ULTRIX

The Big Gray Book: The Next Step with ULTRIX

Order Number: AA-PBKNA-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This manual describes features of the ULTRIX operating system and its related tools for users with some ULTRIX experience.

digital equipment corporation
maynard, massachusetts

Contents

3 Line-Oriented Editors

Part II: Communication with Other Users

6 Mail

7 Interactive Communication

Part III: Other Commands and the Shell

8 Calculators

9 C Shell Scripts

Tables

About This Manual

Welcome

The Big Gray Book: The Next Step with ULTRIX is an intermediate manual on working with the ULTRIX operating system and its related tools. Like *The Little Gray Book: An ULTRIX Primer*, it is based on the theory that you will do most of your work with only a small part of the computer's capabilities. This book goes beyond the *Primer*, introducing you to more advanced tools that will help you to make more and better use of the computer.

Audience

This book is a guide for intermediate users that also serves as a reference for users who have gained more experience. It assumes that you have read the *Primer* or that you are otherwise familiar with the material presented in the *Primer*.

Organization

This book is divided into three parts, and has two appendixes:

Part I – Text Manipulation
: Discusses commands and utilities useful for manipulating text files and the material in them, including editors and searching tools.

Part II – Communication with Other Users
: Discusses commands and utilities that provide ways to communicate with other users, including mail and interactive communication.

Part III – Other Commands and the Shell
: Describes assorted useful commands and the C shell itself, with emphasis on creating your own commands in the form of shell scripts.

Appendix A
: Contains examples illustrating use of some of the tools and utilities described in the book.

Appendix B
: Describes solutions for difficulties commonly encountered with using the ULTRIX system.

Related Documents

The Little Gray Book: An ULTRIX Primer introduces the ULTRIX operating system and some of the tools and utilities discussed here, and is a handy reference as you read this book.

The *Guide to the nawk Utility* is a thorough tutorial description of an enhanced version of the `awk` utility discussed in Chapter 4.

The *ULTRIX Reference Pages* provide details of the commands and utilities described in this book.

The ULTRIX operating system *Supplementary Documents, Volume 1: General User* contain exhaustive descriptions of some of the utilities discussed in this book.

Learning the vi Editor, one of the Nutshell Handbooks available from O'Reilly and Associates, Inc., describes the `vi` editor in detail.

Conventions

The following typeface conventions are used in this manual:

`vizier>`	The default user prompt is your system name followed by a right angle bracket. The system name `vizier` is used in this manual.
`user input`	This bold typeface is used in interactive examples to indicate typed user input.
`system output`	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. (Example text enclosed in a box indicates text matching a regular expression.) In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
filename	In examples, syntax descriptions, and function definitions, italics are used to indicate variable values; and in text, to give references to other documents.
macro	In text, bold type is used to introduce new terms.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
RETURN	This symbol is used in examples to indicate that you must press the named key on the keyboard.
CTRL/*x*	This symbol is used in examples to indicate that you must hold down the CTRL key while pressing the key *x* that follows the slash. When you use this key combination, the system sometimes echoes the resulting character, using a circumflex (^) to represent the CTRL key (for example, ^C for CTRL/C). Sometimes the sequence is not echoed.

The Next Step 1

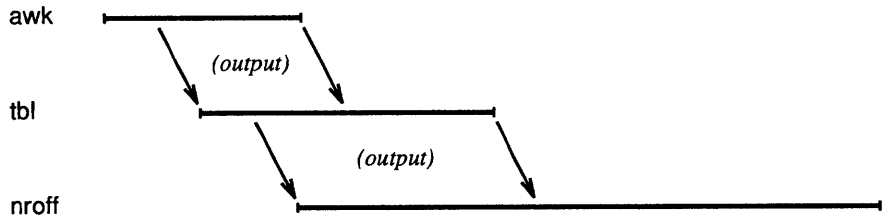
The *The Big Gray Book: The Next Step with ULTRIX* is an intermediate guide to commands, tools, and utilities that are part of the ULTRIX operating system. While not a step-by-step tutorial, this book builds on the skills introduced by *The Little Gray Book: An ULTRIX Primer*. Instead of trying to present a comprehensive and unnecessarily complex view of the entire ULTRIX system, the *Primer* showed you only a few tools and commands that you need for much of the ordinary work you do. This book continues in this vein, introducing you to a relatively small number of additional tools and commands. The topics are those that you are likely to use often in your work.

1.1 The Toolbox Philosophy

One of the most powerful features of the ULTRIX system is the way its various tools work together. Instead of providing a single massive utility that does everything but might not be able to do all its tasks in the most efficient way, the ULTRIX system's "toolbox" philosophy offers many smaller utilities that you can use as you need them. This design allows you to bring only the processing power you need to bear on the task you're performing. For example, if you want to format a document that contains tables, you can use the `tbl` preprocessor, described in Chapter 5, and then the `nroff` text formatter. If you have a document without tables, there is no need to use the special table-formatting capabilities of `tbl`; instead, you use only `nroff`. Selecting only the tools you need saves time, because the system can perform your job faster. It also allows others to work faster, because there are more system resources available for them to use.

Coupling tools with pipelines saves time by allowing the system to overlap the tasks it is performing. As soon as the first tool in a pipeline begins producing output, the next tool can begin working on that output without having to wait for the first tool to finish. As soon as the second tool produces some output, the third tool can begin, and so on. Figure 1-1 illustrates the process.

Figure 1-1: Using a Pipeline to Couple Several Utilities



This book describes many different tools. By learning about several tools that do similar things, you can decide more easily which of them is best suited to the task at hand.

1.2 What's In This Book?

This book's three parts cover commands and utilities for which you will find frequent use in your day-to-day work. Each part deals with several tools that work together or perform similar tasks related to a particular general subject.

1.2.1 Part I – Text Manipulation

The chapters of Part I describe tools for text location, filtering, editing, and organization.

One feature that makes the ULTRIX operating system effective is its ability to find the information you want. There are tools that search for information and, having found it, process it into a different form. For example, you can use the `grep` utility (Chapter 4) to scan a price list of office supplies, creating an order for the supplies you're out of; the `sed` stream editor (Chapter 3) to perform automatic text editing; and the `awk` utility (Chapter 4) to tabulate the order and calculate its total cost. These are all things you can do manually, but it is often easier to let the computer do the work for you while you do something else that the computer cannot do.

A key part of using ULTRIX tools effectively is knowing how to use regular expressions (called "regular" because their formation follows a specific set of rules). The concept of regular expressions is similar to that of the asterisk (`*`) and question mark (`?`) wildcards you use to name files in shell commands, but regular expressions are used to locate information within files instead of finding the files themselves. Chapter 2 describes regular expressions in detail.

1.2.2 Part II – Communication with Other Users

Part II provides a thorough description of the `mail` utility and an introduction to another mail-handling system called MH. With a little exploration, you should emerge from Chapter 6 as an expert in the use of mail.

Part II also describes several commands that are useful for interactive communication. With these utilities, you can send quick messages or have a two-way conversation in real time.

1.2.3 Part III – Other Commands and the Shell

Part III extends the *Primer's* description of the C shell, providing an introduction to creating your own commands by writing shell scripts. Suppose you have a complicated procedure that you have to do once every three months. This job requires you to copy several files, edit some of them, format the results, and finally print the formatted files. You could keep a notebook to remind yourself how to handle this job. Or you could take the time to write a shell script, a program for the shell, so that your job would work automatically. Writing a script has another advantage: When you move on to your next job, the script you leave behind will make your old job easier for the new person.

Also in Part III are descriptions of generally useful commands such as interactive and programmable calculators.

Part I: Text Manipulation

Regular Expressions 2

This chapter describes regular expressions and how to use them. This discussion is basic to using many of the utilities described in later chapters. In this chapter you will meet regular expressions in the context of the `grep` utility, which was introduced in *The Little Gray Book: An ULTRIX Primer.*

2.1 What Is a Regular Expression?

In the *Primer*, you learned how to use the `grep` command to search for strings in a file. You can also search for strings in the standard input stream; for example, you can use the following piped commands to see if a user named `daniels` is logged in.

```
vizier> who | grep daniels
daniels tta1  Jul 31 1989
```

This ability to search for an exact string is useful but limited. For example, it doesn't let you search a file for two or more different strings at the same time. But if you use regular expressions, you can search for strings containing common elements, such as ''board'' and ''beard'', quickly and easily.

In algebraic equations, you can use a letter, such as x or y, to represent any number. When you use a letter like this in an expression such as $3+\sqrt{x}$, the expression takes on different values depending on the value of x. Another kind of expression, although not a mathematical one, is a wildcard. (In the *Primer*, you learned how to use wildcards to represent any characters in a file name.) **Regular expressions** (REs) are an extension of this ability to represent more than one character. They are to strings of characters what mathematical expressions are to numbers.

Regular expressions are called ''regular'' because they conform to a set of rules. The first two rules are as follows:

- Any ordinary character is a simple RE that represents, or matches, itself.
- When you concatenate simple REs, the result is a compound RE that matches the concatenation of the strings matched by each of its components.

The first rule says that any ordinary character is an RE that matches itself; for example, `a` represents `a` and nothing else. That might seem obvious, but this concept is important to understanding more complex REs.

The meaning of the second rule might not be immediately clear. As an example of this rule, the following `grep` command finds file names in `/bin` that contain the letters ''iz''. (In this chapter's examples, we'll put boxes around the text that matches the RE being used.)

```
vizier> ls /bin | grep iz
pagesize
size
```

Specifying `iz` as the RE for `grep` to match finds all strings that match first the "i" and then, immediately after it, the "z". The second rule means that any string of characters, like `iz`, is really a series of simple REs put together to make a compound RE. A simple RE is one that matches a single entity (usually one character) in the text being processed. A compound RE is one that matches a series of entities. (By framing a compound RE as described in Section 2.3, you can make it behave like a simple RE.)

The examples shown so far probably don't look very useful because they merely illustrate searching for the exact string you want. The next rule of REs opens up all sorts of new possibilities:

- A period (.) matches *any* character.

Suppose you are preparing an order for office supplies and need to find the stock numbers for blue pushpins and red felt-tip pens. By using a period as part of your RE, you can search your group's list of supplies for "pin" and "pen" at the same time this way:

```
vizier> grep p.n ~bjornson/supplies
 02141  Felt pen, black                  8/box
 02142  Felt pen, blue                   8/box
 02143  Felt pen, red                    8/box
 02144  Felt pen, green                  8/box
 09667  Three-hole paper punch           Unit
 13785  Pushpin, red                     150/box
 13786  Pushpin, yellow                  150/box
 13787  Pushpin, green                   150/box
 13788  Pushpin, blue                    150/box
 31591  Paint, japan, white              Bottle
 31592  Paint, japan, red                Bottle
 31593  Paint, japan, green              Bottle
 31594  Paint, japan, yellow             Bottle
 31595  Paint, japan, purple             Bottle
 31596  Paint, japan, blue               Bottle
 31597  Paint, japan, black              Bottle
 31598  Paint, japan, orange             Bottle
```

The period matches both the "i" in "pin" and the "e" in "pen". But because the period matches any character, this RE also matches some things you weren't looking for, such as the japan paints. There are other REs that let you avoid undesired matches.

Now we begin to see the power of REs.

2.2 The Rules for Regular Expressions

Ordinary characters and metacharacters together make up the set of simple REs. Table 2-1 describes the rules for creating REs.

Table 2-1: Rules for Regular Expressions

Expression	Name	Rule
`0-9, A-Z, a-z,` most punctuation	Ordinary character	Matches itself.
.	Period (dot)	Matches any single character.

Table 2-1: (continued)

Expression	Name	Rule
char	Backslash	Matches the character following the backslash regardless of whether that character is an RE metacharacter or not.
*	Asterisk	Matches any number of occurrences of the preceding RE, including none.
[*chars*]	Brackets	Matches any one of the characters within the brackets. Ranges of characters can be abbreviated; for example, `[0-9a-z]` matches any digit or any lowercase letter.
^	Circumflex	Matches the beginning of a line when at the beginning of an RE. When used as the first character inside brackets, excludes the bracketed characters from being matched. Otherwise, has no special properties.
$	Dollar sign	Matches the end of a line when at the end of an RE. Otherwise, has no special properties.
+	Plus sign†	Matches one or more occurrences of the preceding simple RE. (Not available to all utilities.)
?	Question mark†	Matches zero or one occurrence of the preceding simple RE. (Not available to all utilities.)
expr expr . . .	Concatenation	Forms a compound RE that matches any string that matches the first simple RE, then the second, and so on.
(*expr*)	Parentheses†	Encloses, or frames, an RE, allowing metacharacters that act on the preceding RE to treat the entire framed RE as a simple RE. (Not available to all utilities.)
|	Vertical bar †	Separates multiple REs. (Not available to all utilities.)

You can combine any or all of these kinds of REs to do the job you need to do. Items that are not marked with a dagger (†) in this table are available to all the utilities that use REs, such as `ed`, `vi` and `ex`, `sed`, and `grep`. The dagger indicates features that are available only to specific utilities such as `awk` and `egrep`. The following sections discuss using the general REs in Table 2-1 in more detail. Later chapters describe the ways the various utilities use REs; the items marked with a dagger are discussed where appropriate.

2.2.1 Matching Any Single Character

In Section 2.1 we showed you how to use a period to match a single character. This way of using the period is exactly the same as using a question mark in file names; you can concatenate more than one period to represent an exact number of characters.

For example, suppose you are writing a report and need to search a list of turn-of-the-century Japanese warships[1] for all ships whose displacement was between 10,000 and 19,999 tons:

```
vizier> grep 1.,... warships
Yashima          1896     12,517   18 kt   4x12in, 10x6in, 16x12-pdrs
Fuji             1896     12,517   18 kt   4x12in, 10x6in, 16x12-pdrs
Shikishima       1898     15,088   18 kt   4x12in, 14x6in, 20x12-pdrs
Asahi            1899     15,443   18 kt   4x12in, 14x6in, 20x12-pdrs
Mikasa           1900     15,362   18 kt   4x12in, 14x6in, 20x12-pdrs
Hatsuse          1899     15,240   18 kt   4x12in, 14x6in, 20x12-pdrs
```

As shown by this example, you do not have to put the periods together; you can place them wherever you need, and you can use as many as you need.

2.2.2 Matching Metacharacters As Ordinary Characters

A backslash (\ ) makes the character following it lose its special RE properties, if it had any, so that you can search for actual occurrences of characters such as the period. For example, suppose you are looking for cross-references in a series of recipes[2]:

```
vizier> grep 'No\.' egg-with-liver
half a pint of Madiera sauce (No. 103); and let cook for five
minutes; make an omelet of twelve eggs, as for No. 46, and
```

Note that we have used apostrophes (single quotation marks) to enclose the RE in this example. Some of the metacharacters used in REs are also shell metacharacters; for example, the backslash is also used by the shell to disable a following character's special properties. To prevent the shell from attempting to interpret metacharacters in an RE, enclose the entire RE in apostrophes.

You can also make the shell pass metacharacters by preceding each one with a backslash; for example:

```
vizier> grep No\\. egg-with-liver
half a pint of Madiera sauce (No. 103); and let cook for five
minutes; make an omelet of twelve eggs, as for No. 46, and
```

In this example, the first backslash forces the shell to pass the second backslash to `grep`. It is usually less confusing to use apostrophes, especially when the metacharacter you want to pass is a backslash, as shown here. Remember that the apostrophes are not part of the RE syntax; they're just used to make the shell ignore metacharacters in an RE.

2.2.3 Matching Any Number of Occurrences of a Character

In file names, an asterisk (*) stands for any string of characters, even a null one. As part of a compound RE, it's a little different. It stands for any number of occurrences of the preceding RE, even none. Suppose you are preparing a new American edition of *The Coming Race,* by Edward Bulwer-Lytton. The original edition used British spellings of words like "colour," and your task is to find and change all these usages[3]. If you have completed part of the job and want to find where you left off, you could check all the book chapter files with this command:

---

[1] From *Universal Cyclopedia and Atlas,* Volume 10. D. Appleton and Company, 1903.

[2] From *100 Ways of Cooking Eggs,* by Filippini. Charles L. Webster & Company, New York, 1892.

[3] From *The Coming Race,* first edition, published anonymously. Francis B. Felt & Co., New York, 1871.

```
vizier> grep 'colou*r' tcr.ch*
tcr.ch3:have seen above the earth; the color of it not green,
tcr.ch5:of gold in the colors, like pictures by Louis Cranach.
tcr.ch5:rich in colouring, showing a perfect knowledge of
tcr.ch5:intermediate valleys of mystic many-coloured herbage,
  .
  .
  .
```

This command shows that you left off partway through chapter 5. Using an RE with an asterisk after the "u" causes `grep` to find every instance of either "color" or "colour". This command would also have found any mistakes like "colouur" because the asterisk matches any number of occurrences of the individual RE before it. (Note that this example also uses an asterisk as a file name wildcard.)

When an asterisk follows a period, the combination indicates a match on any sequence of characters, even none. The period matches any character and the asterisk says to match any number of them. Suppose you need to scan a list of your computer's users to find a person named John Smith. There are several John Smiths. If the list were organized properly, you could search for "Smith, John" – but someone has made the list with first names first. You could use this command to find all the John Smiths:

```
vizier> grep 'John.* Smith' /usr/users/names
John Andrew Smith       Office 237      Ext 1234
John Charles Smith      Office 118      Ext 2835
John Smith              Office 533      Ext 7614
John Smith Smith        Office 101      Ext 7814
John Wellington Smith   Office 976      Ext 5476
```

In this example, any sequence of "John*anything* Smith" is matched, including "John*nothing* Smith". For John Smith Smith, the first occurrence of "Smith" is enough to trigger the match.

2.2.4 Matching Only Selected Characters

A period represents any character in an RE. But sometimes you don't want to search for every possible combination that your RE will match. Placing the desired match characters inside brackets ([ ]) allows you to restrict the match to only those characters you really care about. Each set of bracketed characters is a single-character RE that matches any *one* of the bracketed characters. Suppose you want to search a story file for the words "bare" and "byre". The following example does what you need:

```
vizier> grep 'b[ay]re' story
The girl studied his bared head for a few moments and
the byre with the cattle."
```

In this example, the bracketed expression matches the "a" in "bare" on one line and the "y" in "byre" on another. All other possible characters between "b" and "re" are ignored, so the RE doesn't match words like "arboreal".

Sometimes you need to match a string regardless of the case (upper or lower) of some of the letters in it. You can do this by using a bracketed RE consisting of just the upper- and lowercase versions of the character you want. For example:

```
vizier> grep '[Kk]ing' bible-report
books chronicle the history of the Jews under their kings.
language found only in the King James Version.  Such usage is
```

By using a series of bracketed pairs you can create an entire compound RE that is case insensitive:

```
vizier> grep '[Gg][Ii][Nn]' miscellaneous-file
BEGINNING EMBROIDERY TECHNIQUE
birth of Virginia Dare in Roanoke.
Janice's Super Ginger Snaps
```

2.2.5 Using the Circumflex in Regular Expressions

The circumflex (^) has two functions in REs:

- Matching the beginning of a line
- Excluding a match on certain characters

2.2.5.1 Matching the Beginning of a Line – Sometimes you want to match an expression only at the beginning of the line. For example, suppose you want to find "Roberts, Kenneth" in a list of authors. You could look for "Kenneth" but that would also find "Galbraith, John Kenneth". You could look for "Roberts" but that would also find "Rinehart, Mary Roberts". By using a circumflex at the beginning of an RE, you can force a match on "Roberts" only if it occurs at the beginning of a line:

```
vizier> grep '^Roberts' authors-list
Roberts, Kenneth                American historical fiction
```

Note that if the circumflex is not the first character of the RE, it is not a special character. In this case, it matches itself just as any ordinary character does.

2.2.5.2 Excluding a Match on Certain Characters – As described in Section 2.2.4, placing a series of characters in brackets forms a single-character RE that matches any one of the bracketed characters. If you use a circumflex as the first character inside the brackets, however, the RE you construct will match any character *except* those in the brackets. The following example searches the list of supplies we used in Section 2.1, but it excludes the letters "a" and "u" from its search so that you will see only the things you want.

```
vizier> grep 'p[^au]n' ~bjornson/supplies
 02141  Felt pen, black                  8/box
 02142  Felt pen, blue                   8/box
 02143  Felt pen, red                    8/box
 02144  Felt pen, green                  8/box
 13785  Pushpin, red                     150/box
 13786  Pushpin, yellow                  150/box
 13787  Pushpin, green                   150/box
 13788  Pushpin, blue                    150/box
```

2.2.6 Matching the End of a Line

Although matching the end of a line is a less common task than matching the beginning, it is still useful. (You use an end-of-line match most often when you are editing a file, as described in Chapter 3.) Suppose you are writing a paper on poetry and want to scan a file of limericks to find a line that rhymes with "sonnet." You can use an RE that ends in a dollar sign ($) to force this kind of match:

```
vizier> grep 'onnet$' limericks
There was a Young Lady whose bonnet
```

This line is from a limerick by Edward Lear[4].

Note that if the dollar sign is not the last character of the RE, it is not a special character; in this case it matches itself just as any ordinary character does.

2.2.7 Matching Exact Numbers of Occurrences of Characters

We have shown how to match any number of occurrences of a character. Sometimes you want to limit the number of occurrences; you can do this by using a plus sign (+) or a question mark (?).

2.2.7.1 Matching an Expression That Appears One or More Times – The plus sign matches one or more occurrences of the simple RE that it follows.

As indicated in Table 2-1, the plus sign is not valid for all the utilities that use REs. The `grep` command does not use them, so you would have to use `egrep`, described in Chapter 4, with this RE. For example:

```
vizier> egrep '[Ss].l+' boxing-report
Even the great John L. Sullivan was not immune to flattery.
in the middle of a solo tour of New England, Dempsey met
```

Here, the plus sign says that the "l" must occur one or more times. By using the plus sign instead of an asterisk, we prevent a match on words like "so".

2.2.7.2 Matching an Expression That Appears Once or Not At All – The question mark matches exactly one occurrence or zero occurrences of the RE that it follows.

The question mark is also not available to the `grep` command; you must use `egrep` to search for REs using the question mark. For example:

```
vizier> egrep '[Ss].l?' boxing-report
   .
   .
   .
in the middle of a solo tour of New England, Dempsey met
His ankle was weakened so that when he stepped on the
   .
   .
   .
```

The question mark says that the "l" must occur once or not at all. This requirement means that the RE in this example matches not only three-character sequences like "sol", but also any two-character sequence beginning with "S" or "s" *unless* it is followed by "ll". This exclusion is the reason this example does not match "Sullivan".

---

[4] *A Book of Nonsense*, by Edward Lear, 1846. Reprinted in *The Complete Nonsense of Edward Lear*, edited by Holbrook Jackson. Dover Publications, Inc., 1951.

There was a Young Lady whose bonnet
Came untied when the birds sate upon it;
But she said, 'I don't care! all the birds in the air
Are welcome to sit on my bonnet.'

2.3 Making a Compound Regular Expression Simple

As noted in Table 2-1, metacharacters that apply to the preceding RE, such as the plus sign, apply only to the preceding simple RE, not to an entire compound RE. By **framing** a compound RE, you can make it behave like a simple RE so that a following metacharacter can act on it. You frame an RE by enclosing it in parentheses.

For example, suppose you want to search your `.mailrc` file to recall the mail alias you assigned to your group manager, whose login name is `jane`. But there is also a user named `janene` on the system, and you've also assigned that user an alias. You could search for the name `jane`, but that would list both aliases. To see only the one alias, you could use a framed RE followed by a question mark. Framed REs are not used by the `grep` command, so you would use the following `egrep` command:

```
vizier> egrep 'ja(ne)?' .mailrc
alias jane boss
```

The question mark in this example excludes two occurrences of the framed RE.

2.4 Building Complex Regular Expressions

Once you are familiar with all the REs and their rules, you can combine them in any way you need to make a very specific compound RE. For example, you can search a list of names for every occurrence of the names Jean, Joan, Jeanne, or Joanne. An RE to find just these four strings while excluding everything else is easy to construct, but it's not as obvious as it might seem. One user we know tried this RE:

```
J.*an*
```

This RE works, but it also finds many more strings:

```
vizier> grep 'J.*an*' /usr/users/names
Jan Blankenship          Office 724      Ext 7633
Jian Lee Chen            Office 761      Ext 7523
Jane Crawford            Office 854      Ext 2996
Joann Daugherty          Office 451      Ext 7612
Joan Davis               Office 562      Ext 4345
Jacqueline Deschamps     Office 734      Ext 6781
Joanna Exeter            Office 423      Ext 6512
Jack Heisler             Office 422      Ext 7611
Mary Jameson             Office 414      Ext 8763
Desmond Jeannotte        Office 292      Ext 2722
Janene Leighton          Office 612      Ext 2323
Jeanne Sexton            Office 993      Ext 1111
Joanne Stevens           Office 438      Ext 6485
Jake Willis              Office 765      Ext 1752
Jean Wilson              Office 124      Ext 7826
```

Because it allows for any number of occurrences (including zero) of both the "n" and the character following the "J", the RE in this example is not restrictive enough.

To construct the RE we want, let's look at the names Jean, Joan, Jeanne, and Joanne piece by piece:

1. The RE starts with `J`.
2. Then, to find only `e` or `o`, we use the bracketed characters `[eo]`.
3. Next comes `an`.

4. Last, to match the optional `ne` at the end of the name, we create a framed, or parenthesized, RE, `(ne)`.
5. To prevent more than one match on the `(ne)`, we follow it with a question mark, which matches only zero occurrences or one occurrence.

The final compound RE, then, is this:

```
J[eo]an(ne)?
```

This RE will not match any string except the four we are looking for.

As indicated in Table 2-1, parenthesized REs and the question mark are not valid for all the utilities that use REs. The `grep` command does not use them, so you would have to use `egrep`, described in Chapter 4, with this RE. For example:

```
vizier> egrep 'J[eo]an(ne)?' /usr/users/names
Joan Davis              Office 562      Ext 4345
Jeanne Sexton           Office 993      Ext 1111
Joanne Stevens          Office 438      Ext 6485
Jean Wilson             Office 124      Ext 7826
```

2.5 Separating Regular Expressions

It is often useful to be able to match two or more radically different REs in a single operation. For example, suppose you are writing a treatise on light and color. You want to rework all the places where color is mentioned, but you have used several different words to refer to different aspects of it. You can find all the references with REs separated by vertical bars (|). The vertical bar isn't used by `grep`, so you would have to use `egrep`, as in this example:

```
vizier> egrep 'color|hue|shade|tint' light-report
discovered the relationship of hue to intensity by setting
successful color photograph by using three separate films,
difference between shade and tint as applied to paints is
```

Although this example uses only simple strings for its REs, you can use both simple and compound REs that are as complex as required to match the text you want to find.

Line-Oriented Editors 3

This chapter describes three text editors that work differently from the screen-oriented `vi` editor that was introduced in the *Primer*. These editors are called line-oriented editors or **line editors** because they work on one line of text at a time. While `vi` can do all of the things these editors do, it is often inefficient to use `vi` in the ways in which line editors excel.

This book does not discuss the `vi` editor in detail; for more information on `vi`, refer to the ULTRIX reference documentation or to a `vi` book such as *Learning the vi Editor,* one of the Nutshell Handbooks available from O'Reilly and Associates, Inc.

The editors discussed in this chapter make use of regular expressions (REs) for addressing and pattern matching. If you are not already familiar with REs, read Chapter 2 before reading this chapter.

3.1 Types of Line Editors

Line editors can be divided into two types: interactive and noninteractive. You are familiar with the way an interactive editor works: You give the editor a command, and the editor performs it and waits for the next command. A noninteractive, or **stream,** editor does not accept commands from you; it does its job by reading a program, or script, that you prepare before you invoke the editor.

Although you can use line editors for any editing task you can do with `vi`, they are particularly useful for making quick edits such as fixing a typographical error (typo) on line 327 of a file, or for making global changes such as correcting that same typo everywhere it appears in a file. The ULTRIX stream editor, called `sed`, provides a mechanism not only for quick fixes of this type but also for extended repetitive editing tasks such as processing a series of mail messages to remove header information and compile a single report file.

The ULTRIX operating system offers two interactive line editors and one stream editor:

- `ed` and its restricted version `red`

  The `ed` program is the standard interactive line editor. It allows you to make any desired change in a file. You can enter your edits by selecting a specific line or group of lines by number, or by searching for a pattern of text that identifies the line or lines you want to alter.

  The `red` editor is a restricted version of `ed` that allows you to edit only files that are in your current working directory. You cannot execute shell commands while using `red`.

- `ex`

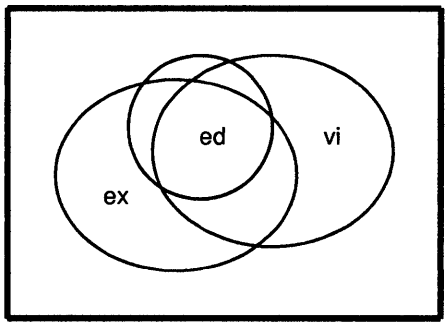
  The `ex` editor is a superset of `ed`; it is also the root of a family of editors that includes `vi`. You can switch back and forth between `ex` and `vi` in a single editing session.

- `sed`

  The `sed` program is an optimized stream editor that performs commands specified as option arguments on the command line or in a script that you supply.

Figure 3-1 illustrates how `ed`, `ex`, and `vi` are related. The `sed` stream editor is very much like `ed`.

Figure 3-1: The Relationship of ed, ex, and vi



The following sections describe the `ed` and `ex` editors and the `sed` stream editor. The examples in this chapter illustrate editor features using excerpts from the collection of poems known as *Rubáiyát of Omar Khayyám*[1].

3.2 The ed Editor

If you are familiar with `vi`, you will find a number of similarities between its colon-introduced commands and the commands for `ed`. This similarity is is due to the fact that `vi` is a member of the `ex` editor family, which is an extended version of `ed`.

Note that, as with `vi`, commands for `ed` are case sensitive. For example, the `p` command does something different from the `P` command.

When you invoke the `ed` editor to work on a file, the editor creates a temporary copy of the file, called a **buffer.** All editing is performed in the buffer; the real file is altered only if you give an explicit write command. This design protects you in case you make edits and then change your mind or in case of a system crash caused by, for example, a power failure.

[1] From *Rubáiyát of Omar Khayyám, the Astronomer-Poet of Persia,* rendered into English verse. Empire State Book Company, New York, 1924.

3.2.1 Starting the ed Editor

The first thing to know about using `ed` is that it does not normally give you an obvious prompt. When you invoke `ed`, as in the following example, it displays the number of characters in the file and then just sits there waiting for a command. For example:

```
vizier> ed rubaiyat
1265
```

Note the position of the cursor, indicated here by a box (□). You can make `ed` give you a more noticeable prompt by entering the `P` (Prompt) command:

```
P RETURN
*
```

The `P` command also turns off the asterisk prompt if it has been turned on. A command that turns a feature on and off alternately like this is called a **toggle**.

If you don't like the asterisk, you can invoke `ed` with the `-p` *string* option; this option makes `ed` use *string* for its prompt. For example:

```
vizier> ed-p
1265
ed>
```

You must end each `ed` command by pressing the RETURN key. We will not show the RETURN key in our examples unless we're indicating that you should enter a blank line. We'll also omit the cursor box in the remaining examples.

To create a file from scratch using the `ed` editor, start the editor using the name of your new file. The editor responds with a question mark and the file name, to say that the file does not currently exist. For example:

```
vizier> ed sample1
?sample1
```

Enter an `a` command (discussed in Section 3.2.3.1) with no address and then enter the text for the new file. The editor wants to append the text you enter after the current location (the last line of the buffer), but since there is nothing in the file, your new text becomes the entire contents of the file. Once you've created your file, you can leave the editor as described in Section 3.2.12.5.

3.2.2 Moving Around the Buffer

The `ed` editor locates lines in the buffer by means of addresses. An address can be a line number or a regular expression (RE). Line numbers can be relative or absolute. In addition to all the forms of REs shown in Chapter 2, you can use a special form of RE that includes two or more matches of the same identical string. You can also mark lines with single-character identifiers and return to those lines later.

Line numbers are not very useful unless you have a listing of the file with numbers printed on it. You can make such a listing by using the `cat` command's `-n` option:

```
vizier> cat -n rubaiyat
     1                          I
     2
     3  Wake!  For the Sun, who scatter'd into flight
     4  The Stars before him from the Field of Night,
     5    Drives Night along with them from Heav'n, and strikes
```

```
  6  The Sultan's Turret with a Shaft of Light.
     .
     .
     .
 43                          VII
 44
 45  Come, fill the Cup, and in the fire of Spring
 46  Your Winter-garment of Repentance fling;
 47    The Bird of Time has but a little way
 48  To flutter -- and the Bird is on the Wing.
```

To make a printed copy of the listing, you can pipe the `cat` command's output to the `lpr` command.

3.2.2.1 Moving in the Buffer Using Line Numbers – The editor starts out at the last line of the buffer. To display this line, enter the `p` (print) command:

```
p
To flutter -- and the Bird is on the Wing.
```

To select a different line, enter its number:

```
31
Iram indeed is gone with all his Rose,
```

When you move to a line, `ed` displays the line for you without requiring a `p` command. The editor interprets a plain line number as if it included the `p` command.

The period (`.`) command displays the current line. Using this command is one way to see changes you have made after you make them. (Edited lines are not redisplayed automatically.) You can also display the current line by entering the `p` command.

3.2.2.2 Moving in the Buffer Using Relative Addresses – Besides giving an absolute address (line number) to select a line, you can also use a relative address. To select a line relative to the current line, use a minus sign (-) or a plus sign (+) before the number you enter:

```
+2
  But still a Ruby kindles in the Vine,
-8
The thoughtful Soul to Solitude retires,
```

There are some ways to make moving short distances a little quicker. If you are making many edits in a file by reading each line and then stepping to the next, you can just press RETURN with no command. The editor understands this to mean the same as a `+1` command. A command consisting of just a plus sign also means the same as `+1`:

```
p
The thoughtful Soul to Solitude retires,
+
  Where the White Hand of Moses on the Bough
RETURN
Puts out, and Jesus from the Ground suspires.
RETURN
(blank line)
```

A command consisting of just a minus sign means the same as a `-1` command:

```
-
Puts out, and Jesus from the Ground suspires.
```

The editor also understands the dollar sign to mean the last line of the file:

```
$
To flutter -- and the Bird is on the Wing.
```

3.2.2.3 Moving in the Buffer Using Regular Expressions – Using line numbers is an effective way to move through a file unless your editing removes or adds lines in the file. If that happens, all the line numbers after the added or deleted material are changed. In many cases, you can get around this problem by planning all the changes you intend to make and then working through the buffer backward.

But going through the buffer backward doesn't always work; for instance, you might be moving chunks of text around. There is a second way to find the line you want: by using a regular expression. Suppose you want to find the first line in the file that contains the letters "ou". You can do that by specifying an RE. To indicate to `ed` that you're entering an RE, type a slash as the first character. In examples using REs, we will indicate the matching text by enclosing it in a box. For example:

```
/ou
Methought a Voice within the Tavern cried,
```

As with `vi`, the slash causes a forward search. Entering a slash alone repeats the search to find the next occurrence of the same RE:

```
/
Why nods the drowsy Worshipper outside?"
/
The Tavern shouted -- "Open then the Door!
```

If you enter a search command for which there are no more matches after your current location in the buffer, `ed` goes to the end of the buffer and then continues its search from the beginning.

You can search backward by using a question mark instead of a slash at the beginning of your RE. This feature provides an easy way to back up to the last edit you made if it was on an earlier line.

3.2.3 Adding and Deleting Text

With a line editor, you can't add or delete text in the middle of a line. You can only work in terms of complete lines. The `ed` editor has three commands for adding and deleting information:

Command	Addresses	Description
`a`	0, 1	Appends text after the specified line.
`i`	0, 1	Inserts text before the specified line.
`d`	0, 1, 2	Deletes lines of text.

Throughout this chapter, commands are listed in tables with the number of addresses they can accept. See Section 3.2.5 for a discussion of using addresses with commands.

3.2.3.1 Adding Text – Suppose you want to add another stanza after the last line of the file. You can do this by moving to the line and then entering the `a` (append) command. To end your addition, enter a line containing only a period:

```
$
To flutter -- and the Bird is on the Wing.
a
RETURN
TAB TAB TAB IX
RETURN
Each Morn a thousand Roses brings, you say;
Yes, but where leaves the Rose of Yesterday?
And this first
Summer month that brings the Rose
Shall take Jamshyd and Kaikobad away.
```

When you finish adding text, the editor leaves you positioned on the last line you added.

This example has an intentional error; the sixth and seventh lines of the new text should be a single line. You can correct this error with the `j` (join) command, described in Section 3.2.4.2. This example is also wrong because it is out of order; stanza VIII should go here instead of stanza IX. Section 3.2.7 shows how to correct this error by moving blocks of text.

You can append text after any line in the file. You can also insert text before any line in the file by using the `i` (insert) command. This command works exactly like the `a` command except that it inserts the new text before the current line instead of after it.

3.2.3.2 Deleting Text – To delete lines, position the editor on the first line you want to delete and enter the `d` command once for each line to remove.

3.2.4 Changing Text

Although you can't add or delete text within a line, the `ed` editor has commands for changing text in ways other than adding or deleting lines:

Command	Addresses	Description
`s`	0, 1, 2	Substitutes the second argument for the first.
`c`	0, 1, 2	Changes the addressed lines by deleting them and replacing with new text.
`j`	0, 1, 2	Joins lines together, making them one line.
`u`	0	Undoes the previous edit.

3.2.4.1 Changing Text by Substitution – To substitute one string for another, use the `s` (substitute) command. This command requires two arguments, one to tell it what is to be changed and one to describe how the change is to be made. You use slashes to set off the arguments. Suppose you want to correct a typo on line 17 of the file:

```
17
And, as the Cock crow, those who before
s/crow/crew/
```

This command finds the first occurrence of the characters "crow" on the line (the first argument) and substitutes "crew" for them (the second argument). Note that the changed line is not displayed automatically. You can make `ed` display the changed line by adding a `p` command to the end of the `s` command's arguments:

```
s/crow/crew/p
And, as the Cock crew, those who before
```

Using the `s` command allows you to simulate adding or deleting text within a line. To add a word, for example, you can substitute for the last few characters of the word before your new text, using those same characters and your new word as the second argument. For example, the line changed in the previous example is missing a word. The following command corrects this error:

```
s/who/who stood/p
And, as the Cock crew, those who stood before
```

The `s` command allows you to use a null first argument. When you use a null first argument, `ed` uses the last expression it searched for. For example, line 39 of the file looks like this:

```
High-piping Pehlevi, with "Wine, Wine, Wine!
```

The first two instances of "Wine" should be followed by exclamation points instead of commas. The following example corrects these errors:

```
/Wine,
High-piping Pehlevi, with "Wine, Wine, Wine!
s//Wine!/p
High-piping Pehlevi, with "Wine! Wine, Wine!
s//Wine!/p
High-piping Pehlevi, with "Wine! Wine! Wine!
```

In this example, the `s` command is entered twice. Normally, the `ed` editor changes only the first occurrence of the first argument that it finds on the line. You can make the same series of changes more efficiently by using the `g` (global) option for the `s` command. Using the `g` option makes `ed` apply the specified change everywhere on the line. The following example makes the same change as the previous example:

```
/Wine,
High-piping Pehlevi, with "Wine, Wine, Wine!
s//Wine!/gp
High-piping Pehlevi, with "Wine! Wine! Wine!
```

Note that you can combine the `g` and `p` options to make `ed` redisplay the line after making your change.

In the preceding examples, we have entered a null RE (two slashes with nothing between them) for the first argument to all but the first command. When you enter a null RE in this way, `ed` reuses the last RE it searched for. In these examples, the first command searches for `Wine,` – using a null RE in sebsequent commands causes `ed` to search for the same string again.

You can often simplify text substitution by using an ampersand (&) in the second argument to represent the text that was matched by the first argument, as in the following example:

```
/Sev
And Jamshyd's Sevn-ring'd Cup where no one knows;
s//&'/p
And Jamshyd's Sev'n-ring'd Cup where no one knows;
```

The ampersand in the second argument of the s command duplicates the string matched by the first argument. In this case, the first argument is null, so the match is on the string "Sev" that was matched by the command that found the line. The final result of the substitution is to add an apostrophe after this string.

When you use the ampersand as in this example, it makes no difference whether the first argument was an explicit string or an RE that might have matched more than one string. The ampersand represents the actual text that was matched, not the expression that matched it.

When you are addressing lines to be edited, you can use either line numbers or regular expressions as the addresses. You can also use an RE as the first argument for an s command. Because the second argument for an s command is the exact text you want to use, metacharacters have no special meanings in the second argument; you do not need to precede them with backslashes. If you want to include a backslash in the second argument, however, you must precede it with another backslash.

3.2.4.2 Changing Text by Replacing and Joining Lines – You can alter an entire line by using the c (change) command. This command deletes the line you specify and then replaces it with everything you type until you end with a line containing only a period. This action is the same as a d command followed by an i command.

You can join two lines together with the j command. For example, the sixth and seventh lines we added to illustrate the a command in Section 3.2.3.1 should actually be a single line. To join the two lines, enter the j command:

```
/And this
  And this first
Summer month that brings the Rose
jp
  And this firstSummer month that brings the Rose
s/tS/t S/p
  And this first Summer month that brings the Rose
```

As with the s command, you can include a p command to make ed display the joined lines. The result of joining the lines isn't exactly what we want, so we use an s command to add the missing space.

3.2.4.3 Correcting Editing Errors – If you make a mistake in your editing, you can use the u (undo) command to reverse the last change you made.

3.2.5 Combining Commands and Addresses

Locating a line and then operating on it, as we've shown you in the preceding sections, is rather cumbersome. This technique also limits you to working with one line at a time. You can work more efficiently by using ed's ability to couple addresses with most of its commands.

3.2.5.1 Using Commands with Single Addresses – To enter an address and a command at the same time, type the address before the command. Suppose you want to add another stanza at the end of the file:

```
$a
RETURN
TAB TAB TABVIII
RETURN
Whether at Naishapur or Babylon,
Whether the Cup with sweet or bitter run,
The Wine of Life keeps oozing drop by drop,
The Leaves of Life keep falling one by one.
.
```

(Note that this stanza is out of sequence; we added stanza IX in Section 3.2.3.1.)

You can use both line numbers and regular expressions as addresses when you enter commands in this way. Knowing that the last line of the file before the previous example is line 57 and that it contains the words "Shall take", you could also have used either of the following `append` commands:

```
57a
```

or

```
/Shall take/a
```

Note that you must use a second slash to separate the RE from the command. If you want to make a change earlier in the file than the current location, you can use two question marks to set off the address RE.

Be careful when using REs as addresses. If you specify an RE whose matching text occurs more than once in the file, you could make your alteration in a location far removed from where you intended.

You can use addresses with many commands. For example, you could correct "revising" to be "reviving" (on line 24 of the file) with this command:

```
/revising/s//reviving/p
Now the New Year reviving old Desires,
```

3.2.5.2 Using Commands with Two Addresses – There are times when you want to work with more than one line at a time. The editor accepts two addresses for many commands. When you enter two addresses, they indicate a group of lines starting at the first address and ending at the second. You separate addresses with a comma. For example, the following command displays the first five lines of the file. The `n` command, shown in this example, lists lines together with their line numbers.

```
1,5n
1                                I
2
3       Wake!  For the Sun, who scatter'd into flight
4       The Stars before him from the Field of Night,
5         Drives Night along with them from Heav'n, and strikes
```

Addressing multiple lines gives you the ability to make changes throughout part or all of a file with a single command. Suppose you want to change the file so that it is suitable for formatting with `nroff`. To do this, you need to insert a `.sp` command wherever there is a blank line. You can do this job as follows:

```
1,$s/^$/.sp/
```

In this example, the `s` command, with an address range from the first line to the end of the buffer, uses an RE to find all the blank lines (lines with nothing between the beginning, indicated by the circumflex, and the end, indicated by the dollar sign) and change them to the `.sp` command.

Almost all of `ed`'s commands work with two addresses in this way. It is largely this ability to make global changes rapidly that makes line editors better than `vi` for some tasks. (You can use `vi` to make global changes, but when you do so you are actually using the `ex` editor's line-oriented commands.)

3.2.6 Marking Lines in the Buffer

It is often convenient to mark lines in the buffer so that you can return to them later. The `k` command marks lines by assigning identifiers consisting of a single lowercase letter that you supply as an argument after the `k` command. Suppose you want to mark line 10 before doing something else:

```
10kx
```

The preceding example marks line 10 with the identifier `x`. Now you go off and make a change elsewhere in the file. Afterward you can return to line 10 without having to remember its number. To return to a marked line, you enter an apostrophe followed by the identifier you gave to the line, as shown in the following example. (This example adds another stanza at the end of the file and then uses the previously set marker `x` to return to line 10.)

```
$a
RETURN
TAB TAB TABx
RETURN
Well, let it take them.  What have we to do
With Kaikobad the Great, or Kaikhosru?
Let Zal and Rustum bluster as they will,
Or Hatin call to Supper -- heed you not.
.
'x
Before the phantom of False morning died,
```

You can mark up to 26 lines in this way. All marks are lost when you leave the `ed` program.

3.2.7 Juggling Blocks of Text

In addition to changing bits or lines of text here and there in the file, you can use the following `ed` commands to pick up a block of text from one location and place it in another:

Command	Addresses	Description
`m`	0, 1, 2	Moves the specified block of text to a different location.
`t`	0, 1, 2	Makes a copy ("takes a picture") of the specified text after the target line, leaving the original text untouched.

In the preceding examples, we added several stanzas to the file. We added stanza IX before stanza VIII. To put these stanzas in the correct order, you can use the `m` command. The `m` command moves the lines specified by one or two addresses preceding the command, placing them after the line specified by the argument following the command. Stanza VIII currently consists of lines 57 to 63; stanza IX begins at line 50. The following command moves stanza VIII to its proper place:

```
57,63m49
```

Note that we use line 49 as the target instead of 50. Remember that the moved text is placed after the target, not before it. Line 49, the end of stanza VII, is the line after which stanza VIII is to be placed.

You can often simplify moving text by using REs or marked line identifiers as the addresses and argument for the `m` command. Suppose that you have marked line 49 using the `k` command and that are you are currently positioned on line 63. The following command would make the same change as the preceding example:

```
/VIII/,.m'x
```

In this example the RE `VIII` finds the line that begins the selected block. The period specifies that the current line is the end of the block. The `'x` identifies line 49.

The `t` command works the same as the `m` command except that it duplicates the indicated text block at the target location instead of actually moving it.

3.2.8 Making Global Changes Interactively

Sometimes you want to make the same change in many places throughout the file. Using an `s` command with an address range from line 1 to the end of the file is the logical way to perform this task, but there are times when you don't want to change every occurrence of the string you are looking for. You can execute such global changes selectively by using the `G` command. This command accepts zero, one, or two addresses and a single optional RE argument. Without the RE, the command works on all the lines in the range you specify; if you include the RE, the command works only on lines that match the RE.

For each valid line, the `G` command displays the line and then accepts a single interactive command from you. After performing this interactive command, `ed` moves to the next valid line specified by the `G` command. In the following example, we want to change a period after "them" into an exclamation point *somewhere* in the file. We could use an RE to find the line, but we've chosen to illustrate the `G` command here:

```
1,$G/them/
  Drives Night along with them from Heav'n, and strikes
RETURN
Well, let it take them.  What have we to do
s/\./!/p
Well, let it take them!  What have we to do
RETURN
With Kaikobad the Great, or Kaikhosru?
```

For the first occurrence of "them" in the file we enter a null command (a blank line) because that is not the line we want to change. The null command makes `ed` move ahead to the next occurrence. For the second occurrence, we enter a command to make the desired change. (Because the period is an RE metacharacter, we must use a backslash before the period we want to change.) The next null command we enter

displays the line containing "Kaikobad the Great". Because it does not contain the string we were searching for, this line indicates that the G command's function is complete.

3.2.9 Error Messages and Help

The ed editor provides two commands to help you figure out what has happened when something goes wrong:

Command	Description
h	Displays an error message explaining the last ? response.
H	Toggles the display of error messages.

If you give the ed editor an invalid command, the editor responds by displaying a question mark (?). This terse response is in keeping with the UNIX philosophy of being concise. You can ask ed what it is complaining about by entering the h (help) command. For example:

```
70s/Hatin/Hatim/p
?
h
line out of range
```

In this example ed is saying you have specified a line that does not exist. (The s command specifies line 70, but there are only 69 lines in the file.) You can make ed always respond with an error message instead of just a question mark by entering the H command. Entering the same command again toggles ed's error display mode, returning to the mode of displaying question marks.

3.2.10 Matching Multiple Occurrences of a String

It is sometimes useful to specify an address that includes matches on more than one occurrence of a given string. For example, there are many lines in the file that contain "The" or "the" more than once. But there is only one line that contains "The" and two other occurrences of "he". You could search for this line by specifying an RE as in the following example:

```
/The.*he.*he
The Tavern shouted -- "Open then the Door!
```

In this instance this RE is the simplest approach, but we want to illustrate a powerful extension that makes use of framed REs.

The ed editor accepts REs that are enclosed, or framed, by parentheses, in the following form:

\(*expr*\)

The backslashes tell ed to interpret the parentheses as metacharacters framing an RE instead of as literal characters. This RE matches exactly the same things as the same RE without the enclosing parentheses, but by framing it in this way you set it off so that ed can refer to it later in a more complex RE. You can frame (and refer to) more than one RE in the same compound RE. To refer to a framed RE later in a compound RE, you use a backslash followed by a number *n* that represents the *n*th

framed RE in the expression. For example, the following RE matches the same line matched by the previous example:

```
/T\(he\).*\1.*\1
The Tavern shouted -- "Open then the Door!
```

The parentheses frame the string "he". A period followed by an asterisk matches any string. The backslash and number 1 refer to the first framed string in the RE; in this example, there is only one. The period, asterisk, backslash, and number 1 are repeated, and the complete RE matches only a line that contains "The" and two other occurrences of "he".

This feature is limited in that you cannot use bracketed characters to restrict what your framed RE matches.

3.2.11 Executing Shell Commands from Within ed

You can execute a shell command by preceding it with an exclamation point:

```
!ls rub*
rubaiyat        rubaiyat.bak
!
```

This example lists all the files in your working directory whose names begin with "rub". The editor displays an exclamation point to indicate that the shell command has finished and that you are now back in the editor.

You can include a percent sign (%) in shell commands that you issue from within ed. The percent sign is replaced with the current buffer name:

```
!ls -l %
ls -l rubaiyat
-rw-r--r--  1 hale         1265 Aug 11 14:30 rubaiyat
!
```

3.2.12 Managing the File and Quitting ed

The following commands let you keep track of your file and leave ed:

Command	Addresses	Description
w [*file*]	0, 1, 2	Writes all or part of the buffer. If *file* is specified, ed writes to a file of that name.
e [*file*]	0	Discards the current state of the buffer and reads the file again as it was before the editing session began. If *file* is specified, ed reads that file instead.
E [*file*]	0	Like e, except that ed does not warn you of impending loss of the current buffer.
r [*file*]	0, 1	Reads a file into the buffer.
f *name*	0	Changes the name of the current buffer to *name*.
q	0	Ends your editing session. If the buffer is unwritten, ed warns you.
Q	0	Like q, but ed does not warn you if the buffer is unwritten.

3.2.12.1 Saving the Buffer – When you are ready to end your editing session, you must save your file before leaving `ed`, or all your changes will be lost. You save the file with a `w` (write) command:

```
w
1810
```

When you enter a `w` command, `ed` replaces the original contents of your file with the contents of the buffer and tells you how many characters are in the file. (This process is called writing the buffer.) You can create another copy of the file under a different name by using the `w` command with a file name:

```
w rubaiyat-save
1810
```

If you specify an address range, the `w` command writes only the specified part of the buffer. This feature is useful for creating files to be included later into other files you edit. For example, the following command writes only the first two stanzas of the poem to a file called `sample1`:

```
1,13w sample1
374
```

3.2.12.2 Rereading the File – If you have been making changes in the buffer and then decide that you want to throw them away, you can clear the buffer and read the file again with an `e` command. When you give the `e` command, `ed` warns you that it expects you to save the old buffer first. Repeating the command tells `ed` that you really intend to destroy what you have and start fresh. You can also use the `E` command; this command is like `e` except that the editor proceeds without warning you.

You can also use the `e` and `E` commands to read in a different file for editing by specifying the name of the file you want to edit. For example:

```
E rubaiyat-save
1810
```

3.2.12.3 Including Other Files – You can include other files as part of the one you are editing by using an `r` (read) command and specifying the file name.

3.2.12.4 Renaming the Buffer – You can change the name of the buffer you are editing with the `f` command (file name):

```
f new-rubaiyat
new-rubaiyat
```

3.2.12.5 Leaving the ed Editor – When you have finished your editing session, you leave `ed` by entering a `q` command. If you have not yet written your buffer, `ed` warns you that the contents are about to be lost. If you repeat the command, `ed` discards the buffer and exits. You can make `ed` exit without warning you by using the `Q` command instead of `q`.

3.2.13 Recovering from a Crash

The buffer file is located in the `/tmp` directory and named `e`*nnnnn*, where *nnnnn* was the process ID number of the editing process; for example:

```
/tmp/e05044
```

When a crash occurs, you can recover the last state of your buffer by locating the buffer file with an `ls -l /tmp` command and then renaming it with an `mv` command before you resume editing. You should use `ls` with the `-l` option to make sure you recover your own buffer and not someone else's.

3.3 The ex Editor

The `ex` editor is a more sophisticated version of `ed`; it includes all of `ed`'s functions and provides more. Its most important facility is that you can use `ex` commands while you are in `vi` and you can switch back and forth between `ex` and `vi` in a single editing session.

The introduction to `vi` given in the *Primer* describes how to use commands that begin with a colon. These are all `ex` commands; the colon is used in `vi` because, unlike `ed`, the `ex` editor always displays a colon as a command prompt.

To switch to `ex` from `vi`, enter the `Q` command (not `:Q`). To return to `vi`, enter the `visual` command.

The `ex` editor accepts all `ed` commands. Most of them can also be entered as a command word instead of a single letter, for example, `undo` instead of `u` or `write` instead of `w`. This might seem contrary to the UNIX philosophy of being concise, but many people are more comfortable thinking in whole words.

The `ex` editor supports options that affect how editing is done. Options are entered as editor commands. For example, the `autoindent` option sets `ex` to prepare indented program code. When you enter a command that adds text (`append`, `insert`, and so on), `ex` looks at the line after which the added text is to be placed and calculates the amount of white space at the beginning of the line. That amount of space is inserted at the beginning of each new line; that is, the edit creates a virtual left margin, simulating a tab stop. If you add more white space to indent nested code further, the virtual margin shifts accordingly. You can back up through the indentation as code nesting levels decrease.

The `ex` editor allows you to create an initialization file that will execute commands and set editing options each time the editor is started. You can use one or more personalized initialization files to tailor the editor for different editing tasks; the environments created in this way prevail whether you are working in `ex` or in `vi`. To use an initialization file, create the file in your home directory and give it the name `.exrc`. The `ex` editor will automatically read the initialization file and perform the commands it contains before beginning to edit the file you want to work on. The `vi` editor uses the same initialization file.

For a thorough discussion of `ex` features, refer to the ULTRIX *Supplementary Documents, Volume 1: General User.*

3.4 The sed Stream Editor

The `sed` stream editor's command syntax is almost identical to that of the `ed` interactive editor. The `sed` editor reads commands from a program, or **script,** that

you prepare before invoking the editor. It compiles the commands to make sure they are all valid and to arrange them in the most efficient fashion, and then it executes them. For quick edits, the editor also accepts a sequence of commands supplied as arguments to a command-line option. You can combine command-line editor commands with a script.

The line or group of lines specified by any editing command is called the command's **pattern space**. The pattern space is equivalent to the line or lines selected by the addresses you use with `ed` commands. The editor proceeds through the file line by line, applying to each line all the commands whose pattern spaces include that line. The output of each command is passed to the next, so that edits are cumulative. This line-by-line editing procedure is why `sed` is called a stream editor. It also explains one of the major limitations of `sed`: you cannot use relative addresses in `sed` commands. Only absolute line numbers or regular expressions (REs) are permitted. When you specify a pattern space by using REs, every line or series of lines matching the REs will be processed by the command. For example, the following command finds and deletes every line containing the string "Kaikobad":

```
/Kaikobad/d
```

3.4.1 Using sed with a Script

You create a `sed` script using the `cat` command or any editor you choose. The script consists of a series of editor commands. For example, the following script makes some changes to the `rubaiyat` file that we used in the section describing the `ed` editor:

```
s/^$/.sp/
1i\
\.RP\
\.TL\
The Rubaiyat of Omar Khayyam\
\
\.nf\
\.na
```

If this script file is named `sedscr`, you would process the `rubaiyat` file by entering a command like this one:

```
vizier> sed -f sedscr rubaiyat | more
```

The `-f` option tells `sed` that you are using a script file. The option requires the script file's name as an argument.

This script substitutes `.sp` commands for all the blank lines in the file, to ensure that the stanzas will be separated in the formatted output. Then it inserts a `.RP` macro call, a `.TL` macro call, the text for the `.TL` macro, and `.nf` and `.na` commands to tell `nroff` not to perform line filling or justification on the poem.

Because `sed` works with lines in a stream, it also expects text that you give it for `a` (append), `c` (change), and `i` (insert) commands to be a single line. If you want to enter more than one line of text for these commands, each line you enter except the last must end with a backslash; this technique "hides" the new-line characters from `sed`. This sample script uses backslashes in this way to insert several lines at the beginning of the file. Note that the `i` command itself is also terminated with a backslash. The last line to be inserted does not end with a backslash; the lack of a backslash there indicates the end of the new text.

Note that unlike ed or ex, the sed editor writes its output to the standard output, leaving the original file unaltered. You could use a script similar to this to process your story for formatting so that you would not have to maintain the title and other front matter in the file; you would redirect the output to a second file name instead of viewing it with the more command. Using a sed script like this would be a handy way of avoiding the need to maintain nroff title information and formatting commands in your file.

3.4.2 Using sed for Quick Edits

You can also use sed for quick editing without a script. If you invoke sed with the -e option, the argument following the -e is a sed command. For example:

```
vizier> sed -e 's/^$/.sp/' rubaiyat > rubaiyat.sp
```

This example changes blank lines to .sp commands throughout the file, the same as if we had included the command in a script. Note that the command is enclosed by apostrophes ('). The apostrophes keep the shell from interpreting metacharacters in the command before passing the command to sed.

You can pass several commands to sed in this way by using a series of -e options:

```
vizier> sed -e 's/^$/.sp/' -e '$r more-stanzas' rubaiyat
```

You can also pass several commands to sed in the same -e argument by separating them with semicolons:

```
vizier> sed -e 's/^$/.sp/;$r more-stanzas' rubaiyat
```

You can include both the -e option and the -f option for the same sed command. This ability lets you create a standard script to use in conjunction with additional edits that you specify at the time you run the command.

3.4.3 Command Syntax and Addressing

The command syntax for sed is almost identical to that for the ed interactive editor. Many commands can have zero, one, or two addresses and zero, one, or two arguments. Some commands do not accept addresses or arguments.

For commands that accept addresses, a single address specifies a one-line pattern space. Every line in the file that matches the pattern will be processed. Two addresses specify a pattern space that includes the first addressed line, the second addressed line, and all the lines between them. Edits are applied to the first group of lines that match the addresses, and then sed searches for a new group of lines on which to work. This process is repeated through the entire file. Supplying no addresses for a command that accepts addresses means that the command is applied to the entire file. The first line of the short sample script we showed to edit the rubaiyat file is a substitute command with no addresses; this command changes each blank line that is found anywhere in the file.

You can specify that a given command is to be performed on every line that does *not* match the addressed pattern space by placing an exclamation point between the address and the command. For example:

```
/Kaikobad/!s/rose/Rose/g
```

This command changes "rose" to "Rose" wherever it occurs unless it is on a line that also contains "Kaikobad".

3.4.4 Compound Commands

There is one exception to the addressing rules described in Section 3.4.3: the use of compound commands. This exception provides a very powerful way to control the scope of editing. It is often useful to apply a series of commands to the same pattern space. You cannot always do this by specifying the same addresses for each command in the series, because one command in the series might delete or alter part of the pattern space so subsequent commands couldn't find a proper match. For example, suppose you made several mistakes in entering this stanza into the `rubaiyat` file:

```
                        XI

With me along the strip of Herbage strewn
That just divides the dessert from the sowm,
  Where name of Slave and sutlan is forgot --
And Peas to Mahmud on his golden Throne?
```

Obviously, you could make the explicit changes one at a time, but let us generalize a little for purposes of illustration. You might try editing the stanza with the following `sed` script:

```
/strewn/,/Throne/s/strewn/strown/
/strewn/,/Throne/s/dessert/desert/
/strewn/,/Throne/s/sowm/sown/
/strewn/,/Throne/s/sutlan/Sultan/
/strewn/,/Throne/s/Peas/Peace/
```

The first command changes the word "strewn" that you are using for the starting address of the pattern space. The remaining commands will work incorrectly because they cannot find their address. They might simply be unable to make the specified changes, but the failure could be catastrophic: if there is a match on "strewn" anywhere else in the file, some or all of your changes could be made in locations far removed from where you intended.

You can avoid this problem by creating a compound command. A compound command begins with an address or pair of addresses followed by a left brace ({). On subsequent lines are the commands to be performed. You end the compound command with a right brace (}) on a line by itself. The following example shows a compound command that will perform the edits attempted by the incorrect example above:

```
/strewn/,/Throne/{
  s/strewn/strown/
  s/dessert/desert/
  s/sowm/sown/
  s/sutlan/Sultan/
  s/Peas/Peace/
}
```

This example works because the addresses specify an area in which to operate; all the commands that follow are applied to the entire area. The second editing command is not looking for an address match, so it does not matter that the beginning address has been altered by the previous command.

Note that the editing commands in this example are indented from the margin. You don't have to indent, but indenting helps you to keep track of what a script is doing, especially if you nest compound commands inside each other. For example, the following script does the same things as the preceding example, but for illustration it uses nested commands to make two changes on the line containing "dessert":

```
/strewn/,/Throne/{
```

```
  s/strewn/strown/
  /dessert/{
    s//desert/
    s/sowm/sown/
  }
  s/sutlan/Sultan/
  s/Peas/Peace/
}
```

As with simple commands, you can make a compound command edit everything that does not match its address by including an exclamation point before the opening brace. When you apply this technique to a command that is within a compound command, the edits are applied only to the parts of the compound command's address space that do not match the address space of the nested command. Lines outside the compound command's address space are not affected. For example:

```
2,6{
 3,4!s/string1/string2/
}
```

This example changes all occurrences of `string1` on lines 2, 5, and 6 of a file. Lines 3 and 4 are excluded by the exclamation point; line 1 and everything after line 6 are excluded by the compound command's address range.

3.4.5 Additional sed Features

Because you cannot interact with `sed` as it edits your file, you cannot always do the things you want to do using only the commands that `sed` has in common with `ed`. For this reason, `sed` has several additional features that allow you to do some very powerful editing. The following sections describe these features.

3.4.5.1 Using the Print Command – The `sed` editor's `p` command does not display the specified pattern space on your terminal; instead, it writes the pattern space to the destination. You can use this feature to produce more than one copy of a pattern space in the output file. You can also use it to produce an output file containing only the lines you specify by invoking `sed` with its `-n` option. This option inhibits normal output; only pattern spaces explicitly written with the `p` command are sent to the output file. In the following example, the output is not redirected, so it comes to your screen:

```
vizier> sed -n 's/Kaikobad/Marvin/p' rubaiyat
Shall take Jamshyd and Marvin away.
With Marvin the Great, or Kaikhosru?
```

3.4.5.2 Joining Lines – You cannot use the `ed` editor's `j` command to join lines in `sed`. Instead, the `sed` editor has the `N` command, which joins the next line to the current one. When this command is executed, it joins the two lines with an embedded new-line character between them. (The `ed` editor does not embed anything between the joined lines.) You can operate on this new-line character by using the special character `\n`. For example, to join lines 2 and 3 of a file and then remove the new-line character, you would use these commands:

```
2{
  N
  s/\n//
}
```

3.4.5.3 Substituting Characters – The `sed` editor's `y` command performs one-for-one character substitutions. The command requires two arguments, which must be strings of exactly the same length. For each occurrence of any character in the first argument, the `y` command substitutes the corresponding character from the second argument. The command performs its change on every matching character in the pattern space; you do not have to specify the `g` option as you would with the `s` command. For example, the following command changes lowercase letters into uppercase letters:

```
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Using a command like this example provides a rapid way to convert an entire file from mixed case or lowercase into all uppercase letters for use, perhaps, as a telephone listing. For an example of this command's use in a practical `sed` script, see Example 3-1.

3.4.5.4 Holding and Getting Text – The `sed` editor does not support the `ed` editor's `m` and `t` commands. To move text from one place to another or for other operations that require temporary storage of a block of text, `sed` has the following commands:

Command	Addresses	Description
`H`	0, 1, 2	Stores the contents of the pattern space in the hold buffer.
`h`	0, 1, 2	Appends the contents of the pattern space to the existing contents of the hold buffer.
`G`	0, 1, 2	Replaces the contents of the pattern space with the contents of the hold buffer.
`g`	0, 1, 2	Appends the contents of the hold buffer after the end of the pattern space. Further edits in a compound command are applied to the appended text as well as the previous pattern-space contents.

These commands are very useful for handling text moves. You can also use them to perform some fairly exotic manipulations. For example, the tool that formatted and typeset this book uses a `sed` script to perform diagnostic checks on the book files. The script uses REs as addresses to locate special formatting commands. When a command is found, the script saves a copy of the pattern space in a temporary storage area (the **hold buffer**). It then tests the command for correct syntax by looking within the pattern space for further matches. If the required information is not found, the script uses an `i` command to insert an error message. Then it gets the saved command back from the hold buffer and writes the pattern space using a `p` command. The final output consists of error messages followed by the lines that caused them. For example:

```
Jul 28 14:58 1990          ***  Diagnostic Report ***          Page 1


*** File: sample.profile ***

*** File: sample.ch1 ***

ERROR: Unterminated tag argument(s)
    32     .\" <include>(example-file
```

```
*** File: sample.ch2 ***

*** File: sample.ch3 ***
```

This usage of `sed` takes advantage of the `sed` editor's `-n` option, described in Section 3.4.5.1. Example 3-1 shows several excerpts from this diagnostic script.

Example 3-1: Sample sed Script

```
/\.\\"[ TAB]*<[A-Za-z].*>/{    1
  h    2
  y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/    3
  /<abstract>/!{
    /<include>/!{    4
      /<style>/!{
        /<title>/!{
          i\    5
\
ERROR: Unrecognized tag
          g; p; d    6
        }
      }
    }
  }
  />(/{
    /)[ TAB]*$/!{    7
      i\
\
ERROR: Unterminated tag argument(s)
      g; p; d
    }
  }
}
```

This script illustrates several pattern matching and text manipulation techniques you can use:

1. This code locates special formatting commands in the file being processed. These commands, called tags, consist of a word delimited by angle brackets, such as `<abstract>`. Tags are case insensitive, so the RE is designed to look for any letter, upper- or lowercase, followed by zero or more other characters between the angle brackets.

2. This line saves the tag in the hold buffer. When tag syntax is being checked by later commands, the pattern space might be altered; this `h` command allows the editor to recover the exact text of an erring tag for display.

3. To simplify processing the tags, this line converts the entire line to lowercase. This way, it is not necessary to use a series of bracketed REs for each character of a tag when comparing against the known tags.

4. This code matches each lowercased tag against the list of known tags. If the tag is not an `<abstract>` tag, it is compared to the `<style>` tag, and so on. If no match is found, the the tag is one that the processing tool does not recognize.

5. Having found an unrecognized tag, the script inserts an error message.

6. The saved text is recovered from the hold buffer and printed along with the inserted error message. Then the pattern space is deleted to prevent further checks from being made on it.

7 Errors like the following example are common:

```
.\" <tag-name> (argument)
```

The brace is not a proper terminator. This part of the `sed` script looks for tags with arguments to ensure that there is a closing parenthesis for each such tag.

See Section A.1 for an example that uses the `sed` stream editor to extract information from a mail message.

Pattern-Matching Utilities 4

This chapter describes grep and its related utilities, and the awk utility. These utilities provide powerful ways of searching for and reorganizing information.

The examples in this chapter use a file containing the following shopping list, called shop, for their input. Columns in the table are separated by tab characters.

```
10 oz    2        .69    frozen peas
10 oz    1        .89    frozen broccoli
doz      1       2.25    fresh corn
lb       1        .45    rice
lb       3       1.89    apples
24 oz    2        .79    fruit cocktail ?
1/2 gal  2       1.89    ice cream
gal      3       2.39    milk
4 oz     1        .89    Fruit Twirls
16 oz    1       2.59    Tasty Squares
lb       1       1.89    rye bread
12 ct    1       1.19    burger buns
lb       2       1.79    hamburger
```

The utilities discussed in this chapter make use of regular expressions (REs). If you are not already familiar with REs, read Chapter 2 before reading this chapter.

The discussion of awk in this chapter is intended only as an introduction to its capabilities; for a thorough tutorial description of an enhanced version of awk called nawk, see the *Guide to the nawk Utility*.

In examples, we will enclose the information for which we are searching in apostrophes to prevent the shell from interpreting any metacharacters within the information. Another name for the information being searched for is **pattern**; in the grep family, the pattern is an RE, while in awk REs form only a subset of the patterns that can be used. In this chapter's examples, text that matches a search pattern is enclosed in boxes.

4.1 The grep Family of Utilities

The name grep stands for "global regular expression printer." You have been introduced to grep in the *Primer* and again in Chapter 2 of this book, where it was used to demonstrate the use of regular expressions.

The simplest use of grep is to search a file for a specific string:

```
vizier> grep 'pea' shop
10 oz    2        .69      frozen peas
```

There are three versions of the grep command. Although they appear to work very much alike, each version has special features that make it better for certain uses. Table 4-1 describes the different versions of grep. One difference between the grep versions is in their use of program space. **Program space** is the amount of

memory used by a program; the more program space a given program uses, the less there is available for other users' programs.

Table 4-1: Versions of the grep Utility

grep Version	Description
`grep`	Patterns can contain a limited set of regular expressions. (See the list immediately following this table.) The `grep` command uses a compact searching method that is fast and requires a minimum of program space.
`egrep`	"Extended `grep`" patterns make use of all the regular expressions. The `egrep` command uses a more complicated searching method that can sometimes require exponential data space. (Searching for twice the number of patterns can require four times the space in memory.)
`fgrep`	"Fixed `grep`" patterns are fixed strings (explicit character sequences instead of REs). "Rubaiyat" is a fixed string; "Ru.*at" is an RE. The `fgrep` command is extremely fast and compact.

The set of REs supported by `grep` is limited so that `grep` can work more efficiently for most uses. The additional power of `egrep` supports the full range of REs, including the following features that are not supported by `grep`:

- You can use a plus sign (+) after an RE, including an RE that is part of a larger compound RE, to require a match on one or more occurrences of the RE.
- You can use a question mark (?) after an RE, including an RE that is part of a larger compound RE, to require a match on exactly zero occurrences or one occurrence of the RE.
- You can separate REs with vertical bars (|) to make `egrep` search for more than one pattern at a time. For example:

```
vizier> egrep 'u[ai]|[pr]ea' shop
10 oz   2        .69    frozen peas
1/2 gal 2       1.89    ice cream
24 oz   2        .79    fruit cocktail ?
4 oz    1        .89    Fruit Twirls
16 oz   1       2.59    Tasty Squares
lb      1       1.89    rye bread
```

- You can frame an RE in parentheses.

The `fgrep` command does not allow REs, but it does allow you to specify more than one string. You surround the strings with apostrophes (single quotation marks) and separate them with a backslash followed immediately by pressing the RETURN key, as in this example:

```
vizier> fgrep 'pea\
ice' shop
Size    Qty     Price   Item
10 oz   2        .69    frozen peas
1/2 gal 2       1.89    ice cream
```

4.1.1 Modifying the Behavior of the grep Utilities

By default, the `grep` commands find each line that matches your pattern or patterns, printing the line on the standard output. Table 4-2 describes command-line options that allow you to specify other results from your searches.

Table 4-2: Options for the grep Utilities

Option	grep Versions	Description
`-b`	All	Precedes each output line with its disk block number. This option is of use primarily to programmers who are trying to identify specific blocks on a disk by searching for the information contained in them.
`-c`	All	Counts matching lines and prints only the count.
`-e` *expression*	All	Uses *expression* as the pattern. Useful if *expression* begins with a minus sign (`-`).
`-f` *file*	`egrep`, `fgrep`	Searches for a list of patterns taken from *file*.
`-i`	`grep`, `fgrep`	Performs a case-insensitive search.
`-l`	All	Lists only the names of files containing matching lines. Each file name is listed only once, even if the file contains multiple matches.
`-n`	All	Precedes each matching line with its line number.
`-s`	All	Performs its search in "silent" mode, printing nothing except error messages.
`-v`	All	Prints only lines that do not match the specified expressions.
`-w` *expression*	`grep`	Matches only if *expression* is found as a separate word in the text.
`-x`	`fgrep`	Prints only lines matched in their entirety.

Some of these options need more explanation.

The `-f` option for `egrep` and `fgrep` allows you to specify the name of a file containing the patterns instead of including them on the command line. This option is useful when you have much information to search for or when you search for the same information repeatedly.

Each pattern to be matched must be entered on a separate line of the pattern file; for example, the sample `egrep` command shown near the end of Section 4.1 could also have been executed this way:

1. Create a file containing the patterns to be matched:

```
vizier> cat > grepfile
u[ai]
[pr]ea
CTRL/D
```

2. Execute the `egrep` command, specifying the pattern file with the `-f` option:

```
vizier> egrep -f grepfile shop
```

The `-i` option makes `grep` or `fgrep` ignore the difference between uppercase letters and lowercase letters. For example:

```
vizier> fgrep -i 'fruit' shop
24 oz   2        .79    fruit cocktail ?
4 oz    1        .89    Fruit Twirls
```

With `grep`, you can also match either uppercase or lowercase letters by creating bracketed REs that contain the appropriate pairs, such as `[Ff]`. Each technique is useful under different circumstances; using the `-i` option makes all matches case insensitive, whereas using bracketed REs treats only the specific characters you bracket.

The `-s` option makes `grep` print nothing to the standard output. Instead, `grep` sets status information so that a subsequent command can determine whether the search was successful. (See Sections 9.5.4.3 and 9.9 for discussion of the status information and how to use it.) This option makes the `grep` commands especially useful in shell scripts because it allows a script to test whether a file contains desired information without actually displaying the information.

The `-w` option for `grep` constrains the search so that the expression being matched must be found as a separate word in the file. For the purposes of this match, the term "word" means that the matching text must be both preceded and followed by nonalphanumeric characters. Nonalphanumeric characters are white-space characters (tabs or spaces) and punctuation, except that the underscore character (_) is treated as if it were a letter. For example:

```
vizier> grep -w 'ice' shop
1/2 gal 2        1.89    ice cream
```

This example finds the line containing "ice cream" but not the line containing "rice".

4.2 The awk Utility and Programming Language

The first thing many new users ask about `awk` is, "What does that name stand for?" The name `awk` is an abbreviation of the last names of Alfred Aho, Peter Weinberger, and Brian Kernighan, the engineers who created the `awk` utility.

This section provides only a brief introduction to the power of `awk`. The `nawk` utility is an enhanced version of `awk`; refer to the *Guide to the nawk Utility* for a thorough tutorial on all the utility's features.

The `awk` utility combines pattern matching with the ability to process the matched information. The processing ability of `awk` is actually a complete programming language in its own right.

An `awk` statement has the following form:

pattern { *action* }

If the action is missing, every line that matches the pattern is printed. Using `awk` this way is like using one of the `grep` commands except that `awk` can use much more sophisticated patterns. If the pattern is missing, the action is performed for every line in the file. If both the pattern and the action are missing, the entire file is printed; the result is the same as if you had used the `cat` command.

4.2.1 What Can awk Do?

To get a quick idea of what awk can do, look at this example, which adds up the cost of all the items in the shopping list file shop:

```
vizier> awk '-F[TAB]' '{s += ($3 * $2)} END {print "Total",s}' shop
Total 33.31
```

What happened in this example? The awk utility has actually performed three distinct operations:

1. The awk command read the file, interpreting each line, or **record,** as a series of columns, or **fields,** separated by field separators. By default, fields are separated by any amount of white space (spaces or tab characters). Each field is identified by a dollar sign ($) followed by the field's number. Field $1 is the first column, $2 is the second, and so on.

   Because there are spaces within the columns of the shopping list file, this example uses awk's -F option to specify that only tabs are to be used as field separators. The -F option and the tab character are enclosed in apostrophes to prevent the shell from interpreting the tab as ordinary white space.

2. Because this example specifies no pattern, awk operated on every record in the file, performing the following action:

   ```
   {s += ($3 * $4)}
   ```

   This action adds up the contents of each record's third field (price) multiplied by the record's second field (quantity).

3. When awk reached the end of the file, it performed the action following the END keyword, printing the string "Total" followed by the result of the addition.

From this example, you can see that awk performs actions that process the input information. It also recognizes patterns; for example, if there were items in the shopping list with prices that you aren't sure of, awk could print a list of just those items. We've marked fruit cocktail with a question mark; the following command prints just the items that are marked this way:

```
vizier> awk '-F[TAB]' '$4 ~ /\?/' shop
24 oz   2        .79    fruit cocktail ?
```

The pattern enclosed in the second set of apostrophes examines field $4. The tilde (~) tells awk to see if the field contains text matching the RE enclosed in the slashes, in this case a question mark. (The question mark is preceded with a backslash to prevent awk from treating it as a metacharacter.) If there is a match, awk prints the line. You could accomplish this same task more easily with a grep command.

The strength of awk is its ability to combine pattern recognition with action. For example, you could create an awk program that would total the prices you are unsure of while totaling all the other prices separately to produce an output like this:

```
Items with uncertain prices:
fruit cocktail ?

Total of uncertain prices =  1.58
Total known prices =        31.73
Estimated total cost =      33.31
```

We'll show you the program that created this output in Section 4.2.4.

4.2.2 Printing with awk

As shown in the example in Section 4.2.1, awk uses the print command to print things. You can print all of a record by using the print statement with no arguments. Using a print command like this is the same as including no action in your awk statement. For example:

```
vizier>  awk '{print}' shop
```

This command prints all of the shopping list file.

To print only selected fields, you specify the fields you want to print. You can also mix text in the output as shown by the following example, which prints a list of just the items and the quantity to buy:

```
vizier> awk '-F[TAB]' '{print "Buy", $2, "of", $4}' shop
Buy 2 of frozen peas
Buy 1 of frozen broccoli
  .
  .
  .
Buy 1 of burger buns
Buy 2 of hamburger
```

Note that the fields and text are separated by spaces. There is a predefined variable in awk called OFS that contains the value of the output field separator. The default value of OFS is a single space, but you can change OFS to specify a different output field separator, such as a tab character, to provide for uniform alignment of columnar output. For example:

```
vizier> awk '-F[TAB]' '{OFS = "[TAB]"; print $2, $4}' shop
2       frozen peas
1       frozen broccoli
  .
  .
  .
1       burger buns
2       hamburger
```

Another useful value for the output field separator is the at sign (@); this character is commonly used for separating columns in source files for the tbl table-formatting processor discussed in Chapter 5.

4.2.3 Using Pattern Recognition in awk

The awk utility's pattern-matching ability supports the full range of regular expressions listed in Table 2-1. You must always enclose the pattern being matched in slashes; if your pattern includes a slash or a metacharacter that you want interpreted literally, precede that character with a backslash. For example:

```
/[Cc]olou?r\?/
```

This pattern matches "Color", "Colour", "color", or "colour" when the matching word is followed by a question mark. (The question mark after the "u" is treated not as an ordinary character but as an RE metacharacter calling for zero occurrences or one occurrence of the preceding character.)

The awk utility converts freely between numbers and strings as it sees the need. You can use this feature to match on mathematical expressions. To search the shopping list for all items whose quantities are greater than 1, use this command:

```
vizier> awk '-F[TAB]' '$2 > 1 {print $2, $4}' shop
2 frozen peas
3 apples
2 fruit cocktail ?
2 ice cream
3 milk
2 hamburger
```

You can test expressions using the following set of comparison operators:

==	Equals
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal to
~	Contains RE
!~	Does not contain RE

You can combine tests for more than one expression in a single pattern by using parentheses and the following logical operators:

||	Or
&&	And

For example, the following pattern matches shopping-list items whose quantities are less than 3 and whose prices are greater than $1.00, or that are marked with a question mark:

```
($2 < 3) && ($3 > 1) || ($4 ~ /\?/)
```

4.2.4 Programming awk

An awk statement consists of a pattern and an action. All of the actions we have shown to this point are variations on printing. But you can create very complex actions by using the programming features of awk.

An action can be a single statement or a sequence of statements. You separate statements on a line with semicolons. Among the programming features of awk is the ability to use flow-control structures (if-else, while, and for) to create powerful programs. These structures are implemented for awk exactly the same as they are for the bc calculator, described in Chapter 8.

As patterns and actions become more complex it is often easier to create a file containing the patterns and actions you want awk to work with; this file is called an awk program. The following example shows the program that created the neatly separated totals of uncertain and known prices shown in Section 4.2.1:

```
BEGIN {FS = "[TAB]"      1
print "Items with uncertain prices:"}
{st += ($3 * $2)}          2
{if ($4 ~ /?/) {       3
  sq += ($3 * $2)
  print $4}
else                   4
```

```
  sr += ($3 * $2)}
END {print                    5
printf "Total of uncertain prices = %5.2f\n", sq    6
printf "Total known prices =        %5.2f\n", sr
printf "Estimated total cost =      %5.2f\n", st}
```

The statements in this program perform the following functions:

1 Actions preceded by the `BEGIN` keyword are performed before `awk` processes the file. In this case, a line is printed to identify the lines following it as items whose prices are uncertain. The `FS` variable specifies the input field separator in the same way as the `OFS` variable specifies the output field separator. Setting `FS` has the same effect as using the `-F` command option.

2 This line adds prices to produce the total cost for the entire list. The variable `st` holds the total.

3 The `if` statement and the lines that follow it select items whose prices are indicated as questionable, accumulating the total cost for just those items. The variable `sq` holds this total. Any matching items are printed. Note that the lines controlled by the `if` statement are enclosed in braces to form a single action.

4 The `else` statement and the line following it accumulate the cost for all the items not marked as questionable. This total is stored in the variable `sr`.

5 The `END` keyword identifies actions to be taken after the file processing is complete. In this case, the program prints a blank line and then formats and prints the three cost totals with identifying text.

6 The `printf` command uses the first argument, in quotation marks, as a format string. Percent signs (`%`) introduce formatting controls for variables following the format string, and the remaining text inside the format string is printed as-is. This example formats the accumulated costs so that they will print in a neat five-character column with two digits after the decimal point.

You use the `-f` option for `awk` to tell the command to use your program. For example, if this program is called `add-prices`, you would use this command:

```
vizier> awk -f add-prices shop
Items with uncertain prices:
fruit cocktail ?

Total of uncertain prices =  1.58
Total known prices =        31.73
Estimated total cost =      33.31
```

Note that this example does not need the `-F` option because the `add-prices` program already specifies that the input field separator (`FS`) is to be a tab character.

The tbl Table Creation Utility 5

Tables are an effective way to present certain kinds of information in documents. This chapter discusses how to use the `tbl` preprocessor to create tables for documents that will be formatted by the `nroff` text formatter. You can use this tool to create simple tables that look like lists, as well as complex formal tables.

The commands and functions for `tbl` that are described here also work with special typesetting text formatters; this book was typeset with one such formatter, part of Digital's optional ULTRIX Documentation Tools product.

This chapter assumes that you are familiar with the `nroff` formatter. If you are not, you should read the chapter in the *Primer* that discusses `nroff`.

5.1 Why Use tbl?

A table is a collection of information presented as a multicolumn list. Usually, but not always, the first column contains a list of items that are described or explained by other columns in the table. The following table is used for the examples in Chapter 4 of this book.

```
10 oz    2        .69    frozen peas
10 oz    1        .89    frozen broccoli
doz      1       2.25    fresh corn
lb       1        .45    rice
lb       3       1.89    apples
24 oz    2        .79    fruit cocktail ?
1/2 gal  2       1.89    ice cream
gal      3       2.39    milk
4 oz     1        .89    Fruit Twirls
16 oz    1       2.59    Tasty Squares
lb       1       1.89    rye bread
12 ct    1       1.19    burger buns
lb       2       1.79    hamburger
```

This table was created without the `tbl` preprocessor. It is just a collection of text lines. For quick notes that you scratch out for yourself, this method is adequate. But as tables become more complex or as you create documents that you will change many times, it is easier to use `tbl`, which automatically takes care of establishing the proper columns, drawing lines or boxes, and allocating the proper space. If your table contains descriptions that extend over several lines in one column as shown in Example 5-1, using `tbl` saves a great deal of time. (The code that produced Example 5-1 is shown at the end of this chapter as Example 5-10.) With the exception of the shopping list shown at the beginning of this chapter and Chapter 4, all the tables in this book, even the simple lists of mathematical and relational operators in several chapters, were created with the `tbl` preprocessor.

Example 5-1: Table with Multiline Entries

Feature	Problem in Manual Formatting	Benefit with tbl Formatting
Simplicity	User must keep track of column alignment to make attractive display.	Columns are aligned by the formatter.
Long blocks of text	User must align each line within its column, then go on to the next column.	Formatter automatically calculates how much fits on a line and breaks text blocks for you.
Underlining	Each underlined section must be coded manually.	Formatter can underline selected columns automatically.

We have used the word "preprocessor" several times in this discussion. What is a preprocessor, and what's it good for? Part of the UNIX philosophy is the idea of using several simple tools, each designed to be very good at its job, instead of one massive tool that can do everything but perhaps not very well. The `nroff` text formatter is an example of this philosophy. It is good at formatting text, but it does not know how to format tables. You could give it all the commands to make it format a table, but they are very complex and confusing.

A preprocessor is a program that interprets information destined for another processor. It provides an intermediate step of processing in order to simplify the job you must do. The `tbl` preprocessor is another example of the UNIX philosophy of one-job tools. It knows how to create formatting commands to make `nroff` produce a table by translating a special set of table-formatting commands that you put in your file.

To format a document containing tables, you process your file with the `tbl` preprocessor and pipe the output to `nroff`. For entries that are more than one line long, the `nroff` output has the first column's text followed by the second column's text, and so on, as in the following illustration from the file that created Example 5-1:

```
Simplicity
^[7
              User must keep track of
              column alignment to make at-
              tractive display.
^[7^[7^[7
                                          Columns are aligned by the
                                          formatter.
```

The odd-looking strings (`^[7`) tell a printer how far to back up after printing the first column to print the next column. But because most line printers cannot move their paper backward, there is another tool, the `col` postprocessor, that reformats `nroff` output by storing the first column's information until it has gathered the rest of what should be printed on the same line. Once `col` knows what each complete line should look like, then it outputs the line. The following example shows how you would process a file with tables and print the result on the default printer:

```
vizier> tbl file | nroff | col | lpr
```

5.2 Creating Tables

Creating a table involves three steps:

- Setting off the table information
- Defining the table format
- Entering the table information

Example 5-2 shows the code for a simple table; the following sections describe the steps in creating a table.

Example 5-2: Code for a Simple Table

```
.TS
center,tab(@);
l l l r.
Item@Size@Qty@Price
frozen peas@10 oz@2@.69
frozen broccoli@10 oz@1@.89
fresh corn@doz@1@2.25
.TE
```

This example produces the following result:

```
Item             Size    Qty   Price
frozen peas      10 oz   2       .69
frozen broccoli  10 oz   1       .89
fresh corn       doz     1      2.25
```

5.2.1 Setting Off the Table Information

You set off your table information by enclosing it between `.TS` (table start) and `.TE` (table end) commands. These are commands that `nroff` does not recognize. The `tbl` preprocessor looks for them, and it formats only the material between matched sets of `.TS` and `.TE` commands. Also, as `nroff` is formatting the file, the `ms` macro package invokes some special macros that allow you to do things beyond the normal functions, such as these:

- Create a table header so that if your table spans more than one page the header will appear on each page.
- Create boxed sections of text, like this:

```
|This text is in a box.  You can enclose as much text as  you |
|want in this way.                                            |
```

5.2.2 Defining the Table Format

Defining the table format consists of two different tasks, specifying `tbl` options and specifying the columns of the table.

5.2.2.1 Specifying tbl Options – The first thing in a table is a line containing a comma-separated list of `tbl` options and terminated with a semicolon. For example:

```
center,tab(@);
```

The first option in this example says to center the table between the margins. You can also specify `expand`, which makes the table span the entire length of a line. If you don't specify any placement, your table will be placed flush with the left margin.

The second option, `tab(@)`, defines the character that you will use to separate the information for one column from that for the next. This character is referred to as the "tab character," but if you use a real tab character your files can be a little confusing to work with, especially if there are places where a given column is blank, because tabs are invisible on the terminal display. You can specify whatever character you like; many ULTRIX users use the at sign (`@`) because it is used for little else in most documents.

By using other options, you can specify that your table is to be boxed. Example 5-3 shows two versions of the same small table; the first was created with the `box` option and the second with `allbox`, which boxes each object in the table separately.

Example 5-3: Boxed Tables

```
| Item             Size    Qty    Price|
| frozen peas      10 oz   2        .69|
| frozen broccoli  10 oz   1        .89|
| fresh corn       doz     1       2.25|
```

```
| Item           | Size  | Qty | Price|
| frozen peas    | 10 oz | 2   |   .69|
|                |       |     |      |
| frozen broccoli| 10 oz | 1   |   .89|
| fresh corn     | doz   | 1   |  2.25|
|                |       |     |      |
```

(These tables are oddly boxed because `nroff`'s idea of vertical spacing is not entirely consistent. We'll show you how to deal with this inconsistency in Section 5.3.4.3.) The first of these tables was produced with the following options:

```
center,box,tab(@);
```

The second table was produced with these options:

```
center,allbox,tab(@);
```

If you specify neither `box` nor `allbox`, your table will be printed with no boxing at all, as in Example 5-2.

5.2.2.2 Specifying the Table Columns – After the `tbl` options, you specify the number and alignment of the table's columns, or **fields.** Each field can be specified as `l`, `c`, `r`, or `n`, for left, center, right, or numerical alignment. The tables in Example 5-3 are both specified with four fields, and the right field is right-aligned. There is one line of specification for each of these tables:

```
l l l r.
```

Each specification line controls one line of the table. But if there are more lines in the table than there are specifications, the last specification controls all the way to the

end of the table. The last specification line is terminated with a period to tell `tbl` that the actual table information begins on the next input line.

You can create other attractive effects with special characters that `tbl` understands. (See Examples 5-4 and 5-5 for an illustration of these effects and how to achieve them.)

- Column separation

  You can specify that columns are to be separated by a vertical bar (|) by including a bar between the field descriptions in your specification lines. For example:

  ```
  l | l | l.
  ```

  This line produces a three-column table with bars between the columns.

- Spanned headings

  You can specify that a heading is to span multiple columns by using the letter `s` in your specification lines for fields into which the header can span. For example:

  ```
  c s s
  ```

  You can use this example to produce a three-column table with a single centered heading that spans all three columns.

5.2.3 Entering the Table Information

Once you have specified the table's format, you enter the information. Each line of the table is represented by one line in your source file. You separate the fields with the tab character you selected in the table specification. For example, the table lines in Examples 5-2 and 5-3 were created this way:

```
Item@Size@Qty@Price
frozen peas@10 oz@2@.69
frozen broccoli@10 oz@1@.89
fresh corn@doz@1@2.25
```

You can create additional effects with special characters that `tbl` understands. (See Examples 5-4 and 5-5 for an illustration of these effects and how to achieve them.)

- Table-width rules

  If you include a line containing nothing but an underscore, `tbl` will create a horizontal rule all the way across your table. The rules in Example 5-1 were created in this way. For example:

  ```
  Column-1 Header@Column-2 Header

  _
  Column-1 entry@Column-2 entry
  ```

- Column-width rules

  If you create a table entry that has an underscore all by itself in a particular field, `tbl` will produce a rule just the width of that field. For example:

  ```
  Column-1 entry@_@Column-3 entry
  ```

As shown by the table in Example 5-1, you can include more text than will fit on a single table line by using a **text diversion**. Text diversions are blocks of text that

tbl stores as it reads them, holding the stored text until the end of the block is found. Then, tbl knows how much text must be output, and it calculates the proper arrangement before outputting the text.

You create a text diversion by placing the letter T and a left brace (T{) at the end of the line before the desired text, and the letter T and a right brace (T}) at the beginning of the line following the text. The table in Example 5-1 was created with text diversions; the following example shows a portion of that table's source file.

```
Simplicity@T{
User must keep track of column alignment to make
attractive display.
T}@T{
Columns are aligned by the formatter.
T}
```

In this example, the material for the middle column is enclosed in one text diversion and the last column's material is in another diversion. The tbl preprocessor reads and saves each diversion until it knows how to format the entire table entry; then it outputs each diversion's text.

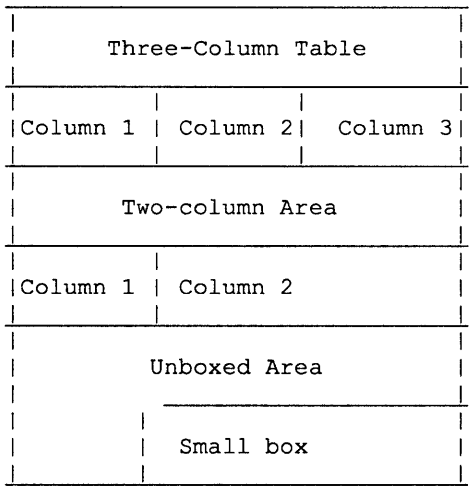
5.3 Advanced Techniques

This section explores some of the things you can do to make your tables more useful.

5.3.1 Combining Effects

You can combine the techniques and effects shown earlier to produce quite sophisticated tables like the compound table in Example 5-4. The code for this table is given in Example 5-5.

Example 5-4: Compound Table



Example 5-5: Code for the Compound Table

```
.TS
box,tab(@);
c s s
l | l | l
l | l | l
c s s
l | l s
l | l s
c s s
l | l s.
.sp
Three-Column Table
.sp
_
.sp
Column 1@Column 2@Column 3
_@_@_
.sp
Two-column Area
.sp
_
.sp
Column 1@Column 2
_@_
.sp
Unboxed Area
.sp
@_
.sp
@Small box
.TE
```

The actual combination of commands and characters you must use to create the effect you desire is not always obvious; you might need to format the table and then adjust its control information to achieve the final appearance you want.

5.3.2 Creating Multipage Tables

If your table is longer than a single page, or if it happens to fall across a page break in the document, you can improve its appearance by specifying that a complete header is to be printed at the top of each new page. You do this by using a special form of the `.TS` macro together with the `.TH` macro.

To start the header for a multipage table, add the letter `H` as an argument to the `.TS` macro. The `H` tells `nroff` to begin assembling text in a temporary buffer. To end the header, use the `.TH` macro. Example 5-6 shows the code for a table heading using these macros.

Example 5-6: Code for Multipage Headings in a Table

```
.TS H
tab(@);
l l.
_
Column-1 Heading@Column-2 Heading
_
.TH
First column-1 entry@First column-2 entry
Second column-1 entry@Second column-2 entry
```

Example 5-6: (continued)

```
Third column-1 entry@Third column-2 entry
   .
   .
   .
```

When you use this multipage capability, you must format your document using the `ms` macros. For example:

```
vizier> tbl file | nroff -ms | col | lpr
```

5.3.3 Creating Boxed Text Blocks

When you format a document with the `ms` macros, you can take advantage of `tbl`'s text-formatting features to create boxed text that is not part of a table. The sample boxed text segment in Section 5.2.1 was created in this way. To box text, precede the desired information with the `.B1` macro and follow it with the `.B2` macro. For example:

```
.B1
This text is in a box.  You can enclose as much text
as you want in this way.
.B2
```

This example produces the following result:

```
|This text is in a box.  You can enclose as much text as  you |
|want in this way.                                            |
```

5.3.4 Adding the Final Touch

Sometimes the spacing of your table is not as attractive as you would like. Vertical problems can happen in any table; horizontal problems occur most often in tables containing text diversions. You can alter the spacing by using blank columns, by specifying field widths, or by using `.sp` commands.

5.3.4.1 Using Blank Columns – You can insert blank columns to serve as placeholders. The table in Example 5-1 has blank columns two characters wide inserted between its three visible columns in order to provide space between the text blocks in adjacent columns. Without these blank columns, the table entries would have been squeezed together until the words of adjacent columns ran together.

5.3.4.2 Specifying Column Widths – When you use text diversions, often they are not placed in columns that produce attractive or easily readable output. Column width can also be unsatisfactory under other circumstances.

When your columns don't work out properly to produce an attractive table, you can alter the width of any field or fields in the table. You do this by using the `w` modifier for your field headings. Suppose you have a table that you have specified using the `expand` option. If there are text diversions, your table might come out looking like Example 5-7.

Example 5-7: Table with a Text Diversion

Column 1	Column 2
Entry 1	This is text describing Entry 1. It is long enough that it requires a text diversion.

Example 5-8 shows the code that produced Example 5-7.

Example 5-8: Code for the Table with a Text Diversion

```
.TS
expand,tab(@);
l l.
_
.sp
Column 1@Column 2
.sp
_
.sp
Entry 1@T{
This is text describing Entry 1.  It is long enough that it
requires a text diversion.
T}
_
.TE
```

To change the width of the second column, modify the field specification line like this:

```
l lw(50).
```

The w modifier requires a width value that is enclosed by parentheses. This example specifies 50 characters. The result of this change is as follows:

Column 1	Column 2
Entry 1	This is text describing Entry 1. It is long enough that it requires a text diversion.

This second table is far more attracive and readable than the first. You might notice that the text-diversion table shown here is justified so that its text aligns with the right margin, whereas Example 5-1 is not justified in this way. This difference results from the use of the `.ad` command for `nroff`. You can use `.ad` and other `nroff` commands to affect the appearance of your tables; see the ULTRIX *Supplementary Documents, Volume 1: General User* for a complete discussion of both `tbl` and `nroff`.

5.3.4.3 Handling Vertical Spacing Problems – The odd vertical spacing that `nroff` creates in some tables is due to the fact that `nroff` works in terms of 1/240-inch units of vertical spacing; the location of a printed line does not always translate to an integral number of units from the top of the page, and `nroff` sometimes rounds off to the same line instead of the next one. You can insert `.sp` commands or table lines with blank entries in your input file to cause more attractive spacing. Example 5-9 illustrates how the appearance of the `allbox` table in Example 5-3 can be improved by the addition of the two `.sp` commands shown in bold type.

Example 5-9: Improved Spacing in allbox Table

```
.TS
center,allbox,tab(@);
l l l r.
Item@Size@Qty@Price
.sp
frozen peas@10 oz@2@.69
frozen broccoli@10 oz@1@.89
.sp
fresh corn@doz@1@2.25
.TE
```

Item	Size	Qty	Price
frozen peas	10 oz	2	.69
frozen broccoli	10 oz	1	.89
fresh corn	doz	1	2.25

5.4 Example tbl Code

The code that produced Example 5-1 is shown here as Example 5-10.

Example 5-10: Code for the Table Shown in Example 5-1

```
.ll 72
.ad l
.TS
expand,tab(@);
l l l l l.
_
.sp
Feature@  @Problem in Manual Formatting@  @Benefit with tbl Formatting
.sp
_
.sp
Simplicity@  @T{
User must keep track of column alignment to make attractive display.
T}@  @T{
Columns are aligned by the formatter.
T}
.sp
T{
Long blocks of text
T}@  @T{
User must align each line within its column, then go on to the next
```

Example 5-10: (continued)

```
column.
T}@  @T{
Formatter automatically calculates how much fits on a line and breaks
text blocks for you.
T}
.sp
Underlining@  @T{
Each underlined section must be coded manually.
T}@  @T{
Formatter can underline selected columns automatically.
T}

_
.TE
```

The `.ll` command in this example tells `nroff` to use a line length of 72 characters; the `.ad` command says to justify to the left margin only.

Part II: Communication with Other Users

Mail 6

This chapter discusses advanced features of ULTRIX mail. If you are not familiar with the `mail` program, you should read the *Primer* chapters on mail and on customizing your environment.

There are several mail-handling systems available on the ULTRIX system in addition to `mail`. This chapter gives a brief discussion of one of these systems, called MH.

For examples in this chapter, we will use the login name `hale` to indicate the user.

6.1 Where Is My Mail?

Your mail is kept in any of several places. New and unread mail, or mail that you have explicitly kept there, is in your system mailbox. On most systems, your system mailbox is a file in the `/usr/spool/mail` directory, but this may not always be true.

Unless you have deleted it or saved it elsewhere, mail that you have read is in your `mbox` file, located in your home directory. If you are using folders, then mail that you have read and saved is in the folders you have specified. You can also specify an explicit file for saving mail, so you could have many scattered files containing mail messages.

6.2 Using the Mail System

You start the mail program with the `mail` command. If you just want to send a message, you enter the name or alias to which you want to send a message as an argument to the command. For example:

```
vizier> mail evelyn
```

Depending on the mail system options in force, you might be prompted for the various parts of your message. You can change these options to suit yourself; see Section 6.5.

If you want to read mail, you enter the `mail` command with no argument. If there is no mail for you, the system tells you so. The *Primer* explained how to read mail in your `mbox` file or in folders.

The `mail` command has several command-line options that let you alter its behavior. Table 6-1 lists the most useful command-line options. For a complete list, see the `mail`(1) reference page.

Table 6-1: Command-Line Options for the mail Program

Option	Description
`-f [`*folder*`]`	Reads `mbox` or the folder you specify.
`-i`	Ignores CTRL/C interrupts, echoing them as at signs (`@`).
`-n`	Inhibits reading `/usr/lib/Mail.rc`, a systemwide file that specifies `mail` options. See Section 6.4.
`-s "`*subject*`"`	Specifies the subject text on the `mail` command line. Enclose the entire subject in quotation marks. For example: `vizier> mail -s "Meeting tomorrow" evelyn`

6.3 Commands for the Mail Program

The `mail` program has a large set of commands. You are already familiar with most of the commands for sending, reading, deleting, and saving mail. The additional commands provide facilities for manipulating your mail environment. Table 6-2 describes most of the `mail` commands. The `mail`(1) reference page lists some other commands that are useful only under special circumstances.

Table 6-2: Commands for the mail Program

Command	Description
`!`*command*	Executes the shell command you enter.
`-[`*n*`]`	Selects and displays the previous message or the *n*th previous message. For example, `-4` backs up four messages.

`alias`
`alias` *alias*
`alias` *alias name...*
`g`
	With no arguments, lists the current aliases. With one argument, displays only that alias. With two or more arguments, creates an alias with the first argument as its name and all subsequent arguments as the members of the alias. For example:
	`& alias`
	`eddie haskell@beaver`
	`eve evelyn`
	`group daniels johnson janacek pinkham`
	`service operators@muezzin`
	`& alias eve`
	`eve evelyn`
	`& alias manager daniels`
	`& alias manager`
	`manager daniels`
	The `g` command is an alternate for `alias` with no arguments.

`chdir` *path* `ch` *path*	Changes your current directory to the pathname specified, as if you had executed the `cd` shell command except that the directory you specify with `chdir` prevails only while you are in the mail environment.

Table 6-2: (continued)

Command	Description
`copy` [*message...*] *file* `co` [*message...*] *file*	Copies the current message or the specified messages into a file. If *file* exists, the messages are appended. This command works like `save` except that it does not mark copied messages for deletion when you quit from `mail`.
`delete` [*message...*] `d` [*message...*]	Deletes the current message or the specified messages. You can use the `undelete` command to recover messages you have accidentally deleted.
`dp` `dt`	Deletes the current message and prints the next active message.

`exit`
`ex`
`x`
	Exits mail without updating your system mailbox.

`file` [*file*] `fi` [*file*] `folder` [*file*] `fo` [*file*]	Selects a mail file or folder. If you do not specify a file, this command prints your current path and file name and the number of messages in your current file. If you specify a file or folder, this command displays any changes you have made to your current file and switches to the specified file for reading.
`folders`	Lists the names of the folders in your folder directory.
`from` [*login*] `f` [*login*]	Prints the active message header. If you specify a login name, this command prints all the active messages from the specified name.
`headers` [*n*] `h` [*n*]	Lists active message headers, using the value of the `screen` variable as the number of headers to display. See Table 6-4 for a description of the `screen` variable. If you have more than one screenful of messages, you can move forward or backward one screenful with the `z` command. If you specify a message number, the `headers` command displays the screenful that includes the specified message.
`help`	Displays help information.
`hold` [*message...*] `ho` [*message...*] `preserve` [*message...*] `pre` [*message...*]	Holds, or preserves, the current message or the specified specified messages in your system mailbox instead of moving them to your `mbox` file.
`ignore` [*field...*]	Sets `mail` to display messages without the specified fields of the header when you use the `print` or `type` command. For example: `ignore Status Received Message-id` Note that this command is different from the `ignore` variable described in Table 6-4. If you enter the `ignore` command with no arguments, it displays the current list of ignored fields.
`mail` *user...* `m` *user...*	Sends a message.
`mbox` [*message...*]	Marks the current message or the specified messages to be moved to your `mbox` file. This is helpful if you have set the `hold` variable in your `.mailrc` file.

Table 6-2: (continued)

Command	Description

`next`
`n`
`+`
RETURN
	Displays the next message.

`Print` [*message*] `P` [*message*] `Type` [*message*] `T` [*message*]	Displays the current message or the specified message, including any header fields specified by the `ignore` command.
`print` [*message*] `p` [*message*] `type` [*message*] `t` [*message*]	Displays the current message or the specified message without any header fields specified by the `ignore` command.
`quit` `q`	Leaves the `mail` program and updates your system mailbox. If you do not have the `hold` variable set, all messages that you have not deleted, saved, or preserved are moved to your `mbox` file. If you do have `hold` set, all these messages will be left in your system mailbox and marked as having been read.
`Reply` `R`	Replies to a message. If the original message was addressed to a group of people, replies sent with the `Reply` command are sent only to the originator of the message.

`reply`
`r`
`respond`
	Replies to a message. If the original message was addressed to a group of people, replies sent with the `reply` and `respond` commands are sent to everyone who received the original message.

`save` [*message...*] *file* `s` [*message...*] *file*	Saves the current message or the specified messages in the file. Note that the messages are added to the specified file so that you will not delete the contents of the file.
`set` [*variable*] `se` [*variable*]	If entered with no variables, the `set` command displays all the options you have set. If you specify a variable, the option will be set. (Table 6-4 lists the available variables.)
`shell` `sh`	Invokes the shell interactively.
`source` *file* `so` *file*	Reads mail commands from a file (usually `.mailrc`).
`top` [*message...*] `to` [*message...*]	Displays the first five lines in the current message or each of the specified messages.
`undelete` *message...* `u` *message...*	Undeletes the specified messages.
`unset`	Unsets (turns off) options. For example, if your `.mailrc` file includes a `set hold` command, you can use the `unset` command to disable the `hold` variable for the current `mail` session.
`visual`	Invokes the editor specified by the `VISUAL` mail variable to edit the current message.

Table 6-2: (continued)

Command	Description
`write` [*message...*] *file* `w` [*message...*] *file*	Saves the current message or the specified messages in the named file. This is similiar to the `save` command, except that `write` saves only the body of each message; the headers are deleted.
`z[+]` `z-`	Moves forward or backward one screenful of messages. You can specify the number of messages in a screenful with the `screen` variable. (See Table 6-4.) To move forward one screenful, enter `z` or `z+`; to move backward, enter `z-`.

6.4 Escape Commands for Mail Messages

There is a special set of commands, called **escape commands** or **escapes,** that perform functions while you are in the process of writing a message.

You use an escape by entering it as the first thing on a line, with a tilde (~) as the very first character. The tilde is called an escape character because it signals `mail` that an escape command follows. If you want to type a real tilde as the very first character on a line in your message, you must type two tildes.

Table 6-3 describes the escape commands.

Table 6-3: Escape Commands in mail

Command	Description
`~!`*command*	Executes the shell *command* you enter.
`~?`	Prints a brief summary of escape commands.
`~:`*command*	Executes the specified mail command. This is useful for performing housekeeping tasks such as redisplaying a message. For example, entering `~:10` selects and displays message number 10 just as if you had entered its number at the `&` mail prompt.
`~c` *name . . .*	Adds the specified names to the `Cc:` list.
`~d`	Includes the file named `dead.letter`, in your home directory, into the message.
`~e`	Invokes the editor specified by the `EDITOR` mail variable to edit the message.
`~f` [*message...*]	Reads the current message or the specified messages into your message.
`~h`	Edits the message header fields. This command displays the fields one at a time so you can alter them by adding text to the end, by using the DELETE key, or by using CTRL/U to erase the entire field and then retyping it. Use this command with caution.
`~m` [*message...*]	Includes the current message or the specified messages, shifted one tab stop to the right. This is useful to set off messages your are forwarding as part of your new message.

Table 6-3: (continued)

Command	Description
~p	Displays the message you are composing on your terminal. This is useful to see that the message looks the way you want it to and that it includes the right subject heading and lists of recipients.
~q	Aborts the current message as if you had pressed two CTRL/C interrupts.
~r *file*	Includes the named file in your message.
~s *subject*	Makes *subject* be the new subject heading, replacing the previous heading.
~t *name...*	Adds the names to the `To:` list of your message.
~v	Invokes the editor specified by the `VISUAL` mail variable to edit the message.
~w *file*	Writes the message to the named file.

~| *command*	Pipes the message through the named command. This is useful to make global changes in the message; for example, if you are including a message in your new message you can use the `sed` editor to prefix each line with an angle bracket and a space by using the following command:
	`~|sed 's/^/> /'`
	You can then add your own text; the result will look like this:
	`> This is the text of the message`
	`> you have included.`
	`>`
	`This is the text you add yourself.`

6.5 Customizing the mail Program

The `mail` program provides options that allow you to customize the way it responds to you. For example, by using the shell command `biff`, you can have the system notify you immediately when new mail arrives or wait until the next time you receive a shell prompt. Entering `biff y` enables asynchronous notification, and entering `biff n` disables it. Entering `biff` with no arguments reports the current setting. For example:

```
vizier> biff
is n
vizier> biff y
```

Most of the things you can do to customize your interaction with the `mail` program are controlled by mail variables, or options, that you set in your `.mailrc` file. There is another options file for the `mail` program, called `/usr/lib/Mail.rc`. Your system administrator decides what options to set in `Mail.rc` so that those options will be set for all users. You can override the settings in `Mail.rc` by placing counteracting commands in your own `.mailrc` file. A typical `Mail.rc` file looks like this:

```
set append dot save ask askcc save SHELL=/bin/csh \
EDITOR=/usr/ucb/ex metoo hold
```

(The backslash at the end of this example's first line "hides" the new-line character at the end of the line, so that `mail` will read the second line as if it were a continuation of the first.)

To override a variable that is set by `Mail.rc`, include an `unset` *variable* command in your `.mailrc` file. For example:

```
unset dot
```

Some of the `mail` variables, such as `dot`, are binary variables; they are either set or unset. Others are either string or numeric variables; they have values associated with them. For example, `crt` is a numeric variable that tells `mail` how many lines of a message to display before pausing with the `--More--` prompt, and `folder` is a string variable that tells `mail` what mail folder you are reading.

Other commands that can be useful in your `.mailrc` file are `alias` commands to specify frequently used names, and the `ignore` command to specify header fields that you don't want to see.

The following is a typical `.mailrc` file:

```
set ask
set hold
set crt 20
set askcc
set save
set SHELL=/bin/csh
set EDITOR=/usr/ucb/vi
set metoo
alias group daniels johnson janacek pinkham
alias eve evelyn
ignore Status Received Message-id
```

You can also respecify any of `mail`'s options interactively when you are using the `mail` program. Settings you make in this way prevail only until you leave `mail` with an `exit` or `quit` command. To make your changes permanent, include your desired settings in your `.mailrc` file. Table 6-4 describes the mail options.

Table 6-4: Variables for Customizing the mail Program

Variable	Type	Description
`append`	Binary	Saves messages in your `mbox` file in the order of arrival; the earliest message is the first message in the file. When this variable is unset, messages are saved in reverse order; the first message in the file is the most recent. The `mail` program runs faster if `append` is set.
`ask`	Binary	Prompts you for a subject line when you send a message. Enter a blank line to send a message with no subject.
`askcc`	Binary	Prompts you for carbon-copy recipients for each message you send.
`autoprint`	Binary	Automatically displays the next message when you delete the current message. When `autoprint` is unset, `mail` does not display the next message when you delete a message. In either case, the next message becomes your new current message.

Table 6-4: (continued)

Variable	Type	Description
crt	Numeric	For use with a video display (CRT) terminal. Reads your mail one screenful at a time using the more program. The value tells mail how many lines to display each time. For example: set crt 20
debug	Binary	Displays debugging information.
dot	Binary	Interprets a period on a line by itself to be the end of a message. Do not unset dot and also set ignoreeof.
EDITOR	String	Specifies the pathname for the text editor to be used when you use the edit command or the ~e escape. For example: set EDITOR=/usr/ucb/ex If your terminal is a CRT terminal, you can specify a screen editor for this variable. See the VISUAL variable later in this table.
escape	String	Allows you to specify the escape character (the character that starts an escape command when you are in the middle of writing a message). The default is the tilde (~). You must specify a single character.
folder	String	Specifies the directory for storing mail folders. A name beginning with a slash, such as /usr/users/hale, is a absolute pathname. A name without an initial slash is a pathname relative to your home directory. For example, the command set folder=folder indicates the directory /usr/users/hale/folder.
hold	Binary	Prevents messages from being moved to your mbox file after you read them. Messages you have read are held in your system mailbox.
ignore	Binary	Ignores CTRL/C interrupts, echoing them as at signs (@). Note that this variable is different from the ignore command described in Table 6-2.
ignoreeof	Binary	Ignores CTRL/D as the end of an outgoing message. Do not set ignoreeof and also unset dot.
keep	Binary	Allows mail to truncate your system mailbox instead of deleting it when it is empty. This is useful if you have set special permissions on your system mailbox for security reasons. If keep is unset, your system mailbox is deleted when it becomes empty; the next time it is created, you must reestablish your desired permissions.
keepsave	Binary	Prevents deletion of saved messages when you quit mail. Normally, the mail program marks messages when you save them in other files or folders, and then deletes them from your system mailbox when you leave mail. Setting keepsave makes mail leave these messages in your system mailbox.

Table 6-4: (continued)

Variable	Type	Description
metoo	Binary	Includes you in the list of recipients when you send mail to an alias of which you are a member. If metoo is unset, you will not receive copies of messages sent to aliases of which you are a member.
msgprompt	Binary	Prompts you for the text of an outgoing message and indicates how to end the message.
noheader	Binary	Inhibits display of the header and version identification when you invoke mail.
nosave	Binary	Prevents mail from saving aborted messages as dead.letter in your home directory.
quiet	Binary	Supresses printing the version when first invoked and the message number when you use the type command.
record	String	Specifies the name of a file into which mail will save copies of all outgoing messages.
SHELL	String	Specifies the pathname of the shell to use when you use the ! command and the ~! escape.
screen	Numeric	Specifies the number of messages to be displayed in one screenful when you enter the headers command.
sendmail	String	Specifies the pathname of the program to use to send your mail messages. If this variable is not specified, mail uses the default delivery system. See your system administrator for information about alternate delivery systems.
toplines	Numeric	Specifies the number of lines the top command prints. The default is 5.
verbose	Binary	Invokes mail in verbose mode; mail then announces expansion of aliases as messages are sent. For example: & **set verbose** & **mail eve** Subject: **Meeting this afternoon** Enter message. Use <ctrl>D to terminate the letter. **Just a reminder, we're meeting at 2.** CTRL/D Cc: /usr/users/evelyn/.forward: line 0: evelyn... forwarding to evelyn@vizier evelyn... Connecting to .local... evelyn... Sent
VISUAL	String	Specifies the pathname for the screen editor that will be used when you use the visual command or the ~v escape. For example: set VISUAL=/usr/ucb/vi If your only terminal is a CRT, you can specify a screen editor for the EDITOR variable, too; then either edit (~e) or visual (~v) will invoke the same editor.

6.6 Getting Notification of Mail at Login Time

On most systems, when you log in you are notified if you have mail. The system does not check to see whether the mail is new since the last time you logged in; but if there is anything in your system mailbox the system displays this message:

```
You have mail.
```

If you are using the C shell, you must tell the shell the location of your system mailbox to receive this notification. Usually your system administrator will have included a line like this in your `.cshrc` or `.login` file:

```
set mail=/usr/spool/mail/hale
```

If you are not being notified that you have mail, check these two files to see if the line is there. If it isn't, you can add it, substituting your own login name for `hale`. If you are being notified but don't want to be, you can find this line and delete it.

6.7 Sending Mail to Files

You are not limited to sending mail to other people. You can also send mail directly to a file. Sending mail directly to a file is one way to send yourself carbon copies as if you had the `record` variable set, except that you can direct the message to any file you like. If you send mail to a name that has a slash (/) embedded in it or to a name that begins with a plus sign (+), the mail program understands this to be a file name. To send mail to a file in your current directory, precede the file name with a period and a slash (./). For example, the following command sends mail to the file `notes` in your current directory as well as to another user:

```
vizier> mail ./notes daniels
```

If you send mail to a new file, the file is created. If you send mail to a file that exists already, the message is appended to the existing file. If you send mail to a file beginning with a plus sign (+), the mail system assumes that the file is a folder.

You can also include file names in an alias. For example, the following command creates an alias for `project-team`:

```
alias project-team john evelyn /usr/users/project-team/mail
```

An alias like this saves the members of the project team from having to save mail sent to the team; they will know that it is available for review in the `project-team/mail` file, and they can read that file with this command:

```
vizier> mail -f /usr/users/project-team/mail
```

6.8 Sending Mail Across Networks

In the *Primer*, you were introduced to the concept of sending mail across a network so that you can send messages to people working on other systems. Sending mail across a network is like sending mail to other users on your own system except that the addressing is different. There are two commonly used network addressing schemes; when you send mail across a network, you must use the right one:

- UUCP addressing
- Internet addressing

6.8.1 UUCP Addressing

The term UUCP stands for UNIX-to-UNIX Copy Program. The UUCP protocol has existed almost since the first UNIX systems were built. It allows ULTRIX systems to communicate with other UNIX systems over ordinary telephone lines. A UUCP address consists of the system name, an exclamation point, and the user's login name, like this:

```
aladdin!joan
```

This is the address for a user named `joan`, who works on a system called `aladdin`.

Because UUCP communication uses ordinary telephone lines, UUCP systems must know the telephone numbers of the systems they want to communicate with. It is not reasonable to keep a list of perhaps 30,000 other computers' phone numbers (most of which might be called once or twice a year), so the UUCP protocol allows systems to share the information. To send a message to a system whose number it does not know, your system can send it to a system that does know the destination system's number. For example, suppose you want to sent a message to a user named `arnold` on a system called `minaret`. Your system, `vizier`, does not know `minaret`'s phone number but it does know the number of `aladdin`, and `aladdin` knows the number for `minaret`. You can send mail to `aladdin`, asking that your mail be sent onward to `minaret`, by including `aladdin` in the address like this:

```
& m aladdin!minaret!arnold
```

You can include as many systems in the path as you need to get your message where it must go. This addressing scheme is called **explicit routing** because you must explicitly specify each system along the entire route your message will travel.

6.8.2 Internet Addressing

Internet addressing uses **implicit routing.** The Internet addressing scheme divides all the possible addresses into domains, each consisting of one or more networks. Internet systems know how to contact domains other than the one they reside in. This routing information includes the paths to use in contacting other domains' systems directly; there is no need to specify each system in the route by name. Internet addressing places the user's login name first, followed by an at sign (`@`) and then the system name and domain information. An Internet address might look like this:

```
arnold@kaaba.BLIVIT.COM
```

The parts of this address, called **fields,** are as follows:

Field	Description
`arnold`	This is the user's login name.
`kaaba`	This is `arnold`'s system name. It is separated from the user's name by an at sign (`@`).
`BLIVIT`	This is the domain of which `kaaba` is a part.
`COM`	This is the type of domain that `BLIVIT` is. There are commercial domains (`COM`), educational domains (`EDU`), military domains (`MIL`), and others.

Note that the parts of the domain information are separated by periods.

Some domains contain subnetworks that are "hidden" behind particular machines in the domain's network. You can send mail to hidden systems, provided you know their addresses, by including the hidden system's name in the address with a percent sign (%), like this:

```
& m arnold%aladdin@kaaba.BLIVIT.COM
```

6.9 The MH Message-Handling System

Having learned almost all there is to know about the `mail` program, you might come to the conclusion that you would like to try a different system for handling your mail. One such system is MH. Instead of being a single program that executes all the functions to handle mail, the MH system is a series of small programs. You use MH by entering the command you want to execute while you are at the shell prompt.

The MH system is optional; it may not be installed on your system. To find if MH is available, look for the `/usr/new/mh` directory. To use MH, you must add that directory to your path by editing the `set path` line in your `.cshrc` or `.login` file. Then you must tell the shell about the change in your path; you can do this by logging out and logging back in, or by entering the following command:

```
vizier> source .login
```

If your path is set in `.cshrc`, use that name instead of `.login` in this command. See Section 9.7 for a description of the `source` command.

Remember as you read this discussion that each of the MH commands is a separate system command and has its own reference page. To remind you, we will introduce the command names using the reference page naming convention of `command`(number).

The MH system uses folders, as does `mail`, but the folders are organized a little differently. New and unread mail is kept in a folder called `+inbox`, into which you move the mail that arrives in your system mailbox by using the `inc`(1mh) command. You must enter the `inc` command every time you want to include new mail; this provides a handy way of combining the MH system with `mail` because you can use `mail` to weed out messages quickly before including your system mailbox into your `+inbox` folder.

You select a folder with the `folder`(1mh) command; this command also shows you what folder is currently selected if you enter it without a folder name. Folder names are the same as in `mail`; each begins with a plus sign (+). If you enter the option `-all`, the `folder` command displays a list of your folders and the number of messages in each. (You can also use the `folders` command to list your folders.) The `scan`(1mh) command lists the messages in your current folder.

You use the `show`(1mh), `prev`(1mh), and `next`(1mh) commands to read the current, previous, and next messages in your current folder. If you enter a message number with the `show` command, that message becomes your current message. For example:

```
vizier> show 7
Message 7:
From evelyn Mon Jul 23 10:02:10 1990
Date: Mon, 23 Jul 90 10:01:25 edt
To: hale
Subject: Cafeteria hours
Cc:
Status: R
```

```
I'm sorry you didn't ask that sooner.  The cafeteria
closes its breakfast service at 10.  Lunch starts
at 11:30.

vizier>
```

The rmm(1mh) command removes messages from your current folder. If you use the rmm command with no argument, it deletes the current message. If you specify one or more message numbers, the messages you specify are removed. For example:

```
vizier> rmm 2 5 7
```

Table 6-5 lists most of the MH commands. For a full listing, see the mh(1mh) reference page, and see the individual commands' reference pages for complete information.

Table 6-5: Commands for the MH Message-Handling System

Command	Description
ali	Searches the specified alias files and displays the addresses corresponding to the specified aliases.
anno	Annotates messages to keep track of distribution, forwarding, and replies for your messages.
burst	Extracts the original messages from a forwarded message, discards the forwarding header, and places the burst messages at the end of the current folder.
comp	Creates a new mail message, providing a template for you to fill in and invoking an editor to finish the message.
dist	Redistributes the current message to addresses that are not on its original distribution list.
folder	Selects a folder or displays the contents of your current folder.
folders	Lists all your folders and the number of messages each one contains.
forw	Forwards messages to recipients who were not the original addressees. The message is encapsulated (included with a "Forwarded message" notice) and a header is added.
inc	Incorporates mail from you rsystem mailbox into your +inbox folder.
mark	Assigns a name to a sequence of messages in your current folder. You can then use pick(1mh) to select messages marked in this way.
mhl	Lists formatted MH messages. You can use this command as a replacement for more(1) to display messages.
mhmail	Sends mail to the specified users. If you do not specify any users, mhmail works like the inc command.
msgchk	Checks your system mailbox and any other files that can receive new mail for you, looking for new messages. If any new messages are found, msgcheck reports like this: vizier> **msgchk** You have new mail waiting, last read on *date*
next	Displays the next message in the current folder or in the specified folder.

Table 6-5: (continued)

Command	Description
packf	Compresses a folder into a single file. (Each message is normaly stored as a separate file.) Do not confuse packf(1mh) with the pack(1) command.
pick	Selects messages based on content, sequence name, or other criteria.
prev	Displays the previous message in the current folder.
prompter	Invokes a simple editor designed for composing messages. The prompter command is invoked by comp, dist, forw, and repl; you do not need to call prompter directly.
rcvstore	Incorporates a message from the standard input directly into a folder.
refile	Moves messages from the current folder to one or more other folders.
repl	Replies to either the current message or the message you specify.
rmf	Removes all of the messages in a folder and then removes the folder itself.
rmm	Removes messages from a folder. The message files are not actually destroyed; instead, rmm renames them by inserting a number sign (#) as the first character of the file names. On most systems, files whose names begin with a number sign are deleted once a day by an automatic process. Until they are actually deleted, you can recover removed messages by using the mv command to rename the files.
scan	Displays a list of the messages in a folder.
send	Sends a message that you have created using comp(1mh), prompter(1mh), or another editor.
show	Displays the contents of a message.
sortm	Sorts messages in a folder into chronological order according to the Date: field of the message header.
whatnow	Prompts you for what to do with a message you have just composed. You can reexamine an original message to which you are replying, resume editing the new message, or do other tasks associated with sending the message.
whom	Expands the header of a message into a set of addresses and optionally checks to see that the message can be delivered to those addresses.

You can tailor the features of MH by creating a .mh_profile file in your top-level directory. The MH reference pages describe the features that you can modify.

Interactive Communication 7

There are occasions when you must communicate directly with another user. Sometimes, however, you might be in different buildings, or one of you might be logged in using a modem connected to the only available telephone. In situations like these, you cannot simply get up and go to that person's office or use the telephone. This chapter discusses two utilities that provide the ability to communicate immediately, and it also describes how to "take your phone off the hook" so that these two utilities cannot reach you.

7.1 The write Command

Often, you can convey a message to another user by sending a mail message. But mail can take a long time to arrive at its destination, depending on how often both your system and the destination system check their mail queues. If you need instant communication, you can use the `write` command. To use the `write` command, you enter the command and the login name of the person you want to send a message to. Then you write your message, finishing with a CTRL/D. For example:

```
vizier> write daniels
The copier service person is in my office *now* and
needs you to explain what the problem is.
CTRL/D
```

The system sends an announcement line and rings the bell on the recipient's terminal when you enter the `write` command. Each line of the mesage is sent as soon as you press RETURN at the end of the line. This example appears on on `daniels`' screen this way:

```
Message from vizier!hale on tty27 at 15:42 ...
The copier service person is in my office *now* and
needs you to explain what the problem is.
EOF
```

You might recognize the address from which the message comes as being a UUCP address. Despite this addressing, the `write` command lets you communicate only with users on your own system.

You can use the `write` command to hold a two-way conversation by waiting until the end of the conversation to press CTRL/D. But because communication using `write` is line-by-line half-duplex (one-way) communication, it is best to establish a protocol with users to whom you send `write` messages often. One good protocol is to wait after you send your first message until your recipient writes back, and to use some signal such as a pair of slashes to signal that you are waiting for a reply. When you eventually press CTRL/D, the "EOF" that appears on the other person's screen signals that you have ended your part of the conversation[1].

[1] Another common protocol uses "o" to signal "Over" and "oo" to signal "Over and out." This protocol is often used for radio conversations in war movies and police dramas, but it's actually pretty silly. "Over" means "It's your turn now," and "Out" means "Goodbye," so when you say, "Over and out," you're really saying, "You can talk now, but I just hung up."

For example:

```
vizier> write daniels
The copier service person is in my office *now* and
needs you to explain what the problem is.
//
Message from vizier!daniels on tty18 at 15:43 ...
Okay, I'll come immediately.  What office are
you in?
//
I'm in office L23, on the third floor.
CTRL/D
vizier>
On my way.
EOF
```

Note that the system gave you a new shell prompt as soon as you entered CTRL/D, without waiting for `daniels` to send the last message. If you didn't expect an immediate reply, you could go on and do something else while you were waiting for `daniels` to respond.

7.2 The talk Command

The `write` command works for two-way communication, but it is inefficient at best, and its limitation to communicating only with users on your own system can be a real headache. Its major advantage is that it is not limited to video display terminals that use a television-like screen; it also works with teletypewriters and other terminals that print their messages on paper instead of displaying them on a screen. The `talk` command, on the other hand, is a serious two-way communication program that works only with video display terminals. (Many users refer to these terminals as CRT terminals; CRT is an acronym for cathode-ray tube.) The `talk` command is designed to work somewhat like a telephone.

You start the `talk` command the same way you start `write`; to talk to `daniels`, you enter this command:

```
vizier>  talk daniels
```

The program divides your screen in half, assigning you to the top half and the person you address to the bottom half. A message appears at the top of the screen:

```
[No connection yet]
```

When the connection is established, this message is replaced by a different message:

```
[Waiting for your party to respond]
```

When this message appears on your screen, the `talk` program rings the bell on the other person's terminal and displays a message announcing that you are calling and explaining how to answer:

```
Message from Talk_Daemon@vizier at 16:18 ...
talk: connection requested by hale@vizier
talk: respond with:  talk hale@vizier
```

If the person at the other end is slow to answer, the system will ring again:

```
[Ringing your party again]
```

When the person at the other end responds, the system tells you that your connection has been established. You can then converse as long as you want. If you fill up your

half of the screen, `talk` goes back to the top of that half and overwrites lines you sent earlier. To end the conversation, press CTRL/C. The system will tell you that the conversation has finished:

```
[Connection closing. Exiting]
```

You can see from this example that you respond to a `talk` call exactly as if you had originated the call yourself; the system figures out who called whom and takes the appropriate action.

You might recognize the address displayed in the announcement on `daniels'` screen in the last example as an Internet address (`hale@vizier`). In this way, `talk` is different from `write`; besides providing full-duplex (two-way) communication, `talk` also allows you to converse with users on other systems. To use `talk` to communicate with a user named `arnold` on a system named `muezzin`, enter the `talk` command this way:

```
vizier> talk arnold@muezzin
```

You can also use UUCP addressing to converse with users on other systems. For example, if `muezzin` is also on a UUCP network, you could start a `talk` session this way:

```
vizier> talk muezzin!arnold
```

7.3 The mesg Command

When you are at home, you sometimes want to take your phone off the hook so nobody can call you. On the ULTRIX system, you can figuratively "take your phone off the hook" by using the `mesg` command. If you enter `mesg n`, the system will not allow `talk` or `write` messages to arrive at your terminal. To enable these messages again, enter `mesg y`. If you forget the current `mesg` state of your terminal, you can enter `mesg` with no arguments; the system will respond with either `is y` or `is n`. The default state is to allow messages.

Part III: Other Commands and the Shell

Calculators 8

This chapter describes two calculator utilities, `bc` and `dc`. Both utilities work much like a desk calculator. The `bc` calculator, as well as being an interactive calculator, is also a compiler and programming language that allows you to write sophisticated calculating applications without having to use complex programming languages.

8.1 The bc Calculator

Most ordinary hand-held calculators use some variation of a system called algebraic notation. You enter problems in much the same way you would write an algebraic equation. For example, to find the sum of 3 and 4, you would think of the problem this way:

$$3 + 4 = ?$$

To solve this problem, you would press the 3 key, then the plus key (+), then the 4 key, and finally the equals key (=). Algebraic calculators provide parentheses for solving more complex problems such as this one:

$$(2 + 3) \times (4 + 5) = ?$$

Table 8-1 shows the steps for one solution of this problem.

Table 8-1: Solving a Problem Using Algebraic Notation

Key	Description of Operation
[(]	Start the first sum, using parentheses to make sure the multiplication doesn't happen at the wrong time.
[2]	Enter the first operand for the first sum.
[+]	Enter the addition operator for the sum. The calculator stores this operator until later.
[3]	Enter the second operand.
[)]	Close the first set of parentheses. The calculator recalls the addition operator and performs the addition, returning a sum of 5.
[×]	Enter the multiplication operator. The calculator stores both the first sum and this operator until later.
[(]	Start the second sum.
[4]	Enter the first operand for the second sum.
[+]	Enter the addition operator for the sum. The calculator stores this operator until later.
[5]	Enter the second operand.

Table 8-1: (continued)

Key	Description of Operation
)	Close the second set of parentheses. The calculator recalls the addition operator and performs the addition, returning 9.
=	Calculate the final result. The calculator recalls the first sum and the multiplication operator it stored earlier and performs the multiplication to give you the final answer, 45.

You could also add the first sum, store the result in a memory, add the second sum, and then multiply by the recalled first sum. With bc, you can use either of these methods just as you would with a hand-held calculator.

There are problems in which algebraic notation is ambiguous and can be misinterpreted. For example:

$$2 + 3 \times 4 + 5 = ?$$

This problem looks like the example used in Table 8-1, but it's actually an entirely different problem because algebraic hierarchy gives precedence to multiplication over addition. The correct way to interpret this problem is this:

$$2 + (3 \times 4) + 5 = 19$$

But unless you use parentheses, some algebraic calculators might actually interpret the problem this way:

$$((2 + 3) \times 4) + 5 = 25$$

Unlike most commercial hand-held calculators, bc is a true algebraic calculator. It handles problems correctly according to the rules of algebraic hierarchy. In an expression not delimited by parentheses, exponentiation is done first; then multiplication, division, and remaindering; and finally addition and subtraction. Portions of an expression that are enclosed in parentheses are evaluated before being used to evaluate the portions outside the parentheses.

The bc calculator works with numbers of arbitrary precision; this means that you can use decimal points, with as many digits as needed after the point.

8.1.1 Starting and Stopping bc

To start the bc calculator, enter the bc command at the shell prompt. For example:

```
vizier> bc
```

Note that bc does not give you any visible prompt. In this example, the box represents the position of the cursor as bc waits for you to enter commands.

To stop bc, enter the quit command. The bc utility will return you to the shell prompt. For example:

```
vizier> bc
quit
vizier>
```

8.1.2 Using bc

To use `bc` as an interactive calculator, you must enter your entire problem on a single line. To work the problem that illustrates algebraic ambiguity in Section 8.1, start `bc` and type in the problem just as you see it. The problem is this:

$$2 + 3 \times 4 + 5 = ?$$

Enter the problem this way:

```
2+3*4+5 RETURN
19
```

The asterisk (*) is the `bc` calculator's times (×) key, and pressing RETURN is the equivalent of pressing the equal key (=) on a hand-held calculator. The `bc` calculator uses proper algebraic hierarchy to return the correct answer.

The calculator uses six standard mathematical operators:

Operator	Description	Examples
+	Addition	3+5=8 6245+713=6958
–	Subtraction – also used to indicate a negative number	7–2=5 42*–14=–588
*	Multiplication	4*18=72 72*393=28296
/	Division	27/9=3 355/113=3.14159...
%	Remaindering – integers only (`a%b` returns the remainder of dividing *a* by *b*)	13%3=1 8%3=2
^	Exponentiation	3^3=27 6^6=46656

8.1.2.1 Handling Noninteger Numbers – When performing calculations, `bc` truncates the result toward zero, maintaining the number of digits to the right of the decimal point (fractional digits) that appear in the operand having the greatest number of fractional digits. For example, multiplying 1.2345 by 3.67 returns a result of 4.5306 instead of the complete result, 4.530615. Four fractional digits are returned because 1.2345 contains four.

This method of truncation can produce some startlingly incorrect answers; for example, when you divide 1 by 3, the result is normally 0 instead of 0.333... .

The `bc` calculator allows you to control how many decimal places it will maintain by using a predefined variable (register) named `scale`. You can specify how many fractional digits `bc` will maintain in its calculations by assigning a value to the `scale` register. For example, the following command changes the number of decimal digits maintained to eight:

```
scale=8
```

The scale value you set remains in effect until you change it:

```
scale=8
1.9375/1.3124
```

```
1.47619047
1/3
.33333333
scale=5
1.9375/1.3126
1.47619
```

8.1.2.2 Creating and Using Registers – You can assign values to temporary storage locations (called **registers**) in `bc`; this feature allows you to save intermediate results and recall them for later calculations. You use a single lowercase letter to name a register. For example, the following command assigns the value of 337 to register `r` and then asks `bc` to display the contents of the register:

```
r=337
r
337
```

Note that when you assign a value to a register in this way, `bc` does not display any result. You can ask `bc` to display the register, as in this example, or you can make `bc` display the result of the assignment by enclosing the entire statement in parentheses; then `bc` evaluates it as a complete value and prints it. For example:

```
(r=337)
337
```

You can perform a complete calculation and assign the result to a register. The following example calculates an approximate value of π (pi) and stores the result in register `p` for use in calculating the volume of a 24-inch sphere on the next line. (The formula for a sphere's volume is $^4/_3\pi r^3$). This example assumes that `scale` is set to 8.

```
p=355/113
4/3*p*12^3
7238.23005312
```

(This example's result is very close to the correct value of 7238.22947387... . The easily-remembered fraction $^{355}/_{113}$ is a much more accurate approximation for π than 3.14 or $^{22}/_7$.)

Because the calculator saves only the number of fractional digits you specify with the `scale` command, intermediate results can be truncated, causing you to get different results by working a problem in a different order. For example, working the volume example straight through instead of using a stored value for π gives the following answer:

```
4/3*355/113*12^3
7238.23007040
```

When you are working complex problems, you should always specify a scale that is several digits greater than the precision you need for your final result.

8.1.2.3 Using Other Radices – The `bc` calculator provides the facility for operating in number radices other than base 10. You specify the input and output radices by assigning values to the `ibase` and `obase` registers as you would to any other register. The `ibase` register affects the radix of numbers you enter, and `obase` affects the results displayed. The following example converts octal numbers (base 8) first to decimal and then, after `obase` is changed, to hexadecimal (base 16):

```
ibase=8
52746
```

```
21990
77125
32341
obase=16
52746
802A
77125
BB01
```

You can use unusual radices such as 100,000 to perform tasks like grouping digits in sets of five. For example:

```
obase=100000
123456789012345
 12345 67890 12345
```

Output of large numbers in radices other than 10 is slow; nondecimal conversion of a 100-digit number can take several seconds on some systems.

8.1.2.4 Creating and Using Functions – If you have formulas that you use often, such as the one we've shown for calculating the volume of a sphere, you can create a command file, or **program,** for `bc` that will automatically initialize those formulas so you can use them when you start the calculator. Formulas initialized in this way are called **functions.** A function can do anything that `bc` can do. You create a function with the `define` command. The following example creates a function to calculate the volume of a sphere:

```
vizier> cat > vol-sphere
define v(x){        [1]
auto z              [2]
z=4/3*355/113*x^3   [3]
return(z)           [4]
}                   [5]
CTRL/D
```

The lines in this example demonstrate all the parts of creating a function:

[1] The `define` command starts the function definition. The name of the function is `v`. Function names must consist of a single lowercase letter. The `x` in parentheses names the variable parameter that will be input to the function. In this example, `x` is the radius of the sphere. There can be as many parameters as you need; separate them with commas. The left brace starts the function's actual definition.

[2] The `auto` command defines the name of an "automatic" variable that will be used inside the function. It is initialized to zero when the function is called and is thrown away when the function has finished. You can define as many automatic variables as you need, but you must do so using only one `auto` command. The `auto` command must be the first line in the function definition. Automatic variable names can be the same as register names outside the function.

[3] This line calculates the desired value and assigns it to the variable `z`.

[4] This line tells the function to return the calculated variable to you when you call the function.

[5] The right brace ends the function definition.

After you have created one or more functions, you can gather them into one file. When you start `bc`, include this file's name as an argument on your command line.

The bc calculator will start, read your function-definition file, and then wait for input from you. For example:

```
vizier> bc vol-sphere
scale=8
v(12)
7238.23005312
```

The bc calculator has a built-in square root function consisting of the word sqrt followed by the value whose square root you want, like this:

```
sqrt(64)
8
```

In addition to the built-in sqrt function and any functions you create, there is a library that includes the following functions:

Function	Name	Function	Name
Sine	s(x)	Natural logarithm (base *e*)	l(x)
Cosine	c(x)	Exponential (e^x)	e(x)
Arctangent	a(x)	Bessel function of integer order	j(n,x)

To use this library, start bc with the -l command option. You can use this library together with your own function definitions; be sure all your functions are named differently from the functions in the library.

8.1.3 Programming bc

This section discusses the advanced features of bc that allow you to write sophisticated programs for it. These features are not limited to use in programs; you can also use them interactively. If you are familiar with the C programming language, you will find few surprises in programming bc. This discussion of bc programming does not describe all of bc's features; for a complete explanation, refer to the ULTRIX *Supplementary Documents, Volume I: General User.*

Having read the description of functions in the preceding section, you already know the basics of programming bc. The more advanced bc features allow you to write more complex and powerful functions. The complete syntax of a function call is as follows:

function-name ([*expression* [, *expression* ...]])

(The spaces in this syntax diagram are for clarity only; do not include them when defining functions.) An expression can be an explicit value or a variable.

Any function that is defined with no parameters always returns a zero result, but it can produce other indirect results by operating on variables (registers) that are not declared as automatic inside the function. This ability allows you to calculate several things in a single function. For example, you could calculate both the surface area and the volume of a sphere, storing the results into registers a and v, which you create outside the function.

Statements in a function definition can be separated by semicolons or placed on separate lines. To make your function definition more easily maintainable, you can include comments. Begin a comment with a slash and an asterisk (/*) and end it with an asterisk and a slash (*/). This convention is the same as the style for including comments in C language programs.

The following sections describe the basic programming constructs of `bc`.

8.1.3.1 Control Structures – The `bc` calculator provides three control structures for conditional execution. Each of these structures allows you to execute one or more statements based on satisfaction of the condition being tested.

You test a condition by expressing it as a relationship between two values. A relational operator compares the value to the left of the operator against the value to its right. These are the standard relational operators:

==	Equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

For example, `(x>=y)` is true if `x` is greater than or equal to `y`. You can use complete expressions as values for testing. For example:

```
(x==y+32)
```

In this example, the expression *y*+32 is evaluated before being tested to see if it is equal to *x*. When you are making tests, do not confuse the relational operator == with the mathematical operator =, which works but doesn't do what you expect.

Each of the three control structures tests the relationship the same way, but the actions that result are different:

- `if` (*relation*) *statement*
  `if` (*relation*) {*statements*}

  The `if` command causes execution of *statement* or *statements* if the specified relationship between values is true. For example:

  ```
  if (x>r^3) r=r^3
  ```

  This statement checks whether register `x` is greater than the cube of register `r`. If it is, then `r` is cubed.

- `while` (*relation*) *statement*
  `while` (*relation*) {*statements*}

  The *statement* or *statements* are executed repeatedly as long as the relationship is true. Somewhere in the body of the code being executed there must be a statement that alters one or both of the tested values, or this construct will loop forever. The relationship is tested before each pass through the loop. For example:

  ```
  while (x>r^3) r=r^3
  ```

  This statement checks whether register `x` is greater than the cube of register `r`. If it is, then `r` is cubed. The relationship is tested again and `r` is cubed repeatedly until the cubed value equals or exceeds the value of `x`.

- `for` (*expression1*; *relation*; *expression2*) *statement*
  `for` (*expression1*; *relation*; *expression2*) {*statements*}

  The `for` command executes *expression1* once to initialize conditions. Then *relation* is tested; if it is true, *statement* or *statements* are executed. Then *expression2* is executed and the relationship tested again. If it is still true, *statement* or *statements* are executed again. Then *expression2* is executed, and

so on. This loop is repeated until the relationship is no longer true. This construct is usually used for controlled iteration, as in this example:

```
for (r=1;r<=5;r=r+1) r^3
1
8
27
64
125
```

This example displays the cubes of the integers from 1 to 5.

8.1.3.2 C Language Constructs – The constructs shown in Table 8-2 work in `bc` exactly as they do in C.

Table 8-2: C Language Constructs in bc

Construct	Result	Construct	Result
`x=y=z`	`x=(y=z)`	`x =^ y`	`x = x^y`
`x =+ y`	`x = x+y`	`x++`	`(x=x+1)-1`
`x =- y`	`x = x-y`	`x--`	`(x=x-1)+1`
`x =* y`	`x = x*y`	`++x`	`x = x+1`
`x =/ y`	`x = x/y`	`--x`	`x = x-1`
`x =% y`	`x = x%y`		

Note that in some of these constructs spaces are meaningful; for example, `x=-y` sets the value of `x` to be `x-y`, whereas `x= -y` sets the value to be `-y`.

You can use these constructs to simplify expressions; for example, the following `for` statements are equivalent:

```
for (r=1;r<=5;r=r+1) r^3
for (r=1;r<=5;++r) r^3
```

8.1.3.3 Arrays – For complex calculations involving many values, you can create an array to hold the values. Using arrays allows you to manipulate any or all of the values by using a control construct at a later point in your program. Array names are lowercase letters like regular registers, except that they also have subscripts. A subscript is an expression in brackets that identifies the specific element. The first element in an array is named *array-name* `[1]` . The second is *array-name* `[2]` , and so on. The following example creates a five-element array called `c`, loads its elements with the cubes of the integers 1–5, and then displays the results:

```
for (r=1;r<=5;r++) c[r]=r^3
for (r=1;r<=5;r++) c[r]
1
8
27
64
125
```

8.2 The dc Calculator

The dc calculator is an interactive utility. It performs operations step by step as you input information to it. The way you enter numbers and commands to dc, called **reverse Polish notation** (RPN), is different from the algebraic notation most ordinary calculators use. There are calculators that use RPN; if you are familiar with them, you might not need to read the following discussion.

RPN is a modified form of the notation invented by the Polish mathematician Jan Lukasiewicz. An RPN calculator works with just one or two operands at a time. Operands are stored on a push-down stack until needed; this means that they are recalled (popped) in the reverse of the order in which they were stored (pushed).

Consider the following problem:

$$(2 + 3) \times (4 + 5) = ?$$

Table 8-3 shows the steps you would follow to solve this problem using an RPN calculator. The third column of the table illustrates the contents of the stack as each operation is performed.

Table 8-3: Solving a Problem Using Reverse Polish Notation

Key	Description of Operation	Stack Contents	
2	Key in the first operand for the first sum.	Top:	2.0000
Enter	Push this operand down in the stack to make room for the next one.	Top: 2nd:	2.0000 2.0000
3	Key in the second operand. This operand goes on top of the first one.	Top: 2nd:	3.0000 2.0000
+	Perform the addition. The sum goes back on the stack, with nothing under it.	Top:	5.0000
4	Key in the first operand for the second sum. This operand goes on top of the first sum, which is automatically pushed down to make room for it.	Top: 2nd:	4.0000 5.0000
Enter	Push this operand down in the stack.	Top: 2nd: 3rd:	4.0000 4.0000 5.0000
5	Key in the second operand. This operand goes on top of the previous operand, which is in turn on top of the first sum.	Top: 2nd: 3rd:	5.0000 4.0000 5.0000
+	Perform the addition. The sum is on the stack, above the first sum.	Top: 2nd:	9.0000 5.0000
×	Perform the multiplication. This gives the final result.	Top:	45.0000

Each time you press an operator key, the calculator pops the top two values from the stack and operates on them, placing the result back on the top of the stack.

Because RPN requires fewer keystrokes than algebraic notation, it is often more efficient. The two algebraic methods we described in Section 8.1 (using parentheses

or using memory) both take 12 keystrokes. Solving the same problem on a hand-held RPN calculator takes only nine keystrokes. Because the bc calculator uses true algebraic hierarchy, there are some problems for which bc needs fewer keystrokes than dc, but usually the RPN calculator is more efficient.

RPN has another advantage over algebraic notation. As described in Section 8.1, algebraic notation can produce different results depending on whether you use parentheses. The bc calculator avoids most problems of this kind by using strict algebraic hierarchy, but with dc's RPN no such ambiguity exists. The calculation order you want is observed automatically because the calculator doesn't store pending operations until a convenient time to perform them.

Once you get used to it, you might also find RPN to be more intuitive than algebraic notation; generally, RPN lets you enter your calculation from left to right and from top to bottom without having to spend a lot of time decomposing the problem before you start.

Like bc, the dc calculator works with numbers of arbitrary precision; you can use decimal points, with as many digits as needed after the point. There is also no limit to the depth of the stack; you can enter a dozen, or two dozen, values and then apply all the operators you need. It is usually more efficient, however, to remember that dc works only at the top of the stack; entering your calculations in the general way shown in Table 8-3 is the most efficient way of working.

8.2.1 Starting and Stopping dc

To start the dc calculator, enter the bc command at the shell prompt. For example:

```
vizier> dc
```

Like bc, dc does not give you any visible prompt.

To stop dc, enter the q command.

8.2.2 Using dc

Using dc is much like using an RPN calculator as described in the introduction to Section 8.2. Because you're really dealing with a program that reads your input line by line, you have to end each command line by pressing RETURN. And because this calculator doesn't have a display that it can update constantly, you have to tell it to print results that you want to see (with a p command). To solve the problem we show in Table 8-3, you would give the following dc commands:

```
2
3
+
4
5
+
*
p
45
```

With all the RETURNs, this seems rather inefficient, especially since we commented earlier on the increased efficiency of RPN over algebraic notation. There is a better way. You can use the RETURN key as if it were the calculator's ENTER key, pressing it only to separate two operands as you place them in the stack. This technique allows you to key in more than one item on a line. When you're entering two operands in a row, you have to end the first one with a RETURN to tell dc

which digits belong to which operand. Otherwise, since operators separate operands, you can enter as many items on a line as you like. You could solve our example in this way:

```
2
3+4
5+*p
45
```

If you compare this example to Table 8-3, you will find that they both require exactly the same number of keystrokes except for the `p` command that tells `dc` to print the result.

The following sections describe how to use the `dc` calculator's features.

8.2.2.1 Using dc Commands – The `dc` calculator has many capabilities beyond the simple arithmetic operators; it supports memory storage, stacked storage, command file execution, and more. Table 8-4 lists all the commands that `dc` accepts.

Table 8-4: dc Commands

Key	Description of Operation
number	A number is an unbroken string of digits, with or without a decimal point. To indicate a negative number, precede the number with an underscore (_).
`+ - * / % ^`	The top two values in the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The values are popped from the stack and replaced by the result. See Section 8.2.2.2 for a description of how numbers with decimal points are handled.
`c`	The entire stack is popped and becomes empty.
`d`	The top value on the stack is duplicated.
`f`	All values in the stack and in registers are printed.
`i`	The top value on the stack is popped and used as the number radix for further input. If you use `I` instead of `i`, the value is used but not popped.
`k`	The top value is popped from the stack and used as the number of decimal places that are maintained during multiplication, division, and exponentiation. If you use `K` instead of `k`, the value is used but not popped.
`l`*x*	The value in register *x* is placed onto the stack. The contents of the register are not altered.
`L`*x*	The top value on a storage stack named *x* is popped and placed onto the stack. See the `S`*x* command.
`o`	The top value on the stack is popped and used as the number radix for further output. If you use `O` instead of `o`, the value is used but not popped.
`p`	The top value on the stack is printed.
`q`	The program stops and returns you to the shell.

Table 8-4: (continued)

Key	Description of Operation
s*x*	The top value in the stack is popped and stored in a register named *x*. You can use any character except DELETE for *x*, including a space or a RETURN, so you can have as many as 127 registers.
S*x*	The top value in the stack is popped and placed on top of a storage stack named *x*. See the `L`*x* command.
[*string*]	The bracketed string is placed on the stack as a string instead of as a number. See the `x` command.
`v`	The top value on the stack is replaced by its square root.
`x`	The top value on the stack is popped and executed as a series of `dc` commands. This value is assumed to be a string.
`z`	The number of elements in the stack is placed on top of the stack.
<*x* >*x* =*x*	The top two values in the stack are compared in this order: `top value` *relational-operator* `second value` If the values satisfy the test, register *x* is executed. To negate the tested relationship, precede the operator with an exclamation point (!); thus, `!>x` tests to see that the top value in the stack is not greater than the second value in the stack.
`?`	A line of input is taken from the input source and executed.
`!`	The remainder of the line is passed to the shell for execution as a shell command line.

8.2.2.2 Handling Noninteger Numbers – When performing calculations, `dc` truncates the result toward zero, maintaining the number of digits to the right of the decimal point (fractional digit) that appear in the operand having the greatest number of fractional digits. For example, multiplying 1.2345 by 3.67 returns a result of 4.5306 (instead of the complete result, 4.530615).

This method of truncation can produce some startlingly incorrect answers; for example, when you divide 1 by 3, the result is normally 0 instead of 0.333... .

You can control how noninteger results are handled by using the `k` command to tell `dc` how many digits you want maintained (the **scale**). For example, you can specify that eight fractional digits are to be maintained by entering a `k` command as shown in this example:

```
8k
```

After you have entered a `k` command, dividing 1 by 3 returns a more sensible answer:

```
1
3/p
.33333333
```

The scale you specify with the `k` command remains in effect until you change it with another `k` command.

8.2.2.3 Entering Commands and Operands

– The dc calculator treats commands the same as a hand-held calculator treats its function keys; there is no special syntax for dc commands. For example, to set a scale of 8, clear the stack, and then calculate the volume of a 24-inch sphere, you would enter the following key sequence:

```
8kc4
3/355
113/*12
3^*p
7238.23005312
```

When dc uses two values from its stack for a command, the stack's top value is always applied to the next-to-top value. This means that to divide 4 by 3, as in the preceding example, you enter the 4 first, then the 3. The slash following the 3 performs the division. The only time this order could be confusing is in exponentiation, where the first value entered (next to top) is raised to the power of the second value (top of stack). The previous example illustrates the correct order for exponentiation operands; the 12 is entered first and then raised to the power of 3.

8.2.2.4 Using Other Radices

– The dc calculator provides the facility for operating in any number radix (base) from 2 to 16. For example, if you are a programmer, you might have frequent need to work in octal (base 8) or hexadecimal (base 16) notation.

You can change the input and output radices independently; the following example shows how you can enter octal numbers and receive hexadecimal results. The i and I commands change the input radix, that is, the radix for your entries. The o and O commands change the output radix, that is, the radix for display:

```
16o8i177p
7F
```

Note that if you change both the input and output radices as in this example, you must enter the change to the output radix first.

For any radix greater than 10, dc uses the uppercase letters A–F to represent the decimal values 10–15.

8.2.3 Programming dc

Stacks, strings, comparison commands, and other features are included in dc to allow you to write programs for the calculator; they are not generally useful when you are using the calculator interactively. Because the technique for programming dc is at an intimately detailed level, we recommend that you write programs using bc. In fact, bc uses dc for its calculations. Each line of a bc problem or program is translated from the form in which you enter it into a problem for dc. Then the dc calculator is used to make the actual computation.

C Shell Scripts 9

This chapter describes how to write C shell scripts. Scripts can save time and effort by pulling together the functions of many commands into one command that is tailored to do exactly what you need to do. This entire book was formatted and typeset for printing by a shell script.

The discussion in this chapter assumes that you are a moderately experienced shell user. If you are not familiar with the basics of communicating with the shell, you should read the chapter on the C shell in *The Little Gray Book: An ULTRIX Primer.*

Most of the material presented here is also useful when you are using the shell interactively, and many of the techniques described here are equally adaptable to other shells.

9.1 Creating and Using Shell Scripts

Scripts are programs for the shell. But instead of being written in a complicated programming language, a script is simply a file that contains a series of the same commands you would type on your keyboard. The shell reads a script file, interprets each command just as if it had come from the keyboard, and executes the command. The concept of creating shell scripts to simplify tasks you do repetitively was introduced in the *Primer* with the following script, called `swap`:

```
#
mv $1 swap.tmp
mv $2 $1
mv swap.tmp $2
```

This `swap` script interchanges two files, naming each with the other's former name. To create this script, follow these steps:

1. Create a file called `swap` in your `bin` directory:

   ```
   vizier> cat > ~/bin/swap
   mv $1 swap.tmp
   mv $2 $1
   mv swap.tmp $2
   CTRL/D
   ```

2. Use the `chmod` command to make the file executable:

   ```
   vizier> chmod u+x ~/bin/swap
   ```

3. Check your `.login` file to make sure your `bin` directory is included in your path; if it's not there, add it. Then log out and log back in to make the change take effect. (Another way to make changes like this take effect is discussed in Section 9.7.)

Once you have created the `swap` script, you use it as you would use any other command:

```
vizier> swap file1 file2
```

In essence, you have created a new ULTRIX command for your own use.

Many standard ULTRIX commands are shell scripts. Because a script is interpreted by the shell each time it is used, scripts run more slowly than programs written in a language such as C. Often the time difference is little enough that it is not important; the ease of creating and maintaining scripts outweighs the loss of speed.

The scripts for many tasks can be as short and straightforward as the `swap` script. For tasks that involve making decisions, the shell provides control structures that test conditions or relationships in order to decide whether to perform further commands. There are also structures that allow a script to perform actions repetitively. For larger tasks, scripts can become quite complex; the script that formatted this book is designed to be generic so that it can be used by many writers, and it is more than 1000 lines long.

The following sections describe how to write C shell scripts that make effective use of the shell's features.

9.2 Using Comments in Shell Scripts

It is a good idea to include comments in your shell scripts. Comments help you remember what the script does and how to use it. They can also make it easier for someone else to modify a script. The shell interprets a number sign (#), also called an octothorpe, as a comment introducer. Anything that follows this character on the same line is a comment. For example:

```
# This is a comment.
#
cp file1 file2     # This is also a comment.
```

The shell ignores the first two lines of this example. It executes the `cp` command on the third line and then ignores everything after the command. Note that you cannot use the number sign this way interactively; the shell does not interpret it as a comment introducer when you type it at the shell prompt.

The number sign has another use in C shell scripts; see Section 9.3 for more information.

9.3 Specifying Use of the C Shell

Because the ULTRIX system has several different shells, the first thing you must know about writing C shell scripts is how to specify that they are for the C shell. You can do this by including a number sign as the first character in your script file. But because it is also the character that introduces shell comments, the number sign can cause confusion when used in this way. One solution is to place the number sign alone on the first line of the file, as in the `swap` script shown in Section 9.1.

A better solution takes advantage of an ULTRIX object called a **magic number**. The magic number of a file tells the system what kind of file it is. Every executable file has its magic number in the first two bytes of the file. The combination of a number sign and an exclamation point (#!) is a magic number that tells the system to execute the rest of the line as if it were a normal shell command. You can use a magic number in your script as in the following example:

```
#! /bin/csh
#
# This is a script that will execute under the C shell
# as a result of magic number interpretation.
#
   .
   .
   .
```

Every time you enter a shell command, the ULTRIX system starts a new shell for that command to run in. Because the shells themselves are ordinary binaries, they can be executed just as any other command; you can type `csh` at the shell prompt and start a new shell. When the script in this example is run, its magic number tells the shell to execute the command named `/bin/csh`; this command is the C shell, so a new C shell is started for the script.

9.4 Creating and Using Shell Variables

Shell variables are names that the shell uses to keep track of the objects it is manipulating. There are three types of shell variables:

- String variables

  String variables are character strings that the shell uses for a variety of purposes. For example, the `prompt` variable contains the text string that the shell uses to prompt you for commands.

- Numeric variables

  Numeric variables are used by the shell in the same ways you use numbers. They can identify things, or they can be used to count things.

- Binary variables

  Binary variables act like switches; they can be set (on) or unset (off). The `noclobber` variable, for example, prevents the shell from redirecting output onto a file that already exists.

Some shell variables are built into the shell or created by your `.login` and `.cshrc` files. Binary variables, such as `noclobber`, are built into the shell. Users do not normally create new binary variables, but you can set or unset these variables (turn them on or off) as needed. You can also manipulate other built-in variables such as `path`. For a discussion of all the shell's built-in variables and what they do, see the `csh`(1) reference page. Your script can create other variables as it needs them. You can use any names you like, except for those already given to built-in variables.

9.4.1 Setting String Variables

To create a string variable or change the value of an existing one, you use the `set` command, like this:

```
set myvar=Testing
```

This command creates a variable named `myvar` and assigns the string value "Testing" to it. If the string you are assigning contains spaces, you must surround it with quotation marks, like this:

```
set myvar="Testing one two three"
```

The shell treats a variable created in this way as a single "word" even though to you it consists of four words. You can create a variable that the shell will see as having multiple words by enclosing your value in parentheses:

```
set myvar=(Testing one two three)
```

Section 9.5.1 explains how to use multiword variables.

9.4.2 Setting and Manipulating Numeric Variables

To create a numeric variable, you set it as if it were a string variable:

```
set lines=8
```

Once you have created a numeric variable, you can manipulate it by using the shell's `@` command. Although the variable was originally created as a string, the shell converts between numbers and strings as needed. Numeric variables must be integers, and any non-integer results are truncated to return integers. Note that the shell observes proper algebraic hierarchy, performing multiplication and division before addition and subtraction. You can use parentheses to alter the sequence of operations. For example:

```
set lines=8
@ lines = ($lines + 2) / 3
set lines=8
@ lines = $lines + 2 / 3
```

In the first `@` command, the addition is performed first because it is within parentheses. Then the division is performed, giving a result of $3^1/_3$. The shell truncates this result to the integer value of 3. The second `@` command performs the division first, making the result equal to $8^2/_3$. The shell then truncates this value to 8.

As shown in this example, you must separate the elements of your expression with spaces. A list of available mathematical operators is shown in Section 8.1.2.

9.4.3 Setting Binary Variables

To set a binary variable, you use a `set` command with the variable's name and no argument, as in this example:

```
set noclobber
```

As with string and numeric variables, setting a binary variable actually creates the variable; before you set it, the variable does not exist.

9.4.4 Removing Variables

To remove a variable, use the `unset` command:

```
unset myvar
unset lines
unset noclobber
```

After you enter one of these commands, the variable named by the command will no longer exist. (When the shell tests a binary variable such as `noclobber`, it is actually just testing to see if that variable exists.)

If you refer to a variable that you have unset, the shell returns this error:

variable-name`: Undefined variable.`

9.5 Using Shell Variables

Once you have created a variable, you can use it in other commands by referring to its name. When you refer to a variable, you must precede the name with a dollar sign ($) to indicate that you are referring to a variable instead of a file, a command, or an ordinary text string. For example:

```
set myvar="Testing one two three"
set lines=8
echo $myvar
echo $lines
```

The first `echo` command returns *Testing one two three,* and the second returns the string *8*; in this example, these values are displayed on the standard output. You can redirect output just as when using the shell interactively. If you try to refer to a binary variable in this way, the shell returns nothing.

9.5.1 Using Multiword Variables

Section 9.4.1 shows how to set multiword string variables. You can use a multiword variable either as a single entity or as multiple words, and you can modify individual words. To address a single word, you use the variable's name followed by the numeric position of the word in brackets. For example:

```
set myvar=(Testing one two three)
echo $myvar[2]
```

The shell returns the value of the second word, *one*. This ability to address individual words of a variable is very useful when dealing with command-line variables, as explained in Section 9.5.3.

If you modify a word of a multiword variable, that word becomes part of the complete variable. For example:

```
set myvar=(Testing one two three)
set myvar[2]=ONE
echo $myvar
```

The shell returns *Testing ONE two three*. You can specify a range of words in a multiword variable by using a construct like this:

```
echo $myvar[2-3]
```

The shell returns *one two*. You can also use a variable as the bracketed index as in the following example:

```
set myvar=(Testing one two three)
set index=2
echo $myvar[$index]
@ index = $index + 1
echo $myvar[$index]
```

The first `echo` command returns *one,* and the second returns *two*.

9.5.2 Testing Variables

You can find out if a variable exists by using the notation `$?`*name*; this construct returns the string *1* if a variable called *name* exists, and *0* if not. You can also find out the number of words in a variable with `$#`*name*. For example:

```
set myvar=(Testing one two three)
set othervar="The quick brown fox"
```

```
echo $?myvar          Returns 1 to indicate that myvar exists.
echo $?thirdvar       Returns 0 because thirdvar does not exist.
echo $#othervar       Returns 1 because othervar contains one word.
echo $#myvar          Returns 4 because myvar contains four words.
```

The last `echo` command returns *4* because `myvar` is a multiword variable created with parentheses. If `myvar` had been created with quotation marks, the shell would consider it, like `myvar`, as containing only one word.

Some scripts can use variables having only true/false values; this is an application for binary variables. You can create binary variables and later use the `$?` *name* form to test them. For example:

```
echo $?binaryvar      Returns 0 because binaryvar does not exist yet.
set binaryvar         Setting binaryvar creates the variable.
echo $?binaryvar      Returns 1 because binaryvar now exists.
unset binaryvar       Unsetting binaryvar removes the variable.
echo $?binaryvar      Returns 0 because binaryvar no longer exists.
```

Section 9.8.1.1 shows how to test binary variables in a script.

9.5.3 Using Command-Line Variables

When you enter a command interactively, the shell creates a variable called `argv`. This variable is a multiword variable that contains the entire command line except the command name itself. For example:

```
vizier> swap sample1 sample2
```

When you enter this command, the `argv` variable receives the string value "sample1 sample2". You can refer to individual words of `argv` as `$argv[1]`, `$argv[2]`, and so on. You can also use `$1`, `$2`, and so on as shorthand names. Although the command name is not included in `argv`, it is assigned to the variable named `$0` so that you can use it if you need it. The `swap` script demonstrates how to use these command-line variables. When you enter the `swap` command, the shell substitutes the values of its command-line variables for the variable names. If you entered the `swap` command with `sample1` and `sample2` as arguments, what would actually happen is this:

```
mv sample1 swap.tmp
mv sample2 sample1
mv swap.tmp sample2
```

Although you should seldom find the need, you can modify `argv` before its individual components are used in the same way that you can modify any other shell variable. For example, suppose you have a script that expects a comma-separated list such as `joe,marge,bev,bill` for its first argument. Because typing a space after a comma is a common habit, you might inadvertently enter your command this way:

```
vizier> writemail joe, marge, bev, bill
```

If you did this, the information would be broken into four words (`$1`, `$2`, `$3`, and `$4`) instead of being assigned to only one (`$1`). By passing `argv` through the `sed` stream editor before using the individual words, you can restore the desired oneness to the list of names:

```
set argv=`echo $argv | sed 's/, */,/g'`
```

This line substitutes a single comma for each occurrence of a comma followed by any number of spaces. Any later use of individual numbered arguments will address

the arguments as you intended. (The use of grave accents in this example is explained in Section 9.6.)

You can use `$*` as a shorthand notation for `$argv`, and you can use ranges of words as with any multiword variable. For example:

```
echo $argv[1-3]
```

9.5.4 Using Special Variables

There are some special variables built into the shell to allow you greater freedom and power in creating scripts. This section describes these variables.

9.5.4.1 Reading User Input – There is a special variable name, `$<`, that is replaced by the next line of input read from the standard input (not the script). This form allows you to create interactive scripts:

```
echo -n "Enter yes or no: "
set yn=($<)
```

This example prompts the user for an entry and then returns the response in the variable `yn`. (The `-n` option for the `echo` command causes the echoed text to be displayed without moving the cursor to the next line.)

9.5.4.2 Using a Script's Process ID – The name `$$` returns the process identification number (process ID) of the current shell. Because a process ID is unique, you can use this variable to create unique temporary file names. For example:

```
set tempfile=cmd_$$.tmp
```

If this command is used by a process whose process ID is 21873, `tempfile` will receive the value `cmd_21873.tmp`.

9.5.4.3 Reading Command Result Status – The shell provides a built-in variable that you can test, called `status`. This variable contains the result of the preceding command. If the command succeeded, `$status` is zero; if the command failed, `$status` is nonzero. Depending on the command, nonzero results can reflect different kinds of errors that the command encountered. The `status` variable is treated as a numeric variable.

9.6 Substituting Command Output

You can use the output of a command as an argument for another command. This is not the same as piping, which uses one command's output as *input* to the next. Suppose you want to know the size of the `nroff` binary. To find this information interactively, you could use the `which` and `ls` commands:

```
vizier> which nroff
/usr/bin/nroff
vizier> ls -l /usr/bin/nroff
-rwxr-xr-x  1 root        58368 Mar  8  1989 /usr/bin/nroff
```

You cannot use two separate commands in this way in a script because there is no way to feed the `which` command's output to the `ls` command. However, there is a way to perform this task, either interactively or in a script, by using grave accents (` ) to include the `which` command into the `ls` command:

```
ls -l `which nroff`
```

The command enclosed by the grave accents is executed, and its output is substituted for the accents and their contents. In this example, `which` returns the location of the `nroff` binary; this information is then used by `ls` to return the directory information.

This substitution technique is useful if you want your script to display messages that look like standard system messages, as in this example:

```
vizier> hits -s
usage: hits [-f index-file] { -m file... | [-s[n]] string }
```

You can display this kind of message with a simple `echo` command:

```
echo "script-name: error text"
```

This technique works, but it is not generic. For example, a script name like `weekly-report-generator` doesn't look much like a typical ULTRIX command name. When you give this script to your system administrator to be installed publicly, it might be named `wkrepg` instead. Any error messages you have included that display the longer name can look out of place. You can use command substitution to make a script find out what its own name is with the `basename` command:

```
set myname=`basename $0`
```

The variable `$0` contains the full path of the command. The `basename` command strips away everything except the base name of the script file. You can use the result as the command's name in messages, like this:

```
set myname=`basename $0`
  .
  .
  .
echo "$myname: error text"
```

If the weekly report generator script displays an error message, that message might look like this:

```
wkrepg: can't find last week's report
```

9.7 Running Other Scripts

Just as you can execute system commands, you can also run other scripts from within your script. Often you will not even know that you are doing this, because scripts that are run this way behave the same as binary programs.

To run another script you can simply use its name, but when you do this a new shell is created to run that script. This means that variables set by the subsidiary script are not set in the calling script's environment. If the subsidiary script changes to a new working directory, for example, the calling script will still be in the original working directory when the subsidiary script finishes. (The name of the current working directory is a C shell variable called `cwd`.) Sometimes this is what you want to do, but sometimes you want the second script to affect the script that called it.

You can run a subsidiary script and have it behave just as if it were part of the main script by using the `source` command. For example, the script that formatted this book begins with a command like this:

```
source $bkl/set_defaults
```

The `set_defaults` script executed by this command specifies what text processing tools and parameters will be used by the formatting script. Because it uses this second script, the complex main script can be installed on different systems without having to be modified; the simple subsidiary script tailors the environment to use the tools provided on each individual system. The `bkl` variable is the name of the directory where the main script and its `set_defaults` subsidiary script are stored. The main script finds what this directory is by using the `dirname` command as in this example:

```
set bkl='dirname $0'
```

The `dirname` command does the opposite of what `basename` does, returning the directory name instead of the command's name.

You can use the `source` command to activate changes you make in your `.login` file. For example:

```
vizier> cat >> .login
set ignoreeof
CTRL/D
vizier> source .login
```

9.8 Making Decisions

The C shell provides control structures that allow your scripts to make decisions and act accordingly. Decisions result from testing expressions. An expression is a string or a number. Shell variables are the most common type of expression; file names are another type. The shell can evaluate single expressions, or it can compare two expressions with each other. You create a test by enclosing the expression or expressions to be tested in parentheses. (The parentheses are required.)

9.8.1 Testing Expressions

The following sections describe how to form expressions. Once you know how to form expressions, you can use them in control structures to make decisions.

9.8.1.1 Testing Expressions Against Primitives – The shell can evaluate a single expression by testing it against one of a predefined set of conditions called primitives. The following example checks to see whether a file called `myfile` exists:

```
( -e myfile )
```

If `myfile` exists, the result of this test is true. The complete list of primitive tests is:

- `-d` The file is a directory.
- `-e` The file exists.
- `-f` The file is a plain file, not a directory.
- `-o` You own the file.
- `-r` The file has read permission.
- `-w` The file has write permission.
- `-x` The file has execute permission.
- `-z` The file has a size of zero.

You can test for the existence of a variable without using a primitive. For example, suppose you have a script that uses a binary variable called `bvar`. You can test `bvar` this way:

```
( $?bvar )
```

This test returns a true result if `bvar` is set (that is, if it exists).

9.8.1.2 Testing Expressions Against Other Expressions – The shell compares expressions using a set of relational operators, most of which are also used in the C programming language. The following example tests to see if the shell variable `fname` is the same as the string ''.cshrc'':

```
( $fname  == ".cshrc" )
```

If the expressions match, the result of the test is true. You can test an expression against a fixed string or number, or you can test one expression against another as in the following example:

```
( $file1 == $file2 )
```

The following list shows the relational operators you can use:

==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=~	Matches an expression containing filename-expansion characters
!~	Does not match an expression containing filename-expansion characters

The magnitude comparisons work only with numbers; you cannot test whether ''a'' is greater than ''A''. The following example tests whether the variable `xx` is greater than 9:

```
( $xx > 9 )
```

The filename-expansion operators are particularly useful because they allow you to use a limited subset of variable expressions in the expression you are testing against. You can use asterisks (*), question marks (?), and expressions in brackets ([ ]) or braces ({ }). For example, you can test a variable against a wildcard expression as follows:

```
( $fname =~ book.* )
```

This test returns a true result if `fname` matches any string beginning with the characters ''book.'' Note that this expression is not a standard RE; the period is not a metacharacter, and it is included in this test. The following example uses filename expansion to test for a match only on six possible strings from `book.1` to `book.6`:

```
( $fname =~ book.[1-6] )
```

While the filename-expansion operators don't give you the full power of REs, they do allow a great deal of freedom. See the `csh`(1) reference page for a discussion of filename expansion.

9.8.1.3 Combining Expressions – Occasionally you need to make two or more tests to determine a single course of action. You can perform this task by using logical operators to combine tests. The two logical operators available are these:

```
&&    and
||    or
```

The following example uses the "and" operator to combine two tests into a single test:

```
(($file1 != "") && ($file2 != ""))
```

This test returns a true result only if both of the variables are not null. Note that both tests are nested within an enclosing set of parentheses. The shell statement can test only a single expression at a time; nesting in this way makes the shell evaluate each of the inside tests first and then evaluate the combination.

You can use the "or" operator in the same kind of structure to test whether one of the arguments is null:

```
(($file1 == "") || ($file2 == ""))
```

This test returns a true result if either variable is null.

9.8.2 Using Control Structures with Expression Tests

Control structures use the results of a test to determine what happens next in a script. By using the shell's control structures you can perform the following actions:

- Choose between two or more courses of action, including (if desired) doing nothing
- Execute a series of commands repeatedly using the same or different parameters each time

The following sections describe the shell's control structures and how to use them.

9.8.2.1 The if Statement – The simplest option available for changing the flow of a script is the `if` statement. You can use the `if` statement to execute a single command based on the results of a test. The following example tests for the existence of a file called `myfile`; if the file is present, the script copies it to a second file as a backup.

```
if ( -e myfile ) cp myfile myfile.backup
```

The `if` statement can be extended to perform more than one command based on the tested condition. By adding the keyword `then` to the `if` line, you can specify a series of commands to be executed if the test is true. You end this series of commands with the `endif` statement. For example, you can maintain and edit files with a short script like this:

```
if ( -e $1 ) then
  cp $1 $1.bak
  vi $1
  nroff -ms $1 | lpr
endif
```

This example expects a command-line argument to specify the file you want to work with. If the file exists, the script saves a copy for backup, invokes the `vi` editor to modify the file, and then processes the changed file through `nroff` and `lpr` to print the results of your changes. Note that the lines within the `if-then-endif`

structure are indented. You don't have to indent like this, but indenting helps you to keep track of control structures, especially if you nest one within another.

You can use the `else` keyword to make your script take alternative action if a test fails. For example, the following script is the same as the one before except that it displays an error message if the desired file does not exist:

```
if ( -e $1) then
  cp $1 $1.bak
  vi $1
  nroff -ms $1 | lpr
else
  echo "File $1 not found"
endif
```

If you fail to specify an argument for this script on the command line, the shell will abort with an error. If this script happened to be part of a much longer script, then this error could be catastrophic; you can protect against it by checking to ensure that there is an argument:

```
if ($1 == "") then
  echo -n "Enter file name: "
  set argv=($<)
endif
if ( -e $1 ) then
  cp $1 $1.bak
  vi $1
  nroff -ms $1 | lpr
else
  echo "File $1 not found"
endif
```

If this example finds that the first command-line argument is null, as it would be if the filename argument were missing, the script prompts for a file name.

9.8.2.2 The while Statement – The last example in Section 9.8.2.1 shows one way to protect from accidental misuse of a script; it asks for a file name if one isn't entered. But if you made a second mistake and pressed RETURN in answer to the `Enter file name:` prompt, the script would abort the same as if it had never asked for a file name. The shell's `while` statement provides a convenient way to execute commands repeatedly as long as a given condition is true; in this case, the condition can be the nonexistence of that all-important file name. You end the list of commands to be controlled by the `while` statement with an `end` statement. The following example shows how the `while` statement can be used to make this script foolproof:

```
while ($1 == "")
  echo -n "Enter file name: "
  set argv=($<)
end
if ( -e $1 ) then
  cp $1 $1.bak
  vi $1
  nroff -ms $1 | lpr
else
  echo "File $1 not found"
endif
```

When you run this script, the `while` statement tests for the argument. If there isn't one, the script prompts for it, and if you enter just a RETURN the script will prompt again. Only when you have entered a non-null response will the `while` statement's test fail, allowing the script to continue. A structure like this that allows you to

execute the same commands repeatedly is called a **loop**; in this example, it is a `while` loop.

9.8.2.3 The foreach Statement – You can design a script that will perform a series of actions repeatedly using a list of items, such as file names. The `foreach` and `end` statements create a loop that will execute a fixed number of times, one for each item in the list. The following example copies each named file for backup:

```
foreach fname (book-chapter1 book-chapter2 book-chapter3)
  cp $fname $fname.bak
end
```

The `foreach` statement creates a variable to control its loop; in this example, the loop variable is named `fname`. The list of file names to be used is in the parentheses. The first file name is assigned to `fname` and the script copies that file, creating `book-chapter1.bak`. Then the second file name is assigned, and the process is repeated until the list of names is exhausted.

You can use filename expansion in the list for a `foreach` statement; for example, `book-chapter*` produces a list of all the matching files. You can also use variables in the list; for example, the following commands could be used in a script that would make backup copies of files you name as command arguments:

```
foreach i ($argv)
  cp $i $i.bak
end
```

9.8.2.4 The switch Statement – One way to take several different courses of action based on the value of an expression is to use a series of `if` statements, as in this example:

```
if ($x == "one") perform choice 1
if ($x == "two") perform choice 2
if ($x == "three") perform choice 3
if ($x == "four") perform choice 4
```

This design works, but it is inefficient because each of the `if` statements must be executed to evaluate the expression repeatedly until a match is found. Each of the `if` statements after the matching one also will be executed. If a given choice involves several commands, then that choice must be implemented with an `if-then-endif` structure. This design also fails to provide for an action to be taken if none of the tests is satisfied.

The last of the shell's control structures, the `switch` statement, simplifies this sort of task. This statement allows you to test an expression against several different expressions and take a different set of actions for each possibility. It also provides for a default course of action. The preceding example can be implemented with the following `switch` structure:

```
switch ($x)
  case "one":
    perform choice 1
    breaksw
  case "two":
    perform choice 2
    breaksw
  case "three":
    perform choice 3
    breaksw
  case "four":
    perform choice 4
```

```
    breaksw
  default:                    4
    perform default action
endsw                         5
```

Note the following items in this example:

1. The `switch` statement evaluates the expression `$x`.
2. This `case` statement looks for a match on the string "one". If `$x` matches this string, then the first choice of action is performed. This action can be as complex as required, involving many commands including other control structures.
3. The `breaksw` statement signifies the completion of the actions taken for this choice. It tells the script to ignore everything from this point until the `endsw` statement. If you omit the `breaksw` statement, the script will fall through and execute the statements controlled by the next `case` (or `default`) statement. You can take advantage of this "omission" feature as in the following example:

   ```
   case "backup"
     cp myfile myfile.bak
   case "edit"
     vi myfile
     breaksw
   endsw
   ```

   If the tested expression matches "backup", the script copies the file for backup. It then ignores the `case "edit"` statement and continues by invoking `vi`. If the expression matches "edit", `vi` is invoked but no backup is done.
4. The `default` statement specifies the action to be taken if none of the cases is matched. This part is optional. You need it only if you want your script to take default action.
5. The `endsw` statement ends the `switch` structure.

9.9 Handling Errors

The `exit` command provides a way for a script to stop after executing only part of its commands, without having to go all the way to the end. This ability is useful for making a script stop gracefully when it detects errors either in the user's use of the script or in executing commands. This is not the same as detecting shell errors such as a missing argument for a `cp` command. Shell errors can make your script do bizarre things; depending on what the error is, you might get an error message, or the script might simply be aborted with no clue as to where in the script the error occurred.

By including commands to check for missing or invalid arguments, you can prevent shell errors so that your scripts will exit gracefully instead of crashing or doing the wrong thing. You can use the `exit` command as in this example:

```
cp $1 $2 >& /dev/null
if ($status != 0) then
  echo "Error during copy operation"
  exit
endif
```

To avoid bothering the user with a system error message that might occur on the `cp` command, this example redirects error output to an imaginary file that serves as a

"bit bucket." (Anything written to /dev/null simply disappears.) The example checks the result of the cp command by examining $status; if the command fails, the script displays an error message and exits.

If no argument is supplied to the exit command, the script's final status is the status of the last non-exit command that was executed. Since the exit command in this example is preceded by an echo command, which will never fail, this script would always return success status upon exiting. But if an argument is supplied, as shown in the following example, the value of that argument is returned as final status.

```
cp $1 $2 >& /dev/null
if ($status != 0) then
  echo "Error during copy operation"
  exit 1
endif
```

The exit command in this example returns nonzero status when the cp command fails. You can examine that final status by looking at the status variable after the script finishes. Also, when you run a script in the background, the shell reports its final status. For example:

```
vizier> copyscr rubaiyat rubaiyat.bak &
  .
  .
  .
[1]  Done                    copyscr rubaiyat rubaiyat.bak
```

If the copy operation fails, the job will return the following error status:

```
[1]  Exit 1                  copyscr rubaiyat rubaiyat.bak
```

In this case, the error status isn't particularly useful because the script will also have reported its own error. But using exit arguments has a second benefit in scripts: Setting status allows your scripts to be called by other scripts; your scripts will work just as if they were standard commands, returning error status that a calling script can act on.

9.10 An Example C Shell Script

The shell script shown here as Example 9-1 employs most of the shell programming techniques discussed in this chapter. Use it as an example of ways to combine functions in a script.

Writers using operating systems other than ULTRIX often use text formatting systems whose input files are not compatible with ULTRIX formatting tools. If this manual had been written using one of these systems, the input file for the section you are reading might look like this:

```
<HEAD1>(An Example C Shell Script\script_example_section)

<P>
The shell script shown here as Example 9-1
employs most of the shell programming techniques discussed
in this chapter.  Use it as an example of ways to combine
functions in a script.

<P>
Writers using operating systems other than <REFERENCE>(u3)
often use text formatting systems whose input files are not
compatible with <REFERENCE>(u3) formatting tools.  If this
```

```
manual had been written using one of these systems, the input
file for the section you are reading might look like this:
  .
  .
  .
```

The script shown in Example 9-1, called `docms`, converts this system's files into files that ULTRIX tools can process. The conversion is accomplished by means of the `sed` stream editor with a set of editor scripts that recognize formatting constructs and translate them into equivalent formatter commands and macros for an enhanced `ms` macro package. The `docms` script uses a series of `sed` scripts because of limitations inherent to the `sed` editor. The result of translating the example text into `ms` form would look like this:

```
.NH 2
An Example C Shell Script
.LP
The shell script shown here as Example 9-1
employs most of the shell programming techniques discussed
in this chapter.  Use it as an example of ways to combine
functions in a script.
.LP
Writers using operating systems other than <REFERENCE>(u3)
often use text formatting systems whose input files are not
compatible with <REFERENCE>(u3) formatting tools.  If this
manual had been written using one of these systems, the input
file for the section you are reading might look like this:
  .
  .
  .
```

Note that not all the foreign constructs are translated; some of them have no direct equivalents in ULTRIX tools, and they are left for the user to handle manually. Example 9-1 shows the shell script that performs the translation.

Example 9-1: Sample C Shell Script

```
#! /bin/csh     1
#
#       This script drives the conversion of an XYZ formatter
#       source file into *roff -ms source format using
#       sed(1).
#
#       Related files:
#
#               doc-ms-global?           main processing
#               doc-ms-last              final pass, cleanup
#               doc-ms-<document_type>?  (opt. for document-type)
#
#       Init variables.
#
        set loc=`dirname $0`   2
        set cmd=`basename $0`
#
        set stamp="$$`date +'%H%M%S'`"   3
        set just="l"
        set document_type=""
        set errors="no"
        set use_err="usage: $cmd [-options] infile [outfile]"
#
#       Parse switches, if any.
#
        foreach i ($argv)                4
```

Example 9-1: (continued)

```
        if ("$i" =~ -[A-Za-z]) then                    5
          switch ($i)
            case "-j":
              shift                    6
              set just="$1"
              switch ($1)
                case "l":
                case "r":
                case "b":
                  shift
                  breaksw
                default:
                  echo "$use_err"
                  exit 1                    7
                  breaksw
              endsw
              breaksw
              case "-d":
              shift
              set document_type="$1"
              switch ($1)
                case "article":
                  set just="b"
                  shift
                  breaksw
                case "software":
                  shift
                  breaksw
                case "special":
                  shift
                  breaksw
                default:
                  echo "$cmd: unsupported document-type"
                  exit 1
                  breaksw
              endsw
              breaksw
            case "-e":
              set errors="y"
              shift
              breaksw
            case "-v":
              set verbose
              shift
              breaksw
            default:
              echo "$use_err"
              exit 1
              breaksw
          endsw
        endif
      end
#
#     Done handling switches - what's left must be the file name.
#
      if ("$1" == "") then
        echo "$use_err"
        exit 1
      endif
      if (! -e $1) then                    8
        echo "$cmd: no input file"
        exit 1
      endif
```

Example 9-1: (continued)

```
#
#       Now create the justification control file and do the
#       processing.
#
        echo ".ds ZJ $just" > $cmd.j          9
        echo ".ad n" >> $cmd.j
        echo ".ad " >> $cmd.j
        cat $cmd.j $1 > $1.tmp
        rm $cmd.j
#
#       Do the optional document-type processing if called for.
#
        if ("$document_type" != "") then                    10
          foreach sedfile ($loc/doc-ms-$document_type?)
            sed -f $sedfile $1.tmp > $stamp.tmp
            mv $stamp.tmp $1.tmp
          end
        endif
#
#       Now do the main processing.
#
        foreach sedfile ($loc/doc-ms-global?)       10
          sed -f $sedfile $1.tmp > $stamp.tmp
          mv $stamp.tmp $1.tmp
        end
        if ($2 != "") then                                  11
          sed -f $loc/doc-ms-last $1.tmp > $2
          if ("$errors" == "y") then
            echo "$cmd: Unprocessed tags" > $2.err
            echo "" >> $2.err
            echo "   Input file:   $1" >> $2.err
            echo "   Output file:  $2" >> $2.err
            echo "   Processed on: `date`" >> $2.err
            echo "" >> $2.err
            echo "Line        Tag or tags" >> $2.err
            echo "----        -----------" >> $2.err
            grep -n "<.*>" $2 | sed 's/:/:        /' >> $2.err
          endif
        else
          sed -f $loc/doc-ms-last $1.tmp
        endif
#
#       Processing complete.  Remove intermediate file.
#
        rm $1.tmp       12
        exit 0
#
#       End of script.
```

Note the following points in this script:

1 This line specifies that the `docms` script is a C shell script.

2 These lines determine the location and actual name of the script file. The `sed` editor script files used by `docms` are all stored in the same directory. This design makes `docms` usable by any user on the system no matter what the user's current working directory is.

3 This line creates a timestamp to use in generating unique temporary file names by combining the time of day with the user's process ID. This timestamp makes sure that even if two users are running `docms` simultaneously in the same directory their temporary files will not conflict.

4 This section of code parses the command options by working through the `argv` variable one word at a time.

5 This `if` statement uses filename expansion to check that the word being processed is a letter preceded by a minus sign (-). Any other string is an argument to one of the options (such as the *document-type* name that follows the `-d` option), a file name, or a mistake. Only words that satisfy the `if` test are processed further.

6 The `shift` statement moves all the words of `argv` one place to the left: `$2` is moved to `$1`, `$3` is moved to `$2`, and so on. The current `$1` is lost. This technique is used in `docms` so that when an option requiring an argument is found, the argument will always appear in a known position, as `$1`. If the words of `argv` were not shifted in this way, the script would have to keep track of the number of each word it processes. Then, if a given word is an option that requires an argument, the script would have to calculate the number of the next word in order to use that next word. The `shift` statement is also used at the end of the `switch` structure to move the words along each time a word is processed, again so that the script will not have to perform the numeric calculation to keep track of the words.

7 This `exit` statement ends the script on detecting an error by the user. Similar `exit` commands are used for all the possible user errors. Nonzero status is returned in each case. Note that the `cmd` variable supplies the script's actual name in the error messages.

8 This code makes sure the user specified an input file that actually exists.

9 This section of code creates the beginning of a temporary file. Several `nroff` commands are inserted to support the `-j` option. This option allows the user to specify whether the output file should be justified to the right, left, or both margins when formatted. (The `.ds ZJ $just` command creates a string variable for `nroff` to use, and the `.ad` commands disable justification and then reenable it in the way specified by the user.)

Note that the option processing done in the `foreach` loop automatically sets for justification to both margins if the `article` document type is specified. You can use this kind of design to do many of the low-level tasks associated with running your script.

10 These two `foreach` loops do the actual translation of the file. If no special document type was specified, then the first loop is not executed. Note that each pass through one of these `foreach` loops processes the file through `sed` and then renames the temporary output using the name of the temporary input. This design allows any number of `sed` scripts to be used in each loop; the `article` document-type, for example, could be processed with a single document-type-specific `sed` script while the `special` document-type might take three scripts. The person who maintains the `docms` tool can create new document-type-specific processing without worrying about the consequences of having different numbers of scripts.

11 This section of code handles the `-e` option. If the user has specified both the `-e` option and an output file name, then `docms` runs `grep` on the final output file to produce a listing of the foreign formatting commands that `docms` might not have processed properly.

12 This line removes the last temporary file that was created during the process. Scripts that create and use temporary files should always remove those files upon completion so that the user's directory will not be cluttered with useless files.

When the docms script is used, it might produce the results shown in the following example:

```
vizier> docms -e sample1.inp sample1.nro
docms: errors detected - see sample.nro.err
vizier> cat sample1.nro.err
docms: Unprocessed tags

   Input file:   sample1.inp
   Output file:  sample1.nro
   Processed on: Wed Jul 25 14:40:00 EDT 1990

Line    Tag or tags
----    -----------
13:     Writers using operating systems other than <REFERENCE>(u3)
15:     compatible with <REFERENCE>(u3) formatting tools.  If this
```

Examples of Using ULTRIX Tools A

The best way to gain facility with ULTRIX tools is to spend time working with them. This appendix shows examples of how the tools described in this manual and *The Little Gray Book: An ULTRIX Primer* can be used effectively. These examples are from real applications.

A.1 Using sed and grep to Create nroff Macros

This section describes how you can use the `sed` stream editor, the `sort` command, and the `grep` pattern-matching utility to extract information from a mail message and reconstruct it into a series of `nroff` text formatting macros. This example was provided by a small amateur chorus.

When the chorus is preparing a concert program, the personnel manager sends a mail message listing the participants to the program designer. This message is not suitable for formatting; it looks like this:

```
From howard@djinn Mon Dec 10 10:40:56 1990
Received: by vizier.chorus.org (5.57/dv.5.yp)
        id AA26651; Mon, 10 Dec 90 10:42:36 EDT
Date: Mon, 10 Dec 90 10:42:36 -0500
Message-Id: <9012111542.AA14867@vizier>
From: howard@djinn (Sandy Howard, Personnel Manager)
To: holland@vizier
Subject: List of concert participants
Status: R

# Performers in holiday concert
#
ogier@kublai            # John Ogier -- conductor
delores@minaret         # Delores Wilson -- accompanying
strangways@kaaba        # Wendy Strangways -- accompanying
#
scott@minaret           # Evelyn Scott -- soprano
peters@kaaba            # Kate Peters -- soprano
  .
  .
  .
bill@kublai             # Bill Hodges -- bass
french@minaret          # Marvin French -- bass
```

The mail message contains all the required information, but there is also much extraneous information, and what is needed is not in the correct order.

The program designer uses this information to create a formatted list of singers for the back page of the program. The list is formatted by a special set of `nroff` macros called the `music` macro package, designed for the chorus by the administrator of their system. The formatted list of personnel is shown at the end of this section. To create this list of performers from the mail message, the personnel manager uses the following shell script:

```
#! /bin/csh
        sed -f crew.sed crewmail > forces.tmp     1
        grep -v '^M' forces.tmp | sort +0 -1 +2 | \     2
        sed 's/A1/.Ss/;s/A2/.Sa/;s/A3/.St/;s/A4/.Sb/' > forces  3
        grep '^M' forces.tmp | sort +0 -1 +3 | \     4
        sed 's/M[56]/.MG/;/Conduct/r extra-management' >> forces
        rm -f forces.tmp     5
```

The callouts in this example indicate the following features:

1 This `sed` command edits a file called `crewmail` to create a temporary file called `forces.tmp`. The editing is done by the following `sed` script, called `crew.sed`:

```
1,/^$/d          6
/@/!d
s/^.*#[ TAB][ TAB]*//     7
/--[ TAB]*[Cc]onduct/{          8
  s/[ TAB]*--.*/"/
  s/^/M5 Conductor /
}
/--[ TAB]*[Aa]ccomp/{
  s/[ TAB]*--.*/"/
  s/^/M6 Accompaniment /
}
/--[ TAB]*[Ss]oprano/{
  s/[ TAB]*--.*//
  s/^/A1 /
}
  .
  .
  .
```

This `sed` script makes the following changes:

6 This command deletes the mail header by deleting everything from line 1 to the first blank line. The next command deletes all lines that do not contain an at sign (`@`), leaving just a list of the concert participants' mail addresses, their names, and the parts they have in the concert.

7 This line deletes the mail addresses by substituting a null string (no characters) for everything from the beginning of the line through any amount of white space after a number sign, leaving just the people's names and the parts they have in the concert. For example, the following line is from the mail message:

```
ogier@kublai            # John Ogier -- conductor
```

It is changed into this:

```
John Ogier -- conductor
```

8 This compound command finds any line containing "-- conduct" or "-- Conduct", deleting everything after the person's name and inserting a dummy character sequence (`M5`) and the word "Conductor". (The dummy sequence will be used for sorting the file, and it will later be replaced by a `.MG` macro call. The line illustrated in the previous step now looks like this:

```
M5 Conductor John Ogier
```

The remaining sections perform similar conversions for accompanists and singers, creating lines starting with `M6`, `A1`, `A2`, and so on, for accompanists, sopranos, altos, tenors, and basses. These dummy

sequences will later be replaced by .MG, .Ss, .Sa, .St, and .Sb macro calls.

2 This grep command finds the sopranos, altos, tenors, and basses; its -v option says to avoid all the lines starting with M. The remaining lines are piped to a sort command that sorts them first in order of the dummy sequences inserted by the earlier sed command and then in order by people's last names. The result is a file sorted by sections (sopranos first, then altos, and so on), and alphabetically within sections. (The backslash at the end of the line "hides" the new-line character from the shell, causing the shell to read the next line as a continuation of this one.)

3 The sorted output from the previous line is piped into another sed command that substitutes the correct macro names for the dummy strings, writing its output to a file named forces.

4 This grep command is like the first except that it passes only the lines beginning with M. Its output is also sorted and piped to a sed command. In addition to replacing the dummy strings, this sed command reads in a separate file after the "Conductor" .MG macro call. This additional file includes the names of administrative people. The output is concatenated onto the end of the forces file.

5 This rm command removes the temporary file, leaving forces as the final file.

The final forces file looks like this:

```
.Ss Carolyn Greenfield
.Ss Sandy Howard
  .
  .
  .
.Sb Bert Holland
.Sb Dan Pinkwater
.MG Conductor John Ogier
.MG "Administration and Music" Ruth Nelson
.MG "Administration and Membership" Sandy Howard
.MG Recording Dan Pinkwater
.MG Program Bert Holland
.MG "Arrangements and composition" Marvin French
.MG Accompaniment Wendy Strangways
.MG Accompaniment Delores Wilson
```

This file is one of several that go together to make up a complete formatted program. These various files are included in the proper places in a master document file with the nroff formatter's .so command; for example:

```
.so forces
```

The program is formatted using the following pipeline:

```
vizier> nroff -ms -music program | col > program.nro
```

The nroff command calls for both the ms and the music macro packages so that macros from both can be used.

The col command aligns columnar information; this function is needed because all the .Sa macros (alto singers) are listed after the .Ss macros (sopranos). The first instance of the .Sa macro backs up on the page so that the altos will be listed beside the sopranos instead of below them. The .St and .Sb macros work similarly. The final result looks like the following:

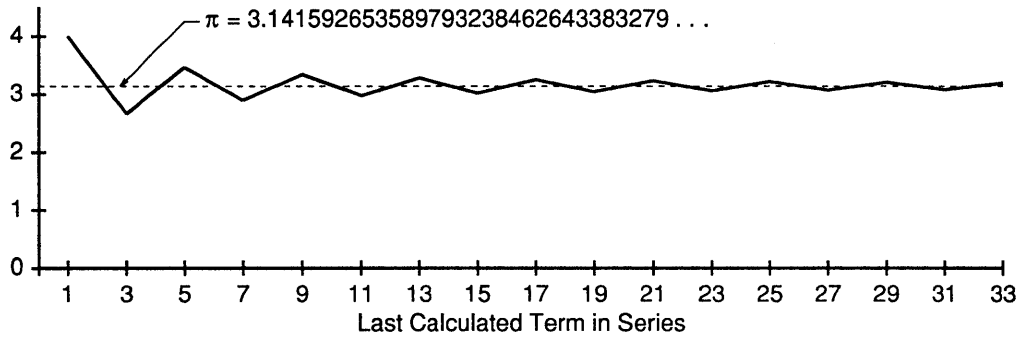
```
                    The Madrigal Singers

Sopranos                        Altos
   Carolyn Greenfield              Debbie Boland
   Sandy Howard                    Lisa Foster
   Diana Layton                    Ginny Hovey
   Ruth Nelson                     Kathy Kitchen
   Kate Peters                     Alice Logan
   Evelyn Scott                    Joanna Myron
                                   Wendy Strangways


Tenors                          Basses
   Tracy Gervaise                  Patrick Barry
   Jim Slattery                    Ron Evans
   Mike Weldon                     Marvin French
                                   Bill Hodges
                                   Bert Holland
                                   Dan Pinkwater

                    ------------------------------

Conductor:                          John Ogier
Administration and Music:           Ruth Nelson
Administration and Membership:      Sandy Howard
Recording:                          Dan Pinkwater
Program:                            Bert Holland
Arrangements and composition:       Marvin French
Accompaniment:                      Wendy Strangways
                                    Delores Wilson
```

A.2 Using the bc Calculator

This section shows how you can take advantage of the `bc` calculator's programmability to solve a repetitive problem quickly. The value of π (pi) can be approximated by several different methods; among the simplest is the following infinite series, described in 1674 by Gottfried Wilhelm Leibniz:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdots$$

The series produces the following curve:



Because this asymptotic series converges very slowly on the value of π (requiring several million terms to achieve merely 10-digit accuracy), it is of no practical use. For recreational purposes, however, you can calculate it using the `bc` calculator with the following function definition, where x is the value of the largest term to be used:

```
define p(x) {
  auto w,y,z
  w = 1
  for (y=1; y<=x; y=y+2) {
    z = z+w*1/y
    w = -w
  }
  return (z*4)
}
```

After creating this `bc` function as a file called `pi-series`, you can issue the following commands to produce the values used for the graph shown earlier in this section:

```
vizier> bc pi-series
scale=30
for (x=1; x<=33; x=x+2) (p(x))
4.000000000000000000000000000000
2.666666666666666666666666666668
3.466666666666666666666666666668
2.895238095238095238095238095240
3.339682539682539682539682539684
2.976046176046176046176046176048
3.283738483738483738483738483740
3.017071817071817071817071817076
3.252365934718875895346483581780
3.041839618929402211135957265992
3.232315809405592687326433456468
3.058402765927331817761216065164
3.218402765927331817761216065164
3.070254617779183669613067917016
3.208185652261942290302723089428
3.079153394197426161270465024912
3.200365515409547373391677146124
quit
vizier>
```

Tips and Tricks B

This appendix describes solutions for some of the more commonly encountered problems or difficulties in using the ULTRIX system:

- Handling files with problem names
- Using loops to rename files
- Creating a prompt with your working directory's name in it
- Redirecting standard output and standard error separately

Many of the techniques described here can be adapted and expanded to make other tasks easier. Some of these techniques make use of special shell syntax; for explanations of shell constructs you don't understand, see the `csh`(1) reference page.

B.1 Tricks with Files

This section describes several things you can do to manage your files more easily:

- Removing or renaming files whose names begin with a minus sign
- Removing or renaming files with odd characters in their names
- Renaming a series of files automatically
- Finding a file somewhere in your directories

B.1.1 Addressing Files Whose Names Begin with a Minus Sign

Many commands, such as `rm` and `mv`, cannot address a file whose name begins with a minus sign because the minus sign causes the command to try to interpret the name as an option instead of as an argument. The easiest solution to this problem is to supply a more complete pathname for the file; if it is your current directory, you can use *./–file* as the name. For example:

```
vizier> rm ./-sample1
```

This way to prevent interpretation of the minus sign works with most commands.

B.1.2 Addressing Files with Odd Characters in Their Names

It is possible to create a file whose name contains unusual characters that you cannot see or, in some cases, characters that you cannot even type. For example, a control character (CTRL/*x*) in a file name is normally represented by a question mark (?) in directory listings displayed on a terminal. In listings sent to other destinations, including commands in a pipeline, these characters are invisible unless you specify the `ls` command's `-q` option. For example:

```
vizier> ls sample*
sample?1        sample1
```

```
vizier> ls sample* | more
sample1
sample1
vizier> ls -q sample* | more
sample?1
sample1
```

In the first and third commands, the question mark in the first file's name represents a CTRL/A.

The usual solution to the problem of addressing files with names like this is to specify the filename argument as an expression unique to the file you want. For example:

```
vizier> rm -i sample*1
```

(Using the `-i` option causes `rm` to ask you for permission before removing each file.) But this does not always work; in the example shown here, this pattern matches both files, and the `rm` command will display both names identically.

There are advanced techniques for addressing this problem, but the simplest way is this:

1. Create a temporary directory using the `mkdir` command.
2. Move all the files from the directory containing the offending file into the temporary directory except the offending file or files. You can use individual file names or wildcards as needed, so long as you don't use a wildcard that will include the offending file or files.
3. Rename or remove the offending file or files, as follows:
   a. If there is only one offending file, remove or rename it by using a wildcard:

      ```
      vizier> mv * newname
      vizier> rm *
      ```

   b. If there are multiple offending files, or if you can't move all the non-problem files to a different directory, use this command sequence instead:

      ```
      vizier> set count=1
      vizier> foreach i (*)
      ? mv $i tempfile_$count
      ? @ count = $count + 1
      ? end
      ```

      This sequence renames each file using a unique temporary name that contains only normal characters, such as `tempfile_1`, `tempfile_2`, and so on. You can now examine each file, removing or renaming it as desired.
4. Move the other files back from the temporary directory and remove the empty temporary directory.

B.1.3 Renaming a Series of Files Automatically

Depending on the kind of renaming you want to do, there are several techniques for renaming multiple files with one command (actually a loop):

- Renaming `*.`*x* to `*.`*y*

  To change one file suffix to another, you can use the following loop:

```
vizier> foreach i (*.x)
? mv $i $i:r.y
? end
```

This technique also works if you want to remove the suffix. The `:r` modifier strips everything after the first period in the filename, returning the directory path and base name of the file. This modifier is similar to the `basename` command (described in Section 9.6), but the modifier works only on variables; you cannot use it to modify a file name you have entered explicitly.

There are several other modifiers available to perform other manipulations of file names. See the `csh`(1) reference page for more information.

- Changing part of the name; for example, renaming `fileA.*` to be `fileB.*`

  To change part of a name, you can use command substitution with a pipeline that includes the `sed` stream editor. For example:

```
vizier> foreach i (fileA.*)
? mv $i `echo $i | sed 's/fileA/fileB/'`
? end
```

You can use other variations of this basic looping technique to create virtually any combination of renaming that you need.

B.1.4 Finding a File Somewhere in your Directories

Sometimes you might have a file somewhere in your directory tree but not remember where it is. You can change from one directory to the next, issuing an `ls` command in each one until you find the file, but there is an easier way: Change to your home directory and use the `ls` command's `-R` option. This option causes `ls` to list the contents of your working directory and all subdirectories of your working directory. If you are in your home directory, the command lists the contents of all your directories.

This technique can only be used to give complete directory listings, however; you cannot include the name of the file you're looking for in the `ls` command. For example, you cannot do this:

```
vizier> ls -R report.txt
```

To find a file using its name, you can use the `find` command, as in this example:

```
vizier> find . -name 'report.txt' -print
```

The first argument for the `find` command is the name of the directory where the search is to start; the command searches that directory and all the subdirectories beneath it. In this example, the period means your working directory. (To find a file anywhere in your entire directory tree, use a tilde (~) instead of the period.) The `-name` option says to look for file names, and `'report.txt'` is the expression to be matched. The `-print` option is an action that says to print the names of all files matching the expression. The `find` command is very powerful; for more information about it, see the `find`(1) reference page.

B.2 Including Your Working Directory's Name in Your Prompt

You can make the shell include the pathname of your working directory in your prompt by inserting the two following commands in your `.login` file:

```
alias cd 'chdir \!* && set prompt="$cwd> "'
cd .
```

You must include the commands in the order shown here, or your prompt will not be right until the first time you enter a `cd` command. To trim the directory name so that only the actual directory name appears instead of the full path, use the following commands instead:

```
alias cd 'chdir \!* && set prompt="$cwd:t> "'
cd .
```

This example makes use of another of the modifiers mentioned in Section B.1.3.

B.3 Redirecting Standard Error and Standard Output Separately

Normally, standard error and standard output are assigned to the terminal. When you redirect standard output with a right angle bracket (>), standard error remains directed to the terminal. Occasionally, you might want to redirect standard error as well. For example, you might use a script for which you want to retain a log file containing any error messages.

You redirect standard error by using an angle bracket and an ampersand together as in this example:

```
vizier> ls sample2.* >& ls.log
vizier> more ls.log
No match.
```

However, when you use this symbol, standard error is redirected to the same file as standard output. Often this result isn't what you want; for example, if you are formatting a file with `nroff`, standard error messages will appear in the formatted file.

You can redirect standard error separately from standard output by enclosing part of your command in parentheses. For example:

```
vizier> (nroff -ms sample1 > sample1.nro) >& sample1.log
```

Enclosing a command in parentheses causes the shell to start a subshell to run the command. The redirection outside the parentheses redirects both standard output and standard error from the subshell, but because standard output is already redirected *inside* the subshell, only standard error messages appear in the error log file.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
The Big Gray Book: The Next Step with ULTRIX
AA-PBKNA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

_____

What do you like best about this manual? _____

_____

_____

What do you like least about this manual? _____

_____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line