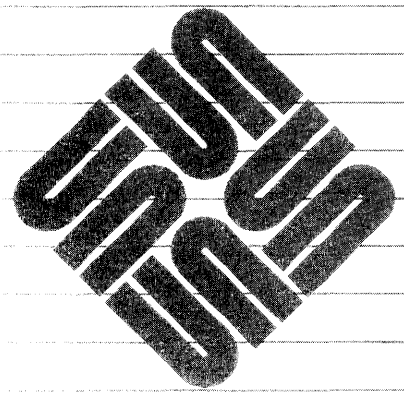
sun® microsystems

Using UNIX Text Utilities *on the* Sun Workstation®



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1322-03
Revision A of 21 January 1986

Credits and Acknowledgements

Material in this *Using UNIX Text Utilities on the Sun Workstation* comes from: *Awk — A Pattern Scanning and Processing Language*, Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, Bell Laboratories, Murray Hill, New Jersey; *Advanced Editing on UNIX*, Brian W. Kernighan, Bell Laboratories, Murray Hill, New Jersey; *Sed — a Non-Interactive Text Editor*, Lee. E. McMahon, Bell Laboratories, Murray Hill, New Jersey; *Introducing the UNIX System*, Henry McGilton, Rachel Morgan, McGraw-Hill Book Company, 1983. *A Practical Guide to the UNIX System*, Mark G.Sobell, Benjamin/Cummings Publishing Company, Inc., 1984. These materials are gratefully acknowledged.

Contents

Tables

Preface

Using UNIX Text Utilities on the Sun Workstation provides reference information for utilities useful with text files. We assume you are familiar with a terminal keyboard and the Sun system. If you are not, see the *Beginner's Guide to the Sun Workstation* for information on the basics, like logging in and the Sun file system. If you are not familiar with a text editor or document processor, read "An Introduction to Text Editing" in *Editing Text Files on the Sun Workstation* and "An Introduction to Document Preparation" in *Formatting Documents on the Sun Workstation* for descriptions of the basic concepts and simple examples that you can try. Finally, we assume that you are using a Sun Workstation, although specific terminal information is also provided.

For additional details on Sun system commands and programs, see the *Commands Reference Manual for the Sun Workstation*.

Summary of Contents

This manual is divided into four sections based on the type of operation you want to perform. The four sections are titled "Comparing Files", "Searching Through Files", "Modifying Files", and "Printing Files". The contents of each section are summarized here:

1. *Comparing Files* — This first section describes the commands `cmp`, `comm`, `diff`, `diff3`, `join`, `look`, and `uniq`.
2. *Searching Through Files* — This second section covers the commands `awk`, `grep`, `rev`, `sed`, and `wc`.
3. *Modifying Files* — This third section explains how to use the commands `colrm`, `compact`, `expand`, `fold`, `sort`, `split`, `tr`, and `tsort`.
4. *Printing Files* — This fourth section clarifies the printing commands `lpq`, `lpr`, `lprm`, and `pr`.

Conventions Used in This Manual

Throughout this manual we use

```
hostname%
```

as the prompt to which you type system commands. **`Bold face typewriter font`** indicates commands that you type in exactly as printed on the page of this manual. `Regular typewriter font` represents what the system prints out to your screen. Typewriter font also specifies Sun system command names (program names) and illustrates source code listings. *Italics*

indicates general arguments or parameters that you should replace with a specific word or string. We also occasionally use italics to emphasize important terms.

1

Comparing Files

1

Comparing Files

Occasionally you want to know whether two files are identical, or if they are not, what the differences are. There are several different UNIX† text utilities for comparing the contents of files. You can choose the command best for the task at hand based on what kind of information it conveys to you. Most of the commands issue no output if the files are the same. Some return terse output stating barely more than the fact that the files differ. Others give a more complete summary of how the files differ and how you would have to modify one file to match the other(s).

The command `cmp` is an example of a command that issues terse output. At most, `cmp` prints the byte and line number where the files differ. Two other functions for directly comparing files are `diff` and `comm`. `comm` compares two files, putting the comparison information into three different columns: column one lists lines only in *file1*, column two lists lines only in *file2*, and column three lists lines common to both files. `diff` compares files and also directories. A special version of `diff`, `diff3`, also compares three files, identifying the differing contents with special flags.

The relational database operator `join` compares a specific field or fields in two files. Each time `join` finds the compared fields in the two files identical, it produces one output line.

For comparing adjacent lines in a single file, UNIX provides the command `uniq`. `uniq` can be made to merely report the repeated lines or to count them or to remove all but the first occurrence.

1.1. cmp

The command `cmp` is for comparing two files. The synopsis of the `cmp` command is:

```
hostname% cmp [-l] [-s] file1 file2
hostname%
```

`cmp` compares *file1* and *file2*. If *file1* is the standard input ('–'), `cmp` reads from the standard input. Under default options, `cmp` makes no comment if the files are the same. If the files differ, `cmp` announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

† UNIX is a trademark of AT&T Bell Laboratories.

The options available with cmp are:

-l Print the byte number (decimal) and the differing bytes (octal) for each difference.

-s Print nothing for differing files; return codes only.

1.2. comm

The comm command prints lines that are common to two files. comm reads *file1* and *file2*, which should be ordered in ASCII collating sequence, but at least ir the *same* order, and produces a three-column output:

Column 1	Column 2	Column 3
lines only in *file1*	lines only in *file2*	lines in both files

The synopsis of the comm command is:

```
hostname% comm [-[123]] file1 file2
hostname%
```

As an example of the comm command's output, consider these files:

```
hostname% cat all
Aaron
Bruce
Dave
Elaine
Greg
Joe
Jon
Kevin
Larry G
Larry K
Linda
Mary
Mike B
Mike F
Niel
Pam
Randy
Sid
Tad
Tom
Wanda
hostname%
```

```
hostname% cat women
Christy
Cyndi
Elaine
Gale
Jeanette
Julia
Katherine
Katy
Linda
Lori
Mary
Pam
Pat
Patti
Rose Marie
Susan
Wanda
hostname%
```

Here is the output of `comm`. The three columns overlap making output from files with long lines a little difficult to read.

```
hostname% comm women all
        Aaron
        Bruce
Christy
Cyndi
        Dave
                Elaine
Gale
        Greg
Jeanette
        Joe
        Jon
Julia
Katherine
Katy
        Kevin
        Larry G
        Larry K
                Linda
Lori
                Mary
        Mike B
        Mike F
        Niel
                Pam
Pat
Patti
        Randy
Rose Marie
        Sid
Susan
        Tad
        Tom
                Wanda
hostname%
```

The filename '–' means the standard input. The flags 1, 2, or 3, suppress printing of the corresponding column. Thus:

```
hostname% comm -12
hostname%
```

prints only the lines common to the two files, and

```
hostname% comm -23
hostname%
```

prints only lines in the first file, but not in the second. (`comm -123` does nothing).

1.3. diff

For summarizing the differences between two files or directories, `diff` is the appropriate tool. To use the `diff` command, you would follow one of these models:

```
hostname% diff [-cefh] [-b] file1 file2
hostname%

hostname% diff [-Dstring] [-b] file1 file2
hostname%

hostname% diff [-l] [-r] [-s] [] [-Sname] [-cefh] [-b] dir1 dir2
hostname%
```

`diff` is a differential file comparator. When run on regular files, and when comparing text files that differ during directory comparison (see the notes below on comparing directories), `diff` tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, `diff` finds a smallest sufficient set of file differences. If neither *file1* nor *file2* is a directory, either may be given as '–', in which case the standard input is used. If *file1* is a directory, a file in that directory whose file-name is the same as the file-name of *file2* is used (and vice versa).

There are several options for output format; the default output format contains lines of these forms:

n1 a *n3,n4*
n1,n2 d *n3*
n1,n2 c *n3,n4*

These lines resemble `ed` commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging 'a' for 'd' and reading backward you can see how to convert *file2* into *file1*. As in `ed`, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these specification lines come all the lines that are affected in the first file flagged by the character '<', then all the lines that are affected in the second file flagged by the '>' character.

If both arguments are directories, `diff` sorts the contents of the directories by name, and then runs the regular file `diff` program as described above on text files that are different. Binary files that differ, common subdirectories, and files that appear in only one directory are listed.

diff — First Form

To produce a script of append (a), change (c), and delete (d) commands for the editor `ed`, which will recreate *file2* from *file1*, use the first form of `diff` with the option `-e`.

Extra commands are added to the output when comparing directories with `diff -e`, so that the result is a Bourne shell (sh) script for converting text files common to the two directories from their state in *dir1* to their state in *dir2*.

To produce a script similar to that using `-e`, but in the opposite order, that is, to recreate *file1* from *file2*, use `diff -f`. The script generated with the `-f` option is not useful with `ed`, however.

To surround the specification lines the simplest use of `diff` puts out with some lines of context, use `diff -c`. The default is to present three lines of context. To change this (to 10, for example), add 10 to the `-c` option (-c10). With the `-c` option, the output format is slightly different from other `diff` output. It begins by identifying the files involved and the dates they were created. Then each change is separated by a line with a dozen stars (*). The lines removed from *file1* are marked with '–'; those added to *file2* are marked '+'. Lines that are changed from one file to the other are marked in both files with '!'.

If you know you've only made small changes to the files you are comparing, and you want to speed up the time `diff` takes to work, you can use `diff -h`. This command only does a fast, half-hearted job. `diff -h` works only when changed stretches are short and well-separated, but does work on files of unlimited length.

Except for the `-b` option, which my be given with any of the others, the options `-c`, `-e`, `-f`, and `-h` are mutually exclusive.

`diff` — Second Form

To create a merged version of *file1* and *file2* on the standard output with C preprocessor controls included, use the second form of `diff` with the option `-D`*string*. Compiling the result without defining *string* is equivalent to compiling *file1*, while compiling the result with *string* defined will yield *file2*.

If you want `diff` to ignore trailing blanks (spaces and tabs), use the option `-b`. Other strings of blanks compare equal. The way `diff` works, when it compares directories with the `-b` option specified, `diff` first compares the files (as in `cmp`), and then decides to run the `diff` algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical, because the only differences are insignificant blank string differences.

`diff` — Third Form

When comparing directories, you might be interested in several different things. If `diff` puts out a lot of output, you probably want to use the `-l` option (for long output). Each text file `diff` is piped through the program `pr` to paginate it, (see "Printing Files" later in this manual). Other differences are remembered and summarized after all text file differences are reported.

To compare directories and subdirectories, use the `-r` option. `-r` applies `diff` recursively to common subdirectories encountered.

Since `diff` ordinarily only outputs information on files and directories that differ, if a file or several files are identical in directories you are comparing, you won't see the identical files listed in the output. The `-s` option reports files that are the same, in addition to the usual `diff` output, which are otherwise not mentioned.

Here are two directories, *macros* and *new*. For this example, here are lists of their contents.

```
hostname% ls macros
Makefile                making.index.msun       summary.msun
SunMacros.msun          mechanisms.msun         test.tr
contents.pic            mmemo.7                 text.effects.msun
contentsfile.msun       model.makefile.msun     troff.msun
document.styles.msun    process.pic
intro.msun              structures.msun
hostname%

hostname% ls new
Makefile                making.index.msun       summary.msun
SunMacros.msun          mechanisms.msun         test.tr
contents.pic            mmemo.7                 text.effects.msun
contentsfile.msun       model.makefile.msun     troff.msun
document.styles.msun    process.pic
intro.msun              structures.msun
hostname%
```

Right now these two directories are identical. The output of `diff` for these two directories *macros* and *new*, if there are no differences is:

```
hostname% diff macros new
hostname%
```

The normal output is nothing, no response. Now if we edit some files and remove some others in the directory *new*, leaving the files like this:

```
hostname% ls macros new
macros:
Makefile                making.index.msun       summary.msun
SunMacros.msun          mechanisms.msun         test.tr
contents.pic            mmemo.7                 text.effects.msun
contentsfile.msun       model.makefile.msun     troff.msun
document.styles.msun    process.pic
intro.msun              structures.msun

new:
Makefile                intro.msun              structures.msun
SunMacros.msun          making.index.msun       summary.msun
contents.pic            mechanisms.msun         text.effects.msun
document.styles.msun    model.makefile.msun     troff.msun
```

The regular `diff` output looks like this:

```
hostname% diff macros new
diff macros/Makefile new/Makefile
7c7
< FORMATTER = /usr/local/iroff
---
> FORMATTER = /usr/doctools/bin/troff
Only in macros: contentsfile.msun
diff macros/intro.msun new/intro.msun
0a1
> .LP
6,10c7,9
< Document preparation at Sun Microsystems relies on variations of the
< .I troff
< text formatter as the underlying mechanism for turning your wishes into
< printed words and outlines on paper.  Using
< .I troff
---
> Document preparation at Sun Microsystems relies on variations of the
> troff text formatter as the underlying mechanism for turning your wishes
> into printed words and outlines on paper.  Using troff
Only in macros: mmemo.7
diff macros/model.makefile.msun new/model.makefile.msun
3,7c3
< The
< .I Makefile
< below is the
< .I Makefile
< used to actually make this document:
---
> The Makefile below is the Makefile used to actually make this document:
Only in macros: process.pic
Only in macros: test.tr
hostname%
```

The output of `diff` is cryptic in true UNIX fashion. But if you look carefully at the specification lines and the direction of the angle brackets, you can decipher the results accurately.

To get a more complete picture of how the two directories compare, you might want to know which files are identical and which files exist only in one directory. For this, you use `diff -s`. The `diff -s` output from our example above looks like this:

```
hostname% diff -s macros new
diff -s macros/Makefile new/Makefile
7c7
< FORMATTER = /usr/local/iroff
---
> FORMATTER = /usr/doctools/bin/troff
Files macros/SunMacros.msun and new/SunMacros.msun are identical
Files macros/contents.pic and new/contents.pic are identical
Only in macros: contentsfile.msun
Files macros/document.styles.msun and new/document.styles.msun are identical
diff -s macros/intro.msun new/intro.msun
0a1
> .LP
6,10c7,9
< Document preparation at Sun Microsystems relies on variations of the
< .I troff
< text formatter as the underlying mechanism for turning your wishes into
< printed words and outlines on paper.  Using
< .I troff
---
> Document preparation at Sun Microsystems relies on variations of the
> troff text formatter as the underlying mechanism for turning your wishes
> into printed words and outlines on paper.  Using troff
Files macros/making.index.msun and new/making.index.msun are identical
Files macros/mechanisms.msun and new/mechanisms.msun are identical
Only in macros: mmemo.7
diff -s macros/model.makefile.msun new/model.makefile.msun
3,7c3
< The
< .I Makefile
< below is the
< .I Makefile
< used to actually make this document:
---
> The Makefile below is the Makefile used to actually make this document:
Only in macros: process.pic
Files macros/structures.msun and new/structures.msun are identical
Files macros/summary.msun and new/summary.msun are identical
Only in macros: test.tr
Files macros/text.effects.msun and new/text.effects.msun are identical
Files macros/troff.msun and new/troff.msun are identical
hostname%
```

To compare two directories beginning somewhere in the middle of the directories, use the option `diff` **–S***filename*. *filename* is one of the files in one of the directories you are comparing. The syntax for this command is

```
hostname% diff –Sfilename dir1 dir2
hostname%
```

For example, comparing the two directories from the example above, and beginning with the file `model.makefile.msun`:

```
hostname% diff -Smodel.makefile.msun macros new
diff macros/model.makefile.msun new/model.makefile.msun
3,7c3
< The
< .I Makefile
< below is the
< .I Makefile
< used to actually make this document:
---
> The Makefile below is the Makefile used to actually make this document:
Only in macros: process.pic
Only in macros: test.tr
hostname%
```

1.4. diff3

If you have three versions of a file that you want to compare at once, use the `diff3` command. The synopsis for the `diff3` command is:

```
hostname% diff3 [-ex3] file1 file2 file3
hostname%
```

`diff3` compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

`====` all three files differ

`====1` *file1* is different

`====2` *file2* is different

`====3` *file3* is different

The type of change required to convert a given range of a given file to a range in some other file is indicated in one of these ways:

f : *n1* a

Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.

f : *n1* , *n2* c

Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a `c` indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the `-e` option, `diff3` publishes a script for the editor `ed` that will incorporate into *file1* all changes between *file2* and *file3*, (that is, the changes that normally would be flagged `====` and `====3`). Option `-x` produces a script to incorporate only changes flagged `====`. Option `-3` produces a script to

incorporate only changes flagged ====3. The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

Note: Text lines that consist of a single dot ('.') will defeat the -e option.

Table 1-1 `diff3` *Option Summary*

OPTIONS	
-e	Publish a script for the editor `ed` that will incorporate into *file1* all changes between *file2* and *file3*, (that is, the changes that normally would be flagged ==== and ====3).
-x	Produce a script for `ed` to incorporate only changes flagged ====.
-3	Produce a script for `ed` to incorporate only changes flagged ====3.

1.5. `join`

To compare two files of database information and output a *join* of two fields, there is a UNIX text utility `join`. `join` is a relational database operator. The synopsis of the command is:

```
hostname% join [-an] [-e string] [-j[1|2] m] [-o list] [-tc] file1 file2
hostname%
```

The program `join` forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is '-', the standard input is used.

file1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*. Fields are separated by blanks, tabs or newlines. Multiple separators count as one, and leading separators are discarded.

Note: With default field separation, the collating sequence is that of `sort -b`. Using the `join -t`, the sequence is that of a plain sort.

Table 1-2 `join` Option Summary

OPTIONS	
`-a`*n*	The parameter *n* can be one of the values: 1 produce a line for each unpairable line in *file1*. 2 produce a line for each unpairable line in *file2*. 3 produce a line for each unpairable line in both *file1* and *file2*. in addition to producing the normal output.
`-e` *string*	Replace empty output fields with *string*.
`-j[1|2]` *m*	Join on the *m*th field of file *n*, where *n* is 1 or 2. If *n* is missing, use the *m*th field in each file. Note that `join` counts fields from 1 instead of 0 like `sort` does.
`-o` *list*	Each output line comprises the fields specifed in *list*, each element of which has the form *n.m*, where *n* is a file number and *m* is a field number.
`-tc`	Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

1.6. `uniq`

If you want to check your input file for repeated lines, use `uniq` `uniq` reports repeated lines in a file.

The synopsis of the `uniq` command is:

```
hostname% uniq [-udc [+n] [-n] ] [input file [output file] ]
hostname%
```

`uniq` reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder of the text (no repeated lines) is written in the output file. Note that repeated lines must be adjacent in order to be found.

Normally, the lines in the input file that were not repeated and the first occurrence of the lines that were repeated forms the output. If you want to isolate either of these functions, you can specify either the `-u` or the `-d` option. `uniq -u` copies only the lines *not* repeated in the original file to the output file. `uniq -d` writes one copy of just the repeated lines to the output file.

In case you are interested in knowing how many occurrences of a given line appear in the input file, you can use the option `uniq -c`. With the `-c` option, you get first the number of occurrences, then the output in default format (all of the unique lines and no adjacent repeated lines).

There is also an option to compare the latter parts of lines rather than entire lines. The *n* arguments specify skipping an initial portion of each line in the comparison:

-n The first *n* fields, together with any blanks before each, are ignored. A field is c string of non-space, non-tab characters separated by tabs and spaces from its ne

+n The first *n* characters are ignored. Fields are skipped before characters.

Table 1-3 `uniq` *Option Summary*

OPTIONS	
-u	Copy only those lines that are *not* repeated in the original file.
-d	Write one copy of just the repeated lines.
-c	Supersedes -u and -d and generates an output report in default style but with each line preceded by a count of the number of times it occurred.
-n	The first *n* fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.
+n	The first *n* characters are ignored. Fields are skipped before characters.

2

Searching Through Files

2

Searching Through Files

Searching through files to find a string or operate on that string or both is a useful facility to have. UNIX provides several different text utilities that approach the problem from several different angles. The first one covered here is the program called `awk`.

`awk` searches for a pattern (a string of characters) in a file and performs a specified action on the pattern. `awk` is actually a programming language so it is very flexible.

There is also a utility for searching for patterns and displaying them (usually on the standard output). This program, called `grep`, doesn't perform any operations on the pattern. To search for a pattern in a file or files with `grep` and perform an operation on the pattern, you would need to pipe the output from `grep` to another program. If you specify more than one input file for `grep` to search, `grep` precedes each line that matches the pattern with the name of the file that it came from.

There are two variations on `grep` that have similar functions: `egrep` and `fgrep`. `egrep` finds full regular expressions and `fgrep` searches only for fixed strings.

For looking up strings of characters quickly in a dictionary file like `/usr/dict/words`, UNIX provides the utility `look`. `look` behaves just like `grep` but unless you give `look` a different input file, it searches through a specific sorted file and prints out all lines that begin with *string*.

To search through a file and reverse the order of characters on every line, use the program `rev`.

UNIX provides a stream editor called `sed` that you can use to search through a file and edit it temporarily. `sed` is particularly useful for transient changes. `sed` commands can reside in a file or can be given on the command line. `sed` edits a file non-interactively and prints out the edited lines on the standard output. The actual file remains unchanged and the changes are not saved permanently unless you redirect the `sed` output to a file.

The last text utility we present here, `wc`, searches through your input file and counts the number of lines, words, and characters.

2.1. Pattern Scanning and Processing with `awk`

`awk` is a utility program that you can program in varying degrees of complexity. `awk`'s basic operation is to search a set of files for patterns based on *selection criteria*, and to perform specified actions on lines or groups of lines which contain those patterns. Selection criteria can be text patterns or *regular expressions*. `awk` makes data selection, transformation operations, information retrieval and text manipulation easy to state and to perform.

Basic `awk` operation is to scan a set of input lines in order, searching for lines which match any of a set of patterns that you have specified. You can specify an action to be performed on each line that matches the pattern.

`awk` patterns may include arbitrary Boolean combinations of regular expressions and of relational and arithmetic operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, `if-else`, `while`, `for` statements, and multiple output streams.

If you are familiar with the `grep` utility (see the *Commands Reference Manual for the Sun Workstation*), you will recognize the approach, although in `awk`, the patterns may be more general than in `grep`, and the actions allowed are more involved than merely displaying the matching line.

As some simple examples to give you the idea, consider a short file called *sample*, which contains some identifying numbers and system names:

```
125.1303    krypton loghost
125.0x0733  window
125.1313    core
125.19      haley
```

If you want to display the second and first columns of information in that order, use the `awk` program:

```
hostname% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
haley 125.19
```

This is good for reversing columns of tabular material for example. The next program shows all input lines with an a, b, or c in the second field.

```
hostname% awk '$2 ~ /a|b|c/' sample
125.1313    core
125.19      haley
```

The material in this chapter is derived from *Awk — A Pattern Scanning and Processing Language*, A. Aho, B.W. Kernighan, P. Weinberger, Bell Laboratories, Murray Hill, New Jersey.

Using awk

The general format for using awk follows. You execute the awk commands in a string that we'll call *program* on the set of named *files*:

```
hostname% awk program files
```

For example, to display all input lines whose length exceeds 13 characters, use the program:

```
hostname% awk 'length > 13' sample
125.1303    krypton loghost
125.0x0733  window
hostname%
```

In the above example, the *program* compares the length of the *sample* file lines to the number 13 and displays lines longer than 13 characters.

awk usually takes its program as the first argument. To take a program from a file instead, use the -f (file) option. For example, you can put the same statement in a file called *howlong*, and execute it on *sample* with:

```
hostname% awk -f howlong hosts
125.1303    krypton loghost
125.0x0733  window
```

You can also execute awk on the standard input if there are no files. Put single quotes around the awk program because the shell duplicates most of awk's special characters.

Program Structure

A program can consist of just an action to be performed on all lines in a file, as in the *howlong* example above. It can also contain a pattern that specifies the lines for the action to operate on. This pattern/action order is represented in awk notation by:

```
pattern   {action }
```

In other words, each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. Thus a line which matches several patterns can be printed several times. If there is no pattern for an action, the action is performed on every input line. A line which doesn't match any pattern is ignored. Since patterns and actions are both optional, you must enclose actions in braces ({action}) to distinguish them from patterns. See more about patterns in the "Specifying Patterns" section later in this chapter.

Records and Fields

awk input is divided into *records* terminated by a *record separator*. The default record separator is a newline, so by default awk processes its input a line at a time. The number of the current record is available in a variable named NR.

Each input record is considered to be divided into *fields*. Fields are separated by *field separators*, normally blanks or tabs, but you can change the input field separator, as described in the "Field Variables" section later in this chapter. Fields are referred to as $X where $1 is the first field, $2 the second, and so

on as shown above. `$0` is the whole input record itself. Fields may be assigned to. The number of fields in the current record is available in a variable named `NF`.

The variables `FS` and `RS` refer to the input field and record separators; you can change them at any time to any single character. You may also use the optional command-line argument *–Fc* to set `FS` to any character *c*.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable FILENAME contains the name of the current input file.

Displaying Text

The simplest action is to display (or *print*) some or all of a record with the `awk` command `print`. `print` copies the input to the output intact. An action without a pattern is executed for all lines. To display each record of the *sample* file, use:

```
hostname% awk '{print}' sample
125.1303    krypton loghost
125.0x0733  window
125.1313    core
125.19      haley
hostname%
```

Remember to put single quotes around the `awk` program as we show here.

More useful than the above example is to print a field or fields from each record. For instance, to display the first two fields in reverse order, type:

```
hostname% awk '{print $2, $1}' sample
krypton 125.1303
window 125.0x0733
core 125.1313
hostname%
```

Items separated by a comma in the `print` statement are separated by the current output field separator when output. Items not separated by commas are concatenated, so to run the first and second fields together, type:

```
hostname% awk '{print $1 $2}' sample
125.1303krypton
125.0x0733window
125.1313core
125.19haley
hostname%
```

You can use the predefined variables `NF` and `NR`; for example, to print each record preceded by the record number and the number of fields, use:

```
hostname% awk '{ print NR, NF, $0 }' sample
1 3 125.1303    krypton loghost
2 2 125.0x0733  window
3 2 125.1313    core
4 2 125.19      haley
hostname%
```

You may divert output to multiple files; the program:

```
hostname% awk '{print $1 >"foo1"; print $2 >"foo2"}' filename
```

writes the first field, `$1`, on the file *foo1*, and the second field on file *foo2*. You can also use the >> notation; to append the output to the file *foo* for example, say:

```
hostname% awk '{print $1 >>"foo"}' filename
```

In each case, the output files are created if necessary. The filename can be a variable or a field as well as a constant. For example, to use the contents of field 2 as a filename, type:

```
hostname% awk '{print $1 >$2}' filename
hostname%
```

This program prints the contents of field 1 of *filename* on field 2. If you run this on our *sample* file, four new files are created. There is a limit of 10 output files.

Similarly, you can pipe output into another process. For instance, to mail the output of an `awk` program to *susan*, use:

```
hostname% awk '{ print NR, NF, $0 }' sample | mail susan
```

(See the *Mail User's Guide* in the *Beginner's Guide to the Sun Workstation* for details on `mail`.)

To change the current output field separator and output record separator, use the variables `OFS` and `ORS`. The output record separator is appended to the output of the `print` statement.

`awk` also provides the `printf` statement for output formatting. To format the expressions in the list according to the specification in *format* and print them, use:

```
printf format, expr, expr, ...
```

To print `$1` as a floating point number eight digits wide, with two after the decimal point, and `$2` as a 10-digit long decimal number, followed by a newline, use:

```
hostname% awk '{printf("%8.2f  %10ld\n",$1,$2)}' filename
```

Notice that you have to specifically insert spaces or tab characters by enclosing them in quoted strings. Otherwise, the output appears all scrunched together. The version of `printf` is identical to that provided in the C Standard I/O library (see *printf* in *C Library Standard I/O* (3S) in the *System Interface Manual for the Sun Workstation*).

Specifying Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. You may use a variety of expressions as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary Boolean combinations of these.

BEGIN and END

awk has two built-in patterns, BEGIN and END. BEGIN matches the beginning of the input, before the first record is read. The pattern END matches the end of the input, after the last record has been processed. BEGIN and END thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by:

```
BEGIN    { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by:

```
END  { print NR }
```

If BEGIN is present, it must be the first pattern; END must be the last if used.

Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete awk program which displays all lines which contain any occurrence of the name 'smith'. If a line contains 'smith' as part of a larger word, it is also displayed. Suppose you have a file *testfile* that contains:

```
summertime
smith
blacksmithing
Smithsonian
hammersmith
```

If you use awk on it, the display is:

```
hostname% awk /smith/ testfile
smith
blacksmithing
hammersmith
```

awk regular expressions include the regular expression forms found in the text editor ed and in grep (see the *Commands Reference Manual for the Sun Workstation*). In addition, awk uses parentheses for grouping, | for alternatives, + for 'one or more', and ? for 'zero or one', all as in lex. Character classes may be abbreviated. For example:

```
/[a-zA-Z0-9]/
```

is the set of all letters and digits. As an example, to display all lines which contain any of the names 'Adams,' 'West' or 'Smith,' whether capitalized or not, use:

```
'/[Aa]dams|[Ww]est|[Ss]mith/'
```

Enclose regular expressions (with the extensions listed above) in slashes, just as in `ed` and `sed`. For example:

```
hostname% awk '/[Ss]mith/' testfile
smith
blacksmithing
Smithsonian
hammersmith
```

finds both 'smith' and 'Smith'.

Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\/.*\//
```

which matches any string of characters enclosed in slashes.

Use the operators `~` and `!~` to find if any field or variable matches a regular expression (or does not match it). The program

```
$1 ~ /[sS]mith/
```

displays all lines where the first field matches 'smith' or 'Smith.' Notice that this will also match 'blacksmithing', 'Smithsonian', and so on. To restrict it to exactly [sS]mith, use:

```
hostname% awk '$1 ~ /^[sS]mith$/' testfile
smith
hostname%
```

The caret `^` refers to the beginning of a line or field; the dollar sign `$` refers to the end.

Relational Expressions

An `awk` pattern can be a relational expression involving the usual relational and arithmetic operators `<`, `<=`, `==`, `!=`, `>=`, and `>`, the same as those in C. An example is:

```
'$2 > $1 + 100'
```

which selects lines where the second field is at least 100 greater than the first field.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
hostname% awk '$1 >= "s"' testfile
smith
```

selects lines that begin with an 's', 't', 'u', etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

performs a string comparison between field 1 and field 2.

Combinations of Patterns

A pattern can be any Boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example, to select lines where the first field begins with 's', but is not 'smith', use:

```
hostname% awk '$1 >= "s" && $1 < "t" && $1 != "smith"' testfile
summertime
```

&& and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

The program:

```
$1 !=prev {print; prev=$1}
```

displays all lines in which the first field is different from the previous first field.

Pattern Ranges

The pattern that selects an action may also consist of two patterns separated by a comma, as in

```
pattern1, pattern2  { ... }
```

In this case, the action is performed for each line between an occurrence of *pattern1* and the next occurrence of *pattern2* inclusive. For example, to display all lines between the strings 'sum' and 'black', use:

```
hostname% awk '/sum/, /black/' testfile
summertime
smith
blacksmithing
hostname%
```

while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

Actions

An awk action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

Assignments, Variables, and Expressions

The simplest action is an *assignment*. For example, you can assign 1 to the *variable x*:

```
x = 1
```

The '1' is a simple expression. awk variables can take on numeric (floating point) or string values according to context. In

```
x = 1
```

x is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance, to assign 7 to *x*, use:

```
x = "3" + "4"
```

Strings that cannot be interpreted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables other than built-ins are initialized to the null string, which has numerical value zero; this eliminates the need for most BEGIN sections. For example, the sums of the first two fields can be computed by:

```
    { s1 += $1; s2 += $2 }
END { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are +, −, *, /, and % (mod). For example:

```
NF % 2 == 0
```

displays lines with an even number of fields. To display all lines with an even number of fields, use:

```
NF % 2 == 0
```

The C increment ++ and decrement −− operators are also available, and so are the assignment operators +=, −=, *=, /=, and %=.

An awk pattern can be a *conditional expression* as well as a simple expression as in the 'x = 1' assignment above. The operators listed above may all be used in expressions. An awk program with a conditional expression specifies conditional selection based on properties of the individual fields in the record.

Field Variables

Fields in awk share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to.

To replace the first field of each line by its logarithm, say:

```
{ $1 = log($1); print }
```

Thus you can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
    $3 = "too big"
  print
}
```

which replaces the third field by 'too big' when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is considered numeric or string depends on context; fields are treated as strings in ambiguous cases like:

```
if ($1 == $2) ...
```

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields. To split the string 's' into 'array[1]' ..., 'array[n]', use:

```
n = split(s, array, sep)
```

This returns the number of elements found. If the `sep` argument is provided, it is used as the field separator; otherwise `FS` is used as the separator.

String Concatenation

Strings may be concatenated. For example:

```
length($1 $2 $3)
```

returns the length of the first three fields. Or in a `print` statement,

```
print $1 " is " $2
```

prints the two fields separated by ' is '. Variables and numeric expressions may also appear in concatenations.

Built-in Functions

`awk` provides several *built-in* functions.

`length` Function

The `length` function computes the length of a string of characters. This program shows each record, preceded by its length:

```
hostname% awk '{print length, $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
hostname%
```

`length` by itself is a 'pseudo-variable' that yields the length of the current record; `length(argument)` is a function which yields the length of its argument, as in the equivalent:

```
hostname% awk '{print length($0), $0}' testfile
10 summertime
5 smith
13 blacksmithing
11 Smithsonian
11 hammersmith
```

The argument may be any expression.

`awk` also provides the arithmetic functions `sqrt`, `log`, `exp`, and `int`, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

displays lines whose length is less than 10 or greater than 20.

substring Function

The function `substr(s, m, n)` produces the substring of *s* that begins at position *m* (origin 1) and is at most *n* characters long. If *n* is omitted, the substring goes to the end of *s*.

index Function

The function `index(s1, s2)` returns the position where the string *s2* occurs in *s1*, or zero if it does not.

sprintf Function

The function `sprintf(`*f, e1, e2, ...*`)` produces the value of the expressions *e1*, *e2*, and so on, in the `printf` format specified by *f*. Thus, for example, to set *x* to the string produced by formatting the values of `$1` and `$2`, use:

```
x = sprintf("%8.2f %10ld", $1, $2)
```

Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have *any* non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the *NR -th* element of the array *x*. In fact, it is possible in principle though perhaps slow to process the entire input in a random order with the `awk` program

```
     { x[NR] = $0 }
END  { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives `awk` a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like 'apple', 'orange', etc. Then the program

```
/apple/      { x["apple"]++ }
/orange/     { x["orange"]++ }
END      { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

Flow-of-Control Statements

`awk` provides the basic flow-of-control statements `if-else`, `while`, `for`, and statement grouping with braces, as in C. We showed the `if` statement in the "Field Variables" section without describing it. The condition in parentheses is evaluated; if it is true, the statement following the `if` is done. The `else` part is optional.

The `while` statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
    print $i
    ++i
}
```

The `for` statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
    print $i
```

does the same job as the `while` statement above.

There is an alternate form of the `for` statement which is suited for accessing the elements of an associative array:

```
for (i in array)
    statement
```

does *statement* with *i* set in turn to each element of *array*. The elements are accessed in an apparently random order. Chaos will ensue if *i* is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an `if`, `while` or `for` can include relational operators like <, <=, >, >=, == ('is equal to'), and != ('not equal to'); regular expression matches with the match operators ~ and !~; the logical operators ||, &&, and !; and of course parentheses for grouping.

The `break` statement causes an immediate exit from an enclosing `while` or `for`; the `continue` statement causes the next iteration to begin.

The statement `next` causes `awk` to skip immediately to the next record and begin scanning the patterns from the top. The statement `exit` causes the program to behave as if the end of the input had occurred.

You may put comments in `awk` programs: begin them with the character # and end them with the end of the line, as in

```
print x, y  # this is a comment
```

2.2. grep

There are many occasions when you will want to determine which file contains something you are looking for, or whether a particular string of characters exists in any of a number of files. One of the most useful text utilities that UNIX provides is `grep`. `grep` stands for 'global regular expression printer', a mouthful of non-mnemonic syllables. However, it is a very useful tool for searching through one or many files for a string of characters.

The synopsis of the `grep` command and its two related commands:

```
hostname% grep [-v] [-c] [-l] [-n] [-b] [-i] [-s] [-h] [-w]
[-eexpression] expression [filename ...]
hostname%
```

```
hostname% egrep [-v] [-c] [-l] [-n] [-b] [-s] [-h]
[-eexpression] [-ffile] [expression] [filename] ... ]
hostname%

hostname% fgrep [-v] [-x] [-c] [-l] [-n] [-b] [-i] [-s] [-h]
[-eexpression] [-ffile] [strings] [filename] ... ]
hostname%
```

grep is a utility program that searches a file or files for lines that contain strings of a specified pattern. When grep finds the lines that match the pattern, it prints them out on the standard output.

The two variations on grep, egrep and fgrep, have functions similar to grep. egrep finds full regular expressions and fgrep searches only for fixed strings. In general, egrep is the fastest of these programs. We will explain these two commands later in this section.

The simplest form of grep searches for a pattern that consists of a fixed character string. grep's power lies in its ability to describe more complex patterns, called regular expressions.

Searching for Character Strings

grep in its simplest form looks for a fixed character string. For example, if you are trying to discover if a specific word exists in a file, you use the form grep *word file*. An example of the command, using the same input files as in the earlier example, is:

```
hostname% grep Linda women
Linda
hostname%
```

This command searches for the string 'Linda' in the file 'women'. Since the grep command uses spaces to separate arguments on the command line, you have to be careful what you tell grep to search for. If the string you want to search for contains spaces or tabs, you must surround the string with some kind of delimiter like quotation marks (single or double). Another example:

```
hostname% grep 'Larry G' all
Larry G
hostname%
```

This command searches for the string 'Larry G' in the file 'all'. Because the string 'Larry G' contains a space, we used single quotes to delimit the second argument to grep.

When any of the grep utilities is applied to more than one input file, the name of the file is displayed preceding each line that matches the pattern. For example:

```
hostname% grep Linda women all
women:Linda
all:Linda
hostname%
```

This command searches through the two files 'women' and 'all' for the string 'Linda'. `grep` displays the names of the files in which it found the string.

Searching for 'Everything except *string*' — Inverted Search

`grep` has an option to print every line *except* those that match *string*. This is done with the `-v` option. An example would be:

```
hostname% grep -v "chicken soup" recipes.file
hostname%
```

if you wanted to list the titles of your recipes to decide what to have for dinner, knowing only that you didn't want chicken soup. This command will print out everything except the line containing the string chicken soup.

Regular Expressions

Many times you can't exactly remember the entire string you want to find. You might remember how it begins, or how it ends, or some other feature. Or, you might want to perform some operation on every occurrence of a particular string in a particular position on a line in the file. You should take advantage of `grep`'s powerful feature of searching for regular expressions in text.

You can ask for patterns like '...all six-letter words starting with 'st'', or '...all strings looking like `.IP` and at the beginning of a line'.

Such a pattern or template is called a 'regular expression'. Regular expressions are possible because certain characters have special meanings. These characters are often called 'metacharacters' because they represent something other than their literal meaning.

Take care when using the characters `$`, `*`, `[`, `^`, `|`, `(`, `)`, and `\`, in the regular expression as these characters are also meaningful to the Bourne and C shells. Enclose the entire expression argument in single quotes (`' '`) to avoid having the shell interpret the metacharacter. Double quotes will work most of the time also.

Match Beginning and End of Line

Two of the simplest metacharacters to use are the caret (`^`) and the dollar sign (`$`). These match the beginning and end of a line, respectively. For example:

```
hostname% grep 'panic' file
hostname%
```

matches any occurrence of the word 'panic' in the file *file*. But if you slightly alter the command to:

```
hostname% grep '^panic' file
hostname%
```

you will locate only occurrences of the word 'panic' at the beginnings of lines. Similarly, `$` appearing at the end of a string matches the end of a line:

```
hostname% grep 'panic$' file
hostname%
```

This last example will find only those occurrences of the word 'panic' that fall at the ends of lines.

Logically, you can specify with:

```
hostname% grep '^Do not push the panic button.$' file
hostname%
```

because of the beginning-of-line and end-of-line match requirement, that you find only lines that consist entirely of this pattern and nothing else. Blank lines can be matched with the pattern `^$`. If there are spaces or tabs or other non-printing characters on the line, the ^$ pattern will not match such lines.

A text pattern that matches at a specific place on a line is called an 'anchored match' because it is anchored to a specific position. The `^` and `$` characters lose their special meanings if they appear in places other than the beginning of the pattern, or the end of the pattern, respectively.

Match Any Character

The period, or dot character, as it's usually known in the UNIX system, is a meta-character that matches any character at all. So the string `" st.... "` selects all words beginning with 'st' and having four other characters, provided the word is preceded and followed by a space. To find such words at the beginning of a line, you use

```
hostname% grep '^st....' file
hostname%
```

or the end of a line

```
hostname% grep 'st....$' file
hostname%
```

What `grep` really finds is not only words starting with 'st', but any string of six characters starting with 'st' and preceded by a space. So

```
hostname% grep ' st.... ' file
hostname%
```

finds any of the patterns:

```
string st[10]
starti stop-g
search      story!
```

Specifying that you only want to search for letters is possible with character classes explained in the next section. Text patterns never match across lines; they only match within a line. This is because the dot metacharacter never matches a newline character.

Character Classes

Characters enclosed in brackets (`[ ]`) specify a set of characters that `grep` is to search for. The match is on any one of the characters inside the brackets. For example:

```
hostname% grep [Tt]his file
hostname%
```

finds both 'this' and 'This'. The expression `^[abcxyz]` finds all lines beginning with 'a' or 'b' or 'c' or 'x' or 'y' or 'z'. Inside square brackets, the hyphen character (–) specifies a range of characters. The patterns:

```
[a-z]    all lower-case letters
[A-Z]    all upper-case letters
[0-9]    all digits
```

are very common regular expressions. So, in the previous example of words beginning with 'st', to really limit the search to letters, we could specify:

```
hostname% grep ' st[a-z][a-z][a-z][a-z] ' file
hostname%
```

If the caret character (^) is the first character inside the square brackets, it does not mean 'beginning of line' anymore. Instead, it means anything *except* the search string. For example, the pattern:

```
hostname% grep ^[^a-z] file
hostname%
```

finds all lines *except* those beginning with lower-case letters.

Note that ranges of letters refer to the ASCII character set so the range `[A-z]` not only finds all upper- and lower-case letters, but also all the other characters that fall in that range of ASCII character values, namely:

```
[    \    ]    ^    _    `
```

There are a few pitfalls you can avoid by paying close attention to syntax in specifying ranges of characters. For example, the pattern:

```
[1-30]
```

does *not* mean 'numbers in the range 1 through 30'. It means 'digits in the range 1 through 3, OR 0'. This is the same as specifying the pattern:

```
[1230]
```

or

```
[0-3]
```

If you want to include the hyphen character (–) in the class of characters, you just need to ensure that it won't be confused with a range specification. For example, a hyphen at the beginning of the pattern stands for itself:

```
[-ab]
```

This example means the pattern '–' or 'a' or 'b'. You should threat the characters `[` and `]` with this same caution.

Closures — Repeated Pattern Matches

A number enclosed in braces { } following an expression specifies the number of times the preceding expression is to be repeated. For example, in the earlier search for six-letter words beginning with 'st' could be expressed:

```
' st[a-z]{4} '
```

This repeat number specification is known as a 'closure'. The general format of the closure is {*n*,*m*}, *where n is the minimum number of repeats and m is assumed to be infinity (or at least huge). There are shorthand ways of expressing some closures:*

asterisk *	is equivalent to {0,}, meaning the preceding pattern is to be repeated zero or more times.
plus sign +	is equivalent to {1,}, meaning the preceding pattern is to be repeated one or more times.
question mark ?	is the same as {0,1}, which means that the preceding pattern can be repeated zero or once only.

Closures are the reason that text patterns do not span across lines. If you just type a `grep` pattern like this:

```
hostname% grep '.*' file
hostname%
```

the pattern is trying to specify 'match zero to infinity amounts of any character'. If patterns could span lines, this would try to digest an entire file. Like any other utility, `grep` has some limit to the size of the pattern it can hold internally. A whole file could be too large for `grep`.

Since patterns can not match a newline, the `grep '.*'` command in the example above finds and displays every line in *file*.

Fast Searching for Fixed Strings — `fgrep`

The `fgrep` utility is another text processing utility in the same family as `grep` and `egrep` (described in the following section). The `fgrep` command only handles fixed character strings as text patterns. The `grep` command cannot process wild-card matches, character classes, anchored matches, or closures. For these reasons, `fgrep` is faster than `grep` when all you want to search for is a fixed character string.

An example of `fgrep` usage:

```
hostname% fgrep 'comma in' awk.msun
Items separated by a comma in the print statement
hostname%
```

You can also give `fgrep` a file of fixed strings. Each string appears on a line by itself, but the newline characters have to be escaped with the backslash character (\).

Finding Full Regular Expressions — `egrep`

Another variation on the basic `grep` utility is `egrep`. `egrep` stands for 'extended `grep`'. The `egrep` command is an extension to the basic `grep` to allow full regular expressions.

`egrep` can handle more complex regular expressions, of the form: 'find a pattern, followed by this or that or one of those, followed by something else'. Alternative patterns are specified by separating the alternative patterns with the | (vertical bar) character. This form of regular expression is technically called 'alternation'.

Alternate patterns within regular expressions can be grouped by enclosing the patterns within parentheses ( ). For example:

```
hostname% egrep '(Roman (type|font)' font.change
This paragraph might appear in either Roman font or Italics
If this is Roman type, .LP resets the font; if Italic, .LP
hostname%
```

In this example, `egrep` searches through the file font.change either for the string 'Roman type' or the string 'Roman font'. In the example, `egrep` found both so it printed two different lines each containing one of the patterns it searched for.

Note that the alternatives are in parentheses. If you had typed the command:

```
hostname% egrep 'Roman type|font' font.change
hostname%
```

you would be searching for the strings 'Roman type' or 'font' and you would get a different result:

```
hostname% egrep 'Roman type|font' font.change
This paragraph might appear in either Roman font or Italics
depending on whether a .LP macro request resets the font.
If this is Roman type, .LP resets the font; if Italic, .LP
hostname%
```

Here the first and second lines matched the pattern 'font' and the third line matched the pattern 'Roman type'.

There are other less-used options to `grep`, not covered in depth in this section, and they are summarized below.

Table 2-1 `grep` *Option Summary*

OPTIONS	
`-v`	Invert the search to only display lines that *do not* match.
`-x`	Display only those lines that match exactly — that is, only lines that match in their entirety (`fgrep` only).
`-c`	Display a count of matching lines.
`-l`	List once the names of files with matching lines separated by newlines.
`-n`	Precede each line by its relative line number in the file.
`-b`	Precede each line by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
`-i`	Ignore the case of letters in making comparisons — that is, upper- and lower-case are considered identical. This applies to `grep` and `fgrep` only.
`-s`	Work silently, that is, display nothing except error messages. This is useful for checking the error status.
`-w`	Search for the expression as a word as if surrounded by '\<' and '\>' — `grep` only. (See `ex`).
`-e` *expression*	Same as a simple *expression* argument, but useful when the *expression* begins with a dash (–).
`-f` *file*	Take the regular expression (`egrep`) or string list (`fgrep`) from *file*.

Table 2-2 `grep` *Special Characters*

Characters	
\	Escape character.[1] Backslash (\) followed by any single character other than newline matches that character.
^	Anchored match: matches the beginning of a line.
$	Anchored match: matches the end of a line.
.	Dot (or period). Matches any character.
c	Matches any single character not otherwise endowed with special meaning.
[*string*]	Character class: match any single character from *string*. Ranges of ASCII character codes may be abbreviated as in `[a-z0-9]`. A right-side square bracket (]) may occur only as the first character of the string. A literal – must be placed where it can't be mistaken as a range indicator. A caret (^) character immediately after the open bracket negates the sense of the character class, that is, the pattern matches any character *except* those in the character class.
*	Closure: a regular expression followed by an asterisk (*) matches a sequence of zero or more matches of the regular expression.
+	Closure: a regular expression followed by a plus (+) matches a sequence of one or more matches of the regular expression.
?	Closure: a regular expression followed by a question mark (?) matches a sequence of zero or one matches of the regular expression.
concatenation	Two regular expression concatenated match a match of the first followed by a match of the second.
|	Alternation: two regular expressions separated by a vertical bar (|) or newline match either a match for the first or a match for the second (`egrep` only).
()	A regular expression enclosed in parentheses matches a match for the regular expression.

[1] In this table, the term 'character' excludes newline.

The order of precedence of operators at the same parenthesis level is

`[ ]`	character classes
`* + -`	closures
concatenation	
`|` and newline	alternation

2.3. `look`

For looking up strings of characters quickly in a dictionary file like `/usr/dict/words`, UNIX provides the utility `look`. `look` behaves just like `grep` but unless you give `look` a different input file, it searches through a specific sorted file and prints out all lines that begin with *string*.

`look`'s function is to find lines in a sorted list. The synopsis of the `look` command is:

```
hostname% look [-df] string [file]
hostname%
```

The options to `look` are:

`-d` 'Dictionary' order: only letters, digits, tabs and blanks participate in comparisons.

`-f` Fold: upper-case letters compare equal to lower-case.

If no file is specified, look uses `/usr/dict/words` with collating sequence `-df`.

2.4. `rev`

To search through a file and reverse the order of characters on every line, use the program `rev`.

To search through a file and reverse the order of characters on every line, use the program `rev`.

The synopsis of the `rev` command is:

```
hostname% rev [file] ...
hostname%
```

`rev` copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

2.5. Using `sed`, the Stream Text Editor

This chapter[2] describes `sed`, the non-interactive context or *stream* editor. Use `sed` for editing files too large for comfortable interactive editing, editing any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode, and performing multiple global editing functions efficiently in one pass through the input. Because the default mode is to

[2] The material in this chapter is derived from *Sed — a Non-Interactive Text Editor*, L.E. McMahon, Bell Laboratories, Murray Hill, New Jersey.

apply edit commands globally, and because its output is to the standard output, your workstation or terminal screen, `sed` is good for making changes of a transient nature, rather than permanent modifications to a file.

You can create a complicated editing script separately and use it as a command file. For complex edits, this saves considerable typing, and its attendant errors. Running `sed` from a command file is much more efficient than any interactive editor even if that editor can be driven by a pre-written script.

Whereas the `ed` editor copies your original file into a buffer, `sed` does not use temporary files so you can edit any size file. The only space requirement is that the input and output fit simultaneously into the available second storage. Additionally, `ed` lets you explore the text in whatever order you want, while `sed` works on your file from beginning to end, and allows you no choice of edit commands once you have started it. Basically `sed` passes some data through a set of transformations called editor *functions*.

By default `sed` copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. You can modify this behavior by adding a command-line option; see the "Command Options" section below.

As a lineal descendant of the `ed` editor, `sed` recognizes basically the same regular expressions as `ed`. The range of pattern matches is called the *pattern space*. Ordinarily, the pattern space is one line of text, but you can read more than one line into the pattern space if necessary. But because of the differences between interactive and non-interactive operation, `ed` and `sed` are different enough that even experienced `ed` users should read this chapter. You cannot use relative addressing with `sed` as you can with an interactive editor because `sed` operates a line at a time. `sed` also does not give you any immediate verification that a command has done what was intended.

Refer to the chapter on "Using the `ed` Line Editor" in *Editing Text Files on the Sun Workstation* for more information on `ed` and to the man pages for `sed` and `ed` in the *Commands Reference Manual for the Sun Workstation.*

Using `sed`

The general format of an editing command is:

```
hostname% sed [line1[,line2]]function [arguments]
```

There is an optional line address, or two line addresses separated by a comma, a single-letter edit function, followed by other arguments, which may be required or optional, depending on which function you use. See the section "Specifying Lines for Editing" for the format of line addresses. Any number of blanks or tabs may separate the line addresses from the function. `sed` ignores tab characters and spaces at the beginning of lines. The function must be present; the available commands are discussed in the "Functions" section under each individual function name. You can either put the edit commands on the `sed` command line or put the commands in a file, which is then applied to the file you want to edit. If the commands are few and simple, put them on the `sed` command line. For example, assume the following input text in a file called *kubla*:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Let's copy the first two lines of input as a simple example:

```
hostname% sed 2q kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

As another example, suppose that you want to change the 'Khan' to 'KHAN.' Then the command:

```
hostname% sed  s/Khan/KHAN/g kubla
```

applies the command 's/Khan/KAN/' to all lines from *kubla* and copies all lines to the standard output. The advantage of using `sed` in such a case is that you can use it with input too large for `ed` to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file or on the command line with a slightly more complex syntax. To take commands from a file, for example:

```
hostname% sed  -f  cmdfile input-files...
```

Command Options

`sed` has three options that modify `sed`'s action. If you invoke `sed` with the `-f` (file) option, the edit commands are taken from a file. For example:

```
hostname% sed -f edcomds oldfile > newfile
hostname%
```

The name of the file containing the edit commands must immediately follow the `-f` option. Here, the edit commands in the *edcomds* file are applied to the file *oldfile*, and the standard output is redirected to *newfile*.

You use the `-e` (edit) option to place editing commands directly on the `sed` command line. If you are only using one edit command, you can omit the `-e`, but we include it in the example below for instructive purposes. For example, to delete a line containing the string 'Khan' from *kubla*, you type:

```
hostname% sed -e /Khan/d kubla > newkubla
hostname%
```

If you put more than one edit command on the `sed` command line, each one must be preceded by `-e`. For example:

```
hostname% sed -e /Khan/d -e s/decree/DECREE/ newkubla
hostname%
```

You can also use both the **-e** and the **-f** options at the same time.

`sed` normally copies all input lines that are changed by the edit operation to the output. If you want to suppress this normal output, and have only specific lines

appear on the output, use the -n option with the p (print) flag. For example:

```
hostname% sed -n -e s/to/by/p kubla
Through caverns measureless by man
Down by a sunless sea.
hostname%
```

As a quick reference, these options are:

-f Use the next argument as a filename; the file should contain one editing command to a line.

-e Use the next argument as an editing command.

-n Send only those lines to the output specified by p functions or p functions after substitute functions (see the "Input-Output Functions" section).

Editing Commands Application Order

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a moderately efficient form for execution when the commands are actually applied to lines of the input file. The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

You can change the default linear order of application of editing commands by the flow-of-control commands, t and b (see the "Flow-of-Control Functions" section). Even when you change the order of application by these commands, it is still true that the input line to any command is the output of any previously applied command.

Specifying Lines for Editing

Use addresses to select lines in the input file(s) to apply the editing commands to. Addresses may be either line numbers or context addresses.

Group one address or address-pair with curly braces '{ }' to control the application of a group of commands. See the "Flow-of-Control Functions" section for more on this.

Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches or 'selects' the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character $ matches the last line of the last input file.

Context Addresses

A context address is a pattern or *regular expression* enclosed in slashes (/). sed recognizes the regular expressions that are constructed as follows:

ordinary character

An ordinary character (not one of those discussed below) is a regular

expression, and matches that character.

^ A circumflex ^ at the beginning of a regular expression matches the null character at the beginning of a line.

$ A dollar-sign $ at the end of a regular expression matches the null character at the end of a line.

\n The characters backslash and en \n match an embedded newline character, but not the newline at the end of the pattern space.

. A period . matches any character except the terminal newline of the pattern space.

* A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.

[*character string*]
A string of characters in square brackets [ ] matches any character in the string, and no others. If, however, the first character of the string is a circumflex ^, the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.

concatenation
A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.

\(\) A regular expression between the sequences \(and \) is identical in effect to the unadorned regular expression, but has side-effects which are described in the section entitled "The Substitute Function s" and immediately below.

d This stands for the same string of characters matched by an expression enclosed in \(and \) earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1 matches a line beginning with two repeated occurrences of the same string.

null The null regular expression standing alone (such as, //) is equivalent to the last regular expression compiled.

To use one of the special characters (^ $. * [ ] \ /) as a literal, that is, to match an occurrence of itself in the input, precede the special character by a backslash \.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

Number of Addresses

The commands described in the "Functions" section can have 0, 1, or 2 addresses. Specifying more than the maximum number of addresses allowed is an error. If a command has no addresses, it is applied to every line in the input. If a command has one address, it is applied to all lines that match that address. If a command has two addresses, it is applied to the inclusive range defined by those two addresses.

The command is applied to the first line that matches the first address, and to all subsequent lines until and including the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated. A comma separates two addresses.

For example:

/an/ *matches lines 1, 3, 4 in our sample* `kubla` *file*

```
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man
```

/an.*an/ *matches line 1*

```
In Xanadu did Kubla Khan
```

/^an/ *matches no lines*

/./ *matches all lines*

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

/\./ *matches line 5*

```
Down to a sunless sea.
```

/r*an/ *matches lines 1,3, 4 (number = zero!)*

```
In Xanadu did Kubla Khan
Where Alph, the sacred river, ran
Through caverns measureless to man
```

/\(an\).*\1/ *matches line 1*

```
In Xanadu did Kubla Khan
```

Functions

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is enclosed in parentheses, followed by the single character function name and possible arguments in italics. The summary provides an expanded English translation of the single-character name, and a description of what each function does.

Whole-Line Oriented Functions

The functions that operate on a whole line of input text are as follows:

(2) d — Delete lines. The d function deletes from the file all those lines matched by its address(es); that is, it does not write the indicated lines to the output, No further commands are attempted on a deleted line; as soon as the d function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2) n — Next line. The n function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the n command.

(1) a\

text — Append lines. The a function writes the argument *text* to the output after the line matched by its address. The a function is inherently multi-line; a must appear at the end of a line, and *text* may contain any number of lines. To preserve the one command to a line, the interior newlines must be hidden by a backslash character (\) immediately preceding the newline. The *text* argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash). Once an a function is successfully executed, *text* will be written to the output regardless of what later commands do to the line that triggered it. The triggering line may be deleted entirely; *text* will still be written to the output. The *text* is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

(1) i\

text — Insert lines. The i function behaves identically to the a function, except that *text* is written to the output *before* the matched line. All other comments about the a function apply to the i function as well.

(2) c\

text — Change lines. The c function deletes the lines selected by its address(es), and replaces them with the lines in *text*. Like a and i, put a newline hidden by a backslash after c; interior new lines in *text* must also be hidden by backslashes. The c function may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of *text* is written to the output, *not* one copy per line deleted. As with a and i, *text* is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

No further commands are attempted on a line deleted by a c function.

If text is appended after a line by a or r functions, and the line is subsequently changed, the text inserted by the c function will be

placed *before* the text of the `a` or `r` functions. See the section "Multiple Input-line Functions" later in this chapter for a description of the `r` function.

Note: Leading blanks and tabs are not displayed in the output produced by these functions. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash does not appear in the output.

For example, put the following list of editing commands in a file called *Xkubla*:

```
hostname% cat > Xkubla
n
a\
XXXX
d
^D
hostname% sed -f Xkubla kubla
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
hostname%
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n
i\
XXXX
d
```

or

```
n
c\
XXXX
```

The Substitute Function `s`

The `s` (substitute) function changes parts of lines selected by a context search within the line. The standard format is the same as the `ed` substitute command:

(2) **s** *pattern replacement flags*

The `s` function replaces *part* of a line, selected by *pattern*, with *replacement*. It can best be read 'Substitute for *pattern*, *replacement*.'

The *pattern* argument contains a pattern, exactly like the patterns described in the "Specifying Lines for Editing" section. The only difference between *pattern* and a context address is that the context address must be delimited by slash (/) characters; you can delimit *pattern* by any character other than space or newline.

By default, only the first string matched by *pattern* is replaced. See the `g` flag below.

The *replacement* argument begins immediately after the second delimiting character of *pattern*, and must be followed immediately by another instance of the delimiting character. Thus there are exactly *three* instances of the delimiting character.

The *replacement* is not a pattern, and the characters which are special in patterns do not have special meaning in *replacement*. Instead, other characters are special:

`&` Is replaced by the string matched by *pattern*.

`\`*d* Is replaced by the *d*th substring matched by parts of *pattern* enclosed in `\(` and `\)` where *d* is a single digit. If nested substrings occur in *pattern*, the *d*th is determined by counting opening delimiters ('\(').

As in patterns, you can make the special characters (`&`, `+`, and `\`) literal by preceding them with a backslash (`\`).

The *flags* argument may contain the following flags:

`g` Substitute *replacement* for all (non-overlapping) instances of *pattern* in the line. After a successful substitution, the scan for the next instance of *pattern* begins just after the end of the inserted characters; characters put into the line from *replacement* are not rescanned.

`p` Print or 'display' the line if a successful replacement was done. The `p` flag writes the line to the output if and only if a substitution was actually made by the `s` function. Notice that if several `s` functions, each followed by a `p` flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

`w` *filename*
Write the line to a file if a successful replacement was done. The `w` flag writes lines which are actually substituted by the `s` function to a file named by *filename*. If *filename* exists before `sed` is run, it is overwritten; if not, it is created. A single space must separate `w` and *filename*. The possibilities of multiple, somewhat different copies of one input line being written are the same as for `p`. You can specify a maximum of 10 different filenames after `w` flags and `w` functions (see below), combined.

For example, applying the following command to the the *kubla* file produces on the standard output:

```
hostname% sed -e "s/to/by/w changes" kubla
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

Note that if the edit command contains spaces, you must enclose it with quotes.

It also creates a new file called *changes* that contains only the lines changed as you can see using the `more` command:

```
hostname% more changes
Through caverns measureless by man
Down by a sunless sea.
hostname%
```

If the nocopy option `-n` is in effect, you see those lines that are changed:

```
hostname% sed -e "s/[.,;?:]/*P&*/gp" -n kubla
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
hostname%
```

Finally, to illustrate the effect of the `g` flag assuming nocopy mode, consider:

```
hostname% sed -e "/X/s/an/AN/p" -n kubla
In XANadu did Kubla Khan
hostname%
```

and the command:

```
hostname% sed -e "/X/s/an/AN/gp" -n kubla
In XANadu did Kubla KhAN
hostname%
```

Input-output Functions

The following functions affect the input and output of text. The maximum number of allowable addresses is in parentheses.

(2) `p` Print. The print function writes the addressed lines to the standard output file. They are written at the time the `p` function is encountered, regardless of what succeeding editing commands may do to the lines.

(2) `w` *filename*

Write to *filename*. The write function writes the addressed lines to the file named by *filename*. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them. Put only one space between `w` and *filename*. You can use a maximum of ten different files in write functions and with `w` flags after `s` functions, combined.

(1) `r` *filename*

Read the contents of a file. The read function reads the contents of *filename*, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If you execute `r` and `a` functions on the same line, the text from the `a` functions and the `r` functions is written to the output in the order that the

functions are executed. Put only one space between the `r` and *filename*. If a file mentioned by a `r` function cannot be opened, it is considered a null file, not an error, and no diagnostic is displayed.

Note: Since there is a limit to the number of files that can be opened simultaneously, put no more than ten files in `w` functions or flags; reduce that number by one if any `r` functions are present. Only one read file is open at one time.

Assume that the file *note1* has the following contents:

```
Note:  Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
```

Then the following command reads in *note1* after the line containing 'Kubla':

```
hostname% sed -e "/Kubla/r note1" kubla
In Xanadu did Kubla Khan
Note:  Kubla Khan (more properly Kublai Khan; 1216-1294)
was the grandson and most eminent successor of Genghiz
(Chingiz) Khan, and founder of the Mongol dynasty in China.
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with *pattern spaces* containing embedded newlines; they are intended principally to provide pattern matches across lines in the input. A pattern space is the range of pattern matches. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the `N` function described below.

The maximum number of allowable addresses is enclosed in parentheses.

(2) `N` Next line. The next input line is appended to the current line in the pattern space; an embedded newline separates the two input lines. Pattern matches may extend across the embedded newline(s).

(2) `D` Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2) `P` Print or 'display' first part of the pattern space. Print up to and including the first newline in the pattern space.

The `P` and `D` functions are equivalent to their lower-case counterparts if there are no embedded newlines in the pattern space.

Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2) `h` Hold pattern space. The `h` function copies the contents of the pattern space into a hold area, destroying the previous contents of the hold area.

(2) `H` Hold pattern space. The `H` function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2) `g` Get contents of hold area. The `g` function copies the contents of the hold area into the pattern space, destroying the previous contents of the pattern space.

(2) `G` Get contents of hold area. The `G` function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2) `x` Exchange. The exchange command interchanges the contents of the pattern space and the hold area.

For example, if you want to add `:In Xanadu` to our standard example, create a file called *test* containing the following commands:

```
1h
1s/ did.*//
1x
G
s/\n/  :/
```

Then run that file on the *kubla* file:

```
hostname% sed -f test kubla
In Xanadu did Kubla Khan  :In Xanadu
A stately pleasure dome decree:  :In Xanadu
Where Alph, the sacred river, ran  :In Xanadu
Through caverns measureless to man  :In Xanadu
Down to a sunless sea.  :In Xanadu
hostname%
```

Flow-of-Control Functions

These functions do not edit the input lines, but control the application of functions to the lines that are addressed.

(2) `!` Called 'Don't', the '`!`' function applies the next command, written on the same line, to all and only those input lines *not* selected by the address part.

(2) `{` Grouping. The grouping command '`{`' applies (or does not apply) the next set of commands as a block to the input lines that the addresses of the grouping command select. The first of the commands under control of the grouping command may appear on the same line as the `{` or on the next line.

A matching `}` standing on a line by itself terminates the group of commands. Groups can be nested.

(0) : *label*

Place a label. The label function marks a place in the list of editing commands which may be referred to by `b` and `t` functions. The *label* may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2) `b` *label*

Branch to label. The branch function restarts the sequence of editing commands being applied to the current input line immediately after the place where a colon function with the same *label* was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A `b` function with no *label* is taken to be a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

(2) `t` *label*

Test substitutions. The `t` function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to *label*; if not, it does nothing. Either reading a new input line or executing a `t` function resets the flag which indicates that a successful substitution has occurred.

Miscellaneous Functions

Two additional functions are:

(1) = Equals. The = function writes to the standard output the line number of the line matched by its address.

(1) `q` Quit. The `q` function writes the current line to the output if it should be, writes any appended or read text, and terminates execution.

2.6. `wc`

UNIX provides a facility, `wc`, which searches through your input file and counts the number of lines, words, and characters.

The synopsis for the `wc` command is:

```
hostname% wc [-lwc] [file ... ]
hostname%
```

`wc` counts lines, words, and characters in the named files, or in the standard input if no file names appear. A word is a string of characters delimited by spaces, tabs, or newlines.

If an argument beginning with one of the letters `l`, `w`, or `c`, is present, `wc` may:

l Count lines.

w Count words.

c Count characters.

The default is to use all of the options in the order `-lwc` (count lines, words, and characters). Some examples are:

```
hostname% wc wc.1
      38     153     943 wc.1
hostname%

hostname% wc -l wc.1
      38 wc.1
hostname%

hostname% wc -w wc.1
     153 wc.1
hostname%

hostname% wc -c wc.1
     943 wc.1
hostname%

hostname% wc wc.1
     943 wc.1
hostname%

hostname% wc awk.1 grep.1 look.1 rev.1 sed.1
     224    1141    6713 awk.1
     246    1113    6548 grep.1
      22      95     614 look.1
      12      58     307 rev.1
     211    1053    6253 sed.1
     715    3460   20435 total
hostname%
```

3

Modifying Files

3

Modifying Files

NAME

colrm – remove columns from a file

SYNOPSIS

colrm [startcol [endcol]]

DESCRIPTION

Colrm removes selected columns from a text file. The text is is taken from standard input and copied to the standard output with the specified columns removed.

If only *startcol* is specified, the columns of each line are removed starting with *startcol* and extending to the end of the line. If both *startcol* and *endcol* are specified, all columns between *startcol* and *endcol*, inclusive, are removed.

Column numbering starts with column 1.

SEE ALSO

expand(1)

NAME

compact, uncompact, ccat – compress and uncompress files, and cat them

SYNOPSIS

compact [filename ...]
uncompact [filename ...]
ccat [filename ...]

DESCRIPTION

Compact compresses the named files using an adaptive Huffman code. If no file names are given, the standard input is compacted to the standard output. *Compact* operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to prepend a decoding tree to the compressed file since the encoder and the decoder start in the same state and stay synchronized. Furthermore, *compact* and *uncompact* can operate as filters. In particular:

... | compact | uncompact | ...

operates as a (very slow) no-op.

When an argument *file* is given, it is compacted and the resulting file is placed in *file.C; file* is removed. The first two bytes of the compacted file code the fact that the file is compacted. This code is used to prohibit recompaction.

The amount of compression to be expected depends on the type of file being compressed. Typical values of compression are: Text (38%), Pascal Source (43%), C Source (36%) and Binary (19%). These values are the percentages of file bytes reduced.

Uncompact restores the original file from a file called *file.C* which was compressed by *compact*. If no file names are given, the standard input is uncompacted to the standard output.

Ccat cats the original file from a file compressed by *compact*, without uncompressing the file.

FILES

*.C	compacted file created by compact, removed by uncompact

SEE ALSO

Gallager, Robert G., 'Variations on a Theme of Huffman', *I.E.E.E. Transactions on Information Theory*, vol. IT-24, no. 6, November 1978, pp. 668 - 674.

NAME

expand, unexpand – expand tabs to spaces, and vice versa

SYNOPSIS

expand [–tabstop] [–tab1,tab2,...,tabn] [file ...]
unexpand [**–a**] [file ...]

DESCRIPTION

Expand copies the named *files* (or the standard input) to the standard output, with tabs changed into spaces (blanks). Backspace characters are preserved into the output and decrement the column count for tab calculations. *Expand* is useful for pre-processing character files (before sorting, looking at specific columns, etc.) that contain tabs.

Unexpand copies the named *files* (or the standard input) to the standard output, putting tabs back into the data. By default only leading spaces (blanks) and tabs are converted to strings of tabs, but this can be overridden by the **–a** option (see the *options* section below).

EXPAND OPTIONS

–tabstop

Specified as a single argument sets tabs *tabstop* spaces apart instead of the default 8.

–tab1,tab2,...,tabn

Set tabs at the columns specified by *tab1...*

UNEXPAND OPTIONS

–a Insert tabs when replacing a run of two or more spaces would produce a smaller output file. This option only applies to *unexpand*.

NAME

fold – fold long lines for finite width output device

SYNOPSIS

fold [*–width*] [file ...]

DESCRIPTION

Fold is a filter which folds the contents of the specified *files*, or the standard input if no files are specified, breaking the lines to have maximum width *width*. The default for *width* is 80. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using *expand*(1) before using *fold*.

SEE ALSO

expand(1)

BUGS

Folding may not work correctly if underlining is present.

NAME

sort – sort or merge files

SYNOPSIS

sort [**–mubdfinrtx**] [*+pos1* [*–pos2*]] ... [**–o** name] [**–T** directory] [file] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '–' means the standard input. If no input *file*'s are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence.

The notation *+pos1 –pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m.n*, optionally followed by one or more of the flags **bdfinr**, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first nonblank in the field; **b** is attached independently to *pos2*. A missing *.n* means .0; a missing *–pos2* means the end of the line. Under the **–tx** option, fields are strings separated by *x*; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

OPTIONS

The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b**.

r Reverse the sense of comparisons.

tx 'Tab character' separating fields is *x*.

These option arguments are also understood:

c Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

m Merge only, the input files are already sorted.

o *name*
name is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.

T *directory*
directory argument is the name of a directory in which temporary files should be made.

u Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

EXAMPLES

Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

```
sort –u +0f +0 list
```

Print the password file (*passwd*(5)) sorted by user id number (the 3rd colon-separated field).

```
sort -t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month day) entries. The options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
sort -um +0 -1 dates
```

FILES

/usr/tmp/stm*, /tmp/* first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **-c**.

BUGS

Very long lines are silently truncated.

NAME

split – split a file into pieces

SYNOPSIS

split [*–number*] [*infile* [*outfile*]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000) onto a set of output files (as many files as necessary). The name of the first output file is *outfile*| with **aa** appended, the second file is *outfile***ab**, and so on lexicographically.

If no *outfile* is given, *x* is used as default (output files will be called *x***aa**, *x***ab**, etc.).

If no *infile* is given, or if – is given in its stead, then the standard input file is used.

OPTIONS

–number Number of lines in each piece.

NAME

tr – translate characters

SYNOPSIS

tr [**–cds**] [string1 [string2]]

DESCRIPTION

Tr copies the standard input to the standard output with substitution or deletion of selected characters. The arguments *string1* and *string2* are considered sets of characters. Input characters found in *string1* are mapped into the corresponding characters of *string2*. When *string2* is short it is padded to the length of *string1* by duplicating its last character.

In either string the notation *a–b* means a range of characters from *a* to *b* in increasing ASCII order. The character '\' followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. A '\' followed by any other character stands for that character.

OPTIONS

Any combination of the options **–cds** may be used:

–c Complement the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal;

–d Delete all input characters in *string1;*

–s Squeeze all strings of repeated output characters that are in *string2* to single characters.

EXAMPLE

The following example creates a list of all the words in 'file1' one per line in 'file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect '\' from the Shell. 012 is the ASCII code for newline.

```
tr –cs A–Za–z '\012' <file1 >file2
```

SEE ALSO

ed(1), ascii(7), expand(1)

BUGS

Won't handle ASCII NUL in *string1* or *string2;* always deletes NUL from input.

NAME

tsort – topological sort

SYNOPSIS

tsort [file]

DESCRIPTION

Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

lorder(1)

BUGS

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

4

Printing Files

4

Printing Files

NAME

lpq – spool queue examination program

SYNOPSIS

lpq [+[*num*]] [**–l**] [**–P***printer*] [*job #* ...] [*user* ...]

DESCRIPTION

lpq examines the spooling area used by *lpd*(8) for printing files on the line printer, and reports the status of the specified jobs or all jobs associated with a user.

Lpq reports on any jobs currently in the queue when invoked without any options. See the OPTIONS section below for a list of options. Arguments supplied that are not recognized as options are interpreted as user names or job numbers to filter out only those jobs of interest.

For each job submitted (that is, invocation of *lpr*(1)) *lpq* reports the user's name, current rank in the queue, the names of files comprising the job, the job identifier (a number which may be supplied to *lprm*(1) for removing a specific job), and the total size in bytes. The –l option causes information about each of the files comprising the job to be printed. Normally, only as much information as will fit on one line is displayed. Job ordering is dependent on the algorithm used to scan the spooling directory and is supposed to be FIFO (First in First Out). File names comprising a job may be unavailable (when *lpr*(1) is used as a sink in a pipeline) in which case the file is indicated as '(standard input)'.

If *lpq* warns that there is no daemon present (that is, due to some malfunction), the *lpc*(8) command can be used to restart the printer daemon.

OPTIONS

Lpq reports on any jobs currently in the queue when invoked without any options.

–P*printer*
: route the output to the printer specified by *printer*. In the absence of the –P option, the default line printer is used (or the value of the PRINTER variable in the environment).

+nnn
: display the spool queue until it empties. Supplying a number *nnn* immediately after the + sign indicates that *lpq* should sleep *nnn* seconds in between scans of the queue.

FILES

/etc/termcap	for manipulating the screen for repeated display
/etc/printcap	to determine printer characteristics
/usr/spool/*	the spooling directory, as determined from printcap
/usr/spool/*/cf*	control files specifying jobs
/usr/spool/*/lock	the lock file to obtain the currently active job

SEE ALSO

lpr(1), lprm(1), lpc(8), lpd(8)

BUGS

The + option doesn't wait until the entire queue is empty; it only waits until the local machine's queue is empty.

Due to the dynamic nature of the information in the spooling directory *lpq* may report unreliably.

Output formatting is sensitive to the line length of the terminal; this can result in widely-spaced columns.

lpq is sometimes unable to open various files because the lock file is malformed.

DIAGNOSTICS

waiting for *printer* to become ready

The daemon could not open the printer device. This can happen for a number of reasons; the most common is that the printer is turned off-line. This message can also be generated if the printer is out of paper, the paper is jammed, and so on. The actual reason is dependent on the meaning of error codes returned by system device driver. Not all printers supply sufficient information to distinguish when a printer is off-line or having trouble (for example, a printer connected through a

serial line). Another possible cause of this message is some other process, such as an output filter, has an exclusive open on the device. Your only recourse here is to kill off the offending program(s) and restart the printer with *lpc*.

***printer* is ready and printing**

The *lpq* program checks to see if a daemon process exists for *printer* and prints the file *status*. If the daemon is hung, a super user can use *lpc* to abort the current daemon and start a new one.

waiting for *host* to come up

Indicates that there is a daemon trying to connect to the remote machine named *host* in order to send the files in the local queue. If the remote machine is up, *lpd* on the remote machine is probably dead or hung and should be restarted as mentioned for *lpr*.

sending to *host*

The files should be in the process of being transferred to the remote *host*. If not, the local daemon should be aborted and started with *lpc*.

Warning: *printer* is down

The printer has been marked as being unavailable with *lpc*.

Warning: no daemon present

The *lpd* process overseeing the spooling queue, as indicated in the "lock" file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer should be checked for a diagnostic from the deceased process. To restart an *lpd*, use

% lpc restart *printer*

NAME

lpr – off line print

SYNOPSIS

lpr [**–P***printer*] [*–#num*] [**–C***class*] [**–J***job*] [**–T***title*] [**–i** [*num*]] [**–1234***font*] [**–w***num*] [**–r**] [**–m**] [**–h**] [**–s**] [*–filter_option*] [*filename ...*]

DESCRIPTION

Lpr uses a spooling daemon to print the named files when facilities become available. *Lpr* reads the stndard input if no files are specified.

OPTIONS

–P*printer*

Force output to the named *printer*. Normally, the default printer is used (site dependent), or the value of the PRINTER environment variable is used.

–#num Produce multiple copies of output, using *num* as the number of copies for each file named. For example,

tutorial% **lpr –#3 new.index.c print.index.c more.c**

produces three copies of the file *new.index.c*, followed by three copies of *print.index.c*, etc. On the other hand,

tutorial% **cat new.index.c print.index.c more.c | lpr –#3**

generates three copies of the concatenation of the files.

–C Print *class* as the job classification on the burst page. For example,

tutorial% **lpr –C Operations new.index.c**

replaces the system name (the name returned by *hostname*(1)) with 'Operations' on the burst page, and prints the file *new.index.c*.

–J*job* Print *job* as the job name on the burst page. Normally, *lpr* uses the first file's name.

–T*title* Use *title* instead of the file name for the title used by *pr*(1).

–i[*num*] Indent output *num* spaces. If *num* is not given, eight spaces are used as default.

–1234*font*

Mount the specified *font* on font position *i*. The daemon will construct a *.railmag* file referencing */usr/lib/vfont/name.size*.

–w*num* Use *num* as the page width for *pr*(1).

–r Remove the file upon completion of spooling.

–m Send mail upon completion.

–h Suppress printing the burst page.

–s Create a symbolic link from the spool area to the data files rather than trying to copy them (so large files can be printed). This means the data files should not be modified or removed until they have been printed. In the absence of this option, files larger than 1 Megabyte in length are truncated. Note that the –s option only works if you are specifically naming data files — it doesn't work if *lpr* is at the end of a pipeline.

filter_option

The following single letter options notify the line printer spooler that the files are not standard text files. The spooling daemon will use the appropriate filters to print the data accordingly.

–p Use *pr*(1) to format the files (equivalent to *print*).
–l Print control characters and suppress page breaks.
–t The files contain data from *troff*(1) (cat phototypesetter commands).
–n The files contain data from *ditroff* (device independent troff).
–d The files contain data from *tex* (DVI format from Stanford).
–g The files contain standard plot data as produced by the *plot*(3X) routines (see also

plot(1G) for the filters used by the printer spooler).

–v The files contain a raster image for devices like the Versatec.

–c This option currently is unassigned.

–f Interpret the first character of each line as a standard FORTRAN carriage control character.

FILES

/etc/passwd	personal identification
/etc/printcap	printer capabilities data base
/usr/lib/lpd*	line printer daemons
/usr/spool/*	directories used for spooling
/usr/spool/*/cf*	daemon control files
/usr/spool/*/df*	data files specified in "cf" files
/usr/spool/*/tf*	temporary copies of "cf" files

SEE ALSO

lpq(1), lprm(1), pr(1), symlink(2), printcap(5), lpc(8), lpd(8)

DIAGNOSTICS

lpr: copy file is too large

A file is determined to be too 'large' to print by copying into the spool area. Use the –s option as defined above to make a symbolic link to the file instead of copying it. A 'large' file is approximately 1 Megabyte in this system.

lpr: ***printer*****: unknown printer**

The *printer* was not found in the *printcap* database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the */etc/printcap* file.

lpr: ***printer*****: jobs queued, but cannot start daemon.**

The connection to *lpd* on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket */dev/printer* to be sure it still exists (if it does not exist, there is no *lpd* process running).

lpr: ***printer*****: printer queue is disabled**

This means the queue was turned off with

tutorial% **lpc disable** *printer*

to prevent *lpr* from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a super-user with *lpc*.

If the **–f** and **–s** flags are combined as follows:

lpr –fs *filename*

copies the file to the spooling directory rather than making a symbolic link.

Placing the **–s** flag first, or writing each as separate arguments makes a link as expected.

NAME

lprm – remove jobs from the line printer spooling queue

SYNOPSIS

lprm [**–P***printer*] [–] [*job #* ...] [*user* ...]

DESCRIPTION

Lprm removes a job, or jobs, from a printer's spool queue. Since the spooling directory is protected from users, using *lprm* is normally the only method by which a user may remove a job.

Lprm without any arguments will delete the currently active job if it is owned by the user who invoked *lprm*.

If the – flag is specified, *lprm* will remove all jobs which a user owns. If the super-user employs this flag, the spool queue will be emptied entirely. The owner is determined by the user's login name and host name on the machine where the *lpr* command was invoked.

Specifying a user's name, or list of user names, will cause *lprm* to attempt to remove any jobs queued belonging to that user (or users). This form of invoking *lprm* is useful only to the super-user.

A user may dequeue an individual job by specifying its job number. This number may be obtained from the *lpq*(1) program. For example:

```
tutorial% lpq –Pimagen
imagen is ready and printing
Rank   Owner   Job   Files                 Total Size
active wendy   385   standard input        35501 bytes
tutorial% lprm –Pimagen 305
```

Lprm announces the names of any files it removes and is silent if there are no jobs in the queue which match the request list.

Lprm will kill off an active daemon, if necessary, before removing any spooling files. If a daemon is killed, a new one is automatically restarted upon completion of file removals.

The **–P** option may be used to specify the queue associated with a specific printer (otherwise the default printer, or the value of the PRINTER variable in the environment is used).

FILES

/etc/printcap	printer characteristics file
/usr/spool/*	spooling directories
/usr/spool/*/lock	lock file used to obtain the pid of the current daemon and the job number of the currently active job

SEE ALSO

lpr(1), lpq(1), lpd(8)

DIAGNOSTICS

lprm: *printer* **: cannot restart printer daemon**

The connection to *lpd* on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket */dev/printer* to be sure it still exists (if it does not exist, there is no *lpd* process running). Use

% ps ax | fgrep lpd

to get a list of process identifiers of running lpd's. The *lpd* to kill is the one which is not listed in any of the "lock" files (the lock file is contained in the spool directory of each printer). Kill the master daemon using the following command.

% kill *pid*

Then remove */dev/printer* and restart the daemon (and printer) with the following commands.

% rm /dev/printer % /usr/lib/lpd

Another possibility is that the *lpr* program is not setuid *root*, setgid *spooling*. This can be checked with

% ls –lg /usr/ucb/lpr

BUGS

Since there are race conditions possible in the update of the lock file, the currently active job may be incorrectly identified.

NAME

pr – print file(s), possibly in multiple columns

SYNOPSIS

pr [**–***n*] [**+***n*] [**–h** *string*] [**–w***n*] [**–f**] [**–l***n*] [**–t**] [**–s***n*] [**–m**] [file] ...

DESCRIPTION

Pr prepares one or more *files*'s for printing. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. *Pr* prints its standard input if there are no *file* arguments.

Inter-terminal messages via *write*(1) are forbidden during a *pr*.

OPTIONS

Options apply to all following *file*'s but may be reset between *file*'s:

–*n* Produce *n*-column output. This option overrides the **–t** option (see below).

+*n* Begin printing with page *n*.

–h *string*
Use *string* as a header for the page instead of the default header.

–w*n* For purposes of multi-column output, take the width of the page to be *n* characters instead of the default 72.

–f Use formfeeds instead of newlines to separate pages. A formfeed is assumed to use up two blank lines at the top of a page. Thus this option does not affect the effective page length.

–l*n* Take the length of the page to be *n* lines instead of the default 66.

–t Do not print the 5-line header or the 5-line trailer normally supplied for each page. Formfeed characters are not generated when this option is used, even if the **–f** option was used. The **–t** option is intended for applications where the results should be directed to a file for further processing.

–s*c* Separate columns by the single character *c* instead of by the appropriate amount of white space. A missing *c* is taken to be a tab.

–m Print all *file*'s simultaneously, each in one column,

EXAMPLES

Print a file called *dreadnaught* on the printer — this is the simplest use of *pr*:

```
krypton% pr dreadnaught | lpr
krypton%
```

Produce three laminations of a file called *ridings* side by side in the output, with no headers or trailers, the results to appear in the file called *Yorkshire*:

```
krypton% pr –m –t ridings ridings ridings > Yorkshire
krypton%
```

FILES

/dev/tty? to suspend messages.

SEE ALSO

cat(1), lpr(1)

DIAGNOSTICS

There are no diagnostics when *pr* is printing on a terminal.

BUGS

The options described above interact with each other in strange and as yet to be defined ways.

Revision History

Revision	Date	Comments
A	17 February 1986	First release of Using UNIX Text Utilities on the Sun Workstation.

Notes