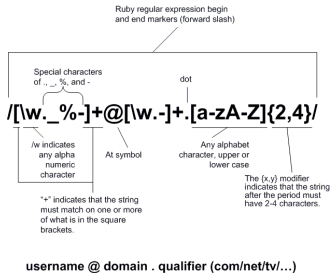
SYST13416
Introduction to Linux
Operating

Regular Expressions

REGULAR EXPRESSIONS

Objectives

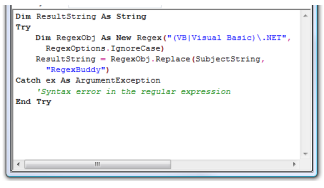
- Understand how to write a basic regular expression
- Search for text in a file or a group of files in the same directory with a regular expression
- Use **modifiers**: asterisk (*) and period (.); plus (+) and question mark (?)
- Use **anchors**: hat (^) and ($); word boundaries (\<) and (\>)
- Use **character sets**, i.e. [abc], [0-9], [^abc]
- Use **escape character** \
- Linux commands with regular expressions: **grep, egrep,** and **sed**
- **fgrep** searches files for one or more pattern arguments. *It does not use regular expressions;* instead, it does direct string comparison to find matching lines of text in the input.



Regular Expressions (REGEX)

- Used to specify patterns of characters; often used in search-and-replace operations.
- As a tool, regular expressions are so useful and so powerful that, mastering the art of using regular expressions is one of the most important things you can do to become proficient with Linux and other computing environments.
- Regular expressions are used with many programming languages, including awk, C, C++, C#, Java, Perl, PHP, Python, Ruby, Tcl, and VB.NET.
- The term 'regular expression" comes from computer science and refers to a set of rules for specifying patterns, comes from work of the eminent American mathematician and computer scientists, Stephen Kleene (1909-1994), later adopted by Ken Thompson in the early versions of Unix.

In Visual basic:



```
Dim ResultString As String
Try
    Dim RegexObj As New Regex("(VB|Visual Basic)\.NET",
      RegexOptions.IgnoreCase)
    ResultString = RegexObj.Replace(SubjectString,
      "RegexBuddy")
Catch ex As ArgumentException
    'Syntax error in the regular expression
End Try
```

Basic and Extended Regular Expressions

- Modern version, extended regular expressions (ERE) is the current standard (IEEE 1003.2 and POSIX)
- The older version, basic regular expressions (BRE) is more primitive, less powerful—replaced by ERE, but supported for backward compatibility
- Most significant differences between ERE and BRE:

ERE	BRE	MEANING
( )	\(\)	Define a bound (brace brackets)
( )	\(\)	Define a group (parentheses)
?	\(0,1\)	Match zero or one times
+	\(1,\)	Match one or more times
|		Alternation: match one of the choices
[:name:]		Predefined character class

Perl example

- extracting values from an associative array (hash) in Linux environment

```
sub Parse_Form {
        if ($ENV('REQUEST_METHOD') eq 'GET') {
                @pairs = split(/&/, $ENV('QUERY_STRING'));
        } elsif ($ENV('REQUEST_METHOD') eq 'POST') {
                read (STDIN, $buffer, $ENV('CONTENT_LENGTH'));
                @pairs = split(/&/, $buffer);

                if ($ENV('QUERY_STRING')) {
                        @getpairs =split(/&/, $ENV('QUERY_STRING'));
                        push(@pairs,@getpairs);
                        }
        } else {
                print "Content-type: text/html\n\n";
                print "<P>Use Post or Get";
        }

        foreach $pair (@pairs) {
                ($key, $value) = split (/=/, $pair);
                $key =~ tr/+/ /;
                $key =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
                $value =~ tr/+/ /;
                $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;

                $value =~s/<!--(.|\n)*-->//g;

                if ($formdata{$key}) {
                        $formdata{$key} .= ", $value";
                } else {
                        $formdata{$key} = $value;
                }
        }
}
1;
```

Javascript example

- Regular expressions used in Windows environment

```
function myValidate(myForm) {
    if (myForm.fullname.value == "") {
        alert ("Please fill in your name");
        return;
    }
    // postal code validation ^ is start $ is end [a-z] is letter \d is number i is ign
    if (!myForm.postal.value.match(/^[a-z]\d[a-z]( |-)\d[a-z]\d$/i)) {
        alert ("Please use a valid postal code in format a2a-2a2");
        return;
    }
    // simple e-mail check - .+ means one or more chars \. means a .
    if (!myForm.email.value.match(/^.+@.+\..+$/)) {
        alert ("Please use a correct e-mail format (simple test)");
        return;
    }
    // complex e-mail check, the ¡-ÿ portion is all these weird characters:
    // ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕ
    // ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
    // the [^ etc] means not all those characters, the (2,4) means two to four
    if (!myForm.email.value.match(/^[^ <>()@¡-ÿ]+@[^ <>()@¡-ÿ]+\.[a-zA-Z](2,4)$/)) {
        alert ("Please use a correct e-mail format (complex test)");
        return;
    }
```


Dictionary file can be found at one of these places:

/usr/share/dict/words

/usr/dict/words

/usr/share/lib/dict/words

GREEDY AND LAZY

*By default, pattern matching is **greedy**, which means that the matcher returns the longest match possible. For example, applying the pattern A.*c to AbcAbcA matches AbcAbc rather than the shorter Abc. To do **non-greedy** (or **lazy**) matching, a question mark must be added to the quantifier. For example, the pattern A.*?c will find the shortest match possible.*

Elements of Regex

- You form a regular expression by combining literals and meta-characters with the rules below.
- Some programs will accept all of these, others may only accept some. Best way is to consult the man page for the program to determine whether you need to use ERE or BRE.
- Regular expressions come in three different forms:

  Anchors—*which tie the pattern to a location on the line*
  Character sets—*which match a character at a single position*
  Modifiers—*which specify how many times to repeat the previous expression*

- The following UNIX commands allow the use of regular expressions:
  grep, egrep, fgrep, sed, awk, perl, vi (when searching and substituting)
- Regular expression syntax is as follows:

*****	match zero or more instances of the single character immediately preceding it
[abc]	match any of the characters enclosed
[a-d]	match any character in the enclosed range
[^expr]	match any character not in the following expression
abc$	the regular expression must end at the end of the line
^abc	the regular expression must start at the beginning of the line
****	treat the next character literally. This is usually used to escape the meaning of special characters sch as "*"
.	match any single character except <newline>
\<abc\>	will match the enclosed regular expression as long as it is a separate word.

- Word boundaries are defined as beginning of a newline or anything expect a letter, digit, or underscore and ending with the same or an end-of-line character. The \< and \> are single operators.

SEARCHING FOR PATTERNS WITH GREP

Objectives

- Linux commands with regular expressions: **grep, egrep**
- **fgrep** searches files for one or more pattern arguments. *It does not use regular expressions;* instead, it does direct string comparison to find matching lines of text in the input.

COMMAND grep

cat	the string cat
.at	any occurrence of a letter, followed by at, i.e., rat, cat, bat
xy*z	any occurrence of an x, followed by 0 or more y's, followed by z
^cat	cat at the beginning of the line
*****	any occurrence of an asterisk
[cC]at	cat or Cat
[^a-zA-Z]	any occurrence of a non-alphabetic character
[A-Z][A-Z]	one or more upper case letters
[A-Z]*	zero or more upper case letters (anything)
cat$	cat at the end of the line
[0-9]$	any line ending with a number

Finding stuff with grep

- Examine the following manual pages for the commands grep, egrep, and fgrep, and note the description and any options used are in these exercises.

Matching Lines and Words

- Regular expression is a compact way of specifying a pattern of characters.
- To create a regular expression, you put together ordinary characters and meta-characters according to certain rules, which are used to search within text files.
- When a regex corresponds to a particular string of characters, we say that it matches the string.
- **RULE**: *All ordinary characters, such as letters and numbers, match themselves.*
- Given the following file named **data** that contains the following:

```
Harry is smart
Harry
His name is Harry
His name is Harry Potter.
The dog likes bones.
```

- To find all lines that contain the pattern "Harry" anywhere on the line:

```
grep Harry data
```

Using Anchors

- You can use **anchors** to specify the location of the pattern on the line.
- The **^** (*circumflex* or *hat*) meta-character is an anchor that matches the **beginning** of a line:

```
grep '^Harry' data
```

- The anchor that matches the **end** of a line is the **$** (*dollar*) meta-character:

```
grep 'Harry$' data
```

- To find lines that contain only the word 'Harry', combine both anchors:
  `grep '^Harry$' data`
- Add the following lines to file **data**
```
I know who you are and I saw what you did.
Who knows what evil lurks in the hearts of men?
color
colour
```
- Using both anchors with nothing in between is an easy way to look for empty lines
  `grep '^$' data`
- The **anchors** to match the **beginning** or **end** of the **word** is the two-character combination **\<** and **\>** respectively.
- To find all lines that contain the word starting with "kn", use
  `grep '\<kn' data`
- To find all lines that contain the word ending with "ow", use
  `grep 'ow\>' data`
- You want to find all the lines in the file data that contain the word "color" (American spelling) or "colour" (British spelling):
  `grep 'colou*r' data`

  `egrep 'colou?r' data`

REPETITION

{n}	*match exactly n times*
{n,}	*match n or more times*
{n,m}	*match n to m times*

Character Sets and Repetition

- What are all the English words that begin with "qu"?
  `grep '^qu' /usr/share/dict/words`
- What are all the English words that end with "y"?
  `grep 'y$' /usr/share/dict/words`
- What are all the English words that begin with "qu" and end with "y"?

```
grep '^quy$' /usr/share/dict/words

grep '^qu[a-z]y$' /usr/share/dict/words
quay

grep '^qu[a-z][a-z]y$' /usr/share/dict/words
query

grep –E '^qu[a-z]{1}y$' /usr/share/dict/words
egrep '^qu[a-z]{1}y$' /usr/share/dict/words
quay

egrep '^qu[a-z]{2}y$' /usr/share/dict/words
query

egrep '^qu[a-z]{3}y$' /usr/share/dict/words
quarry
queasy
quirky

egrep '^qu[a-z]{1,3}y$' /usr/share/dict/words
quarry
quay
queasy
query
quirky

egrep '^qu[a-z]+y$' /usr/share/dict/words
```

Contents of num.list file:

```
_1<tab>15<tab>fifteen
_2<tab>14<tab>fourteen
_3<tab>13<tab>thirteen
_4<tab>12<tab>twelve
_5<tab>11<tab>eleven
_6<tab>10<tab>ten
_7<tab>_9<tab>nine
_8<tab>_8<tab>eight
_9<tab>_7<tab>seven
10<tab>_6<tab>six
11<tab>_5<tab>five
12<tab>_4<tab>four
13<tab>_3<tab>three
14<tab>_2<tab>two
15<tab>_1<tab>one
```

wordlist.txt *is available on the server in the instructor's public directory.*

EXPLORING THE GREP COMMAND

- Test the following commands and analyze the results. Describe the effect of the options. Use the ***num.list*** file you created earlier. If you do not already have the file, create it as it appears on the left side of this page. Note that the fields are separated by a [tab] and _ represents a [space].

```
grep '15' num.list
grep -c '15' num.list
grep '1[125]' num.list
grep '^ ' num.list
grep '^[^ ]' num.list
grep –v '^ ' num.list
grep '^[1-9]' num.list
grep 'te*' num.list
grep 'tee*' num.list
```

- At the command line, execute the following command and note the results:

```
who | grep '^s'
```

- You can copy the file ***wordlist.txt*** from the instructor's directory (ask for instructions) or create it and include the lines listed below among others. To indicate zero or more occurrences of any character, you can use the period and the asterisk together (.*), as in this example:

```
grep 'dog.*bone' wordlist.txt
```

- This command would display lines that contain the following strings:

```
dogbone
dog-bone
doggy bone
My dog has a bone.
My dog is named Rover. Please add a soup bone to the grocery list.
```


SYST13416
Introduction to Linux Operating

Regular Expressions

USING THE STREAM EDITOR

Objectives

- Linux commands with regular expressions: **sed**

Batch editing with SED

- *Sed* is the ultimate **s**tream **ed**itor. It is a very specialized and simple language.
- The syntax is
  sed [options] '{command}' [filename]

Substitute Command

- *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" in the file called **old** to "night" in the file called **new**:
  sed s/day/night/ <old >new
  sed s/day/night/ old >new
  echo day | sed s/day/night/
- I didn't put quotes around the argument because this example didn't need them. However, I recommend you do use quotes. **If you have meta-characters in the command, quotes are necessary.** And if you aren't sure, it's a good habit, and "best practice". Use the strong (single quote) character:
  sed 's/day/night/' <old >new
- This would output the word "Sunnight" bacause sed found the string "day" in the input.
  echo Sunday | sed 's/day/night/' <old >new
- There are four parts to this substitute command:

```
s             Substitute command
/../../       Delimiter
day           Regular Expression Pattern Search Pattern
night         Replacement string
```

  The search pattern is on the left hand side and the replacement string is on the right hand side.

Command Result

a	Append text below current line.
c	Change text in the current line with new text
i	Insert text above current line.
d	Delete text.
p	Print text.
r	Read a file.
s	Search and replace text.
w	Write to a file.

Quicken's .QIF format conversion script:
http://www.gentoo.org/doc/en/articles/l-sed3.xml

Global Changes

- Suppose the message that is to be changed contains more than one occurrence of the item to be changed. By default, the result can be different than what was expected, as the following illustrates:
  echo The tiger cubs will meet this Tuesday at the same time as the meeting last Tuesday | sed 's/Tuesday/Thursday/'
  The tiger cubs will meet this Thursday at the same time as the meeting last Tuesday
- Instead of changing every occurrence of "Tuesday" for "Thursday," the sed editor moves on after finding a change and making it, without reading the whole line. The majority of sed commands function like this.
- In order for every occurrence to be substituted, in the event that more than one occurrence appears on the same line, you specify the action to take place globally:
  echo The tiger cubs will meet this Tuesday at the same time as the meeting last Tuesday | sed 's/Tuesday/Thursday/g'
  The tiger cubs will meet this Thursday at the same time as the meeting last Thursday

Conditional Changes

- When you want to make a change if certain conditions are met—for example, following a match of some other data, consider the following text file (**sample1**):

```
one     1
two     1
three   1
one     1
two     1
two     1
three   1
```

Source of interesting examples:

http://www.oracle.com/technology/pub/articles/dulaney_sed.html

Bear in mind once again that the only thing changed is the display. If you look at the original file, it is the same as it always was. You must save the output to another file to create permanence. It is worth repeating that the fact that changes are not made to the original file is a true blessing in disguise—it lets you experiment with the file without causing any real harm, until you get the right commands working exactly the way you expect and want them to.

- Suppose that it would be desirable for "1" to be substituted with "2," but only after the word "two" and not throughout every line. This can be accomplished by specifying that a match is to be found before giving the substitute command:

```
sed '/two/ s/1/2/' sample1
one     1
two     2
three   1
one     1
two     2
two     2
three   1
```

- And now, to make it even more accurate:

```
sed '
> /two/ s/1/2/
> /three/ s/1/3/' sample_one
one     1
two     2
three   3
one     1
two     2
two     2
three   3
```

- The following saves the changed output to a new file:

```
sed '
> /two/ s/1/2/
> /three/ s/1/3/' sample_one > sample_two
```

- The output file has all the changes incorporated in it that would normally appear on the screen. It can now be viewed with head, cat, or any other similar utility.

To perform multiple commands per address, enter your sed commands in a file, and use the '{ }' characters to group commands, as follows:

Entering multiple commands per address

```
1,20{
    s/[Ll]inux/GNU\/Linux/g
    s/samba/Samba/g
    s/posix/POSIX/g
}
```

The above example will apply three substitution commands to lines 1 through 20, inclusive. You can also use regular expression addresses, or a combination of the two:

Combination of both methods

```
1,/^END/{
    s/[Ll]inux/GNU\/Linux/g
    s/samba/Samba/g
    s/posix/POSIX/g
    p
}
```

This example will apply all the commands between '{ }' to the lines starting at 1 and up to a line beginning with the letters "END", or the end of file if "END" is not found in the source file.

Restricting Lines

- The default is for the editor to look and edit on every line of the input stream. This can be changed by specifying restrictions preceding the command. For example, to substitute "1" with "2" only in the fifth and sixth lines of the sample file's output, the command would be:

```
sed '5,6 s/1/2/' sample_one
one     1
two     1
three   1
one     1
two     2
two     2
three   1
```

- In this case, since the lines to changes were specifically specified, the substitute command was not needed. Thus you have the flexibility of choosing which lines to changes (essentially, restricting the changes) based upon matching criteria that can be either line numbers or a matched pattern.

Prohibiting the Display

- The default is for sed to display on the screen (or to a file, if so redirected) every line from the original file, whether it is affected by an edit operation or not; the "-n" parameter overrides this action. "-n" overrides all printing and displays no lines whatsoever, whether they were changed by the edit or not.
- For example, to print only lines two through six while making no other editing changes:

```
sed -n '2,6p' sample_one
two     1
three   1
one     1
two     1
two     1
```

Addresses

Sed commands can be given with no addresses, in which case the command will be executed for all input lines;

with one address, in which case the command will only be executed for input lines which match that address; or

with two addresses, in which case the command will be executed for all input lines which match the inclusive range of lines starting from the first address and continuing to the second address.

Three things to note about address ranges: the syntax is

addr1,addr2

(i.e., the addresses are separated by a comma); the line which addr1 matched will always be accepted, even if addr2 selects an earlier line; and if addr2 is a regexp, it will not be tested against the line that addr1 matched.

Deleting Lines

- Substituting one value for another is far from the only function that can be performed with a stream editor. There are many more possibilities, and the second-most-used function in my opinion is delete. Delete works in the same manner as substitute, only it removes the specified lines (if you want to remove a word and not a line, don't think of deleting, but think of substituting it for nothing—s/cat//).
- The syntax for the command is:
  '{what to find} d'
- To remove all of the lines containing "two" from the sample1 file:
```
sed '/two/ d' sample1
one     1
three   1
one     1
three   1
```
- To remove the first three lines from the display, regardless of what they are:
```
sed '1,3 d' sample1
one     1
two     1
two     1
three   1
```
- The following command will delete all lines in a file, from the first line through to the first blank line:
  sed '1,/^$/ d' {filename}


```
one     1
two     1
three   1
one     1
two     1
two     1
three   1
This is where we stop
the test
```

```
one     1
two     1
three   1
This is where we stop
the test
one     1
two     1
two     1
three   1
```

```
one     1
two     1
This is where we stop
the test
three   1
one     1
two     1
two     1
three   1
```

Appending and Inserting Text

- Text can be appended to the end of a file by using sed with the "a" option. This is done in the following manner:
```
sed '$a\
> This is where we stop\
> the test' sample1
```
- Within the command, the dollar sign ($) signifies that the text is to be appended to the end of the file. The backslashes (\) are necessary to signify that a carriage return is coming. If they are left out, an error will result proclaiming that the command is garbled; anywhere that a carriage return is to be entered, you must use the backslash.
- To append the lines into the fourth and fifth positions instead of at the end, the command becomes:
```
sed '3a\
> This is where we stop\
> the test' sample1
```
- This appends the text after the third line. As with almost any editor, you can choose to insert rather than append if you so desire. The difference between the two is that append follows the line specified, and insert starts with the line specified. When using insert instead of append, just replace the "a" with an "i," as shown below:
```
sed '3i\
> This is where we stop\
> the test' sample1
```
- The new text appears in the middle of the output, and processing resumes normally after the specified operation is carried out.

```
one     1
We are no longer using two
three   1
one     1
We are no longer using two
We are no longer using two
three   1
```

```
one     1
two     2
three   3
one     1
two     2
two     2
three   3

cat sample_three
one     1
two     2
three   3
```

The Change Command

- In addition to substituting entries, it is possible to change the lines from one value to another. The thing to keep in mind is that substitute works on a character-for-character basis, whereas change functions like delete in that it affects the entire line:
```
sed '/two/ c\
> We are no longer using two' sample1
```

- Working much like substitute, the change command is greater in scale—completely replacing the one entry for another, regardless of character content, or context. At the risk of overstating the obvious, when substitute was used, then only the character "1" was replaced with "2," while when using change, the entire original line was modified. In both situations, the match to look for was simply the "two."

Reading and Writing Files

- The ability to redirect the output has already been illustrated, but it needs to be pointed out that files can be read in and written out to simultaneously during operation of the editing commands. For example, to perform the substitution and write the lines between one and three to a file called sample_three:
```
sed '
> /two/ s/1/2/
> /three/ s/1/3/
> 1,3 w sample3' sample1
```
- Only the lines specified are written to the new file, thanks to the "1,3" specification given to the w (write) command. Regardless of those written, all lines are displayed in the default output.


```
one     1
three   1
one     1
three   1
```

```
two     1
two     1
two     1
```

```
one     1
two     2
three   3
one     1
two     2
```

```
one     1
two     2
three   3
```

Change All but...

- With most sed commands, the functions are spelled out as to what changes are to take place. Using the exclamation mark, it is possible to have the changes take place everywhere but those specified—completely reversing the default operation.
- For example, to delete all lines that contain the phrase "two," the operation is:
```
sed '/two/ d' sample_one
```

- And to delete all lines except those that contain the phrase "two," the syntax becomes:
```
sed '/two/ !d' sample_one
```

- If you have a file that contains a list of items and want to perform an operation on each of the items in the file, then it is important that you first do an intelligent scan of those entries and think about what you are doing. To make matters easier, you can do so by combining sed with any iteration routine (for, while, until).

Quitting Early

- The default is for sed to read through an entire file and stop only when the end is reached. You can stop processing early, however, by using the quit command. Only one quit command can be specified, and processing will continue until the condition calling the quit command is satisfied.
- To perform substitution only on the first five lines of a file and then quit:
```
$ sed '/two/ s/1/2/' -e '/three/ s/1/3/' -e '5q' sample1
```

- The entry preceding the quit command can be a line number, as shown, or a find/matching command like the following:
```
$ sed '
> /two/ s/1/2/
> /three/ s/1/3/
> /three/q' sample1
```

*A nice thing about the s/// command is that we have a lot of options when it comes to those / separators. If we're performing string substitution and the regular expression or replacement string has a lot of slashes in it, we can change the separator by **specifying a different character after the 's'**. For example, this will replace all occurrences of /usr/local with /usr:*

```
sed -e 's:/usr/local:/usr:g' \
mylist.txt
```

Remove HTML tags from a file

```
sed -e 's/<[^>]*>//g' \
myfile.html
```

"a '<' character followed by any number of non-'>' characters, and ending with a '>' character". This will have the effect of matching the shortest possible match, rather than the longest possible one.

Example 1

- Create a file using the command
  `cat > teaormilk`
  and enter the following three lines:

  India's milk is good.
  tea Red-Label is good.
  tea is better than the coffee.

  Use CTRL-D to exit. After creating the file, give command

  `sed '/tea/s//milk/g' teaormilk`

  Note the results. Examine the manual page for sed and explain the results in your own words. For the following three examples, choose or create a file that contains the information used by the command.

Example 2

- This command changes all incidents of a comma (,) into a comma followed by a space (,). Use a database file that has fields delimited by a comma.
  `sed s/,/,\ /g`

Example 3

- Create a file that contains the following:
  Memo
  Date: September 15, 2006
  To: All Staff
  From: John Smith
  Re: Company Picnic
  This is a reminder that the annual company picnic will be held this Sunday at Wonderland.
- To perform multiple operations on the input precede each operation with the –e (edit) option and quote the strings. For example, to filter for lines containing "Date: " and "From: " and replace these without the colon (:), try:
  `sed –e 's/Date: /Date /'  -e 's/From: /From /'`
- To print only the first 10 lines of the input (a replacement for head):
  `sed –n 1,10p`

Remove the first word on each line (including any leading spaces and the trailing space):

```
cat test3.txt | sed -e 's/^ *[^ ]* //'
```

*The initial ^ * is used to match any number of spaces at the beginning of the line. The [^]* then matches any number of characters that are not spaces (the ^ inside the brace reverses the match on the space), so it matches a single word. The trailing space at the end matches the space found at the end of the first word. The empty replace pattern removes the text.*

Remove the last word on each line:

```
cat test3.txt | \
sed -e 's/^\(.*\) .*/\1/'
```

This command introduces the concept of hold buffers. Hold buffers are used to keep parts of the matched text and to insert that text into the result. The pattern that matches the text between the parentheses is recalled in the substitution pattern by the \1. If an additional set of parentheses were in the match pattern, they would be addressed in the substitution pattern as \2, and so on, for more sets of parentheses. Up to nine hold buffers can be specified. In this example, the pattern contained within the parentheses matches from the start of the line up to the last space (the space after the parentheses).

Hold pattern buffer

- So far we worked with three sed concepts:

  (1) The **input stream** or data before it is modified,

  (2) the **output stream** or data after it has been modified, and

  (3) the **pattern space**, or **buffer** containing characters that can be modified and send to the output stream.
- There is one more "location" to be covered: the **hold buffer** or **hold space**. Think of it as a **spare pattern buffer**. It can be used to "copy" or "remember" the data in the pattern space for later. There are five commands that use the hold buffer: x, h, H, g, G.

The "x" command

- **eXchanges** the pattern space with the hold buffer. By itself, the command isn't useful. **Both are changed**.
- The hold buffer starts out containing a blank line. When the "x" command modifies the first line, line 1 is saved in the hold buffer, and the blank line takes the place of the first line. The second "x" command exchanges the second line with the hold buffer, which contains the first line. Each subsequent line is exchanged with the preceding line. The last line is placed in the hold buffer, and is not exchanged a second time, so it remains in the hold buffer when the program terminates, and never gets printed. This illustrates that care must be taken when storing data in the hold buffer, because it won't be output unless you explicitly request it.

The "h" command

- **copies** the pattern buffer into the hold buffer. The **pattern buffer is unchanged**.
- The **"H" command allows you to combine several lines in the hold buffer**. It acts like the "N" command as lines are appended to the buffer, with a "\n" between the lines. You can save several lines in the hold buffer, and print them only if a particular pattern is found later.

The "g" command

- Instead of exchanging the hold space with the pattern space, you can **copy the hold space to the pattern space** with the **"g" command**. This deletes the pattern space.

  If you want to append to the pattern space, use the **"G" command**. This adds a new line to the pattern space, and copies the hold space after the new line.

```
one     1
two     2
three   3
one     1
two     2
two     2
three   3
```

Frequently used options

option	effect
-e SCRIPT	Add the commands in SCRIPT to the set of commands to be run while processing the input.
-f	Add the commands contained in the file SCRIPT-FILE to the set of commands to be run while processing the input.
-n	Silent mode.
--version	Print version information and exit.

Creating Script Files

- sed allows you to create a script file containing commands that are processed from the file, rather than at the command line, and is referenced via the "-f" option. Consider the following script file:

```
cat sedlist
/two/ s/1/2/
/three/ s/1/3/
```

- It can now be used on the data file to obtain the same results we saw earlier:

```
sed -f sedlist sample1
```

- Notice that apostrophes are not used inside the source file, or from the command line when the "-f" option is invoked. Script files, also known as source files, are invaluable for operations that you intend to repeat more than once and for complicated commands where there is a possibility that you may make an error at the command line. It is far easier to edit the source file and change one character than to retype a multiple-line entry at the command line.
- Write sed script that will:
  Insert above the first line the title EMPLOYEE FILE
  Remove the salaries ending with 000.
  Print the content of the file with the last names and the first names reversed
  Append at the end of the file THE END

```
# The file of this sed script is afile.sed
# the comments go at the beginning of the file here
1,$s/[0-9]*000$//
1,$s/\([a-zA-Z]*\) \([a-zA-Z]*\)/\2 \1/
1i\
EMPLOYEE FILE
$a\
THE END
```

- To run:

```
sed –f afile.sed datebook
```


Contents