**Chapter 10**

**Advanced Filters**

**1. grep**

The grep program searches a file or files for lines that have a certain pattern. The syntax is:

**$ grep "pattern" file(s)**

The name "grep" derives from the ed (a Unix line editor) command g/re/p, which means "globally search for a regular expression and print all lines containing it." A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching. When we learn more about regular expressions, we can use them to specify complex patterns of text.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. But let's start with an example reading from files: searching all files in the working directory for a word--say, Unix. We'll use the wildcard \* to quickly give grep all filenames in the directory.

**$ grep "Unix" \***

**ch01:Unix is a flexible and powerful operating system**

**ch01:When the Unix designers started work, little did**

**ch05:What can we do with Unix?**

**$**

When grep searches multiple files, it shows the filename where it finds each matching line of text. Alternatively, if we don't give grep a filename to read, it reads its standard input; that's the way all filter programs work:

**$ ls -l | grep "Aug"**

**-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02**

**-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07**

**-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro**

**-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros**

**$**

First, the example runs ls -l to list our directory. The standard output of ls -l is piped to grep, which only outputs lines that contain the string Aug (that is, files that were last modified in August). Because the standard output of grep isn't redirected, those lines go to the terminal screen.

grep options let we modify the search. Given table lists some of the options.

|  |  |
| --- | --- |
| **Option** | **Description** |
| -v | Print all lines that do not match pattern. |
| -n | Print the matched line and its line number. |
| -l | Print only the names of files with matching lines (lowercase letter "L"). |
| -c | Print only the count of matching lines. |
| -i | Match either upper- or lowercase. |

**Some grep options**

Next, let's use a regular expression that tells grep to find lines with carol, followed by zero or more other characters (abbreviated in a regular expression as ".\*"), then followed by Aug:

[15] Note that the regular expression for "zero or more characters," ".\*", is different than the corresponding filename wildcard "\*". W e can't cover regular expressions in enough depth here to explain the difference--though more-detailed books do. As a rule of thumb, remember that the first argument to grep is a regular expression; other arguments, if any, are filenames that can use wildcards.

**$ ls -l | grep "carol.\*Aug"**

**-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros**

**$**

**2. sed**

sed (which stands for Stream EDitor) is a simple but powerful computer program used to apply various pre-specified textual transformations to a sequential stream of text data. It reads input files line by line, edits each line according to rules specified in its simple language (the sed script), and then outputs the line. While originally created as a Unix utility by Lee E. McMahon of Bell Labs from 1973 to 1974, sed is now available for virtually every operating system that supports a command line.

**2.1 Functions**

sed is often thought of as a non-interactive text editor. It differs from conventional text editors in that the processing of the two inputs is inverted. Instead of iterating once through a list of edit commands applying each one to the whole text file in memory, sed iterates once through the text file applying the whole list of edit commands to each line. Because only one line at a time is in memory, sed can process text files with an arbitrarily-large number of lines. Some implementations of sed can only process lines of limited lengths.

sed's command set is modeled after the ed editor, and most commands work similarly in this inverted paradigm. For example, the command 25d means if this is line 25, then delete (don't output) it, rather than go to line 25 and delete it as it does in ed. The notable exceptions are the copy and move commands, which span a range of lines and thus don't have straightforward equivalents in sed. Instead, sed introduces an extra buffer called the hold space, and additional commands to manipulate it. The ed command to copy line 25 to line 76 (25t76) for example would be coded as two separate commands in sed (25h; 76g), to store the line in the hold space until the point at which it should be retrieved.

**Usage:**

The following example shows a typical usage of sed, where the -e option indicates that the sed expression follows:

**$ sed -e 's/oldstuff/newstuff/g' inputFileName > outputFileName**

The s stands for substitute; the g stands for global, which means that all matching occurrences in the line would be replaced. After the first slash is the regular expression to search for and after the second slash is the expression to replace it with. The substitute command (s///) is by far the most powerful and most commonly used sed command.

Under Unix, sed is often used as a filter in a pipeline:

**$ generate\_data | sed -e 's/x/y/'**

That is, generate the data, but make the small change of replacing x with y.

Several substitutions or other commands can be put together in a file called, for example, subst. sed and then be applied using the -f option to read the commands from the file:

**$ sed -f subst.sed inputFileName > outputFileName**

Besides substitution, other forms of simple processing are possible. For example, the following deletes empty lines or lines that only contain spaces:

**$ sed -e '/^ \*$/d' inputFileName**

This example used some of the following regular expression meta characters:

* The caret (^) matches the beginning of the line.
* The dollar sign ($) matches the end of the line.
* A period (.) matches any single character.
* The asterisk (\*) matches zero or more occurrences of the previous character.
* A bracketed expression delimited by [ and ] matches any of the characters inside the brackets.

Complex sed constructs are possible, to the extent that it can be conceived of as a highly specialised, albeit simple, programming language. Flow of control, for example, can be managed by use of a label (a colon followed by a string which is to be the label name) and the branch instruction b; an instruction b followed by a valid label name will move processing to the block following the label; if the label does not exist then the branch will end the script.

A third one should be added to remove all blanks and tabs immediately before the end of line:

**$ sed -e 's/#.\*//' -e 's/[ ^I]\*$//' -e '/^$/ d'**

The character "^I" is a CRTL-I or tab character. We would have to explicitly type in the tab. Note the order of operations above, which is in that order for a very good reason. Comments might start in the middle of a line, with white space characters before them. Therefore comments are first removed from a line, potentially leaving white space characters that were before the comment. The second command removes all trailing blanks, so that lines that are now blank are converted to empty lines. The last command deletes empty lines. Together, the three commands remove all lines containing only comments, tabs or spaces.

This demonstrates the pattern space sed uses to operate on a line. The actual operation sed uses is:

* Copy the input line into the pattern space.
* Apply the first sed command on the pattern space, if the address restriction is true.
* Repeat with the next sed expression, again

operating on the pattern space.

* When the last operation is performed, write out the pattern space and read in the next line from the input file.

|  |  |
| --- | --- |
| n | replace nth instance of pattern with replacement |
| g | replace all instances of pattern with replacement |
| p | write pattern space to STDOUT if a successful substitution takes place |
| w file | Write the pattern space to file if a successful substitution takes place |

**2.2 Printing with p**

Another useful command is the print command: "p." If sed wasn't started with an "-n" option, the "p" command will duplicate the input. The command

**$ sed 'p'**

will duplicate every line. If we wanted to double every empty line, use:

**$ sed '/^$/ p'**

Adding the "-n" option turns off printing unless we request it. Another way of duplicating head's functionality is to print only the lines we want. This example prints the first 10 lines:

**$ sed -n '1,10 p' <file**

**2.3 Deleting with D**

**$ sed ‘1,10 d’ < file**

**2.4 Substituting with S**

Sed has several commands, but most people only learn the substitute command: s. The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing "day" to"night:"

**$ sed s/day/night/ <old >new**

I didn't put quotes around the argument because this example didn't need them. If we read my earlier tutorial, we would understand why it doesn't need quotes. If we have meta-characters in the command, quotes are necessary. In any case, quoting is a good habit, and I will henceforth quote future examples. That is:

**$ sed 's/day/night/' <filename>**

There are four parts to this substitute command:

|  |  |
| --- | --- |
| s | Substitute command |
| /../../ | Delimiter |
| day | Regular Expression Pattern String |
| night | Replacement string |

We've covered quoting and regular expression. That's 90% of the effort needed to learn the substitute command. To put it another way, we already know how to handle 90% of the most frequent uses of sed. There are a few fine points that must be covered.

**3. awk**

The name awk comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The original version of awk was written in 1977 at AT&T Bell Laboratories. In 1985 a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. This new version became generally available with Unix System V Release 3.1.

.awk is a programming language designed to search for, match patterns, and perform actions on files. awk programs are generally quite small, and are interpreted. This makes it a good language for prototyping.

**input lines to awk:**

When awk scans an input line, it breaks it down into a number of fields. Fields are separated by a space or tab character. Fields are numbered beginning at one, and the dollar symbol ($) is used to represent a field.

For instance, the following line in a file

**I like money.**

Has three fields. They are

**$1 I**

**$2 like**

**$3 money.**

Field zero ($0) refers to the entire line. awk scans lines from a file(s) or standard input.

Note: The most frustrating thing about trying to learn awk is getting our program past the shell's parser. The proper way is to use single quotes around the program, like so:

**$ awk '{print $0}' filename**

The single quotes protect almost everything from the shell. In csh or tcsh, we still have to watch out for exclamation marks, but other than that, we're safe.

**$ awk '{print $2,$1}' filename**

Will print the second field, then the first. All other fields are ignored

* Variables
  + Need not be declared
  + May contain any type of data, their data type may change over the life of the program
  + As in C, case matters; since all the built-in variables are all uppercase, avoid this form.
  + Some of the commonly used built-in variables are:
    - NR-- The current line's sequential number
    - NF-- The number of fields in the current line
    - FS-- The input field separator; defaults to whitespace and is reset by the -F command line parameter
    - OFS-- Output Field Separator default ' '
  + RS-- Input Record Separator default \n
  + FILENAME-- The name of the file currently being processed
* Fields
  + Each record is separated into fields named $1, $2 , etc
  + $0 is the entire record
  + NF contains the number of fields in the current line
* Print
  + print prints each of the values of $1 through $NF separated by OFS then prints a \n onto stdout; the default value of OFS is a blank

**Example:**

See a file(sample) contains these contents.

ramu:8:17:d

RAMU:8:17:D

king:89:37:j

smith:8:17:c

scott:19:4:d

allen:73:99:f

**$ awk –F “:” ‘NR==1{print $1,$4}’ sample**

**ramu d**

**$ awk –F “:” ‘NR==1 || NR==2 {print $0}’ sample**

**ramu 8 17 d**

**RAMU 8 17 D**

BEGIN and END are special patterns. They are not used to match input records. Rather, they are used for supplying start-up or clean-up information to our awk script. A BEGIN rule is executed, once, before the first input record has been read. An END rule is executed, once, after all the input has been read. For example:

**Example1:**

Program to count number of records and Number of fields

**awk 'BEGIN{FS=":";OFS="#";print "Student info"}**

**{print $1,$3}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**END{print "The Number Of records are" NR}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**END{print "The Number Of Fields are" NF}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}' sample**

**Result:**

**Student info**

**ramu#17**

**RAMU#17**

**king#37**

**smith#17**

**scott#4**

**allen#99**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The Number Of records are6**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The Number Of Fields are4**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Example2:**

Program to find out idle users

**who -i|tr -s " "|awk -F " " '$6 !~/\./{print $1,$2,$6}' >smith**

**clear**

**awk 'BEGIN{FS=" ";OFS="\t";print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**BEGIN{print "IDLE USERS ARE";print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";print "username**

**TTy IDLETIME"}**

**{print $1,$2,$3}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}' smith**

**Result:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**IDLE USERS ARE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**username TTy IDLETIME**

**uf027 pts/2 01:48**

**uf088 ttyW1 00:02**

**srujana ttyW1 00:02**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Example3:**

Program to count characters in a given file

**clear**

**echo "Enter the file name"**

**read name**

**awk 'BEGIN{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";print "The length of a given**

**file is";print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**{len=length($0);x=len+x}END{print "the length of is"x}' $name**

**Note:**

Write these programs in a file and then execute them with sh <file name> to get the desired results.

**String functions:**

These are additional functions we can use with awk

|  |  |
| --- | --- |
| sub(regexp,sub) | Substitute sub for regexp in $0 |
| sub(regexp,sub,var) | Substitute sub for regexp in var |
| gsub(regexp,sub) | Globally substitute sub for regexp in $0 |
| gsub(regexp,sub,var) | Globally substitute sub for regexp in var |
| split(var,arr) | Split var on white space into arr |
| split(var,arr,sep) | Split var on white space into arr on sep as separator |
| index(bigvar,smallvar) | Find index of smallvar in bigvar |
| match(bigvar,expr) | Find index for regexp in bigvar |
| length(var) | Number of characters in var |
| substr(var,num) | Extract chars from position num to end |
| substr(var,num1,num2) | Extract chars from num1 through num2 |
| sprintf(format,vars) | Format vars to a string |
| getline | reads in a line each time it is called |