)))
   (if (file_info place exists)
       (if (^ (yesno
          "Do you want to overwrite the file"))
           (return)))
    (write_file place))
```

After the prompt for the filename, EMACS checks to see if the file is in the directory specified. If it is, EMACS asks permission to overwrite.

WINDOW_INFO

The window_info command gets or sets information about a window. It also returns the old value of the property. It takes the form:

```
(window_info property any)
```

Here property is an atom, and the optional any argument is the new value for the property.

The values for property are:

Value	Meaning
top_line	The line number in the buffer at which the current window is displayed.
bottom_line	The last line on which the window is displayed.
left_column	The leftmost column in which the window appears.
right_column	The rightmost column in which the window appears.
is_active	Indicates if the window is being redisplayed.
is_major	Indicates if the window is a major window. This will usually be true.
top_line_cursor	This cursor points into the top line of the text that appears as the top line in the window. This is one of the most useful window_info functions, as will be shown below.
showing_numbers	Returns a true or false value that tells if line numbers are indicated on the screen. This is used by the #on and #off commands.

column_offset	The value of the horizontal column offset. This is used, directly or indirectly, by all commands that scroll the screen horizontally.
last_buffer_cursor	The cursor that {CTRL-X} B will go to.

Example:

This example illustrates the append_to_file command:

```
(defcom append_to_file
     &doc "Appends current region to a file"
     &na (&pass count &default 1)
     (appendf count))
(defun appendf (       (count integer)
                &local (top   cursor)
                       (place string))
     (setq top (window_info top_line_cursor))
     (setq place (prompt "What file do you want to append to"))
     (save_excursion
         (if (= count 1)            ; no arg or 1 is kill before
             (kill_region)
          else
             (copy_region))     ; else copy
         (with_no_redisplay
             (if (file_info place exists)  ;check if file exists
                 (find_file place)
              else          ; if it doesn't, create it
                 (go_to_cursor (find_buffer place))
                 (write_file place))
             (move_bottom)
             (yank_region)
             (save_file)))
      (window_info top_line_cursor top)
      (info_message "Region appended"))
```

This somewhat lengthy command is really pretty simple. The defcom, as others shown before, accepts an argument and transmits it to a function. The argument's sole purpose is telling the function what to do. If the argument is 1, the region to be appended is removed from the buffer; otherwise the region is just copied.

The top variable saves the position of the current top line, so that when EMACS returns to the buffer (using save_excursion), the window will not be shifted. If this command were not there, the line containing the current cursor would be centered in the window.

OTHER INFORMATION COMMANDS

The EMACS commands that return various kinds of information are:

- buffer_name
- cpu_time
- cur_hpos
- cur_cursor
- current_handler
- current_line
- current_major_window
- date
- dt
- file_name
- line_number
- list_dir
- major_window_count
- terminal_type
- uid
- user_name

These are explained in Appendix A.

APPENDIXES

A

EMACS Functions and Commands

This appendix contains a listing of all standard EMACS functions, commands, keywords, global variables, and data types.

Note

The entries in this appendix are arranged alphabetically, except that all names starting with non-alphabetic characters (#, *, >, |, and so forth), as well as those few names starting with capital letters, are placed at the beginning of the list.

For each command or function, the following information is given:

- A brief summary of what the command or function does.
- A command format, if it is a command. Often, two or more command formats are given, especially when there is a standard binding for the command.
- The function format, if it is a function.
- A description of the data types of the arguments.
- The action that EMACS performs when the command or function is executed.
- If relevant, an additional note or example.

- If relevant, the name of the EMACS library in which the source for the command or function may be found.

Although standard LISP functions are described in this appendix when they are available in PEEL, you may wish to refer to a LISP manual for further examples and details.

If the action description of a function states that the value NIL is returned, that is the same as the null list, ().

In the command formats, if [{ESC}n] is shown, the command does not ignore a numeric argument. The numeric argument may be specified either as [{ESC}n] or with any other method for specifying numeric arguments, such as a multiplier.

If [{ESC}n] is not shown in the command format, the command ignores a numeric argument.

Command and Function

The # command or function tells whether line-numbering mode is on.

Command Format: {ESC} X #

Function Format: (#)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS displays either "line numbers are off" or "line numbers are on" in the minibuffer.

The # command returns the value NIL.

#off Command

The #off command turns off line-numbering mode.

Format: (#off)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns off automatic line-numbering mode, and does not show line numbers at the left of the window.

The #off function returns the value NIL.

#on Command

The #on extended command turns on line-numbering mode.

Format: (#on)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns on automatic line-numbering mode, so that line numbers are displayed to the left of the window.

The #on function returns the value NIL.

& Function

The & function is an abbreviation for the "and" function.

&args Keyword

The &args keyword is an abbreviation of the &arguments keyword.

&arguments Keyword

The &arguments keyword starts the argument definition section of a defcom. (See defcom for the complete syntax.)

&char_arg Keyword

The &char_arg keyword is an abbreviation for the &character_argument keyword.

&character_argument Keyword

The &character_argument keyword is used with the defcom function to allow the passing of the keystroke that invoked the command. (See the description of defcom for the syntax.) This keyword is not particularly useful, because the character_argument function can obtain and return this information regardless of whether &character_argument has been specified.

&doc Keyword

The &doc keyword is an abbreviation of the &documentation keyword.

&documentation Keyword

The &documentation keyword is used in the defcom command to allow you to specify a string description of the command. It is used in help facilities, such as apropros and explain_key. (See defcom.)

&eval Keyword

The &eval keyword reverses the effect of the &quote keyword in a defun function definition, and specifies that all the following arguments are to be evaluated. (See defun.)

&ignore Keyword

The &ignore keyword is used in a defcom to specify that numeric arguments to the command being defined are to be ignored.

&integer Keyword

The &integer keyword is used in a defcom with &args to specify that an argument has the integer data type. (See defcom for the complete syntax.)

&local Keyword

The &local keyword is used in a defun definition to specify that there are no further arguments in the function header, and that the remaining entries are local variables used only within the function being defined.

&macro Keyword

The &macro keyword is used in a defun definition to specify that the function returns a list that should be evaluated in the calling context. (See defun for details.)

&na Keyword

The &na keyword is an abbreviation of &numeric_argument.

&numeric_arg Keyword

The &numeric_arg keyword is used in a defcom to specify how a numeric argument to the command is to be handled.

&optional Keyword

The &optional keyword is used in a defun function header to specify that all the following arguments are not required. Optional arguments not specified when the function is invoked are initialized to NIL.

&pass Keyword

The &pass keyword is used in a defcom to specify the name of a local variable to which the value of the numeric argument is to be assigned.

&prefix Keyword

The &prefix keyword is used in a defcom to indicate that the last invocation character is to be inserted into the buffer as a side effect of the command.

&prompt Keyword

The &prompt keyword is used in a defcom to cause a local variable (defined with &args) to be prompted for, if it is not specified in the command's invocation.

&quote Keyword

The &quote keyword is used in the argument list of a defun to specify that EMACS should simply bind all the following arguments, without attempting to evaluate them, when the function is invoked. The &eval keyword reverses the effect of a previous &quote keyword.

&repeat Keyword

The &repeat keyword is used with &numeric_arg in a defcom to specify that the numeric argument to the command indicates the number of times that the body of the defcom code should be executed. (See defcom.)

<u>Format</u>: &na (&repeat)

&rest Keyword

The &rest keyword is used in the argument list of a defun to tell EMACS to take the rest of the arguments and put them into a list.

<u>Format</u>: &rest (v list)

<u>Argument</u>: The argument <u>v</u> is any PEEL variable name. As shown in the format, it must be given the list data type.

<u>Example</u>: &rest (r list)

When the function is invoked, all the following arguments will be placed into a list assigned to the local variable <u>r</u>.

&returns Keyword

The &returns keyword is used in a defun to specify the data type returned by the function being defined. This is the data type of the effect of the function, not any side-effect. The value returned by the function is specified by the return function.

&string Keyword

The &string keyword is used in a defcom with &args to specify that an argument has the string data type.

&symbol Keyword

The &symbol keyword is used with &args in a defcom to specify that the data type of the argument is a symbol.

* Function

The * arithmetic function returns the product of its arguments.

Format: (* x1 [x2 ... x8])

Arguments: * takes one through eight integer arguments.

Action: The * function returns an integer value representing the product of its arguments. If there is only one argument, the value of that argument is returned.

Example: The function

```
(* 3)
```

returns the integer value 3, while

```
(* 3 4 5)
```

returns the integer value 60, obtained by multiplying together 3, 4, and 5.

*_list Function

*_list is a LISP function that constructs a list of its arguments, much like the list function. The last argument becomes the cdr of the last cons used in constructing the list. It is an extended version of cons, and is thus useful for adding elements to the front of a list.

Format: (*_list arg1 arg2 [arg3...arg8])

Arguments: *_list takes two to eight arguments of any data type.

Examples: The following expression

(*_list 'a 'b 'c 'd)

is equivalent to

(cons 'a (cons 'b (cons 'c 'd)))

Both construct the following value:

(a b c . d)

If d is a list (i j k), then the expression becomes:

(*_list 'a 'b 'c '(i j k))

This adds three elements to the front of the list (i j k), constructing:

(a b c i j k)

*catch Function

The *catch function, derived from MACLISP, corresponds roughly to a non-local-goto or an on-unit definition in other high-level languages.

Format: (*catch t sl)

Arguments: The argument t is an atom (called a "tag"), usually quoted.

The argument sl is a PEEL statement.

Action: The *catch function returns a value whose data type depends upon execution of the argument.

If multiple PEEL statements are required at sl, use a progn at sl, and place the multiple statements within it. The value returned by a progn is the value of the last statement in the progn.

EMACS executes the argument sl, stopping if a function of the form

(*throw t v)

is executed.

If no such *throw function is executed, then *catch returns the value of the function sl.

If a *throw function in the format just shown is executed, then execution of sl is terminated immediately, and *catch returns the value v.

Note: The *catch function is like the catch function except that the argument order is different. The order of arguments in *catch is much easier to use.

Examples: Consider the following:

```
(*catch 'hello
        ...
    (if (> a 0)  (*throw 'hello "error"))
        ...
     a)
```

If the value of a is positive at the time the if statement is executed, then execution of *catch will terminate, returning the value "error". Otherwise, execution will continue, and unless stopped for some other reason, will continue to the form a, and *catch will return that value.

The following example shows multiple *catches and *throws, and the use of progn.

```
(print (*catch 'foo
   (progn (setq a (prompt "hello"))
          (if (= a "a") (*throw 'foo 20))
          (print (*catch 'bar
             (progn (setq b (prompt "goodbye"))
                    (if (= b "b") (*throw 'foo 30))
                    (if (= b "c") (*throw 'bar 100))
                    ))))))
```

Note that you can nest catches and throws. For example, an outer *catch might catch "quit" throws, and an inner one might catch argument errors.

*throw Function

The *throw function is a standard LISP function that provides a function similar to invocation of an on-unit in other languages.

Format: (*throw t v)

Arguments: The argument t must be an atom, usually quoted. The argument v can have any data type.

Action: The function is legal only within the argument list of *catch with the tag t. (See the description of *catch for further details.)

Note: The *throw function is like the throw function except that the argument order is different.

+ Function

The + arithmetic function adds together its arguments.

Format: (+ x1 [x2 ...x8])

Arguments: The + function takes at least one argument and no more than eight arguments. All arguments must have the integer data type.

Action: The + function returns an integer value. If there is only one argument, the value of that argument is returned. If there is more than one argument, then + adds together all the arguments and returns their sum.

Example: The function

    (+ 3)

returns the value 3, while the function

    (+ 3 4 5)

returns the value 12, equal to the sum of 3, 4, and 5.

- Function

The - arithmetic function either subtracts two arguments or negates a single argument.

Format: (- x [y])

Arguments: The argument x, and the argument y if specified, must be integer values.

Action: The - function returns an integer value. If y is not specified, - returns the value of -x. If y is specified, - returns the value of (x-y).

Examples: The function (- 2) returns the value -2, while the function (- 10 5) returns the value 5.

/ Function

The / arithmetic function performs integer division.

Format: (/ x y)

Arguments: Both x and y must be integer values.

Action: The / function returns an integer value. The value returned is (x/y), truncating if necessary.

Note: Truncation is always in the direction toward 0. Therefore, for example, (/ 24 5) and (/ -24 -5) each return the value 4, while (/ -24 5) and (/ 24 -5) each return the value -4.

1+ Function

The 1+ arithmetic function returns the value obtained by adding one to its argument.

Format: (1+ x)

Argument: The argument x must be an integer.

Action: The 1+ function returns the integer value (x+1).

Note: The following two expressions are equivalent:

```
(1+ x)
(+ x 1)
```

1- Function

The 1- arithmetic function returns the value obtained by subtracting one from its argument.

Format: (1- x)

Argument: The argument x must be an integer.

Action: The 1- function returns the integer value obtained by computing (x-1).

Note: The following two expressions are equivalent:

```
(1- x)
(- x 1)
```

2d Command and Function

The 2d extended command or function tells whether two-dimensional mode is on. (See 2don.)

Command Format: {ESC} X 2d

Function Format: (2d)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS displays either "2d is off" or "2d is on" in the minibuffer.

2doff Command and Function

The 2doff extended command or function turns off two-dimensional mode (see 2don).

Command Format: {ESC} X 2doff

Function Format: (2doff)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns off two-dimensional mode.

2don Command and Function

The 2don extended command or function turns on two-dimensional mode.

Command Format: {ESC} X 2don

Function Format: (2don)

Arguments: A numeric argument, if specified, is ignored.

Action: EMACS turns on two-dimensional mode. In this mode, you may use the cursor commands to move your cursor anywhere on the screen, even where there are no characters in the buffer. If you insert a character on the screen at a point on a line where characters to the left of the point have not previously existed, then EMACS automatically inserts blanks on the line up to that point.

<, <=, =, ^=, >, >= Functions

These relational operators test the relationship between their arguments.

Format: (op arg1 arg2), where the op is one of the following: <, <=, =, ^=, >, or >=.

Arguments: Each of the relational operators takes two arguments. Both arguments must have the same data type, and the common data can be any.

Action: Each of the relational operators returns a Boolean value, depending upon the result of the comparison. The < function returns true if the first argument is less than the second, and false otherwise. Similarly, the other functions compare and test as follows: <= is the less than or equal comparison; = is the equal comparison; ^= is the not equal comparison; > is the greater than comparison; and >= is the greater than or equal comparison.

If the two arguments are integers, then an ordinary integer comparison is performed.

If the arguments are strings, they are compared according to the rules to the ASCII collating sequence, after padding the string to the length of the longer one with blanks, if necessary.

If the arguments are Boolean, then true is considered to be less than false.

If the arguments are cursor values, then they must be in the same buffer, and one cursor value is considered to be smaller than another cursor value if it precedes the second one in the buffer.

CtoI Function

The CtoI conversion function converts a character to an integer value.

Format: (CtoI c)

Argument: The argument c must have the character or string data type.

Action: The CtoI function returns an integer value. The value is greater than or equal to 0 and less than or equal to 255. The value returned is equal to the position of the character argument c in the ASCII collating sequence.

If the argument c is a string, then the value for first character of c is returned.

ItoC Function

The ItoC function converts an integer between 0 and 255 into a character.

Format: (ItoC x)

Argument: The argument x must have the integer data type.

Action: The ItoC function returns a character value. The character returned is equal to the character in position x modulo 256 of the ASCII collating sequence.

ItoP Function

The ItoP conversion function converts an integer between 0 and 127 into a Prime character with the high-order bit on.

Format: (ItoP x)

Argument: The argument x must be an integer.

Action: The ItoP function returns a character value. Given an argument x,

```
(ItoP x)
```

returns the same value as

```
(ItoC (+ (modulo x 128) 128)
```

NL Global Variable

NL is a global variable having the character data type. The value of NL is the new-line character. You may imbed the value of NL in strings.

PtoI Function

The PtoI conversion function converts a Prime character, having the high bit on, into an integer between 0 and 127.

Format: (PtoI c)

Argument: The argument c must be a character value, usually with the high bit on.

Action: The PtoI function returns an integer value between 0 and 127. Given an argument c, the value of

```
(PtoI c)
```

is c with the high bit off, given by

```
(modulo c 128)
```

If the argument c is a string, then the value for first character of c is returned.

^ Function

The ^ function is an abbreviation for the not function.

^= Function

(See the = function listed under '<'.)

^p_prev_line_command Command

(See prev_line_command.)

^q_quote_command Command

(See quote_command.)

^s_forward_search_command Command

(See forward_search_command.)

| Function

The | function is an abbreviation for the or function. (See the or function for further information.)

abort_command Command

The abort_command command and function aborts a command.

Function Format: (abort_command)

Command Format: {ESC} X abort_command or {CTRL-G}

Argument: A numeric argument, if specified, is ignored.

Action: The abort_command command permits the current command to be aborted. For example, you may use (abort_command) in a PEEL program at any point to terminate execution at that point. EMACS causes the terminal to beep and returns you to command level.

The (abort_command) function does not return any value.

abort_minibuffer Command

(See abort_command.)

abort_or_exit Command

(See abort_command.)

af Function

The af function inserts the results of an active PRIMOS command function invocation into the buffer at the current cursor position.

Format: (af s)

Argument: The argument s must have the string data type.

Action: The af function inserts the result of an active PRIMOS function invocation into your text buffer at the current cursor position. The function returns NIL as its value.

Example: The function

```
(af "[calc 4+5]")
```

inserts the string "9" into your text buffer.

all_modes_off Command

The all_modes_off command turns all modes off for the current buffer.

Format: {ESC} X all_modes_off

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns all modes off in the current buffer. Note that any side-effects of turning a mode on (such as 2don in overlay mode) will not be cancelled.

and Function

The and function is a Boolean operator that returns the logical "and" of its arguments.

Format: (and b1 b2 [b3 ... b8])

Arguments: The and function takes at least two and no more than eight arguments. All arguments must have the Boolean data type.

Action: The and function returns a Boolean value computed by taking the logical "and" of all of its arguments. That is, the and function returns the value true if the value of all its arguments is true; otherwise it returns the value false.

Note that all arguments are evaluated, regardless of whether any one is false, and the order of evaluation is unspecified.

any Data Type

A variable with the any data type can take on any value of any data type supported by PEEL. Note that all undeclared global variables have by default the any data type.

append Function

The append function is a standard LISP function that appends all the items of one list to the end of another list.

Format: (append lstl lst2)

Arguments: Both lstl and lst2 must be lists.

Action: The append function returns a value with the list data type. The value is computed by appending all the items in lst2 to the end of the items in lstl.

Example: The function

```
(append '(a b c) '(d e f))
```

returns the list (a b c d e f).

Note: The append function does not change the value of either list lstl or lst2.

append_to_buf Command and Function

The append_to_buf command or function appends the current region to a buffer.

Command Format: [{ESC}n] {ESC} X append_to_buf
or
[{ESC}n] {CTRL-X} A

Function Format: (append_to_buf [n [b]])

Argument: The argument n, if specified, must be an integer value. The argument b, if specified, must be a string value.

Action: If the argument b is not specified, then EMACS prompts you with "buffer name:". The string that you type is assigned to the variable b.

The string variable b is interpreted as a buffer name. EMACS appends the current region to that buffer. This means that the text in the current region is inserted at the end of that buffer.

If n is not specified, then the text in the current region is deleted, meaning that the append operation is in effect a move. If n is specified, then the text is copied, and the marked region is not deleted.

append_to_file Command

The append_to_file command appends the current region to a file.

Format: [{ESC}n] {ESC} X append_to_file
or
[{ESC}n] {CTRL-X} {CTRL-Z} A

Argument: The argument n, if specified, must be an integer value.

Action: The append_to_file command prompts you for a file name, and then appends the current region to that file. This means that the text in the current region is inserted at the end of that file.

If the argument n is not specified, then EMACS deletes the text in the current region. This means that the append operation is, in effect, a move.

If the argument n is specified, then EMACS copies the text without deleting the marked region.

apply Function

The apply function is a standard LISP function that applies a function to a list of arguments.

Format: (apply f lst)

Arguments: The value of f must be a function. The value of lst must be a list.

Action: EMACS applies the function f to the list of arguments in lst.

Example:

```
(setq add_elements (fsymeval '+))
(setq scores '(1 2 5))
(apply add_elements scores)
```

The apply function returns the value 8, obtained by computing the value of (+ 1 2 5). (See fsymeval.)

apropos Command and Function

The apropos command or function provides an extended help facility that retrieves a list of commands that match a string.

Command Format: {ESC} X apropos

Function Format: (apropos [s])

Argument: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If s is not specified, then EMACS prompts you, with the prompt "Apropos:" in the minibuffer, for a string s.

EMACS goes through all its handlers (defcoms or built-in handlers) that have been loaded and looks for a match between s and either the name of the handler or its documentation string.

EMACS also looks for key bindings for any functions bound by the previous search.

All this information is displayed on your screen.

The apropos function returns NIL.

Note: When you use defcom to define a new command, you may use the &doc operation to specify a documentation string for the new command. The apropos command or function displays that documentation string for an appropriate argument string s.

aref Function

The aref function returns the value of an array element.

Format: (aref a x)

Argument: The argument a must be an array, and is usually a variable name to which an array has been bound by means of setq and make_array. The value of x must be a nonnegative integer less than the number of elements in the array (arrays are indexed from 0).

Action: If the value of x is 0, then aref returns the first element of the array a. In general, aref returns that element of the array a with index x, that is, the (x+1)st element of the array a.

array Data Type

A value with the array data type is created by means of the make_array function.

array_dimension Function

The array_dimension function returns the number of elements in an array.

Format: (array_dimension a)

Argument: The argument a must be an array. Usually a is a variable to which an array has been bound by means of setq and make_array.

Action: The array_dimension function returns an integer value equal to the number of elements in the array.

Example: The function

```
(array_dimension (make_array 'integer 20))
```

returns the value 20.

array_type Function

The array_type function returns the data type of an array.

Format: (array_type a)

Argument: The argument a must be an array. Usually a is a symbol to which an array value has been bound by means of setq and make_array.

Action: The array_type function returns an atom specifying the data type of the array. The data type is the same as was specified in the make_array function that created the array.

aset Function

The aset function stores a value into an array element.

Format: (aset v a n)

Argument: The argument a must be an array value. Usually a is a variable to which an array value has been bound by means of the setq and make_array functions.

The argument v must have a value whose data type is the same as the data type of the array a, as specified in the make_array function that created the array.

The argument n must have a non-negative integer value less than the number of elements in the array.

Action: If the value of n is 0, then aset sets the first element of the array a to the value v. In general, aset sets the value of the (n+1)st element of the array a to the value v.

The aset function returns the value v.

assoc Function

The assoc function looks up an item in a LISP association list.

Format: (assoc k lst)

Argument: The argument lst must be an association list, as described below. The argument k may have any data type.

Action: An association list is really a list of lists. The car of each sublist is called the "key", and the cdr of each sublist is the value associated with the key. The assoc function returns the sublist associated with the key k in the association list lst. If key k is not found in the association list, assoc returns NIL.

Example: The following example illustrates various uses of the assoc function:

```
(setq joe '((name "Joe Smith") (age 27)
                         (phone "234-1234")))
(assoc 'phone joe)         -> (phone "234-1234")
(assoc 'age joe)           -> (age 27)
(assoc 'quux joe)          -> ()
```

assure_character Function

The assure_character function returns the next keyboard character typed without removing it from the input buffer.

Format: (assure_character [raw])

Argument: The argument raw, if specified, must be the atom raw.

Action: The assure_character function waits until there is a typed character in the input buffer. It then returns the value of that character, without removing that character from the input buffer.

If the argument raw is specified, then a "raw read" is performed, and no special handling, such as "help on tap", is supported.

Example: The function

(assure_character raw)

does a raw read to input the next character from the terminal without removing it from the input buffer.

at_white_char Function

The at_white_char function returns a Boolean value indicating whether the cursor is at a whitespace character.

Format: (at_white_char)

Arguments: None.

Action: The at_white_char function returns a Boolean value. The value is true if the character to the right of the cursor is in the string bound to the atom whitespace; otherwise, it is false.

atom Function

The atom function is a standard LISP function that indicates whether or not the argument is an atom.

Format: (atom v)

Argument: The argument v may have any data type.

Action: The atom function returns a Boolean value. The value is true if the argument v is not a list structure, that is, neither a cons nor NIL.

autoload_lib Function

The autoload_lib function fasloads a file and executes the given command.

Format: (autoload_lib c s)

Arguments: The argument c must be an atom representing the name of a command. The argument s must be a string.

Action: The string specified by the argument s must be a PRIMOS pathname for a package containing a redefinition of the atom specified by the argument c. EMACS opens for input the filename obtained by adding the suffix .EFASL to the string s, and then loads and executes that file as a fasload file, thereby executing the command given by the atom c. If EMACS cannot find the .EFASL file, it looks for the corresponding .EM file instead, and compiles it with the pl command.

The autoload_lib function returns the value NIL.

Note: This function is useful when you have an external command for which you wish to defer the loading until it is actually used.

back_char Command and Function

The back_char command or function moves the cursor back by a specified number of characters.

Command Format: [{ESC}n] {ESC} X back_char
or
[{ESC}n] {CTRL-B}

Function Format: (back_char [n])

Argument: The argument n, if specified, must be an integer.

Action: If n is not specified, then let n equal 1.

If the value of n is 0, then no action takes place.

If the value of n is positive, then the cursor is moved back n characters, stopping if the beginning of the buffer is reached.

If the value of n is negative, then the cursor is moved forward (-n) characters, stopping if the end of the buffer is reached.

The back_char function returns the value NIL.

back_page Command and Function

The back_page command or function moves the window back a group of lines, usually 18.

Command Format: [{ESC}n] {ESC} X back_page
or
[{ESC}n] {ESC} V

Function Format: (back_page [n])

Argument: The argument n, if specified, must be an integer.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, the window is moved back n pages, stopping if the beginning of the buffer is reached. The cursor is left at approximately the middle of the window.

If the value of n is negative, the window is moved forward (-n) pages, stopping if the beginning of the buffer is reached. The cursor is left at approximately the middle of the window.

The back_page function returns the value NIL.

back_tab Command and Function

The back_tab command or function moves the cursor back by a specified number of tab stops.

Command Format: [{ESC}n] {ESC} X back_tab

Function Format: (back_tab [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, the cursor is moved back n tab stop positions, stopping if the beginning of the line is reached.

If the value of n is negative, the cursor is moved forward (-n) tab stop positions. If the end of the line is reached, EMACS automatically fills the end of the line with blank characters, up to the desired tab stop position.

The back_tab function returns the value NIL.

back_to_nonwhite Command and Function

The back_to_nonwhite command or function puts the cursor onto the first nonwhite character on a line. (A nonwhite character is any character not bound to the atom whitespace.)

Command Format: [{ESC}n] {ESC} X back_to_nonwhite
or
[{ESC}n] {ESC} M

Function Format: (back_to_nonwhite [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If n does not equal 0 or 1, do the following:

- If n is greater than 1, execute

  (next_line (1- n))

- If n is less than 0, execute

  (next_line n)

EMACS then moves the cursor back or forward to the first nonwhite character on the current line. If there are no nonwhite characters on the line, EMACS moves the cursor to the end of the line.

The back_to_nonwhite function returns the value NIL.

back_word Command and Function

The back_word command or function moves the cursor back by a specified number of words.

Command Format: [{ESC}n] {ESC} X back_word
or
[{ESC}n] {ESC} B

Function Format: (back_word [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is position, EMACS moves the cursor back to the nth occurrence of a character that begins a word, as specified by the list of characters in the whitespace string. (That is, EMACS moves the cursor back n words, and leaves it at the first character on that word.) Cursor movement stops if the beginning of the buffer is reached.

If the value of n is negative, then EMACS moves the cursor forward to the nth occurrence of a character that immediately follows a word. (That is, EMACS moves the cursor ahead n words, leaving the cursor on the whitespace character immediately following the end of the word.) Cursor movement stops if the end of the buffer is reached.

The back_word command returns the value NIL.

Note: The token_chars atom contains the list of characters that define a word or token.

backward_clause Command and Function

The backward_clause command or function moves the cursor back by a specified number of clauses.

Command Format: [{ESC}n] {ESC} X backward_clause
or
[{ESC}n] {CTRL-X} {CTRL-Z} {CTRL-A}

Function Format: (backward_clause [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor back to the character preceding the nth occurrence of a character that delimits a clause. (That is, EMACS moves the cursor back n clauses, and leaves it at the character preceding the last clause delimiter.) Cursor movement stops if the beginning of the buffer is reached.

If the value of n is negative, EMACS moves the cursor forward to the nth occurrence of a character that delimits a clause. (That is, EMACS moves the cursor ahead n clauses, leaving the cursor on the character immediately following the last clause delimiter.) Cursor movement stops if the end of the buffer is reached.

The backward_clause command returns the value NIL.

Note: The characters delimiting a clause are contained in the global string variable clause_scan_table$.

backward_clausef Function

The backward_clausef function moves the cursor back by a specified number of clauses, and returns a Boolean value indicating whether the operation was successful.

Format: (backward_clausef [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

The backward_clausef function returns a Boolean value.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no cursor movement takes place, and backward_clausef returns the value true.

If the value of n is positive, EMACS moves the cursor back to the character preceding the nth occurrence of a character that delimits a clause. (That is, EMACS moves the cursor back n clauses, and leaves it at the character preceding the last clause delimiter.) Cursor movement stops if the beginning of the buffer is reached. If the beginning of the buffer is reached, backward_clausef returns the value false; otherwise, backward_clausef returns the value true.

If the value of n is negative, EMACS moves the cursor forward to the nth occurrence of a character that delimits a clause. (That is, EMACS moves the cursor ahead n clauses, leaving the cursor on the character following the last clause delimiter.) Cursor movement stops if the end of the buffer is reached. If the end of the buffer is reached, backward_clausef returns the value false; otherwise, backward_clausef returns the value true.

Note: The list of characters delimiting a clause may be found in the global string variable clause_scan_table$.

backward_kill_clause Command and Function

The backward_kill_clause command or function kills text from the current cursor position to the beginning of the specified clause.

Command Format: [{ESC}n] {ESC} X backward_kill_clause
or
[{ESC}n] {CTRL-X} {CTRL-Z} {CTRL-H}

Function Format: (backward_kill_clause [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If the value of n is 0, no action takes place.

If the value of n is not 0, EMACS performs the following operations:

```
(mark)
(backward_clause n)
(forward_char 2)
(kill_region)
```

Notice that when n is positive, backward_kill_clause kills all characters back to the character following the last clause delimiter found by the backward_clause operation.

> **Caution**
>
> If n is negative, backward_clausef deletes all characters up to and including the last clause delimiter found, plus two additional characters of the following clause.

The backward_kill_clause function returns the value NIL.

backward_kill_line Command and Function

The backward_kill_line command or function kills all text from the current cursor position back to the beginning of the line.

Command Format: [{ESC}n] {ESC} X backward_kill_line
or
[{ESC}n] {CTRL-X} {CTRL-K}

Function Format: (backward_kill_line [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 0.

EMACS deletes all characters from the current cursor position back to the beginning of the line. In addition, if the value of n is not 0, EMACS deletes the newline character preceding the beginning of the line.

The backward_kill_line function returns the value NIL.

backward_kill_sentence Command and Function

The backward_kill_sentence command or function kills text from the current cursor position to the beginning of the specified sentence.

Command Format: [{ESC}n] {ESC} X backward_kill_sentence
or
[{ESC}n] {CTRL-X} {CTRL-H}

Function Format: (backward_kill_sentence [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is not 0, EMACS performs the following actions:

```
(mark)
(back_sentence n)
(kill_region)
```

The backward_kill_sentence function returns the value NIL.

backward_para Command and Function

The backward_para command or function moves the cursor back by a specified number of paragraphs.

Command Format: [{ESC}n] {ESC} X backward_para
or
[{ESC}n] {CTRL-X} [

Function Format: (backward_para [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor back to the character preceding the nth occurrence of a character that delimits a paragraph. (That is, EMACS moves the cursor back n paragraphs, and leaves it at the character preceding the last paragraph delimiter.) Cursor movement stops if the beginning of the buffer is reached.

If the value of n is negative, EMACS moves the cursor forward to the (-n)th occurrence of a character that delimits a paragraph. (That is, EMACS moves the cursor ahead (-n) paragraphs, leaving the cursor on the character immediately following the last paragraph delimiter.) Cursor movement stops if the end of the buffer is reached.

The backward_para command returns the value NIL.

Note: Paragraphs are defined as lines beginning with a period, a blank line, or a space.

backward_sentence Command and Function

The backward_sentence command or function moves the cursor back by a specified number of sentences.

Command Format: [{ESC}n] {ESC} X backward_sentence
or
[{ESC}n] {ESC} A

Function Format: (backward_sentence [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor back to the character preceding the nth occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor back n sentences, and leaves it at the character preceding the last sentence delimiter.) Cursor movement stops if the beginning of the buffer is reached.

If the value of n is negative, EMACS moves the cursor forward to the (-n)th occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor ahead (-n) sentences, leaving the cursor on the character immediately following the last sentence delimiter.) Cursor movement stops if the end of the buffer is reached.

The backward_sentence command returns the value NIL.

Note: The characters delimiting a sentence are in the global string variable sentence_scan_table$.

backward_sentencef Function

The backward_sentencef function moves the cursor back by a specified number of sentences, and returns a Boolean value indicating whether the operation was successful.

Format: (backward_sentencef [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

The backward_sentencef function returns a Boolean value.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no cursor movement takes place, and backward_sentencef returns the value true.

If the value of n is positive, EMACS moves the cursor back to the character preceding the (-n)th occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor back (-n) sentences, and leaves it at the character preceding the last sentence delimited.) Cursor movement stops if the beginning of the buffer is reached. If the beginning of the buffer is reached, then backward_sentencef returns the value false; otherwise, backward_sentencef returns the value true.

Note: The characters delimiting a sentence are in the global string variable sentencef_scan_table$.

balbak Command and Function

The balbak command or function moves the cursor to the opening parenthesis of the current level.

Command Format: [{ESC}n] {ESC} X balbak
or
[{ESC}n] {ESC} {CTRL-B} (LISP mode only)

Function Format: (balbak [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, let n equal 1.

If n is greater than or equal to 0, EMACS proceeds as follows:

- If point is at a closing parenthesis, EMACS moves point to the corresponding opening parenthesis.
- If point is on text, EMACS moves point back to the last closing parenthesis, and then back again to the corresponding opening parenthesis. This means that if point is between statements or forms, EMACS skips back over the last form or statements.

If the value of n is negative, EMACS executes:

(balfor)

balfor Command and Function

The balfor command or function moves the cursor to the closing parenthesis of the current level.

Command Format: [{ESC}n] {ESC} X balfor
or
[{ESC}n] {ESC} {CTRL-F} (LISP mode only)

Function Format: (balfor arg)

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If the value of n is greater than or equal to 0, EMACS proceeds as follows:

- If point is at an opening parenthesis, EMACS moves point to the corresponding closing parenthesis.
- If point is on text, EMACS moves point forward to the next opening parenthesis, and then forward again to the corresponding closing parenthesis. This means that if point is between statements or forms, EMACS skips point forward over the next form or statement.

If the value of n is negative, EMACS executes:

```
(balbak)
```

begin_line Command and Function

The begin_line command or function moves the cursor back to the start of the line.

Command Format: {ESC} X begin_line
or
{CTRL-A}

Function Format: (begin_line [arg])

Argument: The optional argument, if specified, is ignored.

Action: EMACS moves the cursor to the beginning of the line.

The begin_line function returns the value NIL.

beginning_of_buffer_p Function

The beginning_of_buffer_p function tests whether the current cursor is at the beginning of the buffer.

Format: (beginning_of_buffer_p)

Argument: None.

Action: The beginning_of_buffer_p function returns a Boolean value. The value is true if and only if the current cursor is at the beginning of the buffer.

beginning_of_line_p Function

The beginning_of_line_p function tests whether the current cursor is at the beginning of the line.

Format: (beginning_of_line_p)

Argument: None.

Action: The beginning_of_line_p function returns a Boolean value. The value is true if and only if the current cursor is at the beginning of the line.

bolp Function

The bolp function is an abbreviation of beginning_of_line_p.

Boolean Data Type

A variable with the Boolean data type can have only the values true and false.

buffer_info Function

The buffer_info function either gets or sets information about the current buffer.

Format: (buffer_info p [v])

Arguments: The argument p must have as a value the atom or symbol representing the property to be accessed or changed. A complete list of the valid property names is given below, under action.

The argument v, if specified, must have a data type that is compatible with the property p.

Action: EMACS sets or accesses a specified buffer value. When a value is changed, the function returns the previous value. The argument p is the name of the value to be accessed or changed. The second argument, if specified, is the value to be assigned to the property p.

The legal values for the property p are as follows:

Property	Data Type	Meaning
name	string	Name of buffer. This property may not be modified.

default_file	string	The pathname of the default file associated with the buffer.
modified	Boolean	True if the buffer has been modified. This is indicated by a * on the status line.
modes	list	The list of the modes associated with this buffer.
read_only	Boolean	True if the buffer cannot be modified.
changed_ok	Boolean	If true, the user is allowed to quit the editor with {CTRL-X} {CTRL-C} even if this buffer has been changed.
dont_show	Boolean	If true, the buffer name is suppressed in the {CTRL-X} {CTRL-B} listing.
two_dimensional	Boolean	If true, specifies 2don mode is on.
fill_column	integer	Fill column for word wrapping.
mark	cursor	The current marked position in the buffer.
top_cursor	cursor	Pointer to the beginning of the buffer.
bottom_cursor	cursor	Pointer to the end of the buffer.
user		(See below.)

The "user" option is used for extended values as in:

```
(buffer_info (user frob) 17)
```

This indicates a user variable. It is actually the car of a cons whose cdr is a user-defined per buffer value. The cdr may be the same as the name of a request value, in which case it is identical to using the name directly without "user".

The following example returns or sets the list of modes associated with the current buffer. Note that in the example, mode is an unquoted atom as in:

```
(buffer_info modes (list (find_mode "quux")))
```

So to add a new mode:

```
(buffer_info modes (append (buffer_info modes)
                   (find_mode newmode)))
```

or to remove a mode:

```
(buffer_info modes (remove (find_mode newmode)
                   (buffer_info modes)))
```

Note that the order of the modes in the list is significant!

buffer_name Function

The buffer_name function returns the name of the associated buffer.

Format: (buffer_name c)

Argument: The argument c must have the data type of cursor.

Action: The buffer_name function returns a string value. The string contains the name of the buffer in which the cursor c lies.

capinitial Command and Function

The capinitial command or function changes the first character of a word, if it is a letter, to uppercase.

Command Format: [{ESC}n] capinitial
or
[{ESC}n] {ESC} C

Function Format: (capinitial [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is negative, EMACS sets n=-n, moves the cursor back n words, and proceeds as in the following paragraph.

If the value of n is positive, EMACS starts with the current word (or the next word, if the cursor is on whitespace) and converts the first character of that word to uppercase, if it is a lowercase letter.

The capinitial function returns the value NIL.

car Function

The car LISP function returns the first item in a list.

Format: (car lst)

Argument: The argument lst must be a list.

Action: If lst is a null list, car returns a null list.

If lst is not a null list, car returns the first item of the list. The data type of the value returned by car equals the data type of the first item in the list.

case? Command and Function

The case? command and function displays a message telling whether uppercase and lowercase letters are distinguished in searching.

Command Format: {ESC} X case?

Function Format: (case?)

Argument: A numeric argument, if specified, is ignored.

Action: If cases are being distinguished, the message "Cases are looked at when searching" is displayed. Otherwise, the message "Cases are ignored when searching" is displayed.

The case? function returns the value NIL.

case_off Command and Function

The case_off command causes uppercase and lowercase letters to be treated identically during searches.

Command Format: {ESC} X case_off

Function Format: (case_off)

Argument: A numeric argument, if specified, is ignored.

Action: In subsequent search operations, EMACS ignores the distinction between corresponding uppercase and lowercase letters.

The case_off function returns the value NIL.

case_on Command and Function

The case_on command causes uppercase and lowercase letters to be distinguished during searches.

Command Format: {ESC} X case_on

Function Format: (case_on)

Argument: A numeric argument, if specified, is ignored.

Action: In subsequent search operations, EMACS observes the distinction between uppercase and lowercase letters.

The case_on function returns the value NIL.

case_replace? Command and Function

The case_replace? command and function displays a message telling whether uppercase and lowercase letters are distinguished in replacing.

Command Format: {ESC} X case_replace?

Function Format: (case_replace?)

Argument: A numeric argument, if specified, is ignored.

Action: If cases are being distinguished, the message "Cases are looked at in replace" is displayed. Otherwise, the message "Cases are ignored in replace" is displayed.

The case_replace? function returns the value NIL.

case_replace_off Command and Function

The case_replace_off command causes uppercase and lowercase letters to be treated identically during replace operations.

Command Format: {ESC} X case_replace_off

Function Format: (case_replace_off)

Argument: A numeric argument, if specified, is ignored.

Action: In subsequent replace operations, EMACS ignores the distinction between corresponding uppercase and lowercase letters.

The case_replace_off function returns the value NIL.

case_replace_on Command and Function

The case_replace_on command causes uppercase and lowercase letters to be distinguished during replace operations.

Command Format: {ESC} X case_replace_on

Function Format: (case_replace_on)

Argument: A numeric argument, if specified, is ignored.

Action: In subsequent replace operations, EMACS observes the distinction between uppercase and lowercase letters.

The case_replace_on function returns the value NIL.

catch Function

The catch function is the same as the *catch function, except that the argument order is different.

Format: (catch body tag)

This is like *catch, except that the body and tag arguments are evaluated. Because catch's argument order makes programming quite difficult, the use of *catch is recommended instead. (See *catch for further information.)

catenate Special Form

The catenate special form concatenates its string arguments.

Format: (catenate s1 [s2 ... s8])

Arguments: The catenate special form takes at least one argument and no more than eight arguments. All arguments must have the string or character data type.

Action: The catenate function returns a string value.

If there is only one argument, s1, the value of s1 is returned.

If there is more than one argument, catenate concatenates together all the argument string values and returns the combined string.

Example: The function

```
(catenate "hi" "there")
```

returns the string value "hithere".

cdr Function

The cdr LISP function returns the value of the list argument with the first item removed.

Format: (cdr lst)

Argument: The argument lst must be a list.

Action: The cdr function returns a list value.

If lst is a null list or a list containing only one value, cdr returns a null list.

If lst is a list containing more than one item, cdr returns a list containing all items in lst except the first.

center_line Command and Function

The center_line command or function centers one or more lines of text.

Command Format: [{ESC}n] {ESC} X center_line
or
[{ESC}n] {CTRL-X} {CTRL-Z} S

Function Format: (center_line [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, then n lines of text, beginning with the current one and continuing forward, are centered.

If the value of n is negative, then -n lines of text, beginning with the current one and proceeding backward, are centered.

To center a line of text, EMACS proceeds as follows:

- If the line is null (contains no characters), no action takes place. Otherwise:
- EMACS removes all leading whitespace from the beginning of the line. Then, let k equal the number of characters of text remaining on the line.
- Let f equal the value returned by

  (buffer_info fill_column)

  If that value is 0 or undefined, let f equal 70.
- EMACS inserts (f-k)/2 spaces at the beginning of the line.

The center_line function returns the value NIL.

char_to_string Function

The char_to_string conversion function converts a character argument into a string of length one.

Format: (char_to_string c)

Argument: The argument c must have the character data type.

Action: The char_to_string function returns a string value of length one. The string is computed by converting the character c to a string.

character Data Type

A variable with the character data type can have as a value any character.

character_argument Function

The character_argument function returns the character argument to the current command.

Format: (character_argument)

Argument: None.

Action: The character_argument function returns a value with a character data type. The value is the character argument to the current command; that is, the last character of the keypath used to invoke the command.

charp Function

The charp function tests its argument to determine whether it has the character data type.

Format: (charp arg)

Argument: The argument arg may be any data type.

Action: The charp function returns a Boolean value. The value is true if the argument arg has the character data type; it is false if the argument has a different data type.

clear_and_say Function

The clear_and_say function is the same as the init_local_displays function.

close_paren Command and Function

The close_paren command or function moves the cursor briefly to the opening parenthesis that corresponds to the closing parenthesis just typed.

Command Format: {ESC} X close_paren
or
) (LISP mode only)

Function Format: (close_paren)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS takes the following action:

- Inserts a closing parenthesis at the current point.
- Moves the cursor to the opening parenthesis that corresponds to the closing parenthesis just typed.
- After a pause, returns the cursor to the position immediately following the closing parenthesis just typed.

This command aids you in typing LISP or PEEL programs by helping you match parentheses in LISP mode.

Note: The variable lisp.paren_time controls the duration of the pause at the opening parenthesis. It is normally 750 milliseconds, but can be changed to another value if desired. A value of 0 turns off close_paren.

collect_macro Command

The collect_macro command starts collecting keystrokes typed at the terminal, in order to define a macro.

Command Format: {ESC} X collect_macro
or
{CTRL-X} (

Argument: A numeric argument, if specified, is ignored.

Action: EMACS begins to collect keystrokes typed at the terminal.

Note: Collection of keystrokes is terminated by the finish_macro command, which is bound as {CTRL-X}). The resulting macro may be executed by the execute_macro command, which is bound as {CTRL-X} E.

Be aware that you should not expect this command to work too well when imbedded in a function!

command_abort_handler Atom

The command_abort_handler atom is used in conjunction with the with_command_abort_handler function, which executes its arguments up to the command_abort_handler atom. If an error is encountered, the error flags (and throws) are reset, and execution continues after the command_abort_handler token.

(See the description of with_command_abort_handler for further information.)

cons Function

The cons function is a LISP function that adds a new item to the front of a list.

Format: (cons i lst)

Argument: The argument i is an item with any data type. The argument lst is almost always a list, but may have any data type if you wish to produce a "dotted list".

Action: The data type of the value returned by cons is a list.

If the argument lst is a list, cons returns the list formed by adding the item i to the front of the list lst.

If the argument lst is not a list, cons returns a dotted list (or cons) whose car is the item i and whose cdr is the item lst.

convert_tabs Function

The convert_tabs function takes a list of numbers and converts them to tab stops.

Format: (convert_tabs s)

Argument: The argument s must be a string.

Action: The string s must contain a series of numbers separated by spaces, and the last number must end in a space. The numbers must be positive and in increasing order.

EMACS sets the tab stops at positions specified by numbers in the string.

The convert_tabs function returns the value NIL.

convert_to_base Function

The convert_to_base function converts an integer to a specified numeric base and returns the result as a string.

Format: (convert_to_base n bv [ln])

Argument: The argument n must be an integer. The argument bv must be either a string or a positive integer. The argument ln, if specified, must be an integer.

Action: The convert_to_base function returns a string value.

EMACS proceeds as follows:

- If bv is an integer value, let b equal bv; otherwise, let b equal the length of string bv.
- If bv is a string value, let s equal bv; otherwise, let s equal the first b characters of the string "0123456789abcdefghijklmnopqrstuvwxyz"
- EMACS converts the integer n to a string of base b, using as digits the characters of the string s. Let t be the resulting string of digits.
- Let czero equal the first character of the string s. (This is usually the character "0".)
- If the argument ln is specified, and if the value of ln is longer than the string t, EMACS inserts additional czero characters to the front of the string t, but after a -, if any, so that the length of the string t equals the value of ln.
- If the argument ln is specified and is smaller than the length of the string t, EMACS sets t equal to a string of length ln containing only *'s.

EMACS returns the string t.

Examples:

```
(convert_to_base 255 16)

(convert_to_base 32 8 3)
```

These return the string values "ff" and "040", respectively.

copy_array Function

The copy_array function copies one array to another.

Format: (copy_array a1 a2 [idx1 [n [idx2]]])

Arguments: The arguments a1 and a2 must be array values. Usually, a1 and a2 are assigned array values by setq in conjunction with the make_array function.

The arguments idx1, n, and idx2, if specified, must all be integer values.

Action: The copy_array function returns an array value. This array value is computed as follows:

- Let m1 and m2 equal the number of elements in the arrays a1 and a2, respectively. m1 and m2 are the values specified as the last argument to the make_array function calls that created the arrays a1 and a2.
- If the argument idx1 is not specified, let idx1 equal 0.
- It is an error if idx1 is <0 or >=m1.
- If the argument n is not specified, let n = m1.
- It is an error if either n<0 or idx1 + n >= m1.
- If the argument idx2 is not specified, let idx2 = 0.
- It is an error if idx2 + n >= m2.
- EMACS copies n consecutive elements from the array a2 to the array a1. The n consecutive elements are taken from the array a2, starting at index position idx2, and are copied to the array a1, starting at index position idx1.

The copy_array function returns the array a1.

copy_cursor Function

The copy_cursor function returns a copy of the specified cursor.

Format: (copy_cursor cur)

Argument: The argument cur must be a cursor value.

Action: The copy_cursor function returns a cursor value.

EMACS makes a copy of the cursor value specified by the argument cur, and returns that copy.

Example:

```
(setq old (copy_cursor current_cursor)
(forward_word) ...
(go_to_cursor old)
```

The first line makes a copy of the current cursor position, and assigns that to the variable old. The last line returns the cursor to the original cursor position.

Note: Replacing the first line in the preceding example with

```
(setq old current_cursor)
```

would not have worked, because any cursor movement would have changed the values of both current_cursor and old.

copy_region Command and Function

The copy_region command or function copies the current region into the kill ring.

Command Format: {ESC} X copy_region
or
{ESC} W

Function Format: (copy_region)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS copies the characters between the current cursor and the last marked position in the current buffer into the kill ring without deleting them from the buffer.

The copy_region function returns the value NIL.

cpu_time Function

The cpu_time function returns the current cpu time in milliseconds.

Format: (cpu_time)

Argument: None.

Action: The cpu_time function returns an integer value. The integer value is the cpu time used since login.

cr Command and Function

The cr command or function inserts a newline into the text buffer.

Command Format: [{ESC}n] cr
or
[{ESC}n] {CTRL-M} (carriage return key)

Function Format: (cr [n])

Argument: If the argument n is specified, it must be an integer.

Action: If the argument n is not specified, let n equal 1.

If n is less than or equal to 0, no action takes place.

If n is greater than 0, EMACS inserts n newline characters into the text buffer at the current cursor position. The cr function returns the value NIL.

create_text_save_buffer$ Function

[This is pretty useless, as you cannot determine which buffer it has written into!

The create_text_save_buffer$ function goes to or creates the next buffer in a circular list of ten buffers.

{Format: (create_text_save_buffer$ s)

Argument: The argument s must be a string value.

Action: EMACS goes to or creates the next buffer in a circular list of ten buffers. EMACS deletes the contents of that buffer, and inserts the string s into that buffer.

The create_text_save_buffer$ function returns the value NIL.

cret_indent_relative Command and Function

The cret_indent_relative command or function inserts a new-line character into the text buffer at the current cursor position, and then indents the next line to the first nonwhitespace character of the previous line.

Command Format: [{ESC}n] {ESC} X cret_indent_relative
or
[{ESC}n] {CTRL-X} {RETURN}

Function Format: (cret_indent_relative [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n=0.

Let k equal the number of whitespace characters at the beginning of the current line. If the line is blank, k=0.

EMACS inserts a newline and k spaces into the text buffer at the current cursor position

The cret_indent_relative function returns the value NIL.

cur_hpos Function

The cur_hpos function returns an integer value indicating the current horizontal position of the cursor.

Format: (cur_hpos)

Arguments: None.

Action: The cur_hpos function returns an integer value. The value returned equals one plus the number of characters on the current line to the left of the current cursor position.

current_character Function

The current_character function returns the character at the cursor.

Format: (current_character [cur])

Argument: The argument cur, if specified, must be a cursor value.

Action: The current_character function returns a character value.

If the argument cur is not specified, let cur equal the current cursor.

The current_character function returns the character at the specified cursor position.

current_cursor Variable

The current_cursor variable has the data type cursor and equals the value of the current cursor position.

Note: The current_cursor variable is not normally used with parentheses, because it is not a function. If you bind a variable to current_cursor, the new variable also changes value whenever the current cursor changes. For example,

```
(setq cur current_cursor)
```

binds the variable cur to the current cursor, with the result that any cursor movement command changes the value of cur and current_cursor. To make a copy of a cursor that will not change in this manner, use the copy_cursor function.

> **Caution**
>
> You should never set current_cursor. EMACS may become very confused.

current_handler Function

The current_handler function returns a string value representing the current handler.

Format: (current_handler [chase_atom])

Argument: If an argument is specified, it must be the atom chase_atom.

Action: The current_handler function returns a string value.

This is the object for the current command handler and is used with handler_info. If chase_atom is specified, the function cell of the atom is returned if the object is an atom. This is normal usage.

(See handler_info for more information.)

current_line Function

The current_line function returns a string value containing the current line without the line-ending newline.

Format: (current_line [cur])

Argument: The argument cur, if specified, must have a cursor value.

Action: The current_line function returns a string value.

If the argument cur is not specified, let the value of cur be the current cursor.

EMACS forms a string containing all the characters on the line pointed to by the cursor cur, except the line-ending newline, and returns that string value.

current_major_window Function

The current_major_window function returns the current major window.

Format: (current_major_window)

Arguments: None.

Action: The current_major_window returns a value with the window data type. The value returned is the current major window.

Note: This is the window in which the current_cursor will be redisplayed. You can use the value returned by current_major_window in the go_to_window function. Incidentally, the minibuffer is not a major window.

cursor Data Type

Cursor is a data type. Cursors are pointers to text in buffers. When you assign a cursor value to a variable, such as by using setq with the copy_cursor function, EMACS attempts to make that cursor continue to point to the same text in the buffer, no matter what other changes in text are made to the buffer. A number of operations may be performed on cursors, but most operations operate on the current cursor, which is also the user's cursor. For example, forward_char advances the current cursor forward by one character. It is rather easy to save the current cursor position (using copy_cursor and setq), execute a series of commands that may alter it, and then use go_to_cursor to return the cursor to its old value.

cursor_info Function

The cursor_info function returns information about the current cursor position.

Format: (~~cursor_info~~ p) (cursor-info cursor p [nval])

Arguments: The argument p must be an atom, as described below.

Action: EMACS returns a value whose data type depends upon the atom p.

EMACS returns information about the current cursor position that depends upon the property specified by the atom p. The atom p may have any of the following values:

p	Data Type	Property
buffer_name	string	Name of the buffer
line_num	integer	Line number
char_pos	integer	Character position on line
sticky	Boolean	Cursor moves with text

incorrect. Note: You cannot set any of these properties using cursor_info. Use the make_cursor function to create a new cursor value.

optional 3rd arg can be used to set sticky at least.

cursor_on_current_line_p Function

The cursor_on_current_line_p function returns a Boolean value indicating whether the current cursor is on the same line as the argument cursor.

Format: (cursor_on_current_line_p cur)

Argument: The argument cur must be a cursor value.

Action: The cursor_on_current_line_p function returns a Boolean value. The value is true if the current cursor is on the same line as the argument cur, and is false if the current cursor is on a different line.

cursor_same_line_p Function

The cursor_same_line_p function returns a Boolean value indicating whether two cursors point to the same line.

Format: (cursor_same_line_p cur1 cur2)

Arguments: The arguments cur1 and cur2 must be cursor values.

Action: The cursor_same_line_p function returns Boolean values. The value is true if cur1 and cur2 refer to the same line, and false if they refer to different lines.

Data Types

The PEEL data types are:

1 any	2 Boolean	3 character
4 integer	5 string	6 atom
7 function	8 list	9 cursor
10 buffer	11 dispatch_table	12 handler
13 buffer structure	14 window	15 array
16 PL/I subroutine		

date Command and Function

The date command or function inserts a date into your text buffer at the current cursor position.

Command Format: {ESC} X date

Function Format: (date)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the current date into your text buffer at the current cursor position. The date format is illustrated by the following:

```
TUE, 19 MAY 1981
```

The date function returns the value NIL.

decimal_rep Function

The decimal_rep function is the same as the integer_to_string function.

decompose Special Form

The decompose special form is a special LISP function for arranging the contents of one list into a pattern specified by another list.

Format: (decompose v f s1 else s2)

Arguments: The arguments v and f must be list values. The arguments s1 and s2 are any PEEL code strings.

Action: EMACS evaluates the argument v, but not the argument f.

EMACS then matches the elements of the list f with corresponding elements in the list v, proceeding as follows:

- If the item in list f is NIL, the item in list v must also be NIL.
- If the item in f is an atom, EMACS binds it locally to the corresponding item in the list v.
- If the item is a list, EMACS binds the car of this item to the car of the corresponding item in v, and the cdr of the item to the cdr of the corresponding item in v.

If these local assignments are made without error, EMACS executes the code sequence s1. If an error occurs during the matching, EMACS executes the sequence s2.

```
(decompose '(a b c) (x . y)
           (print x)
           (print y)
       else
           (print "DECOMPOSITION ERROR"))
```

This prints: a
             (b c)

```
(decompose '((a b) (c d)) (x (y z))
           (print (list x y z))
       else
           (print "DECOMPOSITION ERROR"))
```

This prints: ((a b) c d)

Note: This function is particularly useful for writing macros. (See the &macro option of defun.)

def_auto Function

The def_auto function makes another function (defined in a separate file) available for automatic loading when it is invoked.

Format: (def_auto f h s)

Arguments: The argument *f* must be an atom. The arguments *h* and *s* must be string values.

Action: EMACS defines the function *f* without actually loading the function. The function is thus available for use, but the time required to load the function is not spent until it is invoked.

The string *h* is saved as a help string for the function.

The string *s* is the pathname of the .EFASL file containing the actual function definition. The first time the function is invoked, EMACS goes to the file specified by the pathname *s* and fasloads that file, replacing the definition of the function and then executing it. Subsequent invocations of the function use the fasloaded definition directly.

The def_auto function returns the value NIL.

Example:

```
(def_auto poem
        "Anthony's program to write poetry"
        "ANTHONY>EMACS>POEM")
```

This statement defines poem as an EMACS command to load and then execute the file ANTHONY>EMACS>POEM.EFASL. Presumably a file named POEM.EM, containing the actual defcom for the "poem" command, had previously been compiled, by using the dump_file command.

default_tabs Command and Function

The default_tabs command or function restores tab positions to every five spaces.

Command Format: {ESC} X default_tabs

Function Format: (default_tabs)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS restores the default tab positions, set at every five spaces.

The default_tabs function returns the value NIL.

defcom Special Form

The defcom special form defines a new command.

Format:

```
(defcom command_name
     &doc  | &documentation <documentation_string>
     &na   | &numeric_arg
             &repeat | &pass <variable> [&default <value>]
     &args | &arguments ((<name> &prompt <string>
                                 &default <value>
                                 &string | &symbol | &integer )
                         ... )
     &prefix
     &chararg | &character_argument
     <body>
)
```

Arguments: (See Chapter 5.)

Action: EMACS defines a new command, as described in Chapter 5.

The defcom function returns the value NIL.

defun Special Form

The defun special form defines a new function.

Format:

```
(defun name ((<argument1><type1>) ...
              &optional ... &rest ... &quote ...
              &returns <type>
              &local (<variable1><type2>) ... )
        <body>
)
```

Arguments: (See Chapter 5.)

Action: The defun function defines a new function, as described in Chapter 5.

The defun function returns the value NIL.

You may also specify defun with the argument &macro. In this case, the function returns a list that should be evaluated in the calling context. (See page 127 of the second edition of Winston and Horn's LISP for an explanation of the LISP backquote-and-comma syntax used for &macro.)

Example: Consider the following definition:

```
(defun foo (x &macro)
  `(* ,x ,x))
```

In this case, reference to (foo bar) returns (* bar bar).

delete_blank_lines Command and Function

The delete_blank_lines command or function deletes all blank lines around the current cursor.

Command Format: {ESC} X delete_blank_lines
or
{CTRL-X} {CTRL-O}

Function Format: (delete_blank_lines)

Argument: A numeric argument, if specified, is ignored.

Action: If the current cursor is on a nonblank line, EMACS moves the cursor down to the first blank line, stopping if the end of the buffer is reached. EMACS then deletes blank lines until the cursor is again on a nonblank line or at the end of the buffer.

The delete_blank_lines function returns the value NIL.

delete_buffer Function

The delete_buffer function deletes the contents of a buffer without saving it on the kill ring.

Format: (delete_buffer)

Arguments: None.

Action: EMACS deletes the contents of the current buffer, without saving it on the kill ring.

The delete_buffer function returns the value NIL.

delete_char Command and Function

The delete_char command or function deletes one or more characters forward.

Command Format: [{ESC}n] {ESC} X delete_char
or
[{ESC}n] {CTRL-D}

Function Format: (delete_char [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If n is positive, EMACS deletes n characters beginning with the character at the current cursor position and continuing forward, stopping if the end of the buffer is reached.

If the value of n is negative, EMACS deletes -n characters, beginning with the character preceding the current cursor position, and continuing backward, stopping if the beginning of the buffer is reached.

If the absolute value of n is greater than 64, a prompt "Count is <n>. Are you sure?" is displayed.

The delete_char function returns the value NIL.

delete_point_cursor Function

The delete_point_cursor function deletes all the text between the current cursor position and the argument cursor position.

Format: (delete_point_cursor cur)

Argument: The argument cur must be a cursor value.

Action: EMACS deletes all text between the current cursor position and the cursor position specified by the argument cur.

The delete_point_cursor function returns the value NIL.

delete_region Command and Function

The delete_region command and function deletes the current marked region without putting it onto the kill ring.

Command Format: {ESC} X delete_region

Function Format: (delete_region)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS deletes all the text in the region between the current cursor position and the marked position. The text is not saved on the kill ring.

The delete_region function returns the value NIL.

delete_white_left Function

The delete_white_left function deletes all consecutive whitespace to the left of the current cursor position.

Format: (delete_white_left)

Arguments: None.

Action: The delete_white_left function returns a Boolean value.

If there is a whitespace character at the current cursor position and a whitespace character to the left of the current cursor position, EMACS deletes all consecutive whitespace characters to the left of the current cursor position, and returns the value true.

If those conditions are not met, EMACS takes no action and returns the value false.

delete_white_right Function

The delete_white_right function deletes all contiguous whitespace to the right of the current cursor position.

Format: (delete_white_right)

Arguments: None.

Action: The delete_white_right function returns a Boolean value.

If there is a whitespace character at the current cursor position and a whitespace character to the right of the current cursor position, EMACS deletes all consecutive whitespace characters to the right of the current cursor position, and returns the value true.

If those conditions are not met, EMACS takes no action and returns the value false.

delete_white_sides Function

The delete_white_sides function deletes all whitespace characters surrounding the current position.

Format: (delete_white_sides)

Arguments: None.

Action: The delete_white_sides function returns a Boolean value.

If there is a whitespace character at the current cursor position, EMACS deletes it and all consecutive whitespace characters to the right and to the left of the current cursor position, and returns the value true.

If the character at the current cursor position is not a whitespace character, EMACS returns the value false.

Note: The delete_white_sides function has the same effect as the white_delete function.

delete_word Command and Function

The delete_word command or function deletes one or more words.

Command Format: [{ESC}n] {ESC}X delete_word
or
[ESC}n] {ESC} D

Function Format: (delete_word [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes places.

If the value of n is positive, EMACS deletes n words in the text, beginning with the word at the current cursor position and continuing forward. Deletion stops if the end of the buffer is reached.

If the value of n is negative, EMACS deletes -n words beginning with the word preceding the current cursor position, and moving backward. Deletion stops if the beginning of the buffer is reached.

The delete_word function returns the value NIL.

Note: The token_chars atom contains a list of the characters that define a word or token.

describe Command and Function

The describe command or function gives you information about an EMACS command.

Command Format: {ESC} X describe
or
{CTRL-_} D

Function Format: (describe [s])

Argument: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you for a string s.

If the string s does not begin with the character @, EMACS displays a list of all function names that begin with the string s.

If the string s begins with a single @ character, EMACS prints a list of all function names that contain the remaining characters in the string s anywhere in their name.

If the string s begins with two @ characters, EMACS prints a list of all function names for functions that reference any other function beginning with the remainder of the string s.

dispatch Special Form

The dispatch special form compares the text following the current cursor against a set of strings to determine what action to take.

Format:
```
(dispatch
    x1 s1
    x2 s2
    ...
    otherwise sx)
```

Arguments: Each argument, x1, x2, and ... may consist of one or more string or character values.

Each argument s1, s2, ..., and sx may consist of one or more PEEL statements.

Action: EMACS compares the text following point to x1, x2, and ... until a match is found. If a match is found, then the corresponding PEEL statement s1, or s2, or ... is executed, and the value of that statement is returned as the value of dispatch. Otherwise, the statement sx is executed, and the value of that statement is returned as the value of dispatch.

If no match is found and no "otherwise" statement is specified, dispatch returns the value NIL.

(See Chapter 4 for more information.)

dispatch_info Function

The dispatch_info function gets or sets information on a mode or dispatch table. It returns the old value of the property.

Format: (dispatch_info dt p [v])

Arguments: The argument dt must be a dispatch table. The argument p must be an atom, as described below under action. The argument v, if specified, must have a data type compatible with the argument p.

Action: The dispatch_info function can be used to set the handler associated with keys. A mode value is a dispatch table that is found using the find_mode function. The argument to find_mode is either an atom or a string (that is returned by the function). The returned value is the mode value. The following modes are predefined:

main	This is the main character dispatch mode.
x	This is the dispatch table for the {CTRL-X} prefix in the main dispatch table.
esc	This is the dispatch table for ESCAPE in the main dispatch table.
mb_mode	This is the dispatch table for minibuffers.
reader	This is the dispatch table used by the keyboard reader. It is used, for example, to define {CTRL-_} to be help_on_tap. If the function returns a string or character value, the result is returned in place of the character actually read. Otherwise, the reader will read another character from the keyboard. Note that the "raw" reader does not invoke reader functions.

The dispatch_info function can be used to interrogate and modify dispatch tables. The argument dt is a dispatch table (that is, a mode). The argument p is either the quoted atom 'name, in which case the name of the mode is returned as a string, or it is a character, string, or integer value identifying the entry to be interrogated and/or modified.

The argument v, if specified, is the new object to be placed into the dispatch table.

The value returned by dispatch_info is determined by the following rules:

- The function returns an atom if the dispatch table is bound to a command or function. The atom is the command or function.
- The function returns NIL if nothing is bound to the entry.
- The function returns a dispatch table (mode) if the entry is a prefix key, that is, an intermediate part of a key path. (For example, the {ESC} entry in main is bound to escape mode.)

Note: To determine whether a command (handler) or function is bound, use fsymeval on the returned atom. If it is a handler, then handler_info may also be useful.

display_error_noabort Special Form

The display_error_noabort special form is the same as the error_message function.

do_forever Special Form

The do_forever function performs an infinite loop.

Format: (do_forever s1 [s2 ...])

Arguments: The arguments s1, s2, and ..., if specified, are any PEEL statements to be executed in the loop.

Action: PEEL executes all the arguments, s1, s2, and ..., and repeats execution of them in an infinite loop until stop_doing is executed.

The do_forever function returns the value NIL.

do_n_times Special Form

The do_n_times special form executes a loop for a specified number of iterations.

Format: (do_n_times n s1 [s2 ...])

Arguments: The argument n must be an integer value.

The arguments s1, s2, and ..., if specified, may be any PEEL statements.

Action: If the value of n is 0, or negative, no action is taken.

If the value of n is positive, PEEL executes the statements s1, s2, and ... in a loop for n iterations, or until stop_doing is executed.

The do_n_times function returns the value NIL.

downcase Function

The downcase function converts uppercase letters in its string argument to lowercase.

Format: (downcase s)

Argument: The argument s must be a string or character value.

Action: The downcase function returns a string value. The string is computed by returning a copy of the string s after converting all uppercase letters to their corresponding lowercase letters.

dt Command and Function

The dt command or function inserts the current date and time into your text buffer at the current cursor position.

Command Format: {ESC} X dt

Function Format: (dt)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the current date and time into your text buffer at the current cursor position. An example of the format used is as follows:

    05/19/81 11:06:57

dump_file Command

The dump_file command "compiles" PEEL source to a fasdump format file.

Format: {ESC} X dump_file

Argument: A numeric argument, if specified, is ignored.

Action: EMACS partially compiles the PEEL program in the current text buffer and dumps it to a file. If the buffer name is of the form x.EM, EMACS creates the fasdump file x.EFASL. If the buffer name does not have the suffix .EM, EMACS simply adds the suffix .EFASL to the buffer name.

else Atom

The else atom is used in the if function. (See that function description for further information.)

empty_buffer_p Function

The empty_buffer_p function tests if the current buffer is empty.

Format: (empty_buffer_p)

Arguments: None.

Action: The empty_buffer_p function returns a Boolean value. The value returned is true if the current buffer is empty and false if the current buffer contains text.

end_line Command and Function

The end_line command or function moves the cursor to the end of the current line.

Command Format: {ESC} X end_line
or
{CTRL-E}

Function Format: (end_line)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS moves the cursor to the end of the current line.

The end_line function returns the value NIL.

end_of_buffer_p Function

The end_of_buffer_p function tests whether the current cursor is at the end of the buffer.

Format: (end_of_buffer_p)

Arguments: None.

Action: The end_of_buffer_p function returns a Boolean value. The value returned is true if the current cursor is at the end of the buffer; otherwise, it is false.

end_of_line_p Function

The end_of_line_p function tests whether the current cursor is at the end of the line.

Format: (end_of_line_p)

Arguments: None.

Action: The end_of_line_p function returns a Boolean value. The value returned is true if the current cursor is at the end of the line; otherwise, it is false.

eolp Function

The eolp function is an abbreviation of end_of_line_p.

eq Function

The eq function tests whether its arguments are the same object.

Format: (eq arg1 arg2)

Arguments: The arguments arg1 and arg2 may have any data type.

Action: The eq function returns a Boolean value. The returned value is true if arg1 and arg2 are the same object, and false if they are different objects.

Note: In almost all cases, you should use the = function in preference to the eq function. To understand the difference, consider the following two examples:

```
(eq 2 2)
(eq "a" "a")
```

The first of these examples is always true because the integer 2 always becomes the same object. But the second example is never true (if typed in the minibuffer), because a new copy of the string "a" is allocated for each argument by the reader. However, if the = function is used instead of eq in the above two examples, the values of both would be true.

error_message Special Form

The error_message special form displays an error message in the minibuffer. It does not abort the current command.

Format: (error_message s)

Argument: The argument s must be a string.

Action: EMACS displays the string s in the minibuffer.

The error_message function returns the value NIL.

Note: Unlike the info_message function, this function forces a screen update even if redisplay has been suppressed, for example with with_no_redisplay.

Escape Sequences

Escape sequences provide a method to enter control characters so that they are printable. The following escape sequences are available:

~hnn	The character with hexadecimal code of nn
~cC	The control character corresponding to the character C
~n	The newline character
~" or ~q	The string delimiter (double quotation mark)
~~	The string escape character (tilde)
~<nl>	Concealed newline, allowing strings to be continued on the next line without including the newline character in the string

europe_dt Command and Function

The europe_dt command or function inserts the current date in European format into your buffer.

Command Format: {ESC} X europe_dt

Function Format: (europe_dt)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the current date in numerical European format, dd/mm/yr, into your text at the current cursor position. The format is similar to the following:

    19/09/85

The europe_dt function returns the value NIL.

eval Function

The eval function is a LISP function that evaluates its arguments and returns the result.

Format: (eval f)

Argument: The argument f is a form or any executable PEEL program.

Action: PEEL executes the form f and returns the result.

Note: You can evaluate a form f simply by executing it as part of your PEEL program. Therefore, if you are writing a simple PEEL program and explicitly calling the eval function, you are probably doing something wrong. The eval function is primarily useful in programs that deal with LISP or PEEL itself, rather than programs about string manipulation.

evaluate_af Function

The evaluate_af function evaluates an active function in PRIMOS. It is similar to the af function, but returns a string value rather than inserting it into the current buffer.

Format: (evaluate_af s)

Argument: The argument s must be a string value.

Action: EMACS evaluates the string s as an active function in PRIMOS, and returns a string value.

Example: The function

    (evaluate_af "[ATTRIB <0>CMDNC0 -TYPE]")

returns the string value "UFD".

exchange_mark Command and Function

The exchange_mark command or function exchanges mark and point.

Command Format: {ESC} X exchange_mark
or
{CTRL-X} {CTRL-X}

Function Format: (exchange_mark)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS exchanges mark and point. That is, the current cursor position is set equal to the last marked position, and the mark is reset to the cursor position prior to the beginning of this command.

The exchange_mark function returns the value NIL.

execute_macro Command and Function

The execute_macro command or function executes a stored sequence of commands.

Command Format: [{ESC}n] {ESC} X execute_macro
or
[{ESC}n] {CTRL-X} E

Function Format: (execute_macro [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

EMACS repeats the following step n times: it executes the macro consisting of the collection of keystrokes collected by the last collect_macro and finish_macro commands.

exit_minibuffer Command

The exit_minibuffer command exits the minibuffer.

Format: {ESC} X exit_minibuffer
or
{RETURN} (only when in minibuffer)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS exits the minibuffer. You are returned to the current cursor position in your major window or buffer.

expand_macro Command and Function

The expand_macro command or function expands a stored keyboard macro into equivalent PEEL code.

Command Format: {ESC} X expand_macro

Function Format: (expand_macro [s])

Argument: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you in the minibuffer with "macro name:". The string that you type is assigned to the variable s.

EMACS expands the keyboard macro keystrokes collected by the last collect_macro into PEEL code, as a defcom with a name specified by the string s, and stores the resulting PEEL source code into the text buffer at the current cursor position.

explain_key Command and Function

The explain_key command or function invokes the help facility that explains a specified keypath.

Command Format: {ESC} X explain_key
or
{ESC}?

Function Format: (explain_key)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS invokes the help facility that explains a specified keypath. EMACS does not prompt you. The next keypath that you type is explained rather than performed.

The explain_key function returns the value NIL.

extend_command Command

The extend_command command is the EMACS facility that allows you to execute any EMACS function by typing {ESC} X followed by the function's name.

fasdump Function

The fasdump function compiles and dumps the current buffer in fasload format.

Format: (fasdump s [d])

Arguments: The argument s must be a string value. The argument d, if specified, must be a Boolean value.

Action: The fasdump function compiles the current buffer (containing PEEL code) and puts the result into a file. The name of the file is the concatenation of the string s and the string .EFASL.

The fasdump function returns the value false.

fasload Function

The fasload function loads a fasload format file.

Format: (fasload s [d])

Arguments: The argument s must be a string value. The argument d, if specified, must be a Boolean value.

Action: The fasload function loads and executes a fasload-format file. The name of the file is the concatenation of the string s and the string ".EFASL".

If the argument d is not specified, let d equal false. If the value of d is true, then debugging information is printed; otherwise, debugging information is not printed.

The fasload function returns the value false.

file_info Function

The file_info function queries or sets information about a file.

Format: (file_info s p [v])

Arguments: The argument s must be a string value.

The argument s is the pathname of the file.

The argument p (representing the desired property) must be one of the atoms listed below under action.

The argument v, if specified, must have a data type that is consistent with the argument p.

Action: Each possible value of the argument p corresponds to a file property. The different possible values for p and their corresponding properties are as follows:

p	Data Type	File Property
path_name	string	Absolute pathname of the file.
entry_name	string	Entry name of the file.
directory_name	string	The name of the directory that contains the file. (read-only).
type	string	Type of file: none, file, directory, segdir, or unknown. (read-only)
exists	Boolean	True if and only if file exists. (read-only)
dumped	Boolean	True if and only if file has been dumped. (read-only)

The argument v, if specified, must have the data type shown for the argument p in the preceding table.

The file information attribute being set or queried is specified by the argument p, as shown in the preceding table. If the argument v is specified, it is ignored, because at the present time no file attribute values may be changed.

The file_info function returns the value of the file attribute specified by the argument p, as shown in the preceding table.

file_name Function

The file_name function returns the default filename associated with a buffer.

Format: (file_name cur)

Argument: The argument cur must be a cursor value.

Action: The file_name function returns a string value. The string value returned is the file name associated with the buffer into which the cursor cur points.

file_operation Function

The file_operation function deletes a file.

Format: (file_operation s delete)

Argument: The argument s must be a string value.

Action: EMACS deletes the file with the pathname specified by the string s, and returns the PRIMOS error code. The error code equals zero if the delete was successful. The error code equals nonzero if the delete fails for any of the following reasons: a bad pathname, a nonexistent file, a file that is not a SAM or DAM file, or any other deletion error.

fill_array Function

The fill_array function fills an array with a specified value.

Format: (fill_array a v [m [n]])

Arguments: The argument a must be an array value. Usually it is a variable that has been bound to an array by means of the setq and make_array functions.

The argument v is the value to which the elements of the arrays are to be set. The data type of v must be consistent with the data type of the array a, as specified in the make_array function that created the array a.

The arguments m and n, if specified, must be integer values.

Action: The fill_array function returns a value whose data type is the data type of the elements of the array a.

If the argument m is not specified, let m equal 0.

If the argument n is not specified, let n equal the maximum array index, which is one less then the number of elements in the arrays, as specified in the make_array function that created the array a.

The value of m must be greater than or equal to 0 and less than or equal to n. The value of n must be less than the number of elements in the array a, as specified in the make_array function that created the array a.

EMACS assigns the value v to each of the elements of the array a, beginning with index m and ending with index n.

The fill_array function returns the value v.

fill_end_token_insert_left Function

The fill_end_token_insert_left function wraps to the fill column and inserts command characters.

Format: (fill_end_token_insert_left)

Arguments: None.

Action: EMACS wraps to the fill column and inserts command characters.

fill_end_token_insert_pfx Function

The fill_end_token_insert_pfx function wraps to the fill column, preserving the current whitespace prefix on each line, and inserts command characters.

Format: (fill_end_token_insert_pfx)

Arguments: None.

Action: EMACS wraps to the fill column, preserving the current whitespace prefix on each line, and inserts command characters. The indent is similar to cret_indent_relative.

fill_off Command and Function

The fill_off command or function turns off fill mode.

Command Format: {ESC} X fill_off

Function Format: (fill_off)

Arguments: A numeric argument, if specified, is ignored.

Action: EMACS turns off fill mode.

The fill_off function returns the value NIL.

fill_on Command and Function

The fill_on command or function turns on fill mode.

Command Format: {ESC} X fill_on

Function Format: (fill_on)

Arguments: A numeric argument, if specified, is ignored.

Action: EMACS turns on fill mode so that words are automatically moved from line to line in order to fill lines to the specified lengths.

The fill_on function returns the value NIL.

fill_para Command and Function

The fill_para command or function fills and optionally adjusts a paragraph.

Command Format: [{ESC}n] {ESC} X fill_para
or
[{ESC}n] {ESC} Q

Function Format: (fill_para [n])

Argument: The argument n, if specified, must be an integer value.

Action: The fill_para command or function fills the current paragraph so that each line does not have more than the number of characters indicated by the function (buffer_info fill_column) or by the command tell_right_margin. It rearranges words on the line so that each line is about the same length. Use set_right_margin to change the right margin.

If the left margin is greater than zero, the text is indented by that number of characters. The left margin is specified by the variable fill_prefix, and is displayed by the tell_left_margin command. Use set_left_margin to change the left margin.

If the argument n is specified and is greater than one, the paragraph is additionally right-justified.

To undo the results of fill_para, use the untidy command or function.

find_buffer Function

The find_buffer function returns a cursor that points to the start of a specified buffer.

Format: (find_buffer s)

Argument: The argument s must be a string value.

Action: The find_buffer function returns a cursor value.

EMACS interprets the string s as a buffer name. If a buffer with that name does not exist, EMACS creates a new buffer with that name.

EMACS returns a cursor value pointing to the start of the buffer whose name is given by the string s.

Example: The following statement

```
(go_to_cursor (find_buffer "xyz"))
```

moves the current cursor to the start of the buffer xyz, creating that buffer if necessary.

find_file Command and Function

The find_file command or function finds a file either in an EMACS buffer or in the PRIMOS file system.

Command Format: {ESC} X find_file
or
{CTRL-X} {CTRL-F}

Function Format: (find_file s)

Argument: The argument s to the find_file must be a string value.

A numeric argument, if specified, is ignored.

Action: For the find_file command, EMACS prompts you for a file name, and assigns the string you type to the variable s.

If there is a buffer with a name equal to the value of the string s, EMACS makes that buffer the current buffer.

If there is no such buffer, EMACS proceeds as follows:

- It searches the PRIMOS file system for a file with names specified by s. If there is no such file, EMACS terminates this operation with an error message.
- It creates a new buffer with the buffer name s, and loads the text of the file into that buffer, making it the current buffer.

The find_file function returns the value NIL.

Note: You may use PRIMOS conventions in using special characters and options in your filename string. For example, the string

```
foo@@ -after 7/22/85
```

searches for all files beginning with "foo", created since July 22, 1985.

(See the Prime User's Guide manual for complete details.)

find_mode Function

The find_mode function returns a dispatch table to a desired mode.

Format: (find_mode m)

Argument: The argument m must be either a string or a quoted atom.

Action: The find_mode function returns a value with the dispatch_table data type.

The argument m must be the name (either in string or in quoted atom form) of a mode. The find_mode function returns a dispatch table for that mode. If the mode does not exist, it will be created.

finish_macro Command

The finish_macro command terminates a macro whose keystrokes were collected beginning with the collect_macro command.

Format: {ESC} X finish_macro
or
{CTRL-X})

Argument: A numeric argument, if specified, is ignored.

Action: Collection of keystrokes is begun with the collect_macro command and ended with the finish_macro command. The finish_macro command makes the keystroke collection available as a macro that can be invoked by the execute_macro command, which is normally bound as {CTRL-X} E.

first_line_p Function

The first_line_p function returns a Boolean value indicating whether the current cursor position is at the first line in the buffer.

Format: (first_line_p)

Arguments: None.

Action: The first_line_p function returns a Boolean value. The returned value is true if the current cursor position points to a character in the first line of the buffer; otherwise it is false.

firstlinep Function

The firstlinep function is an abbreviation of the first_line_p function.

flush_typeahead Function

The flush_typeahead function flushes pending keyboard input. It is usually used for error clean up.

Format: (flush_typeahead)

Arguments: None.

Action: The flush_typeahead function invokes the PRIMOS supervisor call that flushes typeahead by clearing the keyboard input buffer.

The flush_typeahead function returns the value NIL.

forward_char Command and Function

The forward_char command or function moves the cursor forward by a specified number of characters.

Command Format: [{ESC}n] {ESC} X forward_char
or
[{ESC}n] {CTRL-F}

Function Format: (forward_char [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, the cursor is moved forward n characters, stopping if the end of the buffer is reached.

If the value of n is negative, the cursor is moved back (-n) characters, stopping if the beginning of the buffer is reached.

The forward_char function returns the value NIL.

forward_clause Command and Function

The forward_clause command or function moves the cursor forward by a specified number of clauses.

Command Format: [{ESC}n] {ESC} X forward_clause
or
[{ESC}n] {CTRL-X} {CTRL-Z} {CTRL-E}

Function Format: (forward_clause [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor forward to the nth occurrence of a character that delimits a clause. (That is, EMACS moves the cursor ahead n clauses, leaving the cursor on the character immediately following the last clause delimiter.) Cursor movement stops if the end of the buffer is reached.

If the value of n is negative, EMACS moves the cursor back to the character preceding the (-n)th occurrence of a character that delimits a clause. (That is, EMACS moves the cursor back (-n) clauses and leaves it at the character preceding the last clause delimiter.) Cursor movement stops if the beginning of the buffer is reached.

The forward_clause function returns the value NIL.

Note: The characters delimiting a clause are contained in the global string variable clause_scan_table$.

forward_clausef Function

The forward_clause function moves the cursor forward by a specified number of clauses, and returns a Boolean value indicating whether the operation was successful.

Format: (forward_clausef [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: The forward_clausef function returns a Boolean value.

If n is not specified, let n equal 1.

If the value of n is 0, no cursor movement takes place and forward_clausef returns the value true.

If the value of n is positive, EMACS moves the cursor forward to the nth occurrence of a character that delimits a clause. (That is, EMACS moves the cursor ahead n clauses, leaving the cursor on the last character following the last clause delimiter.) Cursor movement stops if the end of the buffer is reached. If the end of the buffer is reached, forward_clausef returns the value false; otherwise, forward_clausef returns the value true.

If the value of n is negative, EMACS moves the cursor back to the character preceding the (-n)th occurrence of a character that delimits a clause. (That is, EMACS moves the cursor back (-n) clauses leaving it at the character preceding the last clause delimiter.) Cursor movement stops if the beginning of the buffer is reached. If the beginning of the buffer is reached, forward_clausef returns the value false; otherwise, forward_clausef returns the value true.

Note: The list of characters delimiting a clause may be found in the global string variable clause_scan_table$.

forward_kill_clause Command and Function

The forward_kill_clause command or function kills text from the current cursor position to the end of the specified clauses.

Command Format: [{ESC}n] {ESC} X forward_kill_clause
or
[{ESC}n] {CTRL-X} {CTRL-Z} {CTRL-K}

Function Format: (forward_kill_clause [n])

Argument: The argument n, if specified, must be an integer value whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS kills all the text in the region starting with the current cursor position and ending with the end of the nth clause in a forward direction.

If the value of n is negative, EMACS kills all text in the region ending with the current cursor position, and beginning with the (-n)th clause in a backward direction.

The forward_kill_clause function returns the value NIL.

Note: The list of characters delimiting a clause may be found in the global string variable clause_scan_table$.

forward_kill_sentence Command and Function

The forward_kill_sentence command or function kills text from the current cursor position to the end of the specified sentence.

Command Format: [{ESC}n] {ESC} X forward_kill_sentence
or
[{ESC}n] {ESC} K

Function Format: (forward_kill_sentence [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS deletes all text in the region beginning with the current cursor position and ending with the end of the nth sentence in a forward direction.

If the value of n is negative, EMACS deletes all text in the region beginning with the (-n)th sentence in a preceding direction, and stopping at the current cursor position.

forward_para Command and Function

The forward_para command or function moves the cursor forward by a specified number of paragraphs.

Command Format: [{ESC}n] {ESC} X forward_para
or
[{ESC}n] {CTRL-X}]

Function Format: (forward_para [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor forward to the character preceding the nth occurrence of a character that delimits a paragraph. (That is, EMACS moves the cursor forward n paragraphs, leaving the cursor on the character immediately following the last paragraph delimiter.) Cursor movement stops if the end of the buffer is reached.

If the value of n is negative, EMACS moves the cursor backward to the (-n)th occurrence of a character that delimits a paragraph. (That is, EMACS moves the cursor back n paragraphs, leaving it at the character preceding the last paragraph delimiter.) Cursor movement stops if the beginning of the buffer is reached.

The forward_para function returns the value NIL.

Note: Paragraphs are defined as lines beginning with a period, a blank line, or lines beginning with a space.

forward_search Function

The forward_search function searches the text buffer for a specified string, and returns a Boolean value indicating success or failure.

Format: (forward_search s)

Argument: The argument s must be a string argument.

Action: EMACS searches forward for the string s in your text buffer, starting with the character at the current cursor position.

If the search is successful, EMACS moves the cursor to the character following the matching string in the text buffer, and returns the value true.

If the search fails, the cursor is left unchanged, and the function returns the value false.

forward_search_command Command and Function

The forward_search_command command or function searches forward in your text buffer for a string.

Command Format: {ESC} X forward_search_command
or
{CTRL-S}

Function Format: (forward_search_command [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you for a string value, which is assigned to the string s.

EMACS searches forward in the text buffer for the string s, starting with the character in the current cursor position.

If the search succeeds, EMACS moves the current cursor to the character following the string.

If the search is unsuccessful, EMACS displays an error message.

forward_sentence Command and Function

The forward_sentence command or function moves the cursor forward by the specified number of sentences.

Command Format: [{ESC}n] {ESC} X forward_sentence
or
[{ESC}n] {ESC} E

Function Format: (forward_sentence [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor forward to the nth occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor ahead n sentences, leaving the cursor on the character immediately following the last sentence delimiter.) Cursor movement stops if the end of the buffer is reached.

If the value of n is negative, EMACS moves the cursor back to the character preceding the (-n)th occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor back (-n) sentences, and leaves it at the character preceding the last sentence delimiter.) Cursor movement stops if the beginning of the buffer is reached.

The forward_sentence function returns the value NIL.

Note: The characters delimiting a sentence are in the global string variable sentence_scan_table$.

forward_sentencef Function

The forward_sentencef function moves the cursor forward by a specified number of sentences, and returns a Boolean value indicating whether the operation was successful.

Format: (forward_sentencef [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: The forward_sentencef function returns a Boolean value.

If the value of n is not specified, let n equal 1.

If the value of n is 0, no cursor movement takes place, and forward_sentencef returns the value true.

If the value of n is positive, EMACS moves the cursor forward to the nth occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor ahead n sentences, leaving the cursor on the character immediately following the last sentence delimiter.) If the end of the buffer is reached, cursor movement stops, and forward_sentencef returns the value false; otherwise, it returns the value true.

If the value of n is negative, EMACS moves the cursor back to the character preceding the (-n)th occurrence of a character that delimits a sentence. (That is, EMACS moves the cursor back (-n) sentences, leaving it at the last sentence delimited.) If the beginning of the buffer is reached, cursor movement stops and forward_sentencef returns the value false; otherwise, it returns the value true.

Note: The characters delimiting a sentence are in the global string variable sentence_scan_table$.

forward_word Command and Function

The forward_word command or function moves the cursor forward by a specified number of words.

Command Format: [{ESC}n] {ESC} X forward_word
or
[{ESC}n] {ESC} F

Function Format: (forward_word [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS moves the cursor forward to the nth occurrence of a character that immediately follows a word. (That is, EMACS moves the cursor ahead n words, leaving the cursor on the whitespace character immediately following the end of the words.) Cursor movement stops if the end of the buffer is reached.

If the value of n is negative, EMACS moves the cursor back to the (-n)th occurrence of a character that begins a word. (That is, EMACS moves the cursor back (-n) words, leaving it at the first character of that word.) Cursor movement stops if the beginning of the buffer is reached.

The forward_word function returns the value NIL.

Note: The token_chars atom contains a list of the characters that define a word or token.

found_file_hook Command and Function

The found_file_hook command or function checks a suffix and turns on the mode associated with that suffix.

Command Format: {ESC} X found_file_hook

Function Format: (found_file_hook)

Arguments: A numeric argument, if specified, is ignored.

Action: EMACS examines the suffix of the file associated with the current buffer, and turns on the mode associated with that suffix. (See the EMACS Reference Guide, Appendix A.)

The found_file_hook function returns the value NIL.

fset Function

The fset LISP function sets the function cell of an atom.

Format: (fset a f)

Arguments: The argument a is any atom, usually quoted. The argument f is a function.

Action: The fset function sets the function cell of the atom a to the value f, and returns the value f.

Note: Every PEEL symbol has associated with it a function cell (or function value) similar to its ordinary value. Usually you set the function cell by means of a defun or defcom. The fset function lets you set it explicitly, and the fsymeval function lets you obtain the function call explicitly.

fsymeval Function

The fsymeval LISP function returns the contents of the function cell of an atom.

Format: (fsymeval a)

Argument: The argument a must be an atom.

Action: The fsymeval function returns the function cell of the atom a. The value returned may have any data type. If the function cell is not set, fsymeval returns NIL.

fsymeval usually returns a value of type handler for commands and type function for functions.

(See the fset function for further information.)

function Data Type

This is a data type. Functions are objects that contain executable code. A function contains all the information needed to call this code. This includes the number of expected arguments, their data types, and the type of value that will be returned. Built-in functions have their code written in PL/I, while user written functions are written in the PEEL extension language and have list structure as their executable code.

function_info Function

The function_info function returns information about a PEEL function.

Format: (function_info f p [n])

Arguments: The argument f must have the data type function, as returned by the fsymeval function, for example. The argument p is an atom representing the property you wish to determine. The list of legal values for p is given below.

The argument n, if specified, must be an integer value.

Action: The data type of the value returned by the function_info function depends upon the symbol p.

The following table lists the legal values for p, the data type of the value returned by function_info, and the function property corresponding to the value of p.

p	Data Type	Property
return_value	integer	Data type of the returned value. (See typef for meaning of integral value.)
argument_type	integer	Data type of the nth argument.
special_form	Boolean	True if arguments to handler are passed unevaluated in a list (as if &rest was used for all arguments for a defcom).
user_defined_function	Boolean	True if code is PEEL rather than PL/I.
returns_a_value	Boolean	True if the function returns a value.

required_arguments	integer	Number of required arguments.
optional_arguments	integer	Number of optional arguments.
evaluate_argument	Boolean	True if the "&quote" defun option applies to the nth argument, false otherwise.
cleanup_handler	any	The handler to be invoked if the function is aborted. (NIL if none established.) Atom of cleanup_handler is returned.

get Function

The get LISP function returns the value associated with a tag in a given property list (plist).

Format: (get pl t)

Arguments: The arguments pl and t must be atoms, usually quoted.

Action: If the atom pl is an atom with a property list, and if the argument t appears as a tag in that property list, the get function returns the value associated with the tag t. Otherwise, the get function returns the value NIL.

Note: A tag and value can be added to an atom's property list by means of the putprop function, and can be removed by means of the remprop.

Example: Suppose that the property list of the atom Jane is

```
((age 6) (hair blonde) (eyes blue))
```

then the function

```
(get 'Jane 'age)
```

returns the value 6, while the function

```
(get 'Jane 'weight)
```

returns the value NIL.

get_cursor Function

The get_cursor function returns a list of all the cursor properties that may be obtained by the cursor_info function.

Format: (get_cursor [cur])

Argument: The argument cur, if specified, must be a cursor value.

Action: The get_cursor function returns a list value.

If the argument cur is not specified, let cur equal the current cursor position.

The get_cursor function returns a list containing all values that may be obtained by the cursor_info function. These values are:

- A string value containing the buffer name of the cursor cur
- An integer value containing the line number
- An integer value containing the character position on the line
- A Boolean value containing the sticky flag

The get_cursor function returns a list containing those values.

Example: A typical value returned by get_cursor might be:

```
("main" 6 11 false)
```

get_filename Command and Function

The get_filename command or function retrieves the current pathname of the current buffer and inserts it at point.

Command Format: {ESC} X get_filename
or
{CTRL-X} {CTRL-Z} {CTRL-F}

Function Format: (get_filename)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS forms a string containing the current buffer's associated pathname and inserts it into the buffer at the current cursor position.

The get_filename function returns the value NIL.

get_pname Function

The get_pname function returns the print name of an atom.

Format: (get_pname a)

Argument: The value of the argument a must be a quoted atom.

Action: The get_pname function returns a string value that is the name of the atom.

Example: The function

    (get_pname 'xyz)

returns the string "xyz".

get_tab Command and Function

The get_tab command or function restores tabs that were previously saved.

Command Format: {ESC} X get_tab [s] [t]

Function Format: (get_tab [s] [t])

Arguments: A numeric argument, if specified, is ignored.

The arguments s and t, if specified, must be string values.

Action: If the argument s is not specified, EMACS prompts you for a filename. The typed string is assigned to the variable s. If the argument t is not specified, EMACS prompts you for the name by which the tabs are stored in the file. The typed string is assigned to the variable t.

The string s must be the filename of a file that was created with the save_tab command or function, and the string t must be a name of tabs in that file. EMACS restores the tab settings to their value at the time that the file was saved.

The get_tab function returns the value NIL.

go_to_buffer Function

The go_to_buffer function positions the cursor at the beginning of the specified buffer and returns that cursor.

Format: (go_to_buffer s)

Argument: The argument s must be a string value.

Action: The go_to_buffer function returns a cursor value.

EMACS moves the current cursor to the beginning of the buffer specified by the string s. The go_to_buffer function returns that cursor position.

go_to_cursor Function

The go_to_cursor function moves the current cursor to the specified cursor.

Format: (go_to_cursor cur)

Argument: The argument cur must have the cursor data type.

Action: EMACS moves the current cursor to the position specified by the argument cur. The go_to_cursor function returns a cursor value equal to the new cursor position.

go_to_hpos Function

The go_to_hpos function goes to the specified horizontal position on the current line.

Format: (go_to_hpos n)

Argument: The argument n must be an integer value.

Action: The go_to_hpos function returns a Boolean value.

EMACS attempts to move the cursor to the horizontal position specified by the argument n. If this is possible, the go_to_hpos function returns the value true; otherwise, it returns the value false.

It is possible under the following conditions:

- The value of n is positive and smaller than the number of characters on the current line.
- The value of n is positive and smaller than the value of the line margin, and two-dimensional mode is on.

go_to_window Function

The go_to_window function moves the cursor to the buffer associated with the specified window.

Format: (go_to_window w)

Argument: The data type of the argument w must be a window.

Action: The argument w must be a window that is on the screen. The go_to_window function makes the cursor associated with the window the current cursor. It does not change the association of a window and a cursor.

The go_to_window function returns the value NIL.

goto_line Command and Function

The goto_line command or function moves the current cursor to a specified line in the buffer.

Command Format: [{ESC}n] {ESC} X goto_line
or
[{ESC}n] {ESC} G

Function Format: (goto_line [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If n is less than or equal to 1, EMACS moves the cursor to the first line of the buffer.

If n is larger than the number of lines in the buffer, EMACS moves the cursor to the last line of the buffer.

Otherwise, EMACS moves the cursor to line n of the buffer.

The goto_line function returns the value NIL.

handler Data Type

The handler data type represents a handler, which is a command as defined by using defcom.

handler_info Function

The handler_info function gets or sets information about a handler.

Format: (handler_info h p [v])

Arguments: The argument h must have the handler data type, usually as returned either from the current_handler function or from fsymeval of an atom returned from dispatch_info.

The argument p must be one of the atoms listed below under action.

The argument v, if specified, must have a data type that is compatible with the atom p.

Action: The handler_info function returns a value whose data type depends upon the argument p.

Each value of the argument p corresponds to information about the handler h. The following table gives the possible values for the argument p, along with their associated data types and meanings.

p	Data Type	Property
name	string	Name of the handler.
handler	function or PL/I_subroutine	The function code of the command. If internal, it is of type PLI_subroutine; if external, of type function.
is_prefix	Boolean	Command has specified &prefix in its header.
explanation	string	The &doc documentation string associated with the command. If none, null string.
data_value	any	Any data value may be here.
uses_character_argument	Boolean	Command has &chararg in its header.

If the value v is not specified, the handler_info function returns the information associated with the argument p. If the argument v is specified, the handler_info function sets the corresponding property value to v, and returns the old value of the property.

If the command is internal or shared, none of the values may be changed. Otherwise, any value except the name may be changed.

have_input_p Function

The have_input_p function returns a Boolean value indicating if there is pending keyboard input.

Format: (have_input_p)

Arguments: None.

Action: The have_input_p function returns a Boolean value. The value is true if there is pending keyboard input; otherwise, it is false.

hcol Command and Function

The hcol command or function sets or queries the horizontal column, that is, the column number of the leftmost column displayed on your screen.

Command Format: [{ESC}n] {ESC} X hcol

Function Format: (hcol [n])

Argument: The numeric argument n, if specified, must be an integer value.

Action: The argument n, if specified, must be positive. If n is specified, EMACS sets the horizontal column (for horizontal scrolling) to the value specified by n. The result is that the leftmost column displayed on your screen will be text column n.

EMACS displays the current horizontal column value in the minibuffer. (This happens whether the argument n is specified or not.)

The hcol function returns the value NIL.

help_char Command

The help_char command invokes the explain command.

Format: {ESC} X help_char
or
{CTRL-_}

Arguments: A numeric argument, if specified, is ignored.

Action: The help_char command invokes the explain facility that provides you with help information.

high_bit_off Function

The high_bit_off function turns off the high-order bits in each character of a string.

Format: (high_bit_off s)

Argument: The argument s must be a string or character value.

Action: The high_bit_off function returns a string value. The string value is obtained by turning off the high-order bit in each character of string s.

high_bit_on Function

The high_bit_on function turns on the high-order bits in each character of a string.

Format: (high_bit_on s)

Argument: The argument s must be a string or character value.

Action: The high_bit_on function returns a string value. The string value is obtained by turning on the high-order bit in each character of string s.

hscroll Command and Function

The hscroll command sets hcol to the current column position.

Command Format: {ESC} X hscroll

Function Format: (hscroll)

Arguments: A numeric argument, if specified, is ignored.

Action: Let k be the horizontal position of the current cursor. Then EMACS executes

```
(hcol k)
```

The hscroll function returns the value NIL.

if Special Form

The if special form allows you to perform conditional program execution.

Format: (if b s1 else s2)

Arguments: The argument b must have a Boolean value. The arguments s1 and s2 must be PEEL statements.

Action: If the value of b is true, then s1 is executed; otherwise, s2 is executed.

The if special form returns the value NIL.

if_at Special Form

The if_at special form performs conditional execution depending on the string to the right of the current cursor position.

Format: (if_at s s1 else s2)

Arguments: The argument s must be a string value. The arguments s1 and s2 must be PEEL source statements.

Action: If the text to the right of the current cursor position equals the characters of the string s, then s1 is executed; otherwise, s2 is executed.

The if_at special form returns the value NIL.

Note: The use of if_at is the same as

```
(if (looking_at "string") ... ))
```

ignore_prefix Command

The ignore_prefix command aborts a partially completed command sequence.

Format: {ESC} X ignore_prefix

Arguments: None.

Action: This command only makes sense when bound. It is normally bound to the {CTRL-G} entry in dispatch tables referenced through a prefix. For example, {CTRL-X} {CTRL-G} is bound to ignore_prefix, so that you can abort a command prefixed by {CTRL-X}.

indent_line_to_hpos Command and Function

The indent_line_to_hpos command or function indents the current line to the specified horizontal position.

Command Format: [{ESC}n] {ESC} X indent_line_to_hpos

Function Format: (indent_line_to_hpos n)

Argument: The argument n must be an integer value.

Action: The indent_line_to_hpos function returns a Boolean value.

EMACS deletes all whitespace characters from the beginning of the current line, and then inserts n blanks.

The indent_line_to_hpos function returns the value true if the operation succeeds; otherwise, it returns false.

indent_relative Command and Function

The indent_relative command or function indents the current line to the same horizontal position as the preceding line.

Command Format: [{ESC}n] {ESC} X indent_relative
or
[{ESC}n] {ESC} I

Function Format: (indent_relative [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: EMACS performs the following steps:

- Inserts or deletes whitespace characters at the beginning of the current line, so that the first nonblank character of the line is indented exactly as much as the first nonblank character of the preceding line. (Note: If the current line is the first line of the buffer or the preceding line contains only whitespace, then no blanks are inserted at the beginning of the line.)

- Leaves the cursor at the first nonblank character of the line.

- If n is not specified, let n equal 0.

- Executes the function

  (insert_tab n)

  If n is positive, this inserts n tabs. If n is negative, this moves back (-n) tab stops.

The indent_relative function returns the value NIL.

indent_to_fill_prefix Command and Function

The indent_to_fill_prefix command or function indents the current line to the current left margin.

Command Format: [{ESC}n] {ESC} X indent_to_fill_prefix
or
[{ESC}n] {ESC} {CTRL-I}

Function Format: (indent_to_fill_prefix [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, then n lines of text, beginning with the current one and continuing forward, are indented.

If the value of n is negative, then -n lines of text, beginning with the current one and proceeding backward, are indented.

To indent a line of text, EMACS proceeds as follows:

- If the line is null (contains no characters), no action takes place. Otherwise:
- EMACS removes all leading whitespace from the beginning of the line.
- EMACS inserts fill_prefix spaces at the beginning of the line.

Note: The value associated with the atom fill_prefix is set by the set_left_margin command.

The indent_to_fill_prefix function returns the value NIL.

index Function

The index function, which is like the PL/I function of the same name, returns the position of the second string within the first.

Format: (index s1 s2)

Arguments: The arguments s1 and s2 must be string or character values.

Action: The index function returns an integer value.

EMACS searches the string s1 for the string s2 as a substring. (For example, "CDE" is a substring of "ABCDEF", but "CDF" is not.)

If the string s2 appears as a substring of s1, the index function returns the position of the first character of the first occurrence of string s2 within s1. If s2 does not appear as a substring, index returns the value 0.

Example: The function

```
(index "ABCDEF" "CDE")
```

returns the value 3, because "CDE" appears as a substring starting in character position 3 of "ABCDEF".

The function

```
(index "ABCDEF" "CDF")
```

returns the value 0, because "CDF" is not a substring of "ABCDEF".

info_message Special Form

The info_message special form displays an information message in the minibuffer.

Format: (info_message s1 [s2 ... s8])

Arguments: The info_message special form takes at least one argument and no more than eight arguments. All arguments must have the string or character data type.

Action: EMACS concatenates all arguments together, as described with the catenate function, and displays the concatenated string in the minibuffer. The message will not appear on the screen until the next screen redisplay occurs. Use error_message if you want to force a message to be displayed while redisplay is inhibited.

init_local_displays Function

The init_local_displays function clears previous printout material from the screen and starts "printout" mode.

Format: (init_local_displays s [b])

Arguments: The argument s must have a string or character value. The argument b, if specified, must be a Boolean value.

Action: EMACS clears the main text window on your screen of previous printout and information messages, displays the string s on the first line, and enters "printout" mode.

If the argument b is true or unspecified, the line is terminated at that point. If the value of b is specified and false, the next call to local_display_generator will append its data to the end of the current line.

Example: The function sequence

```
(print "The previous message")
(init_local_displays "23")
```

would clear "The previous message" from the screen and print the number 23 at the top.

insert Function

The insert function inserts a string or character into the text buffer at the specified cursor position.

Format: (insert s [cur])

Arguments: The argument s must have a string value. The argument cur, if specified, must have a cursor data type value.

Action: The insert function returns a string value.

If the argument cur is specified, EMACS moves the current cursor to the position specified by the argument cur.

EMACS inserts the characters of string s into the text buffer at the current cursor position, and returns the value of string s.

insert_buf Command and Function

The insert_buf command or function takes the contents of a text buffer and inserts it into your current text buffer at the current cursor position.

Command Format: {ESC} X insert_buf
or
{CTRL-X} {CTRL-Z} I

Function Format: (insert_buf [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is unspecified, EMACS prompts you for the name of a buffer. The resulting string is assigned to the variable s.

EMACS inserts the contents of the buffer specified by the string s into your current text buffer, at the current cursor position. The cursor is left at the beginning of the inserted material.

The insert_buf function returns the value NIL.

insert_buff Command and Function

The insert_buff command or function is an alternate name for the insert_buf command and function.

insert_file Command and Function

The insert_file command or function inserts a disk file at the current cursor position.

Command Format: {ESC} X insert_file
or
{CTRL-X} I

Function Format: (insert_file [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified to the insert_file function, must have a string value.

Action: If the argument s is not specified, EMACS prompts the user for a pathname, and assigns the typed string to the string s.

EMACS opens the file whose pathname is the string s, and inserts the text of that file at the current cursor position. The cursor is left at the beginning of the inserted material.

The insert_file function returns the value NIL.

insert_tab Function

The insert_tab function inserts whitespace from the current cursor position to the next tab stop.

Format: (insert_tab [n])

Argument: The argument n, if specified, must have an integer value.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action is taken.

If n is positive, EMACS repeats the following step n times: insert whitespace at the current cursor position to move the cursor to the next tab stop position.

If n is negative, EMACS performs

```
(type_tab n)
```

which moves the cursor back (-n) tab stops.

The insert_tab function returns the value NIL.

insert_version Function

The insert_version function inserts the current EMACS version number into the text buffer at the current cursor position.

Format: (insert_version)

Arguments: None.

Action: EMACS inserts the current version number into the text buffer at the current cursor position.

The format of the inserted characters is illustrated by the following:

    EMACS version 20.0.4e

The insert_version function returns the value NIL.

integer Data Type

A variable with the integer data type can have only integers as values.

integer_to_string Function

The integer_to_string function converts an integer value to a string value.

Format: (integer_to_string n [k])

Arguments: The argument n, and the argument k if specified, must be integer values.

Action: The integer_to_string function returns a string value.

EMACS forms a new string s as follows:

- EMACS converts the integer n to a string representation in the decimal number system, with a - in front, if negative. Let s equal this string.
- If the argument k is specified, and if the string s has fewer characters than the value of k, EMACS inserts additional blanks in the front of the string s, so that the length of the resulting string is equal to the value of k.
- If the argument k is specified, and if the string s has more characters than the value of k, EMACS replaces the string s with a new string containing k *'s.

EMACS returns the string s.

intern Function

The intern function converts a string argument to an atom with the same name. This is the inverse of the get_pname function.

Format: (intern s)

Argument: The argument s must be a string or character value.

Action: The intern function returns an atom value.

EMACS returns an atom whose name is given by the characters in the string s, creating such an atom if necessary.

kill_line Command and Function

The kill_line command or function deletes text from the current cursor position to the end of the current line.

Command Format: [{ESC}n] {ESC} X kill_line
or
[{ESC}n] {CTRL-K}

Function Format: (kill_line [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If n equals 0, no action is taken.

If n is greater than 0, EMACS deletes text to the end of the line, and at end of line deletes the newline character. Then EMACS deletes (n-1) additional lines.

If n is negative, EMACS deletes n lines above the cursor line and from the cursor to the beginning of the current line.

The kill_line function returns the value NIL.

Note: The killed text is placed on the kill ring and can be recalled via yank.

kill_region Command and Function

The kill_region command or function deletes all text between the mark and the current cursor position, placing it on the kill ring.

Command Format: {ESC} X kill_region
or
{CTRL-W}

Function Format: (kill_region)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS deletes all text in the current region, that is, all text between the mark and the current cursor position.

The kill_region function returns the value NIL.

Note: The killed text is placed on the kill ring and can be recalled via yank_region.

kill_rest_of_buffer Command and Function

The kill_rest_of_buffer command or function deletes all text from the current cursor position to the end of the buffer.

Command Format: {ESC} X kill_rest_of_buffer
or
{ESC} {CTRL-D}

Function Format: (kill_rest_of_buffer)

Arguments: None.

Action: EMACS deletes all text from the current cursor position to the end of the current text buffer.

The kill_rest_of_buffer function returns the value NIL.

Note: The killed text is placed on the kill ring and can be recalled via yank.

lambda Special Form

This is the same as LISP lambda. It is a special function that builds a function object from an argument list and program body.

```
(lambda (argument_list) body_1 body_2 ...)
```

Note: The syntax for lambda is the same as for defun, except that lambda defines a function that is unnamed. Defun creates a named function with a global scope. You use lambda in let lists or for fsets, to bind them to a function or a handler.

last_line_p Function

The last_line_p function tests whether the current line is the last line in the buffer.

Format: (last_line_p)

Arguments: None.

Action: The last_line_p function returns a Boolean value. The value is true if the line at the current cursor position is the last line in the buffer; otherwise, it returns false.

lastlinep Function

The lastlinep function is an abbreviation of the last_line_p function.

ld Command and Function

The ld command or function, which is permitted in SUI only, does a PRIMOS LD command.

Command Format: {ESC} X ld

Function Format: (ld)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS executes a PRIMOS LD command at the current attach point. This command is available only in SUI.

leave_one_white Command and Function

The leave_one_white command or function deletes all extra whitespace characters around point.

Command Format: {ESC} X leave_one_white
or
{ESC} {SPACE}

Function Format: (leave_one_white)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS deletes all whitespace around the cursor position and inserts a single blank.

The leave_one_white function returns the value NIL.

let Special Form

The let special form is the same as LISP let. It locally binds variables to values, and, using those bindings, executes a body of expressions. The first thing in the special form is a list of lists. Each of the sublists consists of a variable and a value to bind to it, as in:

```
(let ((variable1 value1)
      (variable2 value2)
        ...            )

        body)
```

The values are evaluated and are bound to their respective variables, and then the expressions of the body are executed in the order specified. The variables may assume values of any type, not just those of the values bound to them. When the let body is finished or otherwise executed, the variable bindings are popped.

Note: The difference between let and set is that let defines variables with local scope and set defines variables with global scope.

Example: The form

```
(let ((x 5))(print x))
```

prints the value 5.

line_is_blank Function

The line_is_blank function tests whether the current line is blank.

Format: (line_is_blank)

Arguments: None.

Action: The line_is_blank function returns a Boolean value. The value is true if the current line contains only whitespace characters; otherwise, it is false.

line_number Function

The line_number function returns the line number at a specified cursor position.

Format: (line_number cur)

Argument: The argument cur must have a cursor value.

Action: The line_number function returns an integer value. The value returned equals the line number of the line to which the cursor value cur points.

Note: This function is the same as

```
(cursor_info cur line_num)
```

lines_in_file Function

The lines_in_file function returns the number of lines in a file.

Format: (lines_in_file)

Arguments: None.

Action: The lines_in_file function returns an integer value equal to the number of lines in the current buffer.

lisp_comment Command and Function

The lisp_comment command or function moves the cursor to the LISP comment column and places a semicolon (;) in that position.

Command Format: {ESC} X lisp_comment
or
{ESC};

Function Format: (lisp_comment)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts blanks into your text buffer so that the cursor position is moved to the LISP comment column, usually column 40. EMACS inserts a semicolon (;) at that point.

The lisp_comment function returns the value NIL.

Note: If you wish to change the default LISP comment column, set the value of the internal variable lisp_comment_column to the desired new value.

lisp_off Command and Function

The lisp_off command or function turns off LISP mode.

Command Format: {ESC} X lisp_off

Function Format: (lisp_off)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns off LISP mode in the current buffer.

lisp_on Command and Function

The lisp_on command or function turns on LISP mode.

Command Format: {ESC} X lisp_on

Function Format: (lisp_on)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns on LISP mode in the current buffer. In this mode, EMACS assumes that your buffer contains a PEEL program and provides additional syntax checking capabilities.

list Function

The list function returns a list of its arguments.

Format: (list v1 [v2 ... v8])

Arguments: There must be at least one argument and no more than eight arguments. The arguments may have any data type.

Action: The list function returns a value with a list data type. The value returned is a list of the arguments.

Example: (list 'a 'b 'c 'd) evaluates to (a b c d)

list_buffers Command

The list_buffers command lists all user-created text buffers.

Command Format: {ESC} X list_buffers
or
{CTRL-X} {CTRL-B}

Argument: A numeric argument, if specified, is ignored.

Action: EMACS overwrites the text screen with a list of all active external (that is, user-created) buffers. To restore the screen, you may use {CTRL-G}.

Note: This command will not show empty buffers or buffers in which (buffer_info dont_show) is true.

list_dir Function

The list_dir function lists directory information.

Format: (list_dir p [opt1 ... opt7])

Arguments: The argument p must be a string value. The arguments opt1 through opt7, if specified, must be among the atoms listed below.

Action: EMACS lists directory information for the wildcarded pathname specified by the argument p. The following options may be specified:

Option	Effect
files	A list of filenames is returned.
directories	A list of directories is returned.
segdirs	A list of segment directories is returned.
entry_names	Only the entry name portion of each pathname is retained.
insert_names	The directory information is inserted into the current text buffer at the current position, one entry per line.
sorted	Names are sorted alphabetically.
no_error_messages	Error messages are suppressed.

The list_dir function returns a list of all the strings of directory entries.

load_compiled Command and Function

The load_compiled command or function loads a fasload file that was saved by means of the dump_file command.

Command Format: {ESC} X load_compiled

Function Format: (load_compiled [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string or character value.

Action: If the argument s is not specified, EMACS prompts you in the minibuffer with "Fasdump file name:". The characters that you type are assigned to the string s.

EMACS forms a file name by appending the suffix .EFASL to the string specified by the variable s. EMACS loads and executes the PEEL program that was saved in that file, usually with a previous dump_file command.

load_lib Command and Function

The load_lib command or function loads a fasload format file.

Command Format: {ESC} X load_lib

Function Format: (load_lib s)

Arguments: A numeric argument, if specified, is ignored.

The argument s to the load_lib function must be a string.

Action: EMACS opens for input a file whose name is obtained by adding the suffix .EFASL to the string s. EMACS loads and executes that file as a fasload file. EMACS prints a message indicating whether the load was successful.

The load_lib function returns the value NIL.

load_package Command and Function

The load_package command or function loads a package.

Command Format: {ESC} X load_package

Function Format: (load_package [s])

Argument: The argument s, if specified, must be a string or character value.

Action: If the string s is not specified, EMACS prompts you for a pathname and assigns the string you type to the variable s.

EMACS opens the pathname specified by the string s, loads that package, and then executes it.

The load_package function returns the value NIL.

load_pl_source Command and Function

The load_pl_source command or function loads and executes a PEEL source file.

Command Format: {ESC} X load_pl_source

Function Format: (load_pl_source [s])

Argument: The argument s, if specified, must be a string or character value.

Action: If the argument s is not specified, EMACS prompts you for a file name and assigns the string you specify to the variable s.

EMACS opens the file specified by the string s and executes the text in that file as a PEEL source program.

The load_pl_source function returns the value NIL.

local_display_generator Function

The local_display_generator function displays a line on your screen in "printout" mode.

Format: (local_display_generator s [b])

Arguments: The argument s must be a string or character value. The argument b, if specified, must be a Boolean value.

Action: EMACS displays the string s on your screen in "printout" mode.

If the argument b is true or unspecified, the output line is terminated at that point. If the argument b is false, the next call to local_display_generator will append its data to the end of the current line.

Note: The major difference between the local_display_generator function and the print function is that local_display_generator does not put quotation marks around displayed string arguments.

looked_at Function

The looked_at function tests whether the text preceding the current cursor position is the same as a string argument.

Format: (looked_at s)

Argument: The argument s must be a string or character value.

Action: The looked_at function returns a Boolean value. If the characters in the text buffer preceding the current cursor position are the same as the string s, then looked_at returns true; otherwise, it returns false.

Example: The function

```
(if (looked_at "house") ...)
```

returns true if the characters preceding the current cursor position are the string "house".

looking_at Function

The looking_at function tests whether the characters in your buffer beginning at the current cursor position equal a given string value.

Format: (looking_at s)

Argument: The argument s must have a string or character value.

Action: The looking_at function returns a Boolean value. If the characters in your text buffer, beginning with the character at the current cursor position and continuing to the right, are equal to the characters of the argument s, then the value returned is true; otherwise, the value is false.

looking_at_char Function

The looking_at_char function tests the character at the current cursor position.

Format: (looking_at_char c)

Argument: The argument c must be a character value.

Action: The looking_at_char function returns a Boolean value. The value is true if the character at the current cursor position is the same as the argument c; otherwise the value is false.

Note: This is exactly the same as

    (looking_at c)

where c is character type.

lowercase_region Command and Function

The lowercase_region command or function changes all the uppercase letters in a region to lowercase.

Command Format: {ESC} X lowercase_region
or
{CTRL-X} {CTRL-L}

Function Format: (lowercase_region)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS changes all the uppercase letters in the region between mark and point to lowercase.

Caution: This command must be used with extreme care. If it is mistakenly applied to the wrong region of text in uppercase and lowercase, its effect must be undone manually.

The lowercase_region function returns the value NIL.

lowercase_word Command and Function

The lowercase_word command or function changes the text from the current cursor position to the end of the word into lowercase.

Command Format: [{ESC}n] {ESC} X lowercase_word
or
{ESC} L

Function Format: (lowercase_word [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, let n equal 1.

If the value of n is positive, EMACS converts to lowercase all uppercase letters in the region from the beginning of the current word (or the next word, if the cursor is on whitespace) to the end of the nth word, moving forward. The cursor moves to the end of that region.

If the value of n is negative, EMACS changes to lowercase all uppercase letters in the region ending at the end of the current word (or the previous word, if the cursor is on whitespace or the beginning of a word) and beginning at the beginning of the (-n)th word preceding the current cursor position. The cursor is left unchanged.

The lowercase_word function returns the value NIL.

major_window_count Function

The major_window_count function returns the number of major windows presently on the screen.

Format: (major_window_count)

Arguments: None.

Action: The major_window_count function returns an integer value. The integer value equals the number of major windows. You may use this number with do_n_times and select_any_window to loop through the windows.

make_array Function

The make_array function creates an array and returns it.

Format: (make_array d n)

Arguments: The argument d must be a quoted atom representing a legal PEEL data type. The argument n must be a positive integer.

Action: The make_array function returns an array value. The make_array function creates an array of n elements, each of which has the data type d, and returns the resulting array.

Example: The statement

```
(setq boxes (make_array 'integer 5))
```

creates an array called boxes. This array contains five integer elements, numbered 0 through 4.

make_cursor Function

The make_cursor function returns a cursor value generated from the arguments.

Format: (make_cursor s ln cp [st])

Arguments: The argument s must be a string value. The arguments ln and cp must be integer values. The argument st, if specified, must be a Boolean value.

Action: The make_cursor function returns a cursor value.

If the argument st is not specified, let st equal true.

EMACS forms a cursor value with buffer name specified by the string s, line number specified by the integer value ln, character position specified by the integer value cp, and sticky flag specified by the value st. EMACS returns that cursor value.

Note: The sticky flag currently has no effect.

mark Command and Function

The mark command or function either sets the mark or pops a mark.

Command Format: [{ESC}n] {ESC} X mark
or
[{ESC}n] {CTRL-@}

Function Format: (mark [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, EMACS sets the mark to the current cursor position.

If the argument n is specified, EMACS performs the pop mark function. This function pops the last mark saved with the push mark function, and moves the cursor to that position.

The mark function returns the value NIL.

mark_bottom Command and Function

The mark_bottom command or function places a mark at the bottom of the buffer, leaving the cursor unchanged.

Command Format: {ESC} X mark_bottom
or
{CTRL-X} {CTRL-Z} >

Function Format: (mark_bottom)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS places a mark at the bottom of the current text buffer, leaving the current cursor unchanged.

The mark_bottom function returns the value NIL.

mark_end_of_word Command and Function

The mark_end_of_word command or function places a mark at the end of the word, leaving the cursor unchanged.

Command Format: {ESC} X mark_end_of_word
or
{ESC} @

Function Format: (mark_end_of_word)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS places a mark at the end of the word, leaving the current cursor unchanged.

The mark_end_of_word function returns the value NIL.

mark_para Command and Function

The mark_para command or function puts a mark at the end of a paragraph and moves the cursor to the beginning of the paragraph.

Command Format: [{ESC}n] mark_para
or
[{ESC}n] {ESC} H

Function Format: (mark_para [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

EMACS puts a mark at the end of the nth paragraph following the current cursor position, and then moves the cursor back to the beginning of the current paragraph.

The mark_para function returns the value NIL.

mark_top Command and Function

The mark_top command or function places a mark at the top of the buffer, leaving the cursor unchanged.

Command Format: {ESC} X mark_top
or
{CTRL-X} {CTRL-Z} <

Function Format: (mark_top)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS places a mark at the top of the current text buffer, leaving the current cursor unchanged.

The mark_top function returns the value NIL.

mark_whole Command and Function

The mark_whole command or function places a mark at the end of the buffer and moves the current cursor to the beginning of the buffer.

Command Format: {ESC} X mark_whole
or
{CTRL-X} H

Function Format: (mark_whole)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS places a mark at the end of the current text buffer, and moves the current cursor to the beginning of the text buffer.

The mark_whole function returns the value NIL.

member Function

The member function tests whether a specified item is in a list.

Format: (member i lst)

Arguments: The argument i may have any data type. The argument lst must be a list.

Action: The member function returns a Boolean value. The value returned is true if the item i appears in the list lst; otherwise, the value is false.

Note: The test for whether the item i appears in the list lst is made using the = function, not the eq function.

merge_lines Command and Function

The merge_lines command or function merges two lines together.

Command Format: [{ESC}n] {ESC} X merge_lines
or
[{ESC}n] {ESC} ^

Function Format: (merge_lines [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, let n equal 1.

If the value of n is positive, EMACS repeats the following step n times: merge the current line with the next line in the current text buffer by replacing the newline characters separating them with a space.

If the value of n is negative, EMACS repeats the following step (-n) times: merge the preceding line of the current text buffer with the current line by replacing the newline characters separating them with a space.

The merge_lines function returns the value NIL.

minibuf_response Function

The minibuf_response function is an alternate name for the prompt function.

minibuffer_print Special Form

The minibuffer_print special form is an alternate name for the info_message function.

minibuffer_response Function

The minibuffer_response function is an alternate name for the prompt function.

mod_one_window Command and Function

The mod_one_window command or function transforms the current multiwindow display into a one-window display, saving information about the other window.

Command Format: {ESC} X mod_one_window
or
{CTRL-X} 1

Function Format: (mod_one_window)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS expands the current window on your display to the full screen, saving information about the other windows so that they can be restored by a future mod_split_window command.

The mod_one_window function returns the value NIL.

mod_split_window Command and Function

The mod_split_window command or function restores you screen to the two-window display in effect prior to the last mod_one_window command or function.

Command Format: {ESC} X mod_split_window
or
{CTRL-X} 2

Function Format: (mod_split_window)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS retrieves the information saved by the last mod_one_window command, and restores your screen to a two-window display.

If no previous mod_one_window command has been executed, EMACS creates a two-window display using, as a default, a buffer name of ALTERNATE for the second window.

If there were more than two windows on the screen at the time of the last mod_one_window display, EMACS chooses one of the windows that was removed and uses it as a second window. It is unpredictable which window is chosen.

The mod_split_window function returns the value NIL.

mod_write_file Command and File

The mod_write_file command and file writes the current buffer to a specified output file, prompting you in case the file already exists.

Command Format: {ESC} X mod_write_file
or
{CTRL-X} {CTRL-W}

Function Format: (mod_write_file [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified to the mod_write_file function, must be a string value.

Action: If the string s is not specified, EMACS prompts you for a filename, and assigns the string you typed to the variable s.

If the variable s is a null string (no characters), let s equal the name of the file currently associated with the current buffer.

If the file with pathname s already exists as a PRIMOS file, EMACS prompts you, asking whether you wish to replace the existing disk file. If you respond "n", the command is aborted.

EMACS saves the current text buffer to the output file specified by the pathname s.

The mod_write_file function returns the value NIL.

modulo Function

The modulo function computes the remainder obtained when one integer is divided by another.

Format: (modulo n d)

Arguments: The arguments n and d must be integer values.

Action: The modulo function returns an integer value, computed as follows:

- If the value of d is 0, let r equal n.
- If the value of d is nonzero, let r equal the remainder obtained when the numerator n is divided by the denominator d. If the remainder is nonzero, the sign of r always equals the sign of d.

The modulo function returns the value r.

more_args_p Function

The more_args_p function tests to see if more string arguments are pending.

Format: (more_args_p)

Arguments: None.

Action: Whenever you invoke a command, you can specify the string arguments for that command. These arguments are assigned, in turn, to each &args directives and to prompt directives, as well.

more_args_p indicates whether there are any unclaimed arguments from the invocation, and can be used to determine if a prompt will really occur.

move_bottom Command and Function

The move_bottom command or function moves the current cursor to the bottom of the buffer.

Command Format: {ESC} X move_bottom
or
{ESC} >

Function Format: (move_bottom)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS moves the current cursor to the bottom of the buffer.

The move_bottom function returns the value NIL.

move_top Command and Function

The move_top command or function moves the current cursor to the top of the buffer.

Command Format: {ESC} X move_top
or
{ESC} <

Function Format: (move_top)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS moves the current cursor to the top of the buffer.

The move_top function returns the value NIL.

multiplier Command

The multiplier command can be used immediately after a numeric argument specification to multiply the numeric argument by 4. The command can only be used when bound to a key. It is normally bound as {CTRL-U}.

Example: The command

{ESC} 22 {CTRL-U} x

inserts 88 x's into your buffer.

next_buf Command and Function

The next_buf command or function cycles to your next external buffer.

Command Format: {ESC} X next_buf
or
{ESC} N

Function Format: (next_buf)

Argument: A numeric argument, if specified, is ignored.

Action: This command is used to cycle through all your external (user-defined) buffers. The order of searching is as defined in the internal .buffers buffer.

EMACS searches for your current buffer in the .buffers buffer, finds the next buffer, and then changes your current buffer to the next buffer.

The next_buf function returns the value NIL.

next_buff Command and Function

The next_buff command or function is an alternate name for the next_buf command and function.

next_line Function

The next_line function moves the cursor to the next line.

Format: (next_line [n])

Arguments: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: The next_line function returns a Boolean value.

If n is not specified, let n equal 1.

If the value of n is 0, no cursor movement takes place and the value true is returned.

If the value of n is positive, EMACS moves the cursor down n lines and then to the beginning of the line. If the end of buffer is reached, cursor movement stops and the value false is returned; otherwise, the value true is returned.

If the value of n is negative, EMACS moves the cursor up n lines, and then to the beginning of the line. If the top of buffer is reached, cursor movement stops and the value false is returned; otherwise, the value true is returned.

next_line_command Command and Function

The next_line_command command or function moves the cursor to the next line.

Command Format: [{ESC}n] {ESC} X next_line_command
or
[{ESC}n] {CTRL-N}

Function Format: (next_line_command [n])

Arguments: If the argument n is specified, it must be an integer value, which may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If n equals 0, no action takes place.

If the value of n is positive, EMACS moves the cursor vertically down n lines, stopping if the bottom of the buffer is reached.

If the value of n is negative, EMACS moves the cursor vertically up n lines, stopping if the top of the buffer is reached.

next_page Command and Function

The next_page command or function moves the window forward a group of lines, usually 18.

Command Format: [{ESC}n] {ESC} X next_page
or
[{ESC}n] {CTRL-V}

Function Format: (next_page [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place. If the value of n is positive, the window moves forward n pages, stopping if the end of buffer is reached. The cursor is left at approximately the middle of the window.

If the value of n is negative, the window moves back n pages, stopping if the beginning of the buffer is reached. The cursor is left at approximately the middle of the window.

The next_page function returns the value NIL.

not Function

The not function returns the logical negation of its argument.

Format: (not b)

Argument: The argument b must be a logical value.

Action: The not function returns a logical value. If the value of the argument b is true, then not returns the value false; otherwise, not returns the value true.

nth Function

The nth function returns the nth character in a string.

Format: (nth s n)

Arguments: The argument s must be a string or character value. The argument n must be an integer value.

Action: The nth function returns a character value, computed as follows:

- Let k equal the number of characters in the string s.
- If 1<=n<=k, EMACS returns the character from position n of string s.
- Otherwise, EMACS returns a null string.

nthcar Function

The nthcar function returns the nth car of a list.

Format: (nthcar lst n)

Arguments: The argument lst must be a list value. The argument n must be an integer value.

Action: The nthcar function returns a value whose data type depends upon the arguments.

The value returned is the nth car of the list lst. This is the item of the list lst in position n.

If the list's length is shorter than n, the function returns the value NIL.

null Function

The null function tests whether its argument is a null list.

Format: (null v)

Argument: The argument v may have any data type.

Action: The null function returns a Boolean value. The value is true if v is a null list; otherwise, the value is false.

Notes: The null list, (), is often represented in this book as NIL. Either representation works in a PEEL program, as long as you never setq NIL.

You can use the null function in a loop through the elements of a list, applying it to the cdr of the list to test if there are any items remaining.

This function is equivalent to either

(= v NIL)

or

(eq v NIL)

numberp Function

The numberp function tests whether its argument is a number.

Format: (numberp v)

Argument: The argument v may have any data type.

Action: The number p function returns a Boolean value. The value returned is true if v has an integer data type; otherwise, the value is false.

numeric_argument Function

The numeric_argument function returns the numeric argument to a defcom command.

Format: (numeric_argument [n])

Argument: The argument n, if specified, must be an integer value.

Action: The numeric_argument function may return either a numeric argument or NIL.

The numeric_argument function is normally used in the PEEL source for a defcom command. When the defined command is invoked, if there is a numeric argument to the invocation of the command, the numeric_argument function returns the value of that numeric argument. If the command is invoked with no numeric argument, then the argument n serves as a default value:

- If an argument n has been specified, the numeric_argument function returns the value of n.
- If there is no argument n, the numeric_argument function returns the value NIL.

If the numeric_argument function is used other than in the PEEL source for a defcom, it returns the value NIL.

Example: The standard defcom produced by expand_macro uses numeric_argument, with a default of 1, to control the number of times the defined command is executed.

```
(defcom simple_example
  (do_n_times (numeric_argument 1)
  (print "EMACS is extensible.")
))
```

Note: The numeric_argument function is similar to the defcom option &na. For instance, &na (&pass foo &default 2) is equivalent to (setq foo (numeric_argument 2)).

one_window Command and Function

The one_window command or function transforms the current multiwindow display into a one window display.

Command Format: {ESC} X one_window

Function Format: (one_window)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS expands the current window on your display to the full screen.

The one_window function returns the value NIL.

Note: You may use the mod_one_window command to save information about the window being removed, so that it can be restored by a future mod_split_window command.

open_line Command and Function

The open_line command or function inserts a carriage return at the current cursor position without moving the cursor.

Command Format: [{ESC}n] {ESC} X open_line
or
[{ESC}n] {CTRL-O}

Function Format: (open_line [n]

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If n is positive, EMACS repeats the following step n times: insert a carriage return into the buffer at the current cursor position, and then move the cursor back over the character just inserted.

Therefore, this command is equivalent to the cr command followed by the back_char command.

The open_line function returns the value NIL.

or Function

The or function is a Boolean operator that has the logical "or" as its arguments.

Format: (or b1 b2 [b3 ... b8])

Arguments: The or function takes at least two arguments and no more than eight arguments. All arguments must have the Boolean data type.

Action: The or function returns a Boolean value computed by taking the logical "inclusive or" of all of its arguments. That is, the or function returns the value false if the values of all its arguments are false; otherwise, it returns the value true. All arguments are evaluated regardless of whether any is true. Order of evaluation is unspecified.

other_window Command and Function

The other_window command or function switches the cursor between the current window and the last previously used window.

Command Format: {ESC} X other_window
or
{CTRL-X} O

Function Format: (other_window)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS switches the cursor from the current window to the last window on the screen that was previously used.

The other_window function returns the value NIL.

otherwise Keyword

The otherwise keyword specifies an action to be taken if none of the criteria in a select or a dispatch have been met. It must be the last item in these special forms.

overlay_off Command and Function

The overlay command or function turns off overlay mode.

Command Format: {ESC} X overlay_off

Function Format: (overlay_off)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns off overlay mode, so that when you type characters to be inserted into your text buffer, the new characters do not overlay existing characters.

The overlay_off function returns the value NIL.

overlay_on Command and Function

The overlay_on command or function turns on overlay mode.

Command Format: {ESC} X overlay_on

Function Format: (overlay_on)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS turns on overlay mode. In this mode, any character that you type at the keyboard overlays the existing character on your screen, rather than being inserted between existing characters.

The overlay_on function returns the value NIL.

overlay_rubout Command and Function

The overlay_rubout command or function deletes a character in overlay mode.

Command Format: [{ESC}n] {ESC} X overlay_rubout

Function Format: (overlay_rubout [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the value of n is not specified, let n equal 1.

EMACS deletes the character to the left of the current cursor and inserts a space to replace it. The cursor is left at the space that was inserted.

The overlay_rubout function returns the value NIL.

overlayer Command and Function

The overlayer command or function performs the actual overlaying in overlay mode.

Command Format: {ESC} X overlayer

Function Format: (overlayer [c] [n])

Arguments: The argument c, if specified, must be a character value. The argument n, if specified, must be an integer value.

Action: The last character of a key sequence bound to the overlayer command (or, in the case of the overlayer function, the character c), replaces the current character on the screen.

Example: See the self_insert function.

pending_reenter Function

The pending_reenter function is the same as the suppress_redisplay function.

pl Command and Function

The pl command or function compiles and executes the contents of the current buffer.

Command Format: {ESC} X pl

Function Format: (pl)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS compiles and executes the contents of the current text buffer.

The pl function returns the value NIL.

Note: The buffer cannot exceed 32767 characters.

pl_minibuffer Command

The pl_minibuffer command lets you type an expression into the minibuffer that EMACS executes as a PEEL statement.

Format: {ESC} X pl_minibuffer
or
{ESC} {ESC}

Argument: A numeric argument, if specified, is ignored.

Action: EMACS prompts you in the minibuffer for a line containing a PEEL statement. EMACS then executes the statement that you type.

point_cursor_to_string Function

The point_cursor_to_string function returns the text between the argument cursor and the current cursor.

Format: (point_cursor_to_string cur)

Argument: The argument cur must be a cursor value in the same buffer as the current cursor.

Action: The point_cursor_to_string function returns a string value consisting of all the text in the current buffer between the current cursor position and the argument cur.

popmark Command and Function

The popmark command or function pops a mark off the top of the mark stack.

Command Format: {ESC} X popmark

Function Format: (popmark)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS pops the mark off the top of the mark stack and sets the mark to that position.

The popmark function returns the value NIL.

Note: Use the pushmark command or function to store a mark on the mark stack.

prepend_to_buf Command and Function

The prepend_to_buf command or function prepends the current region to a buffer. The word "prepend" means that the insertion is made at the beginning of the buffer.

Command Format: [{ESC}n] {ESC} X prepend_to_buf
or
[{ESC}n] {CTRL-X} P

Function Format: (prepend_to_buf [n [s]])

Argument: The numeric argument n, if specified, must be an integer value.

The argument s, if specified, must be a string value.

Action: If the argument s is unspecified, EMACS prompts you for a buffer name and assigns the resulting string to the variable s.

EMACS prepends the current region to the buffer whose name is specified by the string s. This means that the text in the current region is inserted at the beginning of that buffer.

If n is not specified, the text in the current region is deleted, meaning that the prepend operation is, in effect, a move. If n is specified, the text is copied and the marked region is not deleted.

prepend_to_file Command and Function

The prepend_to_file command or function prepends the current region to a file. The word "prepend" means that the text is inserted at the beginning of the file.

Command Format: [{ESC}n] {ESC} S prepend_to_file
or
[{ESC}n] {CTRL-X} {CTRL-Z} P

Function Format: (prepend_to_file [n [s]])

Argument: The argument n, if specified, must be an integer value.

Action: If the string s is not specified, EMACS prompts you for a filename. The resulting string is assigned to the variable s.

EMACS prepends the current region to that file. This means that the text in the current marked region is inserted at the beginning of that file.

If n is not specified, the text in the current region is deleted, meaning that the prepend operation is, in effect, a move. If n is specified, the text is copied and the marked region is not deleted.

prev_buf Command and Function

The prev_buf command or function cycles you to the previous external buffer.

Command Format: {ESC} X prev_buf
or
{ESC} P

Function Format: (prev_buf)

Argument: A numeric argument, if specified, is ignored.

Action: This command is used to cycle through all your external (user-defined) buffers. The order of searching is in reverse order from the order defined in the internal .buffers buffer.

EMACS searches for your current buffer in the .buffers buffer, finds the preceding buffer in the .buffers buffer, and then changes the current buffer to that buffer.

The prev_buf function returns the value NIL.

prev_line Function

The prev_line function moves the cursor to the previous line.

Format: (prev_line [n])

Argument: The argument n, if specified, must be an integer whose value that may be positive, 0, or negative.

Action: The prev_line function returns a Boolean value.

If n is not specified, let n equal 1.

If the value of n is 0, no cursor movement takes place and the value true is returned.

If the value of n is positive, EMACS moves the cursor up n lines, and then to the beginning of the line. If the top of the buffer is reached, cursor movement stops and the value false is returned; otherwise, the value true is returned.

If the value of n is negative, EMACS moves the cursor down (-n) lines, and then to the beginning of the line. If the end of the buffer is reached, cursor movement stops and the value false is returned; otherwise, the value true is returned.

prev_line_command Command and Function

The prev_line_command command or function moves the cursor to the previous line.

Command Format: [{ESC}n] {ESC} X prev_line_command
or
[{ESC}n] {CTRL-Z}

Function Format: (prev_line_command [n])

Argument: If the argument n is specified, it must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If n equals 0, no action takes place.

If the value of n is positive, EMACS moves the cursor vertically up n lines, stopping if the beginning of the buffer is reached.

If the value of n is negative, EMACS moves the cursor vertically down (-n) lines, stopping if the end of the buffer is reached.

Note: EMACS maintains the horizontal character position as well as possible.

primos_command Command and Function

The primos_command command or function executes a PRIMOS command without leaving EMACS.

Command Format: {ESC} X primos_command
or
{CTRL-X} {CTRL-E}

Function Format: (primos_command [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you with "primos command:". The string that you type in response is assigned to the variable s.

EMACS proceeds as follows:

- If the first character in the string s is not !, EMACS executes

  (primos_external s)

- If the first character of the string s is !, but the second character of the string is not !, EMACS executes

  (primos_internal_como s2)

  where s2 is a string equal to s, but with the leading ! removed.

- If the first two characters of the string s are !!, EMACS runs

  (primos_internal_screen s2)

  where s2 is a string equal to s, but with the first two characters removed.

The primos_command function returns the value NIL.

Note: At Revision 19.4 of PRIMOS, any PRIMOS command except "EMACS" may be executed via primos_internal_como or primos_internal_screen. External commands no longer overwrite EMACS.

primos_external Command and Function

The primos_external command or function executes a PRIMOS command by means of a separate phantom job.

Command Format: {ESC} X primos_external

Function Format: (primos_external [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you with "external command:". The characters that you type in response are assigned to the string variable s.

EMACS runs a separate phantom job to execute the PRIMOS command specified by the string s. It waits for the command to complete and then displays the results in the file_output buffer.

After the phantom job has terminated, EMACS creates or overwrites the text buffer named file_output, loading into it the text of the listing file created by the phantom job. You may continue editing your original file by switching back to the buffer into which that file was loaded.

The primos_external function returns the value NIL.

primos_internal_como Command and Function

The primos_internal_como command or function runs a PRIMOS command with como output.

Command Format: {ESC} X primos_internal_como

Function Format: (primos_internal_como [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you with "internal command:". The characters that you type in response are assigned to the string s.

The characters of the string s must form a PRIMOS command not requiring any interactive keyboard input from the user. EMACS runs the command using CP$.

After execution of the command is completed, EMACS creates or overwrites the text buffer named file_output, loading into it the text or the command output.

primos_internal_quiet Command and Function

The primos_internal_quiet command or function executes a PRIMOS command, overwriting your screen with any terminal output.

Command Format: {ESC} X primos_internal_quiet

Function Format: (primos_internal_quiet [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you with "quiet command:". The characters you type in response are assigned to the string s.

The characters of the string s may contain any PRIMOS command except "EMACS". EMACS runs the command using CP$.

After execution of the command is completed, EMACS overwrites your screen display with the terminal output from the command. You may clear your screen by using {CTRL-L}.

The primos_internal_quiet function returns the value NIL.

Note: This command is useful for executing PRIMOS commands that normally do not produce any terminal output, such as ATTACH.

primos_internal_screen Command and Function

The primos_internal_screen command or function executes a PRIMOS command.

Command Format: {ESC} X primos_internal_screen

Function Format: (primos_internal_screen [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified to the primos_internal_screen function, must be a string value.

Action: If the argument s is not specified to the primos_internal_screen function, or if the primos_internal_screen command is used, EMACS prompts you with "internal command:". The characters that you type in response are assigned to the string variable s.

The characters of the string s may be any PRIMOS command except "EMACS". EMACS executes the command using CP$.

After execution of the command is completed, EMACS clears your display screen and displays the terminal output from the command on your screen. You may restore your screen by typing {CTRL-G}.

The primos_internal_screen function returns the value NIL.

Note: This is used for interactive commands. The terminal is reset, the command is executed, and when command execution is completed, typing any character restores your EMACS screen.

primos_recycle Command

The primos_recycle command runs the PRIMOS rescheduler. It should not normally be used by PEEL programmers.

primos_smsgl Function

The primos_smsgl function sends a PRIMOS message to a specified user.

Format: (primos_smsgl a s)

Arguments: The argument a must be either a string value or an integer value. The argument s must be a string value.

Action: EMACS transmits a PRIMOS message to a specified user. The argument a specifies the addressee either as a string value representing the name or as an integer value representing the process number. (None of the usual PRIMOS MESSAGE command options, such as -ON or -NOW, are supported.)

The argument s is the message to be sent.

If the message was successfully sent, primos_smsgl function returns the value NIL.

prinl Function

The prinl function prints a value or inserts a value into your text. The prinl function is similar to the print function, except that no new line is displayed or inserted.

Format: (prinl v [cur])

Arguments: The argument v may have any data type.

The argument cur, if specified, must be a cursor value.

Action: The prinl function returns a value whose data type equals the data type of the argument v.

If the argument cur is specified, EMACS sets the current cursor to cur, and then inserts a printed representation of the value of the argument v into your text buffer at that cursor position.

If the argument cur is not specified, EMACS displays the value of the argument v on your screen. To restore your screen, type {CTRL-G}.

The prinl function returns the value of v.

Note: The printed line is not terminated. Thus, the results of consecutive prinl invocations will be concatentated on the same line.

print Function

The print function displays or inserts a specified value followed by a newline.

Format: (print v [cur])

Arguments: The argument v may have any data type.

The argument cur, if specified, must be a cursor value.

Action: The print function returns a value whose data type equals the data type of the argument v.

If the argument cur is specified, EMACS changes the current cursor to cur, and then inserts a printed representation the value of the argument v, followed by a newline character, into your text buffer at that position.

If the argument cur is not specified, EMACS displays the value of the argument v, followed by a newline character, on your screen. To restore your screen, type {CTRL-G}.

The format of the value displayed depends upon the data type of the variable v. If v is an atom, the name of the atom is printed; if v is an integer, a number is printed; if v is a string, the string is printed in quotation marks, with appropriate escape sequences; if v is a character, the character (preceded by a backslash) is printed. For all other data types, a bracketed descriptor describing the item is displayed.

The print function returns the value of the argument v.

Note: The difference between the print and local_display_generator functions is that the latter does not display string arguments enclosed in quotation marks.

progn Special Form

The progn function evaluates one or more expressions and returns the value of the last expression.

Format: (progn s1 [s2 ...])

Arguments: The arguments s1, s2, and ..., when specified, may be any PEEL expressions.

Action: EMACS evaluates each of the arguments s1, s2, and ... in turn, and returns the value of the last one.

prompt Function

The prompt function prompts the user for a string value.

Format: (prompt s)

Argument: The argument s must be a string value.

Action: The prompt function returns a string value.

EMACS displays the string s, followed by a colon symbol (:), in the minibuffer, and awaits a typed response. The characters typed in response form the string value that is returned by the prompt function.

Note: This function will return the next pending argument, in place of a prompt, if there are more arguments pending. (See the more_args_p function.)

prompt_for_integer Function

The prompt_for_integer function prompts the user for an integer value.

Format: (prompt_for_integer s n)

Arguments: The argument s must be a string value. The argument n must be an integer value.

Action: The prompt_for_integer function returns an integer value.

EMACS displays the contents of the string s, followed by a colon (:), in the minibuffer, and awaits a typed response. The typed response must be an optional string of characters followed by a newline. The characters must form a legal decimal number, optionally signed.

If no characters are typed prior to the newline, then the prompt_for_integer function returns the value n; otherwise, the prompt_for_integer function returns the value of the integer typed.

Note: This function will return the next pending argument, in place of a prompt, if there are more arguments pending. (See the more_args_p function.)

prompt_for_string Function

The prompt_for_string function prompts the user for a string and returns the value typed.

Format: (prompt_for_string s t)

Arguments: The arguments s and t must be string values.

Action: The prompt_for_string function returns a string value.

EMACS displays the string s, followed by a colon symbol (:), in the minibuffer, and awaits a typed user response. The typed response is an optional string of characters ending with a newline.

If a string of characters is typed prior to the newline, then the prompt_for_string function returns a string containing the characters typed; otherwise, the prompt_for_string function returns the value t.

Note: This function will return the next pending argument, in place of a prompt, if there are more arguments pending. (See the more_args_p function.)

pushmark Command and Function

The pushmark command or function pushes the mark onto the mark stack and sets a new mark.

Command Format: {ESC} X pushmark

Function Format: (pushmark)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS pushes the position of the current mark onto the mark stack, and sets a new mark at the current cursor position.

Note: The stacked mark value may be restored by means of the popmark command or function.

putprop Function

The putprop LISP function assigns a tag and a value to a property list (plist). Plist is an association list.

Format: (putprop pl v t)

Arguments: The arguments pl and t must be atoms, usually quoted. The argument v may have any data type.

Action: The putprop function assigns the tag t and the value v to the property list for the atom pl.

The putprop function returns the value NIL.

Note: If an atom has a property list, the get function can be used to determine the value corresponding to a desired tag in the property list. The remprop function removes a tag and value from the property list. EMACS has no way of accessing the entire contents of an atom's plist.

Example: If the atom Jane is previously undefined, then the statement

```
(putprop 'Jane 6 'age)
```

gives Jane a property list with the value

```
((age 6))
```

query_replace Command and Function

The query_replace command or function replaces occurrences of one string with another, querying you prior to each replacement.

Command Format: {ESC} X query_replace
or
{ESC} %

Function Format: (query_replace [s [t]])

Arguments: A numeric argument, if specified, is ignored.

The arguments s and t, if specified to the query_replace function, must be string values.

Action: If the argument s is not specified, EMACS prompts you with "search for:". The characters you type prior to a newline are assigned to the string s.

If the character string t is not specified, EMACS prompts you with "replace with:". The characters you type are assigned to the string t.

EMACS searches the current marked region, starting from the top of the region, for occurrences of the string s. When the search fails, EMACS displays "not found" and terminates the query_replace function.

For each occurrence found, EMACS positions the cursor at the end of the occurrence of the string s and prompts the user for a single character, which may be any of the following:

- Space to replace the occurrence of s with the string t in the text buffer
- Return for no replacement
- Period to replace the occurrence of s with t, and terminate query_replace
- {CTRL-G} to terminate query_replace with no replacement

When the query_replace function terminates, EMACS moves the cursor back to its original position prior to the execution of the function.

The query_replace function returns the value NIL.

quit Command and Function

The quit command returns you from EMACS to its invocation process, usually PRIMOS.

Command Format: {ESC} X quit
or
{CTRL-X} {CTRL-C}

Function Format: (quit)

Argument: A numeric argument, if specified, is ignored.

Action: If there are modified buffers, EMACS overwrites your screen with a list of modified buffers that have not been saved to files, and asks you if you wish to quit executing anyway. If you type "y", EMACS terminates immediately, returning to the process that invoked it, usually PRIMOS. If you type "n", you simply return to editing.

Note: The function quit simply sets a flag indicating EMACS should exit, rather than actually having the exit occur at that time.

quote Function

The quote function is a LISP function that returns its argument without evaluating it.

Format: (quote x)
or
'x

Argument: The argument x may have any data type.

Action: The quote function prevents the normal evaluation of the argument x. It returns x, not the value of x.

quote_command Command

The quote_command command accepts the next character typed literally for insertion into the text buffer.

Format: [{ESC}n] {ESC} X quote_command
or
[{ESC}n] {CTRL-Q}

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, let n equal 1.

EMACS inserts the typed character n times into the text buffer. Nonprinting characters are usually displayed as a question mark (?), or as a rectangular block.

range_to_string Function

The range_to_string function returns a character string containing the text between two cursors.

Format: (range_to_string cur1 cur2)

Arguments: The arguments cur1 and cur2 must be cursor values.

Action: The range_to_string function returns a character string value containing all the characters in your text buffer between cur1 and cur2. The string cannot be longer than 32767 characters.

Note: It is not required that cur1 precede cur2.

read Function

The read function reads and returns a PEEL form beginning at the specified cursor position.

Format: (read cur)

Argument: The argument cur must be a cursor value.

Action: Beginning at the cursor position indicated by cur, EMACS reads a PEEL form from your text buffer and returns it as a list that can then be evaluated.

read_character Function

The read_character function reads a character from the terminal and returns it.

Format: (read_character [raw])

Argument: An argument, if specified, must be the atom raw.

Action: The read_character function returns a character value.

EMACS reads a single character from the terminal. If raw is specified, the character read is done as a raw read, that is, with no help_on_tap processing.)

EMACS returns this character.

read_file Command and Function

The read_file command or function reads a file into your current text buffer.

Command Format: {ESC} X read_file
or
{CTRL-X} {CTRL-R}

Function Format: (read_file [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a character-string value.

Action: If the argument s is not specified, EMACS prompts you with "read file:", and assigns the characters you type prior to a newline to the string s.

If the current buffer is not empty, EMACS prompts you with the question, "buffer is not empty, delete it?". If you respond with "n", EMACS aborts the command; if you respond "y", EMACS proceeds as follows:

- It deletes the contents of the current buffer, if any.
- If the string s contains a valid pathname, EMACS opens the file specified by that pathname and loads it into your current buffer.

The read_file function returns the value NIL.

Note: If s is a bad pathname, the previous contents of the buffer will have been destroyed.

redisplay Function

The redisplay function forces EMACS to update the screen to reflect the current state.

Format: (redisplay)

Arguments: None.

Action: EMACS updates the screen to reflect the current state. The redisplay function returns the value NIL.

reexecute Command

The reexecute command reexecutes the command entered most recently at the keyboard.

Format: [{ESC}n] {ESC} X reexecute
or
[{ESC}n] {CTRL-C}

Argument: The argument n, if specified, must be a numeric value.

Action: EMACS reexecutes the last command entered at the keyboard. If the argument n is specified and greater than zero, EMACS reexecutes the command n times.

refresh Command and Function

The refresh command or function repaints the display screen.

Command Format: [{ESC}n] {ESC} X refresh
or
[{ESC}n] {CTRL-L}

Function Format: (refresh [n])

Argument: The argument n, if specified, must be a numeric value.

Action: If the value of n is not specified, EMACS totally refreshes the screen. That is, the screen is cleared and repainted. Otherwise, EMACS repaints your screen so that the line in the current cursor position is in the nth line in your window.

reject Command

The reject command is equivalent to executing an undefined command, and it prints "Invalid command:" in the minibuffer. It is typically used to rebind keys that are to be disabled.

remassoc Function

The remassoc function removes an item from a LISP-style association list.

Format: (remassoc k lst)

Arguments: The argument lst must be an association list. The argument k may have any data type.

Action: The remassoc function returns the sublist of the list obtained by removing the first sublist associated with the key k in lst.

Example: After the following:

```
(setq a '((foo bar) (go stop) (hi there) (foo bang)))
(remassoc 'foo a)
```

the value of a becomes

```
((go stop) (hi there) (foo bang))
```

Notice that only the first sublist with key foo is removed.

Notice also that there is no need to assign the result of remassoc to a again with setq. The remassoc function automatically changes the value of a as a side-effect.

remove Function

The remove function removes a member from a list.

Format: (remove i lst)

Arguments: The argument i may have any data type. The argument lst must be a list value.

Action: The remove function returns a list value. The list value returned is obtained by removing the item i from the list lst.

Note: The test for whether the item i appears in the list lst is made with the = function rather than the eq function.

remove_charset Function

The remove_charset function removes all specified characters from a string.

Format: (remove_charset s1 s2)

Arguments: The arguments s1 and s2 must be string values.

Action: The remove_charset function returns a string value. The string value returned is computed by removing from string s1 any and all characters that also appear in string s2.

remprop Function

The remprop LISP function removes a tag and associated value from a property list.

Format: (remprop pl t)

Arguments: The arguments pl and t must be atoms, usually quoted.

Action: If the atom pl has a property list with a tag equal to the tag t, EMACS removes the tag t and the corresponding value from that property list.

The remprop function returns the value NIL.

Note: If an atom has a property list, the get function can be used to determine a value corresponding to a desired tag in the property list.

repaint Command and Function

The repaint command or function moves the cursor to the specified window line.

Command Format: [{ESC}n] {ESC} X repaint
or
[{ESC}n] {CTRL-X} R

Function Format: (repaint [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, let n equal 1.

EMACS moves the cursor to line n on your screen display.

The repaint function returns the value NIL.

replace Command and Function

The replace command or function replaces all instances of one string with another in a current region.

Command Format: {ESC} X replace

Function Format: (replace [s1 [s2]])

Arguments: The arguments s1 and s2, if specified, must be string values.

Action: If the string s1 is not specified, EMACS prompts the user with "search for:". The characters typed prior to a newline are assigned to the string s1.

If the argument s2 is not specified, EMACS prompts you with "replace with:". The characters typed prior to a newline are assigned to the string s2.

EMACS replaces all occurrences of the string s1 in your current region with the characters of the string s2. After all replacements have been made, EMACS returns the cursor to its original position prior to the execution of replace.

The replace function returns the value NIL.

reset Command and Function

The reset command or function puts the screen back in one-window mode, sets hcol to one, and refreshes the screen.

Command Format: {ESC} X reset

Function Format: (reset)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS resets several status values to their defaults. You are returned to one-window mode, hcol (the leftmost column displayed on the screen) is set to one, and the screen is refreshed.

The reset function returns the value NIL.

reset_tabs Command and Function

The reset_tabs command or function sets tabs at every five spaces up to column 130.

Command Format: {ESC} X reset_tabs

Function Format: (reset_tabs)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS sets the tabs at every five spaces, in columns 5, 10, 15, and so forth, up to column 130.

The reset_tabs function returns the value NIL.

rest$ Function

The rest$ function is the same as the suffix$ function.

rest_of_line Function

The rest_of_line function returns a character string containing the rest of the current line, excluding the newline character.

Format: (rest_of_line [cur])

Argument: The argument cur, if specified, must be a cursor value.

Action: The rest_of_line function returns a string value.

If the argument cur is not specified, let cur equal the current cursor position.

EMACS forms a string containing all the characters in the text from cur to the end of the line containing cur, and returns that string value.

restrict_to_sui$ Function

The restrict_to_sui$ function is used in definitions of functions that are restricted to SUI.

Format: (restrict_to_sui$)

Arguments: None.

Action: If you are not using the SUI interface, EMACS displays the error message "That command is not used in this interface." You use this function in definitions of functions and commands that are to be invoked only from the SUI interface.

return Function

The return function terminates a PEEL function by returning to the caller.

Format: (return [x])

Argument: The argument x, if specified, may have any data type.

Action: If the argument x is not specified, let x equal NIL.

EMACS terminates the current function and returns to the caller, with the value x as the function value. The data type of x should match that specified by the function's &returns declaration, if specified.

reverse_search Function

The reverse_search function searches back in the text buffer for a specified string, and returns a Boolean value indicating success or failure.

Format: (reverse_search s)

Argument: The argument s must be a string value.

Action: EMACS searches the text buffer back from the current cursor position for the string s.

If the search is successful, EMACS moves the cursor to the first character of the matching string in the text buffer, and returns the value true.

If the search fails, the cursor is left unchanged, and the function returns the value false.

reverse_search_command Command and Function

The reverse_search_command command or function searches back in the text buffer for a string.

Command Format: {ESC} X reverse_search_command
or
{CTRL-R}

Function Format: (reverse_search_command [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified, must be a string value.

Action: If the argument s is unspecified, EMACS prompts you and assigns the resulting string to the variable s.

EMACS searches the text buffer back from the current cursor position for the string s. If the search succeeds, EMACS moves the current cursor to the first character of the matching string in the text buffer. The matched string cannot end beyond the current cursor position.

If the search is unsuccessful, EMACS displays an error message.

ring_the_bell Function

The ring_the_bell function sends a {CTRL-G} to your terminal.

Format: (ring_the_bell)

Arguments: None.

Action: EMACS sends a {CTRL-G} character to your terminal producing a sound.

The ring_the_bell function returns the value NIL.

rubout_char Command and Function

The rubout_char command or function deletes the character to the left of the current cursor.

Command Format: [{ESC}n] {ESC} X rubout_char
or
[{ESC}n] {CTRL-H}
or
[{ESC}n] {backspace}
or
[{ESC}n] {delete}

Function Format: (rubout_char [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS deletes n characters, beginning with the character preceding the current cursor position and continuing backward, stopping if the beginning of the buffer is reached.

If the value of n is negative, EMACS deletes (-n) characters beginning with the character at the current cursor position and continuing forward, stopping if the end of the buffer is reached.

The rubout_char function returns the value NIL.

rubout_word Command and Function

The rubout_word command or function deletes one or more words backwards in the current buffer.

Command Format: [{ESC}n] {ESC} X rubout_word
or
[{ESC}n] {ESC} {CTRL-H}
or
[{ESC}n] {ESC} {backspace}
or
[{ESC}n] {ESC} {delete}

Function Format: (rubout_word [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If the argument n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS deletes n words beginning with the word preceding the current cursor position and moving backward. Deletion stops if the beginning of the buffer is reached.

If the value of n is negative, EMACS deletes (-n) words in the text, beginning with the word at the current cursor position and continuing forward. Deletion stops if the end of the buffer is reached.

The rubout_word function returns the value NIL.

same_buffer_p Function

The same_buffer_p function returns a Boolean value indicating if two cursors point into the same text buffer.

Format: (same_buffer_p cur1 cur2)

Arguments: The arguments cur1 and cur2 must be cursor values.

Action: The same_buffer_p function returns a Boolean value. If cur1 and cur2 are cursor values within the same text buffer, the function returns the value true; otherwise, it returns false.

save_all_files Command and Function

The save_all_files command or function saves all files that have been modified.

Command Format: {ESC} X save_all_files

Function Format: (save_all_files)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS saves the contents of every modified buffer into its associated file. Unmodified buffers are ignored.

The save_all_files function returns the value NIL.

save_excursion Special Form

The save_excursion special form saves the current cursor position and modes, executes all of its arguments, and restores the original cursor and modes.

Format: (save_excursion s1 [s2 ...])

Arguments: The arguments s1, s2, and ..., if specified, must be legal PEEL statements.

Action: EMACS saves the current cursor and all values and modes. Then EMACS executes s1, s2, and so forth. When completed, EMACS restores the saved cursor, values, and modes. The cursor is repositioned to the center of the window.

The save_excursion function returns the value NIL.

save_file Command and Function

The save_file command or function writes the current buffer to the file associated with that buffer.

Command Format: {ESC} X save_file
or
{CTRL-X} {CTRL-S}
or
{CTRL-X} S

Function Format: (save_file)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS writes the current text buffer out to a file, using the filename associated with the buffer. If the file already exists on disk, this operation overwrites it.

save_position Special Form

The save_position special form is the same as the save_excursion special form, except that the current cursor is not repositioned.

save_tab Command and Function

The save tab command or function saves the current tab positions in a file.

Command Format: {ESC} X save_tab

Function Format: (save_tab)

Arguments: A numeric argument, if specified, is ignored.

Action: EMACS prompts you for a filename. Let the typed string be assigned to the variable s. EMACS next prompts you for the name by which the tabs are to be stored in the file. Let that typed string be assigned to the variable t.

EMACS saves the current tab settings into the file specified by the string s. They are stored in PEEL as a setq statement, which if executed, will assign the list of tab stops to a variable named by the string t.

The save_tab function returns the value NIL.

Note: You may restore the saved tab settings by using the get_tab command or function.

say_more Function

The say_more function is the same as the local_display_generator function.

scan_errors Function

The scan_errors function scans the current buffer for errors in the format of the specified language.

Format: (scan_errors lng)

Argument: The argument lng must be an unquoted atom.

Action: The argument lng must be one of the following atoms: C, RPG, FTN, PMA, or TSI. (TSI refers to all of Prime's common-envelope compilers: PL/1, PL/1G, PASCAL, CBL, VRPG, and SPL.) EMACS scans the current buffer for errors in the format specified by the language atom lng. This is used by COMPILE.

scroll_other_backward Command and Function

The scroll_other_backward command or function scrolls backward that window reached by the other_window command or function.

Command Format: [{ESC}n] {ESC} X scroll_other_backward
or
[{ESC}n] {CTRL-X} V

Function Format: (scroll_other_backward [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS scrolls the other window back n lines.

If the value of n is negative, EMACS scrolls the other window forward (-n) lines.

The scroll_other_backward function returns the value NIL.

scroll_other_forward Command and Function

The scroll_other_forward command or function scrolls forward that window reached by the other_window command or function.

Command Format: [{ESC}n] {ESC} X scroll_other_forward
or
[{ESC}n] {ESC} {CTRL-V}

Function Format: (scroll_other_forward [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS scrolls the other window forward n lines.

If the value of n is negative, EMACS scrolls the other window back (-n) lines.

The scroll_other_forward function returns the value NIL.

search Function

The search function returns the position of the first character of a string that also appears in a second string.

Format: (search s t)

Arguments: The arguments s and t must be string or character values.

Action: The search function returns an integer value.

If no characters in string t occur in string s, search returns the value 0.

If any character in string t occurs in string s, search returns the character position in s of the first character matching any character in t.

Example: The function

```
(search "abcdef" "cb")
```

returns the value 2, while the function

```
(search "abcdef" "ce")
```

returns the value 3.

search_back_first_charset_line Function

The search_back_first_charset_line function is the same as the search_bk_in_line function.

search_back_first_not_charset_line Function

The search_back_first_not_charset_line function is the same as the verify_bk_in_line function.

search_bk Function

The search_bk function searches back in your text buffer for any of a specified set of characters.

Format: (search_bk s [n])

Arguments: The argument s must be a string value. The argument n, if specified, must be an integer value.

Action: The search_bk function returns a Boolean value.

If n is not specified, let n equal 1.

If n is positive, EMACS does the following n times: starting from the current cursor position, it searches back in the text buffer for any character from the set in string s. If no such character is found, the search terminates, EMACS leaves the cursor at the beginning of the buffer, and the function returns the value false. If a character is found matching any in the set s, the cursor is left on this character and the function returns true.

If the value of n is negative, EMACS performs

(search_fd (- n))

search_bk_in_line Function

The search_bk_in_line function is like the search_bk function, except that it only looks back as far as the beginning of the current line.

search_charset_backward Function

The search_charset_backward function is the same as the search_bk function.

search_charset_forward Function

The search_charset_forward function is the same as the search_fd function.

search_fd Function

The search_fd function searches forward in your text buffer for any of a specified set of characters.

Format: (search_fd s [n])

Arguments: The argument s must be a string value. The argument n, if specified, must be an integer value.

Action: The search_fd function returns a Boolean value.

If n is not specified, let n equal 1.

If n is positive, EMACS does the following n times: starting from the current cursor position, it searches forward in the text buffer for any character from the set in the string s. If no such character is found, the search terminates, EMACS leaves the cursor at the end of the buffer, and the function returns the value false. If a character is found matching any in the set s, the cursor is left on this character, and the function returns true.

If the value of n is negative, EMACS performs

```
(search_bk (- n))
```

search_fd_in_line Function

The search_fd_in_line function is the same as search_fd, except that the search stops at the end of the current line.

search_for_first_charset_line Function

The search_for_first_charset_line function is the same as the search_fd_in_line function.

search_for_first_not_charset_line Function

The search_for_first_not_charset_line function is the same as the verify_fd_in_line function.

search_not_charset_backward Function

The search_not_charset_backward function is the same as the verify_bk function.

search_not_charset_forward Function

The search_not_charset_forward function is the same as the verify_fd function.

select Special Form

The select special form evaluates an expression and compares it against a set of constants to determine what action to take.

Format:
```
(select v
        x1 s1
        x2 s2
        ...
        otherwise sx)
```

Arguments: The argument v may have any data type.

Each argument x1, x2, and ... may consist of one or more constants of the same data type as v.

Each argument s1, s2, ..., and sx may consist of one or more PEEL statements.

Action: EMACS evaluates the argument v and compares it to x1, x2, and ... until a match is found. If a match is found, then the corresponding PEEL statement s1, or s2, or ... is executed, and the value of that statement is returned as the value of select. Otherwise, the statement sx is executed, and the value of that statement is returned as the value of select. (See Chapter 4 for additional information and an example.)

If no match is found and no "otherwise" statement is specified, select returns the value NIL.

select_any_window Command and Function

The select_any_window command or function selects any window and, therefore, can be used to cycle through all windows.

Command Format: {ESC} X select_any_window
or
{CTRL-X} 4

Function Format: (select_any_window [n])

Argument: The argument n, if specified, must be a numeric value.

Action: EMACS repeats the following step n times: change the current window to the next major window in its standard order of windows.

Note: Use this command to cycle through all your windows. The major_window_count command can be used to get a count of the number of windows.

select_buf Command and Function

The select_buf command or function lets you change buffers.

Command Format: {ESC} X select_buf
or
{CTRL-X} B

Function Format: (select_buf [s])

Arguments: A numeric argument, if specified, is ignored. The argument s, if specified, must be a string value.

Action: If the argument is not specified, EMACS prompts you with "Buffer:". The characters that you type up to the first newline are stored as a string in the variable s.

If the string s is not a null string, EMACS switches you to the buffer whose name is given by the characters of the string s.

If the string s is a null string, EMACS switches you back to the buffer from which you switched to the current buffer.

The select_buf function returns the value NIL.

self_insert Function

The self_insert function inserts a character into your text buffer.

Format: (self_insert ch) (also 3rd arg = no of times to insert)

Argument: The argument ch must be a character.

Action: EMACS inserts the character ch into your text buffer at the current cursor position. (See also the overlayer function.)

The self_insert function returns the value NIL.

Example:

```
(self_insert \G)
```

This function inserts the character "G" at the current cursor position.

```
(overlayer \H)
```

This function overlays the current character with the character "H".

send_raw_string Function

The send_raw_string function sends a string directly to the terminal.

Format: (send_raw_string s)

Argument: The argument s must be a string value.

Action: The send_raw_string function returns a string value.

EMACS sends the string s to your terminal in raw mode and returns the value of s. Use send_raw_string to send "escape sequences" that directly manipulate features of the terminal, but remember that by doing so you might confuse EMACS.

set Function

The set function assigns the second argument to the first, and returns the value assigned.

Format: (set a v)

Arguments: The argument a must be an atom, usually quoted with .

The argument v may have any data type.

Action: EMACS assigns the value v to the atom a, and returns the value v.

Note: The set function follows the standard PEEL convention that both arguments are evaluated. That is why it is usually necessary to quote the first of the two arguments. This is not always necessary, however, as illustrated by the following example:

```
(set 'a 'b)
(set a 23)
```

The first assignment statement gives the atom a the value b. In the second assignment statement, the argument a is evaluated as yielding the value b; thus the value of b is set to 23. The value of a is unchanged, still equaling the atom b.

The setq function is related. In general, use of the set function with a ' preceding the first argument is equivalent of use of setq with no such '. For example, the first of the two set functions shown above could be changed to:

```
(setq a 'b)
```

set_command_abort_flag Function

The set_command_abort_flag function sets the command_abort flag.

Format: (set_command_abort_flag)

Arguments: None.

Action: EMACS aborts the current command.

set_fill_column Command and Function

The set_fill_column command or function sets the fill column used for fill mode.

Command Format: {ESC} X set_fill_column

Function Format: (set_fill_column [k])

Argument: The argument k, if specified, must be an integer value.

Action: If the argument k is not specified, EMACS prompts you for an integer value, and assigns the result to k. EMACS sets the fill column to k.

Note: The current fill column can be obtained by the following:

(buffer_info fill_column)

set_hscroll Command and Function

The set_hscroll command or function sets the value of hcol, which controls horizontal scrolling.

Command Format: {ESC} X set_hscroll

Function Format: (set_hscroll [k])

Arguments: A numeric argument, if specified, is ignored.

The argument k, if specified, must be a numeric value.

Action: If the argument k is not specified, EMACS prompts you with "What is the horizontal column:", and assigns the result to the variable k.

EMACS uses the value of k to set the value of hcol, which determines the column number of the leftmost column displayed on your screen.

set_key Command and Function.

The set_key command or function binds a keypath to a desired function name.

Command Format: {ESC} X set_key

Function Format: (set_key [p [f]])

Arguments: The arguments p and f, if specified to the set_key function, must be string values.

Action: If the argument p is not specified, EMACS prompts you with "Key path:". The characters you type are assigned as a string value to the variable p.

If the argument f is not specified, EMACS prompts you with "command name:". The characters you type, prior to a newline, are assigned as a string to the variable f.

The string f must be the name of a previously-defined command. EMACS binds the keypath specified by the characters in the string p to the command specified by the string f. This binding applies only to the current buffer.

The key_key function returns the value NIL.

Note: The set_key function is not normally as useful as set_permanent_key. The two functions differ only in that a set_key binding applies to the current buffer, while a set_permanent_key binding applies to all buffers throughout the remainder of your session. Both functions are described in Chapter 2 of this manual.

set_left_margin Command and Function

The set_left_margin command or function sets the left margin.

Command Format: {ESC} X set_left_margin

Function Format: (set_left_margin [k])

Arguments: A numeric argument, if specified, is ignored.

The argument k, if specified, must be a numeric value.

Action: If the argument k is not specified, EMACS prompts you for an integer value with "what is the left margin:", and assigns the result to the variable k.

EMACS uses the value of k as the left-margin value in fill mode.

Note: The left margin is used with the indent_to_fill_prefix and fill_para commands when fill mode is on.

set_mode Command and Function

The set_mode command or function lets you specify the mode that EMACS sets for your current buffer.

Command Format: {ESC} X set_mode

Function Format: (set_mode [m])

Arguments: A numeric argument, if specified, is ignored.

The argument m, if specified to the set_mode function, must be a string value.

Action: If the argument m is not specified, EMACS prompts you with "Mode name:". The characters you type prior to a newline are assigned as a string to the variable m.

EMACS sets the mode for your current buffer to the name specified by string m. All other modes are turned off.

The set_mode function returns the value NIL.

set_mode_key Command and Function

The set_mode_key command or function lets you bind a keypath to a command on a per-mode basis.

Command Format: {ESC} X set_mode_key

Function Format: (set_mode_key [m [p [f]]])

Arguments: A numeric argument, if specified, is ignored.

The arguments m, p, and f, if specified, must be string values.

Action: If the argument m is not specified, EMACS prompts you with "Mode name:". The characters that you type are assigned as a string to the variable m.

If the argument p is not specified, EMACS prompts you with "Key path:". The characters that you type prior to a newline are assigned as a variable to the string p.

If the argument f is not specified, EMACS prompts you with "Command name:". The characters you type prior to a newline are assigned as a string to the variable f.

The string p must define a legal EMACS keypath. The string f must name a previously defined EMACS command.

EMACS binds the command specified by f to the keypath specified by p. The binding is in effect only for those buffers in which the specified mode is turned on.

The set_mode_key function returns the value NIL.

set_permanent_key Command and Function

The set_permanent_key command or function binds a keypath to a desired function name.

Command Format: {ESC}X set_permanent_key

Function Format: (set_permanent_key [p [f]])

Arguments: The arguments p and f, if specified to the set_permanent_key function, must be string values.

Action: If the argument p is not specified, EMACS prompts you with "Key path:". The characters you type are assigned as a string value to the variable p.

If the argument f is not specified, EMACS prompts you with "Command name:". The characters you type, prior to a newline, are assigned as a string to the variable f.

The string f must be the name of a previously defined command. EMACS binds the keypath specified by the characters in the string p to the command specified by string f. This binding affects all buffers.

set_right_margin Command and Function

The set_right_margin command or function sets the right margin for wrapping in fill mode.

Command Format: {ESC} X set_right_margin
or
{CTRL-X} F

Function Format: (set_right_margin [k])

Arguments: A numeric argument, if specified, is ignored.

The argument k, if specified, must be an integer value.

Action: If k is not specified, EMACS prompts you for an integer value with "What is the right margin:", and assigns the value you type to the variable k.

EMACS uses the value of k as the right margin for word wrapping in fill mode.

set_tab Command and Function

The set_tab command or function is an alternate name for the settab command and function.

set_tabs Command and Function

The set_tabs command or function is an alternate name for the settab command and function.

setmark Command and Function

The setmark command or function sets the mark at the current cursor position. It is the same as pushmark, except that it does not push a mark if the mark has not changed.

setq Function

The setq function is a standard LISP function that assigns the second argument to the first after quoting the first.

Format: (setq a x)

Arguments: The argument a is an atom with any data type. The argument x may be any PEEL expression with a matching data type.

Action: EMACS computes

```
(set 'a x)
```

and returns the value of that expression.

settab Command and Function

The settab command or function lets you set your tabs to any values you wish.

Command Format: {ESC} X settab

Function Format: (settab)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS displays a ruler on your screen and lets you set tabs at whatever positions you want. (This command is described in detail in the EMACS Reference Guide.)

settabs_from_table Command and Function

The settabs_from_table command or function sets tab positions based on the column position of words in the current line.

Command Format: {ESC} X settabs_from_table
or
{ESC} X setft

Function Format: (settabs_from_table)
or
(setft)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS sets the tab stop at the first column of every word on the current line. For this command, a word is a sequence of consecutive characters delimited by a space or a punctuation mark.

(This command is described in detail in the EMACS Reference Guide.)

share_library$ Function

The share_library$ function loads a shared fasload format file at EMACS initialization.

Format: (share_library$ s)

Argument: The argument s must be a string value.

Action: This function is for use only in INIT_EMACS when sharing EMACS, and cannot be used for any other purpose.

EMACS opens for input a file whose name is obtained by adding the suffix .EFASL to the string s. That file must have only defun and defcom statements in it. EMACS then loads that file and defines the functions and commands in it.

show_lib_alc$ Command

The show_lib_alc$ command displays the current shared EMACS library segments when initializing EMACS.

Format: {ESC} X show_lib_alc$

Argument: A numeric argument, if specified, is ignored.

Action: This command is used only in INIT_EMACS when sharing EMACS, and cannot be used for any other purpose. It prints at the supervisor terminal the current shared EMACS library segments.

skip_back_over_white Function

The skip_back_over_white function skips back over whitespace characters, returning a Boolean value indicating whether the operation succeeded.

Format: (skip_backing_over_white [n])

Arguments: The argument n, if specified, must be an integer value.

Action: The skip_back_over_white function returns a Boolean value.

If the argument n is not specified, let n equal 1.

If n equals 0, no action is performed.

If n is equal to 1, EMACS proceeds as follows: if the cursor currently points to a whitespace character (as indicated by the atom whitespace), EMACS moves the cursor back to the first preceding nonwhitespace character and returns the value true; otherwise, EMACS returns the value false.

If n is greater than 1, EMACS skips back over (n-1) groups of whitespace characters, and then proceeds as described above for n=1. This is done by alternating between skip_back_over_white and skip_back_to_white.

When n is less than 0, EMACS performs the following:

    (skip_over_white (- n))

Note: This is equivalent to (verify_bk whitespace).

skip_back_to_white Function

The skip_back_to_white function moves the cursor back to the preceding whitespace character, returning a Boolean value indicating whether the operation succeeded.

Format: (skip_back_to_white [n])

Arguments: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If n equals 0, no action takes place.

If n equals 1, EMACS proceeds as follows: if the current cursor is not on a whitespace character, EMACS moves the cursor back until either the beginning of the buffer is reached or a whitespace character is found. If the cursor is now on a whitespace character, the function returns the value true; otherwise, it returns the value false.

If n is greater than 1, EMACS first skips back over (n-1) groups of whitespace characters, and then proceeds as described above for n=1.

If n is less than 0, EMACS performs the following:

    (skip_to_white (- n))

Note: This is equivalent to (search_bk whitespace).

skip_over_white Command and Function

The skip_over_white command or function moves the cursor forward over whitespace characters, returning a Boolean value indicating whether the operation succeeded.

Command Format: [{ESC}n] {ESC} X skip_to_white

Function Format: (skip_to_white [n])

Arguments: The argument n, if specified, must be an integer value.

Action: The skip_over_white function returns the value NIL.

If the argument n is not specified, let n equal 1.

If n equals 0, no action takes place.

If n equals 1, EMACS proceeds as follows: if the cursor currently points to a whitespace character (as indicated by the atom whitespace), then EMACS moves the cursor forward to the first nonwhitespace character, and returns the value true; otherwise, EMACS returns the value false.

If n is greater than 1, EMACS skips forward over (n-1) groups of whitespace characters, and then proceeds as described above when n equals 1.

If the value of n is less than 0, EMACS performs

```
(skip_back_over_white (- n))
```

skip_to_white Command and Function

The skip_to_white command or function moves the cursor forward to the proceeding whitespace character returning a Boolean value indicating whether the operation succeeded.

Command Format: [{ESC}n] {ESC} X skip_to_white

Function Format: (skip_to_white [n])

Arguments: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If n equals 0, no action takes place.

If n equals 1, EMACS proceeds as follows: if the current cursor is not on a whitespace character, EMACS moves the cursor forward until either the end of the buffer is reached or a whitespace character is found. If the cursor is now on a whitespace character, then the function returns the value true; otherwise, it returns the value false.

If n is greater than 1, EMACS first skips forward over (n-1) groups of whitespace characters, and then proceeds as described above for n=1.

If n is less than 0, EMACS performs

```
(skip_back_to_white (- n))
```

sleep_for_n_milliseconds Function

The sleep_for_n_milliseconds function directs EMACS to pause for the specified time interval.

Format: (sleep_for_n_milliseconds n)

Arguments: The argument n must be an integer value.

Action: The sleep_for_n_milliseconds function returns an integer value.

EMACS sleeps for n milliseconds, and returns the value of n.

sort_dt Command and Function

The sort_dt command or function inserts the current date into your text buffer in a format suitable for sorting.

Command Format: {ESC} X sort_dt

Function Format: (sort_dt)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the current date into your text buffer at the current cursor position. The format is as follows:

YY/MM/DD

This format is suitable for sorting.

The sort_dt function returns the value NIL.

sort_list Function

The sort_list function sorts a list and returns the result.

Format: (sort_list lst)

Argument: The argument lst must be a list of lists.

Action: The sort_list function returns a list value.

The list lst is of the following form:

```
((key1 ...) (key2 ...) ...)
```

EMACS forms a new list obtained by rearranging the items of this list so that they are in increasing order by the keys.

Each key is obtained by taking the car of each inner list. The keys must be mutually comparable. In usual practice, they are either all integers, or all characters, or all strings. (Characters are, however, comparable with strings.)

The sort_list function returns the resulting list.

Example:

```
(setq names
      '(("Fred" 37) ("Adam" 14) ("Joe" 34) ("Bert" 296)))
(setq numbers
      '((37 "Fred") (14 "Adam") (34 "Joe") (296 "Bert")))
(print (sort_list names))
(print (sort_list numbers))
```

Executing these statements displays the following results:

```
(("Adam" 14) ("Bert" 296) ("Fred" 37) ("Joe" 34))
((14 "Adam") (34 "Joe") (37 "Fred") (296 "Bert"))
```

split_line Command and Function

The split_line command or function breaks a line at the current cursor position, preserving the horizontal position of the right-hand portion of the line.

Command Format: [{ESC}n] {ESC} X split_line
or
[{ESC}n] {ESC} {CTRL-O}

Function Format: (split_line [n])

Argument: The argument n, if specified, must be a numeric value.

Action: If the argument n is not specified, let n equal 1.

Let k equal the column position of the current cursor. EMACS inserts n newlines, followed by k blanks, so that the current line has been split into n+1 lines, preserving the horizontal position of all characters on the original line.

The cursor is left at the inserted newline.

The split_line function returns the value NIL.

split_window Command and Function

The split_window command or function splits the current window into two, putting the cursor into the new window.

Command Format: [{ESC}n] {ESC} X split_window
or
[{ESC}n] {CTRL-X} 2

Function Format: (split_window [n])

Argument: The argument n, if specified, must be a numeric value.

Action: If the value of n is unspecified, let n equal one half the height of the current window.

EMACS splits the current window into two, moving the cursor into the second window. The split occurs at the nth line of the window.

The split_window function returns the value NIL.

split_window_stay Command and Function

The split_window_stay command or function splits the current window into two, leaving the cursor in the current window.

Command Format: [{ESC}n] {ESC} X split_window_stay
or
[{ESC}n] {CTRL-X} 3

Function Format: (split_window_stay [n])

Argument: The argument n, if specified, must be a numeric value.

Action: If the value of n is not specified, let n equal one half the height of the current window.

EMACS splits the current window into two, leaving the cursor in the original window. The split occurs in the nth line.

The split_window_stay function returns the value NIL.

stem_of_line Function

The stem_of_line function returns the stem (leading portion) of the current line.

Format: (stem_of_line [cur])

Argument: The argument cur, if specified, must be a cursor value.

Action: The stem_of_line function returns a string value.

If the argument cur is not specified, let cur equal the current cursor position.

EMACS forms a string containing the characters preceding cur on the same line as cur. The function returns the resulting string.

stop_doing Special Form

The stop_doing function stops execution of the current do_forever or do_n_times loop.

Format: (stop_doing)

Arguments: None.

Action: EMACS stops execution of the current do_forever or do_n_times loop.

The stop_doing function returns the value NIL.

string Data Type

Variables with the string data type can be assigned character strings.

string_length Function

The string_length function returns the length of a string argument.

Format: (string_length s)

Argument: The argument s must be a string or character.

Action: The string_length function returns an integer value equal to the number of characters in the string s, or 1 if s is a character.

string_of_length_n Function

The string_of_length_n function pads or truncates a string to a specified length.

Format: (string_of_length_n s n [p])

Arguments: The argument s must be a string or character. The argument n must be an integer. The argument p, if specified, must be a string or character.

Action: The string_of_length_n function returns a string value. The length of the resulting string is given by the argument n.

If the argument p is not specified, let p equal the space character.

The value of n must be nonnegative.

If the length of string s is greater than n, EMACS forms a new string by truncating the string s to the length n.

If the length of string s is less then n, EMACS forms a new string by concatenating sufficient copies of the string p to the end of string s so that the length of the new string is greater than or equal to n, and then truncates this result to the length n.

The resulting string is returned.

string_to_integer Function

The string_to_integer function converts a string to an integer.

Format: (string_to_integer s)

Argument: The argument s must be a string value.

Action: The string_to_integer function returns an integer value.

The string s must contain decimal digits, optionally preceded by a sign. EMACS converts the string to the corresponding integer value, and returns the result.

stringp Function

The stringp function tests whether an argument is a string.

Format: (stringp x)

Argument: The argument x may have any data type.

Action: The stringp function returns a Boolean value.

The value returned is true if the data type of x is string; otherwise, the value is false.

sublist Function

The sublist function returns a sublist of a given list. It acts on lists exactly as the substr function does on strings.

Format: (sublist lst n [k])

Arguments: The argument lst must be a list value. The argument n must be an integer. The argument k, if specified, must be an integer.

Action: The sublist function returns a list value.

If the argument k is not specified, let k equal infinity.

EMACS forms a new list by taking items from the list lst, starting at item number n, and continuing for k items, stopping if the end of list lst is reached.

The resulting list value is returned.

substr Function

The substr function returns a substring of a given string. It is like the PL/I substr function.

Format: (substr s n [k])

Arguments: The argument s must be a string or character value. The argument n must be an integer. The argument k, if specified, must be an integer.

Action: The substr function returns a string value.

If the argument k is not specified, let k equal infinity.

EMACS forms a new string by taking characters from string s, starting at character position n, and continuing for k characters, stopping if the end of string s is reached.

The resulting string value is returned.

Example: The function

```
(substr "abcdef" 3 2)
```

returns the string "cd", while the function

```
(substr "abcdef" 3)
```

returns the value "cdef".

suffix$ Function

The suffix$ function returns the substring after the rightmost period (.) in the name of the current buffer or the passed string.

Format: (suffix$ [s])

Argument: The argument s, if specified, must be a string value.

Action: The suffix$ function returns a string value.

If the argument s is not specified, let s equal a string value containing the name of the current buffer. (To be useful, normally s must be a string whose value is a treename.)

EMACS computes a new string value, t, as follows:

- If the string s does not contain any occurrence of the character ".", or if the string s contains the character ">" but does not contain "." to the right of the rightmost ">", let t equal "" (the null string).
- Otherwise, let t equal the substring of the string s containing all characters to the right of the rightmost ".".

The suffix$ function returns the value t.

sui_exchange_mark Command and Function

The sui_exchange_mark command or function is a SUI version of the exchange_mark command or function that provides the user with additional information.

sui_primos_command Command and Function

The sui_primos_command command or function is the SUI version of the primos_command command or function.

sui_set_tabs Command

The sui_set_tabs command establishes the tab settings for SUI users.

Format: {ESC} X sui_set_tabs

Arguments: None.

Action: The sui_set_tabs command establishes tab settings for SUI users.

suppress_redisplay Function

The suppress_redisplay function checks or enables/disables redisplay.

Format: (suppress_redisplay [b])

Argument: The argument b, if specified, must be a Boolean value.

Action: The suppress_redisplay function returns a Boolean value.

Let x equal the current value of the redisplay variable.

If the argument b is specified, set the redisplay variable to the value specified by b.

The suppress_redisplay function returns the value x.

tab Command and Function

The tab command or function moves the cursor to the next tab stop.

Command Format: [{ESC}n] {ESC} X tab
or
[{ESC}n] {CTRL-I}

Function Format: (tab [n])

Argument: The argument n, if specified, must be an integer value.

Action: If n is not specified, let n equal 1.

If n equals 0, no action takes place.

If n is positive, EMACS does the following n times: it moves the cursor to the next tab-stop position on the current line. If there are not enough characters remaining on the current line, EMACS inserts a sufficient number of blanks so that the cursor can move to the next tab-stop position.

tablist Command and Function

The tablist command and function accepts a series of numbers from the terminal and sets tab stops at those positions.

Command Format: {ESC} X tablist

Function Format: (tablist)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS prompts you with "Set tab columns separated by blanks:". You may then type a list of numbers, separated by blanks, ending in a newline. EMACS sets the tab stops at the positions you specify.

The tablist function returns the value NIL.

tablist_to_array Function

The tablist_to_array function takes a list of integers and sets tab stops at the specified positions.

Format: (tablist_to_array lst)

Argument: The argument lst must be a list, normally quoted.

Action: The items in the argument lst must all be integers. EMACS sets the tab stops at positions indicated by those integers.

The tablist_to_array function returns the value NIL.

Example: (tablist_to_array '(5 13 24 60))

tell_left_margin Command

The tell_left_margin command or function prints the current left margin position.

Command Format: {ESC} X tell_left_margin

Function Format: (tell_left_margin)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS prints the current left-margin position in the minibuffer.

Note: The left margin is the value bound to the atom named fill_prefix.

tell_modes Command and Function

The tell_modes command or function displays all modes for the current buffer. This function is needed for those situations in which all modes are not shown on the mode line.

Command Format: {ESC} X tell_modes

Function Format: (tell_modes)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS displays all current modes on your screen. Use {CTRL-G} to restore the display.

tell_position Command and Function

The tell_position command or function displays buffer information, including the current line and character position.

Command Format: {ESC} X tell_position
or
{CTRL-X} =

Function Format: (tell_position)

Argument: A numeric argument, if specified, is ignored.

Action: In the minibuffer EMACS displays current line and character information, buffer size information, and window information.

The tell_position function returns the value NIL.

tell_right_margin Command and Function

The tell_right_margin command or function prints the current right margin position.

Command Format: {ESC} X tell_right_margin

Function Format: (tell_right_margin)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS prints the current right-margin position in the minibuffer.

terminal_info Function

The terminal_info function queries or sets information about the user's terminal.

Format: (terminal_info p [v])

Arguments: The argument p must be an atom chosen from among those listed below under action. This atom should not be quoted.

The argument v, if specified, must have a data type compatible with the atom p, as specified below.

Action: The terminal_info function returns a value whose data type depends upon the argument p.

The argument p is an atom that specifies a terminal property to be queried or set. The legal values for the argument p, their data types, and their meanings are shown in the following table.

p	Data Type	Meaning
type	string	Terminal type
speed	integer	Baud rate
height	integer	Number of rows in the display
width	integer	Number of columns in the display
crtp	Boolean	True if the terminal is a CRT
can_insert	Boolean	Can insert/delete lines, making screen refresh a bit faster

If the argument v is specified, and if the property p is not read only, EMACS sets the property p to the value specified by v.

Note that at the present time all properties are read-only.

The terminal_info function returns the old value of the property p.

terminal_type Function

The terminal_type function returns a string containing the terminal type.

Format: (terminal_type)

Arguments: None.

Action: The terminal_type function returns a string value containing the type of terminal you are using.

terpri Function

The terpri function inserts a newline.

Format: (terpri cur)

Argument: The argument cur, if specified, must be a cursor value.

Action: If cur is not specified, let cur equal the current cursor position.

EMACS inserts a newline character at the cursor position indicated by cur, and sets the current cursor to the character following the newline.

The terpri function returns the value NIL.

throw Function

The throw function is like the *throw function, except that the argument order is different.

Format: (throw body tag)

This is like *throw, except that the body and tag arguments are reversed. Because throw's argument order makes programming quite difficult, the use of *throw is recommended instead. (See *throw for further information.)

tld Command

The tld command is a SUI command that lists your file directory in order of date-and-time last written.

Format: {ESC} X tld

Arguments: None.

Action: EMACS executes the following PRIMOS command at the current attach point:

    LD -LONG -SRTD

toggle_redisp Command and Function

The toggle_redisp command or function toggles redisplay mode. It is usually used with slow display terminals.

Command Format: {ESC} X toggle_redisp
or
{CTRL-X} {CTRL-T}

Function Format: (toggle_redisp)

Argument: A numeric argument, if specified, is ignored.

Action: If the redisplay flag is off, EMACS turns it on. If it is on, EMACS turns it off. (See suppress_redisplay.)

The toggle_redisp function returns the value NIL.

token_chars Variable

The token_chars variable is a global string variable whose value is a string containing all the characters in the basic character set. Specifically, it is set to the following character string:

"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789"

These tokens are considered to be legal characters in a word. The forward_word, back_word, delete_word, and rubout_word commands will treat all characters in this string as components of a word.

translate Function

The translate function translates a string by replacing one set of characters with another. This is similar to the PL/I translate function.

Format: (translate sa sb [sc])

Arguments: The arguments sa and sb, and sc, if specified, must be string or character values.

Action: The translate function returns a string value.

If the argument sc is not specified, let sc equal a string containing the entire collating sequence (character set) supported by EMACS.

If the length of sb is greater than or equal to the length of sc, then let sb2 equal sb; otherwise, let sb2 equal the string obtained by concatenating a sufficient number of blanks to the end of string sb, so that the resulting string is as long as the string sc.

EMACS forms a new string sa2 from the characters in the string sa by performing the following steps for k=1 to the length of string sa:

- Let ch be the character in position k of string sa.
- If ch does not appear in the string sc, leave ch unchanged.
- Otherwise, if ch appears in the string sc, and if the leftmost such appearance is at position m in string sc, let ch equal the character in position m of string sb.
- EMACS sets the character in position k of string sa2 to ch.

The translate function returns the string sa2.

transpose_word Command and Function

The transpose_word command or function inverts the positions of the words before and after the current cursor.

Command Format: {ESC} X transpose_word
or
{ESC} T

Function Format: (transpose_word)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS transposes the words before and after the current cursor position. The current cursor is left at the first character of the second word.

The transpose_word function returns the value NIL.

trim Function

The trim function removes leading and trailing blanks from a string.

Format: (trim s)

Argument: The argument s must be a string value.

Action: The trim function returns a string value. The value returned is computed by removing leading and trailing blanks from the string s.

trim_date Command and Function

The trim_date command or function inserts the current date at the current cursor position.

Command Format: {ESC} X trim_date

Function Format: (trim_date)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the current date into your buffer at the current cursor position, using a format similar to the following:

    19 Sep 1985

The trim_date function returns the value NIL.

trim_dt Command and Function

The trim_dt command or function inserts the current date at the current cursor position.

Command Format: {ESC} X trim_dt

Function Format: (trim_dt)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the current date into your buffer at the current cursor position, using a format similar to the following:

    09/19/85

The trim_dt function returns the value NIL.

turn_mode_off Function

The turn_mode_off function turns off a mode by removing the mode name from the buffer mode list.

Format: (turn_mode_off d)

Argument: The argument d must be a dispatch table.

Action: EMACS removes all occurrences the specified dispatch table d from the buffer mode list.

The turn_mode_off function returns the value NIL.

Example:

```
(turn_mode_off (find_mode 'lisp))
```

In this statement the find_mode function returns a dispatch table for LISP mode, and the turn_mode_off function turns LISP mode off.

turn_mode_on Function

The turn_mode_on function turns a mode on and adds the mode name to the buffer mode list if the mode is not already on.

Format: (turn_mode_on d [f])

Arguments: The argument d must be a dispatch table. The argument f, if specified, must be the unquoted atom first.

Action: EMACS turns the specified mode on, and adds the mode name to the buffer mode list. If the argument f is specified (as the atom first), EMACS places the new mode in the first position on the mode list; otherwise, EMACS places it last.

The turn_mode_on function returns the value NIL.

Example:

```
(turn_mode_on (find_mode 'lisp) first)
```

This statement turns LISP mode on and places the new mode first on the mode list.

Note: The order of modes in the list is the order in which dispatch tables are searched while processing key bindings.

twiddle Command and Function

The twiddle command or function transposes the position of the two characters preceding the current cursor.

Command Format: {ESC} X twiddle
or
{CTRL-T}

Function Format: (twiddle)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS reverses the position of the two characters preceding the current cursor.

The twiddle function returns the value NIL.

type_tab Command and Function

The type_tab command or function moves the cursor forward by a specified number of tab stops.

Command Format: [{ESC}n] {ESC} X type_tab

Function Format: (type_tab [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If the argument n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, the cursor is moved forward n tab stop positions. If the end of the line is reached, EMACS automatically fills the end of the line with blank characters, up to the desired tab stop position.

If the value of n is negative, the cursor is moved back (-n) tab stops, stopping if the beginning of the line is reached.

The type_tab function returns the value NIL.

typef Function

The typef function returns an integer indicating the data type of its argument.

Format: (typef x)

Argument: The argument x may have any data type.

Action: The typef function returns an integer argument. The value returned depends upon the data type of x, as shown in the following table:

Data Type of x	Value Returned
any	1
Boolean	2
character	3
integer	4
string	5
atom	6
function	7
list	8
cursor	9
dispatch_table	11
handler	12
window	14
array	15

uid Function

The uid function returns a string value containing a unique identifier.

Format: (uid)

Arguments: None.

Action: The uid function returns a string value.

The string contains characters that may be used as a unique identifier.

unmodify Command and Function

The unmodify command or function tells EMACS to treat the current buffer as if it were unmodified.

Command Format: {ESC} X unmodify
or
{ESC} ~

Function Format: (unmodify)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS clears its "modified" flag for the current buffer.

The unmodify function returns the value NIL.

untidy Command and Function

The untidy command or function removes indenting and justification from a paragraph.

Command Format: {ESC} X untidy

Function Format: (untidy)

Argument: A numeric argument, if specified, is ignored.

Action: The untidy command or function removes indenting and justification while filling the current paragraph, so that each line does not have more than the number of characters indicated by the function (buffer_info fill_column) or by the command tell_right_margin. It rearranges words on the line so that each line is about the same length. Use set_right_margin to change the right margin.

upcase Function

The upcase function converts a string to uppercase.

Format: (upcase s)

Argument: The argument s must be a string value.

Action: EMACS returns a string value obtained by changing all lowercase letters in s to uppercase, leaving all other characters unchanged.

uppercase_region Command and Function

The uppercase_region command or function converts all lowercase letters in the current region to uppercase.

Command Format: {ESC} X uppercase_region
or
{CTRL-X} {CTRL-U}

Function Format: (uppercase_region)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS changes all lowercase letters in the current region to uppercase, leaving all other characters unchanged.

The uppercase_region function returns the value NIL.

Caution: This command must be used with extreme care. If it is mistakenly applied to the wrong region of text in uppercase and lowercase, its effect must be undone manually.

uppercase_word Command and Function

The uppercase_word command or function changes the text in one or more words to uppercase.

Command Format: [{ESC}n] {ESC} X uppercase_word
or
[{ESC}n] {ESC} U

Function Format: (uppercase_word [n])

Argument: The argument n, if specified, must be an integer whose value may be positive, 0, or negative.

Action: If the argument n is not specified, let n equal 1.

If the value of n is 0, no action takes place.

If the value of n is positive, EMACS converts to uppercase all lowercase letters in the region from the beginning of the current word (or the next word, if the cursor is on whitespace) to the end of the nth word, moving forward. The cursor moves to the end of that region.

If the value of n is negative, EMACS changes to uppercase all lowercase letters in the region ending at the end of the current word (or the previous word, if the cursor is on whitespace) and beginning at the beginning of the (-n)th word preceding the current cursor position. The cursor is left unchanged

The uppercase_word function returns the value NIL.

user_name Variable

The user_name variable is a string variable containing your login name.

using_cursor Special Form

The using_cursor function executes PEEL statements and then resets the cursor.

Format: (using_cursor cur s1 [s2 ...])

Arguments: The argument cur must be a cursor value. The arguments s1, s2, and ... must be PEEL statements.

Action: EMACS executes all the statements s1, s2, and ..., and then performs the following:

(go_to_cursor cur)

The using_cursor function returns the value NIL.

verify Function

The verify function tests a string for legal characters.

This is the PL/I verify built-in function.

Format: (verify sa sb)

Arguments: The arguments sa and sb must be string values.

Action: The verify function returns an integer value.

If all characters of string sa also appear in string sb, then the function returns 0.

Otherwise, the function returns the position of the first character in string sa that is not also in string sb.

Example: The function

(verify "ABACUS" "ABCDEFGHIJKLMNOPQRSTUVWXYZ")

returns the value 0, while

(verify "ABACUS" "ABCD")

returns the value 5.

verify_bk Function

The verify_bk function scans back from the current cursor, searching for a character not in the argument string.

Format: (verify_bk s [n])

Arguments: The argument s must be a string or character value. The argument n, if specified, must be an integer value.

Action: The verify_bk function returns a Boolean value.

If the argument n is not specified, let n equal 1.

If the value of n is 0, no operation takes place.

Starting from the current cursor position and moving backward, EMACS searches for the nth occurrence of a character in the text buffer that does not also appear in the string s. If such a character is found before reaching the beginning of the buffer, then EMACS leaves the cursor at that character position and returns true; otherwise, EMACS leaves the cursor at the beginning of the buffer and returns false.

If the value of n is less than 0, EMACS performs the following:

```
(verify_fd (- n))
```

verify_bk_in_line Function

The verify_bk_in_line function is like verify_bk, except that the search ends at the beginning of the current line.

verify_fd Function

The verify_fd function searches forward from the current cursor for a character not in the argument string.

Format: (verify_fd s [n])

Arguments: The argument s must be a string value. The argument n, if specified, must be an integer value.

Action: The verify_fd function returns a Boolean value.

If the argument n is not specified, let n equal 1.

If the value of n is 0, no operation takes place.

Starting from the current cursor position and moving forward, EMACS searches for the nth occurrence of a character in the text buffer that does not also appear in the string s. If such a character is found before reaching the end of the buffer, then EMACS leaves the cursor at that character position and returns true; otherwise, EMACS leaves the cursor at the end of the buffer and returns false.

If n is less than 0, EMACS performs the following:

```
(verify_bk (- n))
```

verify_fd_in_line Function

The verify_fd_in_line function is the same as verify_fd, except that the search ends at the end of the line.

view_file Command and Function

The view_file command or function allows you to view (look through) a file in read-only mode without being able to modify it.

Command Format: {ESC} X view_file

Function Format: (view_file [s])

Argument: The argument s, if specified, must be a string value.

Action: If the argument s is not specified, EMACS prompts you with "View file:". EMACS assigns all characters typed before a newline is reached to the string variable s.

EMACS opens in view mode the file whose name is specified by the string s, meaning that you may examine the file but not modify it.

view_kill_ring Command and Function

The view_kill_ring command or function lets you view the contents of kill buffers.

Command Format: {ESC} X view_kill_ring
or
{CTRL-X} {CTRL-Z} K

Function Format: (view_kill_ring)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS allows you to review your kill buffers. (This command is described in detail in the EMACS Reference Guide.)

view_lines Command and Function

The view_lines command or function updates your screen on a slow terminal.

Command Format: {ESC} X view_lines
or
{CTRL-X} {CTRL-Z} {CTRL-V}

Function Format: (view_lines)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS updates your display.

The view_lines function returns the value NIL.

Note: This command is used after you have used toggle_redisp to suppress automatic updating of your display.

vld Command

The vld command is a SUI command that provides a verbose listing of your file directory.

Format: {ESC} X vld

Arguments: None.

Action: EMACS executes following the PRIMOS command at the current attach point.

LD -LONG -SRTN

vsplit Command and Function

The vsplit command or function splits your current window vertically into two windows.

Command Format: {ESC} X vsplit

Function Format: (vsplit)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS splits your current window into two at the current cursor position.

The vsplit function returns the value NIL.

wait_for_input Function

The wait_for_input function waits for the user to type a character. The typed character is NOT removed from the input buffer.

Format: (wait_for_input)

Arguments: None.

Action: EMACS waits until the user types a character.

The wait_for_input function returns the value NIL.

wallpaper Command and Function

The wallpaper command or function inserts all help information into your current text buffer.

Command Format: {ESC} X wallpaper

Function Format: (wallpaper)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts all of its apropos help text into your current buffer. This includes all your current bindings.

The wallpaper function returns the value NIL.

white_delete Command and Function

The white_delete command or function deletes all whitespace around point.

Command Format: {ESC} X white_delete
or
{ESC} \

Function Format: (white_delete)

Argument: A numeric argument, if specified, is ignored.

Action: If the character at the current cursor position or the character preceding the current cursor position is a whitespace character, then it and all contiguous whitespace characters are deleted; otherwise, no action takes place.

The white_delete function returns the value NIL.

Note: The white_delete function has the same effect as the delete_white_sides function.

whitespace Variable

The whitespace variable is a string variable that contains all the characters used when searching for whitespace. EMACS initializes it to a space.

whitespace_to_hpos Function

The whitespace_to_hpos function inserts whitespace until the specified horizontal position is reached.

Format: (whitespace_to_hpos n)

Argument: The argument n must be an integer value.

Action: The whitespace_to_hpos function returns a Boolean value.

Let h equal the current horizontal position.

If n is greater than h, EMACS inserts (n-h) blanks at the current cursor position.

After the insertion is completed, the function returns the value true.

window Data Type

A variable with the window data type represents a screen window.

window_info Function

The window_info function sets or queries information about a window.

Format: (window_info p [x])

Arguments: The argument p must be an unquoted atom, chosen from the atom names listed below under action. The argument x, if specified, must have a data type compatible with the argument p, as described below.

Action: The window_info function returns a value whose data type depends upon the argument p.

The following table gives the legal values for the argument p, along with the corresponding function data type and meaning:

p	Data Type	Meaning
top_line	integer	The top line of the screen on which the current window is displayed. (read-only)
bottom_line	integer	The last line on the screen on which the window is displayed. (read-only)
left_column	integer	The leftmost screen column in which the window appears. (read-only)
right_column	integer	The rightmost screen column in which the window appears. (read-only)

is_active	Boolean	Whether the window is being redisplayed. (read-only)
is_major	Boolean	Whether the window is an major window. This will usually be true. The minibuffer window is not a major window. (read-only)
top_line_cursor	cursor	This cursor points into the top line of the text that appears as the top line of the window. (writable)
showing_numbers	Boolean	Whether line numbering is on. (writable)
column_offset	integer	The value of the horizontal column offset for the window. (See hcol.) (writable)
last_buffer_cursor	cursor	The current cursor position for this window. (writable)
mark_bottom	Boolean	Mark is at bottom of window. (writable)

If the argument x is specified, and if the property corresponding to the argument p is not read-only, EMACS sets the property corresponding to the argument p to the value of x. This is permitted only for property shown as "writable" in the above table.

The window_info function returns the old value of the property p for the current window.

with_cleanup Special Form

The with_cleanup function executes PEEL code and then executes a handler that will be run whether an error occurs or not. This is similar to the CLEANUP$ condition in PL/I.

Format: (with_cleanup s1 [s2 ...])
        handler hs1 [hs2 ...])

Arguments: The arguments s1, s2, and ..., and hs1, hs2, and ... must be PEEL statements.

Action: EMACS executes the statements s1, s2, and ... . After completion of that execution, whether normally or because of error, EMACS executes hs1, hs2, and ... .

The with_cleanup function returns the value NIL.

with_command_abort_handler Special Form

The with_command_abort_handler function executes PEEL code with a handler for errors.

Format: (with_command_abort_handler s1 [s2 ...]
command_abort_handler hs1 [hs2 ...])

Arguments: The arguments s1, s2, and ..., and hs1, hs2, and ... must be PEEL statements.

Action: EMACS executes s1, s2, and ... . If no error occurs, execution of the with_command_abort_handler function terminates.

If an error occurs, EMACS resets the error flags and throw, and executes hs1, hs2, and ... .

The with_command_abort_handler function returns the value NIL.

with_cursor Special Form

The with_cursor function executes a body of code after copying the current cursor to a temporary cursor.

Format: (with_cursor c s1 [s2 ...])

Arguments: The argument c must be an unquoted atom. The arguments s1, s2, and ... must be PEEL statements.

Action: EMACS copies the current cursor to c and then executes s1, s2, and ... .

with_no_redisplay Special Form

The with_no_redisplay function executes code while suppressing redisplay.

Format: (with_no_redisplay s1 [s2 ...])

Arguments: The arguments s1, s2, and ... must be PEEL statements.

Action: EMACS executes s1, s2, and ... without updating your display.

wrap Command and Function

The wrap command or function inserts a carriage return if word wrapping should occur.

Command Format: [{ESC}n] {ESC} X wrap

Function Format: (wrap [n])

Argument: The argument n, if specified, must be a numeric value.

Action: If the argument n is not specified, let n equal 1.

EMACS inserts its last invocation character (&character_argument) and checks its horizontal position. If it is greater than the fill column, wrap replaces it with a newline.

wrap_line_with_prefix Function

The wrap_line_with_prefix function specifies the wrap column and the prefix string.

Format: (wrap_line_with_prefix n s)

Arguments: The argument n must be an integer value. The argument s must be a string value.

Action: This is the function that actually performs the wrap operation.

wrapoff Command and Function

The wrapoff command or function is obsolete. Use fill_off.

wrapon Command and Function

The wrapon command or function is obsolete. Use fill_on.

write_file Command and Function

The write_file command or function writes the current buffer to the specified file.

Command Format: {ESC} X write_file

Function Format: (write_file [s])

Arguments: A numeric argument, if specified, is ignored.

The argument s, if specified must be a string value.

Action: If the argument s is not specified, EMACS prompts you with "write file:" and assigns the typed string to the variable s.

If the value of s is a null string, let s equal the filename associated with the current buffer.

EMACS writes the current buffer to the file whose pathname is specified by the string s.

The write_file function returns the value NIL.

Note: This will overwrite an existing file with NO WARNING. (See also mod_write_file.)

yank_kill_text Command and Function

The yank_kill_text command or function inserts the text saved during view_kill_ring at the current cursor position.

Command Format: {ESC} X yank_kill_text
or
{CTRL-X} {CTRL-Z} {CTRL-Y}

Function Format: (yank_kill_text)

Argument: A numeric argument, if specified, is ignored.

Action: EMACS inserts the text saved by view_kill_ring into your text buffer at the current cursor position.

The yank_kill_text function returns the value NIL.

yank_minibuffer Command and Function

The yank_minibuffer command or function inserts the response to a previous minibuffer prompt at the current cursor position.

Command Format: [{ESC}n] {ESC} X yank_minibuffer
or
[{ESC}n] {ESC} {CTRL-Y}

Function Format: (yank_minibuffer [n])

Argument: The argument n, if specified, must be an integer value.

Action: If the argument n is not specified, let n equal 0.

EMACS inserts into your current buffer at the current cursor position the characters typed in response to the (n+1)st previous minibuffer prompt.

The yank_minibuffer function returns the value NIL.

yank_region Command and Function

The yank_region command or function inserts a previously killed region into your current buffer at the current cursor position.

Command Format: {ESC} X yank_region
or
{CTRL-Y}

Function Format: (yank_region [n])

Argument: The argument n, if specified, must be a numeric value.

Action: If the argument n is not specified, let n equal 0.

EMACS inserts the text in the (n+1)st previous kill region into the current buffer, at the current cursor position.

The yank_region function returns the value NIL.

yank_replace Command and Function

The yank_replace command or function replaces the text inserted with yank_region with the text in a preceding kill_ring buffer.

Command Format: {ESC} X yank_replace
or
{ESC} Y

Function Format: (yank_replace [n])

Argument: The argument n, if specified, is ignored.

Action: You use yank_replace after a yank_region command or function.

If the argument n is not specified, let n equal 0.

The yank_replace function replaces the text you have just inserted with text from the (n+1)st preceding buffer on the kill ring.

Notes: You may use yank_replace repeatedly until the text you want from the kill ring appears.

EMACS remembers only the last ten or so responses.

yesno Function

The yesno function prompts the user for a yes or no reply.

Format: (yesno s)

Argument: The argument s must be a string value.

Action: The yesno function returns a Boolean value.

EMACS prompts the user in the minibuffer with the string s. If the reply is "yes", "OK", or "true", then the yesno function returns the value true; otherwise, it returns the value false.

B
Command Cross-reference List

In this appendix, all the commands, functions, and operators described in Appendix A are grouped according to purpose. For example, all the commands dealing with words, such as capinitial, forward_word, and lowercase_word, are listed together in the category WORDS.

Use this appendix when you suspect a command or function might exist, but do not know its name. Select likely names from the categorized lists, and look up their full descriptions in Appendix A.

The following is a list of all the categories:

Arithmetic	Cursor and Mark	Information	Tabs
Arrays	Data Types	LISP	Tests
Booleans and Relationals	Date	Miscellaneous	Type Tests
Buffers	Deletion and Copying	Modes	Whitespace
Clauses	Display	Movement	Windows
Compilation and Loading	Files	Paragraphs	Words
Control	Help	PRIMOS	
Conversion	I/O	Search and Verify	
	Indentation	Sentences	
		Strings/Lines/Regions	

ARITHMETIC

*
+
-
/
1+
1-
modulo
numberp

ARRAYS

aref
array_dimension
array_type
aset
copy_array
fill_array
make_array

BOOLEANS AND RELATIONALS
(See also TESTS)

&
<
<=
=
>
>=
^
^=
|
and
eq
not
or

BUFFERS
See also Files.

2don
2doff
2d
append_to_buf
beginning_of_buffer_p
buffer_info
buffer_name
delete_buffer
empty_buffer_p
end_of_buffer_p
find_buffer
first_line_p
go_to_buffer
go_to_cursor
insert
insert_buf
kill_rest_of_buffer
last_line_p
line_is_blank
list_buffers
mark_whole
next_buf
overlay_off
overlay_on
overlay_rubout
overlayer
prepend_to_buf
prev_buf
same_buffer_p
select_buf
self_insert
tell_position
unmodify
window_info

CHARACTERS
(See STRINGS/LINES/REGIONS)

CLAUSES

backward_clause
backward_kill_clause
forward_clause
forward_kill_clause

COMPILATION AND LOADING

dump_file
expand_macro
fasdump
fasload
load_compiled
load_lib
load_package
load_pl_source

pl
pl_minibuffer
set_key
set_permanent_key

CONTROL

dispatch
do_forever
do_n_times
else
if
if_at
otherwise
return
select
stop_doing

CONVERSION

char_to_string
convert_tabs
convert_to_base
integer_to_string
string_to_integer

CtoI
ItoC
ItoP
PtoI
high_bit_off
high_bit_on

CURSOR AND MARK

copy_cursor
current_cursor
cursor_info
cursor_on_current_line_p
cursor_same_line_p
delete_point_cursor
exchange_mark
get_cursor
go_to_cursor
line_number
make_cursor
mark
mark_bottom
mark_end_of_word
mark_top
mark_whole
point_cursor_to_string
popmark
pushmark
range_to_string
save_excursion
save_position
setmark
using_cursor
with_cursor

DATA TYPES

any
array
atom
Boolean
buffer
character
charp
cursor
dispatch_table
function
handler
integer
list
numberp
string
stringp
typef
window

DATE

date
dt
europe_dt
sort_dt
trim_date
trim_dt

DELETION AND COPYING

copy_array
copy_cursor
copy_region
backward_kill_clause
backward_kill_line
backward_kill_sentence
delete_blank_lines
delete_buffer
delete_char
delete_point_cursor
delete_region
delete_white_left
delete_white_right
delete_white_sides
delete_word
forward_kill_clause
forward_kill_sentence
kill_line
kill_region
kill_rest_of_buffer
leave_one_white
merge_lines
overlay_rubout
rubout_char
rubout_word
view_kill_ring
white_delete
yank_kill_text
yank_minibuffer
yank_region
yank_replace

DISPLAY

#
#off
#on
hcol
hscroll
redisplay
refresh
repaint
set_hscroll
suppress_redisplay
toggle_redisp
view_lines
with_no_redisplay

FILES

append_to_file
file_info
file_name
file_operation
find_file
found_file_hook
get_filename
insert_file
mod_write_file
prepend_to_file
read_file
save_all_files
save_file
unmodify
view_file
write_file

HELP

apropos
describe
explain_key
help_char
wallpaper

I/O

assure_character
clear_and_say
error_message
find_file
flush_typeahead
have_input_p
info_message
init_local_displays
insert
insert_buf
insert_file
load_pl_source
local_display_generator
minibuf_response
minibuffer_print
minibuffer_response
prinl
print
prompt
prompt_for_integer
prompt_for_string

quote_command
read
read_character
read_file
reset
say_more
self_insert
send_raw_string
sleep_for_n_milliseconds
terpri
wait_for_input
yesno

INDENTATION

cret_indent_relative
indent_to_fill_prefix
indent_relative
indent_line_to_hpos
split_line
whitespace_to_hpos

INFORMATION

buffer_info
buffer_name
case?
case_replace?
cpu_time
cur_hpos
current_cursor
current_handler
current_line
current_major_window
cursor_info
date
dispatch_info
dt
file_info
file_name
function_info
get_filename
handler_info
insert_version
line_number
lines_in_file
list_dir
major_window_count
rest_of_line
stem_of_line
tell_left_margin
tell_modes
tell_position
tell_right_margin
terminal_info
terminal_type
tld
uid
user_name
vld
window_info

LISP

*_list
*catch
*throw
append
apply
assoc
car
catch
cdr
cons
decompose
def_auto
defcom
defun
eval
fset
fsymeval
get
get_pname
intern
lambda
let
LISP_comment
LISP_off
LISP_on
list
member
nthcar
null
prinl
print
progn
putprop
quote
remassoc
remove
remprop
set

setq
sort_list
sublist
terpri
throw

LOADING
(See COMPILATION AND LOADING)

MISCELLANEOUS

abort_command
abort_minibuffer
abort_or_exit
character_argument
command_abort
command_abort_handler
debug_info
display_error_noabort
exit_minibuffer
fill_end_token_insert_left
fill_end_token_insert_pfx
hcol
ignore_prefix
multiplier
numeric_argument
quit
reject
restrict_to_sui$
ring_the_bell
scan_errors
send_raw_string
set_command_abort_flag
sleep_for_n_milliseconds
with_cleanup
with_command_abort_handler

MODES

all_modes_off
dispatch_info
find_mode
set_mode
set_mode_key
tell_modes
turn_mode_off
turn_mode_on

MOVEMENT
(See also SEARCH, WHITESPACE)

back_char
back_para
back_tab
back_to_first_nonwhite
back_word
backward_clause
backward_para
balfor
balbak
begin_line
cret_indent_relative
end_line
forward_char
forward_clause
forward_para
forward_sentence
forward_word
go_to_cursor
go_to_hpos
goto_line
indent_line_to_hpos
indent_relative
indent_to_fill_prefix
move_bottom
move_top
next_line
next_line_command
next_page
prev_line
prev_line_command
scroll_other_backward
scroll_other_forward

PARAGRAPHS

backward_para
fill_off
fill_on
fill_para
forward_para
mark_para
set_fill_column
set_left_margin
set_right_margin
untidy

PRIMOS

af
evaluate_af
primos_command
primos_external
primos_internal_como
primos_internal_quiet
primos_internal_screen
primos_smsgl

RELATIONAL OPERATORS
(See BOOLEANS AND RELATIONALS)

SEARCH AND VERIFY

case?
case_off
case_on
case_replace?
case_replace_off
case_replace_on
forward_search
forward_search_command
reverse_search
reverse_search_command
search
search_back_first_charset_line
search_back_first_not_charset_line
search_bk
search_bk_in_line
search_charset_backward
search_charset_forward
search_fd
search_fd_in_line
search_for_first_charset_line
search_for_first_not_charset_line
search_not_charset_backward
search_not_charset_forward
verify
verify_bk
verify_bk_in_line
verify_fd
verify_fd_in_line

SENTENCES

backward_sentence
backward_kill_sentence
forward_sentence
forward_kill_sentence

STRINGS/LINES/REGIONS

NL
begin_line
beginning_of_line_p
capinitial
catenate
center_line
copy_region
cr
cur_hpos
current_character
current_line
downcase
end_line
first_line_p
goto_line
indent_line_to_hpos
index
kill_line
kill_region
last_line_p
line_is_blank
line_number
looking_at
looking_at_char
lowercase_region
lowercase_word
next_line
nth
open_line
prev_line
query_replace
range_to_string
remove_charset
replace
rest_of_line
string_length
string_of_length_n
stringp
substr
tab
translate
trim
twiddle

upcase
uppercase_region
uppercase_word

TABS

back_tab
convert_tabs
default_tabs
get_tab
insert_tab
reset_tabs
save_tab
settab
settabs_from_table
tab
tablist
tablist_to_array
type_tab
whitespace_to_hpos

TESTS

at_white_char
beginning_of_buffer_p
bolp
cursor_on_current_line_p
cursor_same_line_p
empty_buffer_p
end_of_buffer_p
end_of_line_p
eolp
first_line_p
have_input_p
if
if_at
last_line_p
line_is_blank
looked_at
looking_at
more_args_p
null
same_buffer_p

TYPE TESTS

atom
charp
numberp
stringp
typef

WHITESPACE
(See also TABS)

at_white_char
back_to_first_nonwhite
cret_indent_relative
delete_white_left
delete_white_right
delete_white_sides
indent_to_fill_prefix
indent_relative
indent_line_to_hpos
leave_one_white
skip_back_to_white
skip_back_over_white
skip_over_white
skip_to_white
split_line
whitespace
white_delete

WINDOWS

current_major_window
go_to_window
hcol
hscroll
major_window_count
mod_one_window
mod_split_window
one_window
other_window
reset
scroll_other_backward
scroll_other_forward
select_any_window
set_hscroll
split_window
split_window_stay
vsplit
window_info

WORDS

back_word
capinitial
delete_word
forward_word
lowercase_word
mark_end_of_word
rubout_word
token_chars
transpose_word
uppercase_word

INDEX

Index

SURVEY

READER RESPONSE FORM

DOC5025-2LA   EMACS Extension Writing Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent   ___very good   ___good   ___fair   ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand   ___average   ___very clear

   Technical level: ___too simple   ___about right   ___too technical

   Technical accuracy: ___poor   ___average   ___very good

   Examples: ___too many   ___about right   ___too few

   Illustrations: ___too many   ___about right   ___too few

3. What features did you find most useful? ______________________________

   ____

   ____

   ____

4. What faults or errors gave you problems? ______________________________

   ____

   ____

   ____

Name: ______________________________ Position: ____________________

Company: ____

Address: ____

____Zip: ____________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760