MARUDHAR KESARI JAIN COLLEGE FOR WOMEN, VANIYAMBADI

PG DEPARTMENT OF COMPUTER APPLICATIONS

Subject code : 23PCA12

Class : I-MCA

SUBJECT: LINUX AND SHELL PROGRAMMING

UNIT-IV

Regular Expressions:Defining regular Expressions-Looking at the basics –Extending our patterns-creating expressions –Advanced sed: Using Multiline commands-Understanding the hold space –Negating a command –changing the flow replacing via a pattern using variable in gawk-Using structured commands-formatting the printing-working creating sed utilities-Advanced gawk Raexamining gawk.

Regular Expressions:

Linux Regular Expressions are special characters which help search data and matching complex patterns. Regular expressions are shortened as ‘regexp’ or ‘regex’. They are used in many Linux programs like grep, bash, rename, sed, etc.

Types of Regular expressions

For ease of understanding let us learn the different types of Regex one by one.

Basic Regular expressions

Some of the commonly used commands with Regular expressions are tr, sed, vi and grep. Listed below are some of the basic Regex.

Symbol	Descriptions
.	replaces any character
^	matches start of string
$	matches end of string
*	matches up zero or more times the preceding character
\	Represent special characters
()	Groups regular expressions
?	Matches up exactly one character

Regular expressions in Linux, commonly referred to as Reg Ex, are powerful tools used for pattern matching and text manipulation in Linux and other programming languages. Regular expressions allow users to define patterns that can match specific strings of characters or patterns within larger strings of text.

Defining regular Expressions:

The Linux regular expression, basically it is a sequence of characters or string that would define the searching pattern. These searching patterns are used by the string search algorithms like vim, vi, sed, awk, find, grep, etc. It is a very powerful tool in Linux.

Interval Regular expressions

These expressions tell us about the number of occurrences of a character in a string. They are

Expression	Description
{n}	Matches the preceding character appearing ‘n’ times exactly
{n,m}	Matches the preceding character appearing ‘n’ times but not more than m
{n, }	Matches the preceding character only when it appears ‘n’ times or more

Example:

Filter out all lines that contain character ‘p’

```
guru99@guru99-VirtualBox:~$ cat sample|grep p
apple
pant
people
```

We want to check that the character ‘p’ appears exactly 2 times in a string one after the other. For this the syntax would be:

cat sample | grep -E p\{2}

```
guru99@guru99-VirtualBox:~$ cat sample|grep -E p\{2}
apple
guru99@guru99-VirtualBox:~$
```

Note: You need to add -E with these regular expressions.

Extended regular expressions

These regular expressions contain combinations of more than one expression. Some of them are:

Expression	Description
\+	Matches one or more occurrence of the previous character
\?	Matches zero or one occurrence of the previous character

Example:

Searching for all characters ‘t’

```
guru99@guru99-VirtualBox:~$ cat sample|grep t
bat
ant
eat
pant
taste
```

Suppose we want to filter out lines where character ‘a’ precedes character ‘t’

We can use command like

catsample|grep "a\+t"

```
guru99@guru99-VirtualBox:~$ cat sample|grep "a\+t"
bat
eat
guru99@guru99-VirtualBox:~$
```

Brace expansion

The syntax for brace expansion is either a sequence or a comma separated list of items inside curly braces “{}”. The starting and ending items in a sequence are separated by two periods “..”.

Interval Regular expressions

These expressions tell us about the number of occurrences of a character in a string. They are

Expression	Description
{n}	Matches the preceding character appearing 'n' times exactly
{n,m}	Matches the preceding character appearing 'n' times but not more than m
{n, }	Matches the preceding character only when it appears 'n' times or more

Example:

Filter out all lines that contain character 'p'

```
guru99@guru99-VirtualBox:~$ cat sample|grep p
apple
pant
people
```

We want to check that the character 'p' appears exactly 2 times in a string one after the other. For this the syntax would be:

cat sample | grep -E p\{2}

```
guru99@guru99-VirtualBox:~$ cat sample|grep -E p\{2}
apple
guru99@guru99-VirtualBox:~$
```

Note: You need to add -E with these regular expressions.

Extended regular expressions

These regular expressions contain combinations of more than one expression. Some of them are:

Expression	Description
\+	Matches one or more occurrence of the previous character
\?	Matches zero or one occurrence of the previous character

Example:

Searching for all characters 't'

```
guru99@guru99-VirtualBox:~$ cat sample|grep t
bat
ant
eat
pant
taste
```

Suppose we want to filter out lines where character ‘a’ precedes character ‘t’

We can use command like

catsample|grep "a\+t"

```
guru99@guru99-VirtualBox:~$ cat sample|grep "a\+t"
bat
eat
guru99@guru99-VirtualBox:~$
```

Brace expansion

The syntax for brace expansion is either a sequence or a comma separated list of items inside curly braces “{}”. The starting and ending items in a sequence are separated by two periods “..”.

Looking at the basics:

The POSIX ERE engine is often found in programming languages that rely on regular expressions for text filtering. It provides advanced pattern symbols as well as special symbols for common patterns, such as matching digits, words, and alphanumeric characters. The gawk program uses the ERE engine to process its regular expression patterns.

Because there are so many different ways to implement regular expressions, it’s hard to present a single, concise description of all the possible regular expressions. The following sections discuss the most commonly found regular expressions and demonstrate how to use them in the sed editor and gawk program.

Extending our patterns:

Defining BRE Patterns

The most basic BRE pattern is matching text characters in a data stream. This section demonstrates how you can define text in the regular expression pattern and what to expect from the results.

Plain text

Demonstrated how to use standard text strings in the sed editor and the gawk

program to if later data. Here's an example to refresh your memory:

$ echo "This is a test" | sed -n '/test/p'

This is a test

$ echo "This is a test" | sed -n '/trial/p'

$

$ echo "This is a test" | gawk '/test/ {print $0}

This is a test $ echo "This is a test" | gawk '/trial/ {print $0}' $ the first pattern defines a single word, test. The sed editor and gawk program scripts each use their own version of the print command to print any lines that match the regular expression pattern. Because the echo statement contains the word "test" in the text string, the data stream text matches the defined regular expression pattern, and the sed editor displays the line. The second pattern again defines just a single word, this time the word "trial."

Because the echo statement text string doesn't contain that word, the regular expression pattern doesn't match, so neither the sed editor nor the gawk program prints the line. You probably already noticed that the regular expression doesn't care where in the data stream the pattern occurs. It also doesn't matter how many times the pattern occurs. After the regular expression can match the pattern anywhere in the text string, it passes the string along to the Linux utility that's using it.

The key is matching the regular expression pattern to the data stream text. It's important to remember that regular expressions are extremely picky about matching patterns. The first rule to remember is that regular expression patterns are case sensitive. This means they'll match only those patterns with the proper case of characters: $ echo "This is a test" | sed -n '/this/p' $ $ echo "This is a test" | sed -n '/This/p' this is a test $ the first attempt failed to match because the word.

Special characters

As you use text strings in your regular expression patterns, there's something you need to be aware of. There are a few exceptions when defy ning text characters in a regular expression. Regular expression patterns assign a special meaning to a few characters. If you try to use these characters in your text pattern, you won't get the results you were expecting.

These special characters are recognized by regular expressions:

.*[]^${}\+?|()

As the chapter progresses, you'll find out just what these special characters do in a regular expression. For now, however, just remember that you can't use these characters by themselves in your text pattern.

If you want to use one of the special characters as a text character, you need to escape it.

When you escape the special characters, you add a special character in front of it to indicate to the regular expression engine that it should interpret the next character.

For example, if you want to search for a dollar sign in your text, just precede it with a

Backslash character:

$ Cat data2

The cost is $4.00

$ Sed -n '/\$/p' data2

The cost is $4.00

$

Because the backslash is a special character, if you need to use it in a regular expression pattern, you need to escape it as well, producing a double backslash:

$ echo "\ is a special character" | sed -n '/\\/p'

\ is a special character

$

Finally, although the forward slash isn’t a regular expression special character, if you use it

In your regular expression pattern in the sed editor or the gawk program, you get an error:

$ echo "3 / 2" | sed -n '///p'

Sed: -e expression #1

Anchor characters:

As shown in the “Plain Text” section, by default, when you specify a regular expression pattern, if the pattern appears anywhere in the data stream, it matches. You can use two special characters to anchor a pattern to either the beginning or the end of lines in the

Data stream.

Starting at the beginning

The care t character (^) defy nest a pattern that starts at the beginning of a line of text in

The data stream. If the pattern is located any place other than the start of the line of text, the regular expression pattern fails.

To use the caret character, you must place it before the pattern specific end in the regular expression:

$ echo "The book store" | sed -n '/^book/p

$ echo "Books are great" | sed -n '/^Book/p'

Books are great

$

The care t anchor character checks for the pattern at the beginning of each new line of

Data, as determined by the newline character:

$ Cat data3

This is a test line.

This is another test line.

A line that tests this feature.

Yet more testing of this

$ Sed -n '/^this/p' data3

This is another test line.

$

As long as the pattern appears at the start of a new line, the caret anchor catches it.

Looking for the ending

The opposite of looking for a pattern at the start of a line is looking for it at the end of a line. The dollar sign ($) special character defy nest the end anchor. Add this special character after a text pattern to indicate that the line of data must end with the text pattern:

$ echo "This is a good book" | sed -n '/book$/p'

This is a good book

$ echo "This book is good" | sed -n '/book$/p'

$

The problem with an ending text pattern is that you must be careful what you're looking for:

$ echo "There are a lot of good books" | sed -n '/book$/p'

Combining anchors:

In some common situations, you can combine both the start and end anchor on the same

Line. In the if rest situation, suppose you want to look for a line of data containing only a

Specific c text pattern:

$ Cat data4

This is a test of using both anchors

I said this is a test

This is a test

I'm sure this is a test.

$ Sed -n '/^this is a test$/p' data4

This is a test

$

The sed editor ignores the lines that include other text besides the specific end text.

The second situation may seem a little odd at if rest but is extremely useful. By combining

Both anchors in a pattern with no text, you can I filter blank lines from the data stream.

Consider this example:

$ Cat data5

This is one test line.

This is another test line.

$ Sed '/^$/d' data5

This is one test line.

This is another test line.

$

The regular expression pattern that is defined looks for lines that have nothing between the start and end of the line. Because blank lines contain no text between the two newline characters, they match the regular expression pattern. The sed editor uses the d delete command to delete lines that match the regular expression pattern, thus removing all blank lines from the text. This is an effective way to remove blank lines from document

Character classes

The dot special character is great for matching a character position against any character, but what if you want to limit what characters to match? This is called a character class in regular expressions. You can define a class of characters that would match a position in a text pattern. If one of the characters from the character class is in the data stream, it matches the pattern.

To define a character class, you use square brackets. The brackets should contain any character you want to include in the class. You then use the entire class within a pattern just like any other wildcard character. This takes a little getting used to at first, but after you catch on, it can generate some pretty amazing results. The following is an example of creating a character class: $ sed -n '/ [chi] at/p' data6 the cat is sleeping. That is a very nice hat.

Advanced sed:

Using multiline commands:

Looking at Multiline Commands

When using the basic sed editor commands, you may have noticed a limitation. All the sed editor commands perform functions on a single line of data. As the sed editor reads a data stream, it divides the data into lines based on the presence of newline characters. The sed editor handles each data line one at a time, processing the defy need script commands on the data line, and then moving on to the next line and repeating the processing.

Sometimes, you need to perform actions on data that spans more than one line. This is especially true if you're trying to if nod or replace a phrase.

For example, if you're looking for the phrase Linux System Administrators Group in your data, it's quite possible that the phrase's words can be split onto two lines. If you processed the text using a normal sed editor command, it would be impossible to detect the split phrase.

Fortunately, the designers behind the sed editor thought of that situation and devised a

Solution. The sed editor includes three special commands that you can use to process multiline text:

■ N adds the next line in the data stream to create a multiline group for processing.

■ D deletes a single line in a multiline group.

■ P prints a single line in a multiline group.

The following sections examine these multiline commands more closely and demonstrate how you can use them in your scripts.

Navigating the next command

Before you can examine the multiline next command, you first need to look at how the single-line version of the next command works. After you know what that command does, it's much easier to understand how the multiline version of the next command operates.

Using the single-line next command

The lowercase n command tells the sed editor to move to the next line of text in the data stream, without going back to the beginning of the commands. Remember that normally the sed editor processes all the defined commands on a line before moving to the next line of text in the data stream. The single-line next command alters this fl own.

This may sound somewhat complicated, and sometimes it is. In this example, you have a data if le that contains five lines, two of which are empty. The goal is to remove the blank line after the header line but leave the blank line before the last line intact. If you write a sed script to just remove blank lines, you remove both blank lines:

$ Cat data1.txt

This is the header line.

This is a data line.

This is the last line.

$

$ Sed '/^$/d' data1.txt

This is the header line.

This is a data line.

This is the last line.

$

Because the line you want to remove is blank, you don't have any text you can search for to uniquely identify the line. The solution is to use the n command

Combining lines of text

The single-line next command, you can look at the multiline version. The single-line next command moves the next line of text from the data stream into the processing space (called the pattern space) of the sed editor. The multiline version of the next command (which uses a capital N) adds the next line of text to the text already in the pattern space. This has the effect of combining two lines of text from the data stream into the same pattern space.

The lines of text are still separated by a newline character, but the sed editor can now treat both lines of text as one line. Here's a demonstration of how the N command operates: $ cat data2.txt this is the header line. This is the first data line. This is the second data line. This is the last line. $ $ sed '/first/ {N; s/\n/ /}' data2.txt

This is the header line. This is the first data line. This is the second data line. This is the last line. $ The sed editor script searches for the line of text that contains the word "first" in it. When it finds the line, it uses the N command to combine the next line with that line. It then uses the substitution command (s) to replace the newline character with a space. The result is that the two lines in the text file appear as one line in the sed editor output.

Negating a command:

Navigating the multiline delete command:

Introduced the single-line delete command (d). The sed editor uses it to delete the current line in the pattern space. If you're working with the N command, however, you must be careful when using the single-line delete command:

$ Sed 'N ; /System\administrator/d' data4.txt

All System Administrators should attend.

$

The delete command looked for the words System and Administrator in separate lines and deleted both of the lines in the pattern space. This may or may not have been what you intended.

The sed editor provides the multiline delete command (D), which deletes only the first line in the pattern space. It removes all characters up to and including the newline

Character:

$ Sed 'N; /System\administrator/D' data4.txt

Administrator's group meeting will be held.

All System Administrators should attend.

$

The second line of text, added to the pattern space by the N command, remains intact. This comes in handy if you need to remove a line of text that appears before a line that you find a data string in.

Here's an example of removing a blank line that appears before the if rest line in a data

Stream:

$ Cat data5.txt

This is the header line.

This is a data line.

This is the last line.

$

$ Sed '/^$/ {N; /header/D}' data5.txt

This is the header line.

This is a data line.

This is the last line.

$

This sed editor script looks for blank lines and then uses the N command to add the next

Line of text into the pattern space. If the new pattern space contents contain the word

Advanced sed:

Header, the D command removes the first line in the pattern space. Without the combination of the N and D commands, it would be impossible to remove the first blank line without removing all other blank lines.

Navigating the multiline print command

By now, you're probably catching on to the difference between the single-line and multiline versions of the commands. The multiline print command (P) follows along using the sametechnique. It prints only the fi rst line in a multiline pattern space. This includes all characters up to the newline character in the pattern space. It is used in much the same way asthe single-line p command to display text when you use the -n option to suppress output

From the script.

$ Sed -n 'N; /System\administrator/P' data3.txt

On Tuesday, the Linux System

$

When the multiline match occurs, the P command prints only the if rest line in the pattern space. The power of the multiline P command comes into play when you combine it withthe N and D multiline commands.

The D command has a unique feature in that it forces the sed editor to return to the beginning of the script and repeat the commands on the same pattern space (it doesn't read a new line of text from the data stream). By including the N command in the command script, you can effectively single-step through the pattern space, matching multiple lines together.

Next, by using the P command, you can print the if rest line, and then using the D command, you can delete the first line and loop back to the beginning of the script. When you are back at the script's beginning, the N command reads in the next line of text and starts the process all over again. This loop continues until you reach the end of the data stream.

Holding Space:

The pattern space is an active buffer area that holds the text examined by the sed editor while it processes commands. However, it isn't the only space available in the sed editor for storing text.

The sed editor utilizes another buffer area called the hold space. You can use the hold space to temporarily hold lines of text while working on other lines in the pattern space.

The five commands associated with operating with the hold space are shown in

The sed Editor Hold Space Commands

Command Description:

h- Copies pattern space to hold space

H- Appends pattern space to hold space

g -Copies hold space to pattern space

G- Appends hold space to pattern space

i- Exchanges contents of pattern and hold spaces

These commands let you copy text from the pattern space to the hold space. This frees up the pattern space to load another string for processing.

Usually, after using the h or H commands to move a string to the hold space, eventually you want to use the g, G, or x commands to move the stored string back into the pattern space

With two buffer areas, trying to determine what line of text is in which buffer area can sometimes get confusing. Here's a short example that demonstrates how to use the h and g commands to move data back and forth between the sed editor buffer spaces:

$ Cat data2.txt

This is the header line.

This is the first data line.

This is the second data line.

This is the last line.

$

$ Sed -n '/first/ {h; p; n; p; g; p}' data2.txt

This is the first data line.

This is the second data line.

This is the first data line.

Negating a Command:

The sed editor applies commands either to every text line in the data stream or to lines specific call indicated by either a single address or an address range.

You can also configure a command to not apply to a specific address or address range in the data stream.

The exclamation mark command (!) is used to negate a command. This means in situations. Where the command would normally have been activated, it isn't. Here's an example demonstrating this feature:

$ Sed -n '/header/! P’ data2.txt

This is the first data line.

This is the second data line.

This is the last line.

$

The normal p command would have printed only the line in the data2 if le that contained the word header. By adding the exclamation mark, the opposite happens — all lines in the file are printed except the one that contained the word header.

Using the exclamation mark comes in handy in several applications. Recall that earlier in the “Navigating the next command” section showed a situation where a sed editor command wouldn’t operate on the last line of text in the data stream because there wasn’t a line after it. You can use the exclamation point to fix that problem:

$ Sed 'N;

> S/System\administrator/Desktop\nurse/

> S/System Administrator/Desktop User/

> ' data4.txt

On Tuesday, the Linux Desktop

User's group meeting will be held.

All System Administrators should attend.

$

$ Sed '$! N;

> S/System\administrator/Desktop\nurse/

> S/System Administrator/Desktop User/

> ' data4.txt

On Tuesday, the Linux Desktop

User's group meeting will be held.

All Desktop Users should attend.

Changing the Flow:

The sed editor processes commands starting at the top and proceeding toward the end of the script (the exception is the D command, which forces the sed editor to return to the top of the script without reading a new line of text). The sed editor provides a method for altering the fl own of the command script, producing a result similar to that of a structured programming environment.

Branching

In the previous section, you saw how the exclamation mark command is used to negate the effect of a command on a line of text. The sed editor provides a way to negate an entire section of commands, based on an address, an address pattern, or an address range.

This allows you to perform a group of commands only on a specific c subset within the data stream.

Here's the format of the branch command:

[Address] b [label]

The address parameter determines which line or lines of data trigger the branch command. The label parameter defy nest the location to branch to. If the label parameter is not present, the branch command proceeds to the end of the script.

$ Cat data2.txt

This is the header line.

This is the first data line.

This is the second data line.

This is the last line.

$

$ Sed '{2,3b; s/this is/Is this/; s/line./test?/}' data2.txt

Is this the header test?

This is the first data line.

This is the second data l

Testing

Similar to the branch command, the test command (t) is also used to modify the flow of the sed editor script. Instead of jumping to a label based on an address, the test command jumps to a label based on the outcome of a substitution command. If the substitution command successfully matches and substitutes a pattern, the test commands branches to the specified label.

If the substitution command doesn't match the specific end pattern, the test command doesn't branch. The test command uses the same format as the branch command: [address] t [label] like the branch command, if you don't specify a label, sed branches to the end of the script if the test succeeds. The test command provides a cheap way to perform a basic if-then statement on the text in the data stream. For example, if you don't need to make a substitution if another substitution was made, the test command can help:

$ sed'

{> s/first/matched/ > t > s/This is the/No match on/ >

}' data2.txt No match on header line

This is the matched data line

No match on second data line

No match on last line

Replacing via a Pattern:

How to use patterns in the sed commands to replace text in the data stream.

However, when using wildcard characters it’s not easy to know exactly what text will match the pattern.

For example, say that you want to place double quotation marks around a word you match in a line. That’s simple enough if you’re just looking for one word in the pattern to match:

$ echo "The cat sleeps in his hat." | sed’s/cat/"cat"/'

The "cat" sleeps in his hat.

$

But what if you use a wildcard character (.) in the pattern to match more than one word?

$ echo "The cat sleeps in his hat." | sed’s/.at/".at"/g'

The ".at" sleeps in his ".at

$

Using the ampersand:

The sed editor has a solution for you. The ampersand symbol (&) is used to represent the matching pattern in the substitution command. Whatever text matches the pattern defy need, you can use the ampersand symbol to recall it in the replacement pattern.

This lets you manipulate whatever word matches the pattern defy need: $ echo "The cat sleeps in his hat." | sed’s/.at/"&"/g' the "cat" sleeps in his "hat". $ When the pattern matches the word cat, “cat” appears in the substituted word. When it matches the word hat, “hat” appears in the substituted word.

Replacing individual words:

The ampersand symbol retrieves the entire string that matches the pattern you specify in the substitution command. Sometimes, you’ll only want to retrieve a subset of the string. You can do that, too, but it’s a little tricky. The sed editor uses parentheses to define a substring component within the substitution pattern.

You can then reference each substring component using a special character in the replacement pattern. The replacement character consists of a backslash and a number. The number indicates the substring component's position. The sed editor assigns the if rest component the character \1, the second component the character \2, and so on.

Look at an example of using this feature in a sed editor script:

$ echo "The System Administrator manual" | sed '

> s/\ (System\) Administrator/\1 User/'

The System User manual

$

This substitution command uses one set of parentheses around the word System identifying it as a substring component. It then uses the \1 in the replacement pattern to recall

The first identify end component. This isn't too exciting, but it can really be useful when working with wildcard patterns.

Placing sed Commands in Scripts

Now that you've seen the various parts of the sed editor, it's time to put them together

And use them in your shell scripts. This section demonstrates some of the features that you

Should know about when using the sed editor in your bash shell scripts.

Using wrappers

You may have noticed that trying to implement a sed editor script can be cumbersome,especially if the script is long. Instead of having to retype the entire script each time you want to use it, you can place the sed editor command in a shell script wrapper. The wrapper acts as a go-between for the sed editor script and the command line.

Once inside the shell script, you can use normal shell variables and parameters with your sed editor scripts. Here's an example of using the command line parameter variable as the

input to a sed script:

$ cat reverse.sh

#!/bin/bash

Shell wrapper for sed editor script.

to reverse text files lines.

#

Sed -n '{1! G; h; $p}' $1

#

$

The shell script called reverse uses the sed editor script to reverse text lines in a data stream. It uses the $1 shell parameter to retrieve the if rest parameter from the command

Line, which should be the name of the if, le to reverse:

$ ./reverse.sh data2.txt

This is the last line.

This is the second data line.

This is the first data line.

This is the header line.

$

Now you can easily use the sed editor script on any if le, without having to constantly retype the entire command line.

Redirecting sed output

By default, the sed editor outputs the results of the script to STDOUT. You can employ all the standard methods of redirecting the output of the sed editor in your shell scripts.

You can use dollar sign/parenthesis, $(), to redirect the output of your sed editor command to a variable for use later in the script. The following is an example of using the sed

Script to add commas to the result of a numeric computation:

$ Cat fact.sh

#! /bin/bash

add commas to number in factorial answer

#

Factorial=1

Advanced sed

Counter=1

Number=$1

#

While [$counter -le $number]

Do

 Factorial=$[$factorial * $counter]

 Counter=$[$counter + 1]

Done

#

Result=$(echo $factorial | sed '{

: start

```
S/\ (.*[0-9]\)\([0-9]\{3\}\)/\1, \2/

T start

}')

#

Echo "The result is $result"

#

$

$ ./fact.sh 20

The result is 2,432,902,008,176,640,000

$
```

After you use the normal factorial calculation script, the result of that script is used as the input to the sed editor script, which adds commands.

Creating sed Utilities:

As you've seen in the short examples presented so far in this chapter, you can do lots of cool data-formatting things with the sed editor. This section shows a few handy well-known sed editor scripts for performing common data-handling functions. Spacing with double lines to start things off, look at a simple sed script to insert a blank line between lines in a text if le: $ sed 'G' data2.txt this is the header line. This is the first data line. This is the second data line.

You may have noticed that this script also adds a blank line to the last line in the DataStream, producing a blank line at the end of the if le. If you want to get rid of this, you canuse the negate symbol and the last line symbol to ensure that the script doesn't add the blank line to the last line of the data stream:

```
$ sed '$!G' data2.txt

This is the header line.

This is the first data line.
```

This is the second data line.

This is the last line.

Advanced gawk:

Using Variables

One important feature of any programming language is the ability to store and recall values using variables. The gawk programming language supports two different types of variables:

■ Built-in variables

■ User-defi ned variables Several built-in variables are available for you to use in gawk. The built-in variables contain information used in handling the data fields and records in the data if le. You can also create your own variables in your gawk programs. The following sections walk you through how to use variables in your gawk programs.

Built-in variables

The gawk program uses built-in variables to reference specifi c features within the program data. This section describes the built-in variables available for you to use in your gawk programs and demonstrates how to use them. The field and record separator variables .demonstrated one type of built-in variable available in gawk: the data field variables. The data if eld variables allow you to reference individual data fields within a data record using a dollar sign and the numerical position of the data if eld in the record.

Thus, to reference the if rest data if eld in the record, you use the $1 variable. To reference the second data if eld, you use the $2 variable, and so on. Data if elds are delineated by a if eld separator character. By default, the if eld separator character is a whitespace character, such as a space or a tab. IT showed how to change the if eld separator character either on the command line by using the -F command line parameter or within the gawk program using the special FS built-in variable.

User-defi ned variables:

Just like any other self-respecting programming language, gawk allows you to defi ne your

own variables for use within the program code. A gawk user-defi ned variable name can be any number of letters, digits, and underscores, but it can’t begin with a digit. It is also

important to remember that gawk variable names are case sensitive.

Assigning variables in scripts

Assigning values to variables in gawk programs is similar to doing so in a shell script, using an assignment statement:

```
$ gawk '
> BEGIN{
> testing="This is a test"
> print testing
> }'
This is a test
$
```

The output of the print statement is the current value of the testing variable. Like shell

script variables, gawk variables can hold either numeric or text values:

```
$ gawk '
> BEGIN{
> testing="This is a test"
> print testing
> testing=45
> print testing
> }'
```

This is a test

45

$

Structured Commands:

The gawk programming language supports the usual cast of structured programming commands. This section describes each of these commands and demonstrates how to use them

within a gawk programming environment.

The if statement

The gawk programming language supports the standard if-then-else format of theif statement. You must defi ne a condition for the if statement to evaluate, enclosed in parentheses. If the condition evaluates to a TRUE condition, the statement immediately following the if statement is executed. If the condition evaluates to a FALSE condition, the statement is skipped. You can use this format:

if (condition)

statement1

Or you can place it on one line, like this:

if (condition) statement1

Here's a simple example demonstrating this format:

$ cat data4

10

606

```
50

34

$ gawk '{if ($1 > 20) print $1}' data4

50

34

$
```

Not too complicated. If you need to execute multiple statements in the if statement, you

must enclose them with braces:

```
$ gawk '{

> if ($1 > 20)

> {

> x = $1 * 2

> print x

> }

> }' data4

100

68

$
```

Be careful that you don’t confuse the if statement braces with the braces used to start and

stop the program script. The gawk program can detect missing braces and produces an error message if you mess up:

```
$ gawk '{

> if ($1 > 20)
```

```
> {
> x = $1 * 2
> print x
> }' data4
gawk: cmd. line:6: }
gawk: cmd. line:6: ^ unexpected newline or end of string
$
```

The gawk if statement also supports the else clause, allowing you to execute one or more statements if the if statement condition fails. Here's an example of using the else clause:

```
$ gawk '{
> if ($1 > 20)
> {
> x = $1 * 2
> print x
> } else
> {
> x = $1 / 2
> print x
607
```



```
> }}' data4
5
2.5
```

```
6.5
100
68
$
```

You can use the else clause on a single line, but you must use a semicolon after the if statement section:

```
if (condition) statement1; else statement2
```

Here's the same example using the single line format:

```
$ gawk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' data4
5
2.5
6.5
100
68
$
```

This format is more compact but can be harder to follow.

The while statement:

The while statement provides a basic looping feature for gawk programs. Here's the formatof the while statement:

```
while (condition)
{
 statements
}
```

The while loop allows you to iterate over a set of data, checking a condition that stops the iteration. This is useful if you have multiple data values in each record that you must use

in calculations:

```
$ cat data5
130 120 135
160 113 140
145 170 215
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
```



```
> total += $i
> i++
> }
> avg = total / 3
> print "Average:",avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$
```

The while statement iterates through the data if elds in the record, adding each value to the total variable and incrementing the counter variable, i. When the counter value is equal to 4, the while condition becomes

FALSE, and the loop terminates, dropping through to the next statement in the script. That statement calculates the average andprints the average. This process is repeated for each record in the data file.

The gawk programming language supports using the break and continue statements in while loops, allowing you to jump out of the middle of the loop:

```
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
> total += $i
> if (i == 2)
> break
> i++
> }
> avg = total / 2
> print "The average of the first two data elements is:",avg
> }' data5
The average of the first two data elements is: 125
The average of the first two data elements is: 136.5
The average of the first two data elements is: 157.5
$
```

The break statement is used to break out of the while loop if the value of the i

variable is 2.

The do-while statement

The do-while statement is similar to the while statement but performs the statements

before checking the condition statement. Here's the format for the do-while statement:

do

{

609

Chapter 22: Advanced gawk

22

statements

} while (condition)

This format guarantees that the statements are executed at least one time before the condition is evaluated. This comes in handy when you need to perform statements before evaluating the condition:

$ gawk '{

> total = 0

> i = 1

> do

> {

> total += $i

> i++

> } while (total < 150)

> print total }' data5

250

160

315

$

The script reads the data if elds from each record and totals them until the cumulative value

reaches 150. If the if rest data if eld is over 150 (as seen in the second record), the script is

guaranteed to read at least the if rest data if eld before evaluating the condition.

The for statement

The for statement is a common method used in many programming languages for looping.

The gawk programming language supports the C-style of for loops:

for(variable assignment; condition; iteration process)

This helps simplify the loop by combining several functions in one statement:

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
> total += $i
> }
> avg = total / 3
> print "Average:",avg
> }' data5
Average: 128.333
Average: 137.667
```

Average: 176.667

$

By defi ning the iteration counter in the for loop, you don’t have to worry about incrementing it yourself as you did when using the while statement.

610