LECTURE 7

REGULAR EXPRESSIONS AND GREP

7.1 Regular Expressions

7.1.1 Metacharacters, Wild cards and Regular Expressions

Characters that have special meaning for utilities like grep, sed and awk are called metacharacters. These special characters are usually alpha-numeric in nature; a few example metacharacters are: caret (^), dollar ($), question mark (?), asterisk (*), backslash (\),forward slash (/), ampersand (&), set braces ({ and }), range brackets (`[...]`). These special characters are interpreted contextually by the utilities. If you need to use the special characters as is without any interpretation, they need to be prefixed with the escape character (\). For example, a literal $ can be inserted as \$; a literal \can be inserted as \\. Although not as commonly used, numbers also can be turned into metacharacters by using escape character. For example, a number 2 is a literal number two, while \2 has a special meaning.

Often times, the same metacharacters have special meaning for the shell as well. Thus we need to be careful when passing metacharacters as arguments to utilities. It is a standard practice to embed the metacharacter arguments in single quotes to stop the shell from interpreting the metacharacters.

```
$grep a*b file       ...asterisk gets interpreted by shell
$grep 'a*b' file     ...asterisk gets interpreted by grep utility
```

Although single quotes are needed only for arguments with metacharacters, as a good practice, the *pattern* arguments of the grep, sed and awk utilities are always embedded in single quotes.

```
$grep 'ab' file
$sed '/a*b/' file
```

Wild card	Meaning given by shell
*	Zero or more characters
?	Any single character
`[...]`	Range of characters; `[ab]` selects a or b
	home directory of the user

Table 7.1: Wild cards for Bourne again shell

```
$awk '{print $1}' file
```

As stated before, the shell uses metacharacters for its own work. Typically shell uses metacharacters to provide services like I/O redirection, command substitution, file name substitution etc. Of all the services provided by the shell, file name substitution is very popular and handy. **The metacharacters used for file name substitution are called wild cards**. Three wild cards for Bourne again shell are given in table 7.1.

The patterns containing the characters or metacharacters are called regular expressions. The users can build up regular expressions using characters or metacharacters. A simple regular expression is a*; this regular expression combines the character 'a' and the metacharacter '*' to form a string 'a*'. One regular expression used in the grep example is 'a*b'. Another simple form of regular expression is 'abc'. As a good practice, all the regular expression arguments to the utilities are placed in single quotes. The single quotes stop the shell from possibly interpreting the metacharacters in the regular expressions.

We must understand that **regular expressions describe sequence of characters**. Even though sometimes the sequence of characters form meaningful words, the proper context of understanding the word is that it is a sequence of characters.

7.1.2 Regular Expression Components

A regular expression is made of **atoms** and **operators**. The **atoms** specify what we are looking for and where in the text the match is to be made. The **operators**, which are not required in all regular expressions, combine atoms into complex expressions.

Five types of atoms are encountered in regular expressions. They are:

Single Character This is the simplest form of regular expression. When a single character appears in a regular expression, it matches itself.
Example: s - matches 's'

Dot(.) A dot matches any single character except the newline character - \n. The dot can be combined with other types of atoms to make useful regular expressions.

Examples:

.	matches any single character except \n
a.	matches 'a' followed by any character
si.	matches 'sid', 'sip', sit', 'sir', 'sic' etc.
.t	matches 'it', 'at' etc.

Class The class matches a predefined set of characters or a range of characters. The class of characters are placed in `[ ]` braces. The dash (-) character can be used to specify continuous sequence (range) of characters. The caret(^) character works as **not** operator and can be used to indicate exclusion of characters.
Examples:

`[abc]`	matches a or b or c
`[afz]`	matches a or f or z
`[a-z]`	matches all the lowercase alphabets.
`[a-zA-Z]`	A way to specify multiple ranges in the class. The expression matches all the lower and the uppercase alphabets.
`[a-z0-9]`	matches all the lowercase alphabets and the numbers.
`[^0-9]`	all the characters except the numbers.

Because of the way the class is represented, certain characters (namely -,[,] and ^) need special treatment to represent themselves in the character class.

`-`	The dash must appear as first character of the class or it must be escaped by \
`]`	The] must appear as first character of the class or it must be escape by \
`[`	The [must appear as first character of the class or it must be escapes by \
`^`	It is best to always escape the ^ with \

Examples:

`[-0-9]`	matches the dash and all the numbers
`[]a-z]`	matches the] and all the lowercase alphabets
`[[a-z]`	matches the [and all the lowercase alphabets
`[b\^]`	matches b and ^

Anchor Anchors are position specifiers. Anchors specify the position of the next character in the matching text. Four kinds of anchors are very handy; They are: ^ (beginning of a line), $ (end of a line), \< (beginning of a word) and \> (end of a word). From the semantic point of view, there is a special restriction on location of the ^ and $ characters in the regular expression. A ^ must appear at the beginning of a regular expression to acquire special meaning; if ^ appears anywhere else, it is treated as a normal character. Similarly, a $ must always appear at the end of a regular expression; Otherwise, it is just a normal character.

Back Reference Regular expressions have the capability to mark and save matched subexpressions into saved buffers. Back references provide a way to refer to these

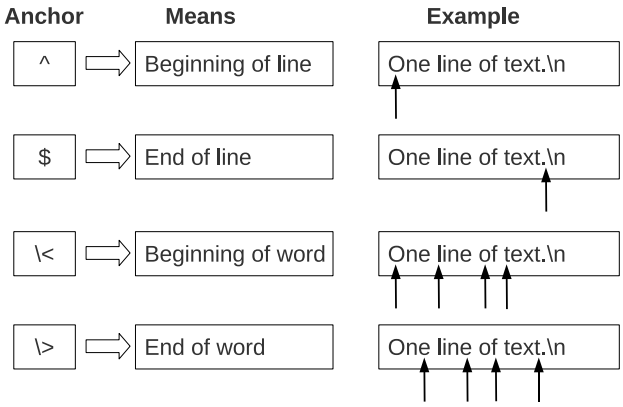

Figure 7.1: Anchors for regular expressions

saved buffers. A total of nine saved buffers are available for usage. These nine buffers can be referenced by using escaped digits. The saved buffers are represented as \1, \2,..., \9.

The second component of regular expressions is **operator**. Regular expression operators are similar to mathematical operators; they combine atoms to form interesting regular expressions. There are five types of operators.

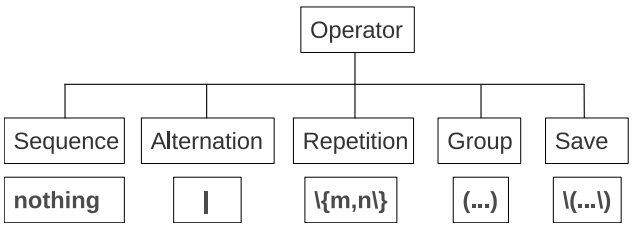

Figure 7.2: Operators for regular expressions

Sequence Sequence operator is a null operator. Sequence implies non-existent join operator between a series of atoms.
Examples:

dog	matches the pattern 'dog'
a..b	matches the pattern a followed any two characters ending with b
[0-3][0-9]	matches all the numbers between 00 and 39
^$	matches a blank line
\<book\>	matches the word 'book'
^[Ll]ong	matches lines with 'Long' or 'long' at the beginning of the line

Alternation The alternation operator (|) defines one or more alternatives.
Example: UNIX|unix - matches either UNIX or unix

Repetition The repetition operator specifies the number of repetitions of an atom or a regular expression. It takes the form \{m,n\}. The various possibilities of the repetition operator are:

\{m,n\}	The number of repetitions allowed is **between m and n**.
\{,n\}	The number of repetitions allowed is **between 0 and n**.
\{m,\}	The number of repetitions allowed are **m or more**.
{m,n}	The number of repetitions allowed is **between m and n**.
{m,}	The number of repetitions allowed are **m or more**.
{,n}	The number of repetitions allowed is **between 0 and n**.
{n}	The number of repetitions allowed is **exactly 'n'**.

The most popular of the repetition possibilities have been given a short notation.There are three widely used short forms.

Long form	Short form	Meaning
\{0,\}	*	zero or more times
\{1,\}	+	one or more times
\{0,1\}	?	zero or one times

The ability to match "zero" or "more" of something is known as closure. Thus *,+,? are called closure operators. The closure operators give elasticity to a regular expression.

Group The group operator pairs up the parts of the regular expression. The braces ( ) indicate the group operator.
Examples:

Reg Ex	Matching String
A(BC){3}	matches ABCBCBC
(F(BC)\{2\}G) \{2\}	FBCBCGFBCBCG

Save The save operator is indicated by escaped parentheses, \(...\). The save operator copies a matched text string to one of nine buffers for later reference. With in an expression, the first saved text is copied to buffer 1, the second saved text is copied to buffer 2, and so on. These saved buffers can be accessed through back references in the later part of the regular expression.
Examples:

Reg Ex	Matching String
^\(.\).*\1$	matches all the lines on which the first character is the same as the last character.

7.1.3 Regular Expression List

The UNIX power tools like grep, sed, awk and vi depend on regular expressions a lot. All the atoms and operators of regular expressions can be separated into two categories: Basic Regular Expressions (BRE) and Extended Regular Expressions (ERE). The grep, sed tools and vi editor support basic regular expressions (BRE); the egrep and awk tools support both basic regular expressions(BRE) and extended regular expressions (ERE).

The BRE are shown in table 7.2 and the ERE are shown in table 7.3. Modern versions of the grep utility are capable of supporting the ERE through '-e' option. The ERE extends the BRE to avoid the clumsy \escape character for group and introduces the +,?, alternation operators.

Metacharacter	Name	Matches
Items to match single character		
.	Dot	Any one character
`[...]`	Character Class	Any characters listed in brackets
`[^...]`	Negated Character Class	Any characters listed in brackets
\char	Escape Character	The characters after the slash literally; used when we want to search for a special character.
Items that match a position		
^	Caret	Start of a line
$	Dollar Sign	End of a line
\<	Backslash less-than	Start of a word
\>	Backslash greater-than	End of a word
The Quantifiers		
*	Asterisk	zero or more of the preceding expression; sometimes used as a wild card
\{min,max\}	Specified Range	Match between *min* and *max* occurrences of the previous regular expression
Others		
-	Dash	Indicates a range
\(...\)	Tagged Expression	Store subexpression matched between parentheses in the next saved buffer.
\1, \2,...,\9	Back Reference	Matches the text previously matched using tagged expressions.

Table 7.2: Metacharacters for Basic Regular Expressions

7.1.4 POSIX Character Classes

POSIX - Portable Operating System Interface - defines regular expression character classes as special metasequences for use within a POSIX bracket expression. The table 7.4 shows the useful ones.

Inside the class (square) brackets, standard metacharacters lose their meaning.

Metacharacter	Name	Matches
Quantifiers		
?	Question Mark	zero or one of the preceding expression; sometimes used as a wild card
+	Plus	one or more of the preceding expression
{m,n}	Specified Range	Match between *min* and *max* occurrences of the previous regular expression
Others		
|	Alternation	Matches either expression given
(...)	Parentheses	Used to limit scope of alternation; also selects subexpression for quantifiers

Table 7.3: Additional Metacharacters for Extended Regular Expressions

Character Class	Meaning
[:alnum:]	alphabetic characters and numerical character
[:alpha:]	alphabetic characters
[:blank:]	space and tab
[:cntrl:]	control characters
[:digit:]	digits
[:lower:]	lowercase alphabetics
[:upper:]	uppercase alphabetics
[:xdigit:]	digits allowed in hexadecimal number (i.e., 0-9a-fA-F]

Table 7.4: POSIX Character Classes

Character	Function
\	Escapes any special characters (awk only). Ex: [a\1] matches a,],1.
–	Indicates a range when not in the first or last position
∧	Indicates a reverse match only when in the first position

7.1.5 Regular Expression Matching

The regular expressions written must match the strings that we want matched and not match anything else. You can evaluate the regular expression pattern matching as follows:

Hits The lines that you wanted to match. A properly written regular expression must have 100% hits.

Misses The lines that you did not want to match. A properly written regular expression must have 100% misses.

Omissions The lines that you did not match but wanted to match. A properly written regular expression must have 0% omissions.

False Alarms The lines that you matched but did not want to match. A properly written regular expression must have 0% false alarms.

Another property of regular expressions is that an expression always tries for greedy match. Sometimes, the result of greedy approach is counter-intuitive.

In regular expressions, a period (.) is equivalent to '?': both match any single character.

7.1.6 Regular Expression Examples

A few interesting regular expressions and the patterns they match are:
(Please note that for the sake of clarity, one empty space is indicated as □ and one tab space is indicated as →. To match tab space, select tab space from the key board; there is no special character for the tab space.)

Regular Expression	Matches
□ □ *	Any number of spaces
.*	Any number of characters
^ □ *	line consisting of spaces
^cat$	lines containing only word 'cat'
^. * $	all lines
".*"	two extreme double quotes on a line
"[^"]*"	two matching double quotes on a line
(subject|date):□	matches 'subject:□' and 'date:□'
^(From|Subject):□	matches 'From:□' and 'Subject:□'
sep[ea]r[ea]te	matches seperete, seperate, separete, separate
colou?r	matches 'color' and 'colour'
(fifteenth|15th|15)	matches 'fifteen', '15th' and '15'
<H[1-6]□*>	matches HTML headers with any number of spaces between the Hn and the closing tag.
Rs[̇0-9]+([̇0-9][0-9])?	matches money represented as rupees and optional paise
[^:]*::	matches beginning of a line, any number of non-colons followed by a double colon

Table 7.5: Regular Expression Examples

7.2 tr

Use:- Translate or delete characters
Syntax:- tr [OPTION]... SET1 [SET2]

Translate, squeeze, and/or delete characters from standard input, writing to standard output.

-c, -C, --complement
first complement SET1

-d, --delete
delete characters in SET1, do not translate

-s, --squeeze-repeats
replace each input sequence of a repeated character that is listed in SET1 with a single occurrence of that character

SETs are specified as strings of characters. Most represent themselves. Interpreted sequences are:

\\	backslash
\f	form feed
\n	new line
\r	return
\t	horizontal tab
CHAR1-CHAR2	all characters from CHAR1 to CHAR2 in ascending order

Translation occurs if -d is not given and both SET1 and SET2 appear. -t may be used only when translating. SET2 is extended to length of SET1 by repeating its last character as necessary. Excess characters of SET2 are ignored.

NOTE: It is a common mistake to place the [ ] brackets around SET1 and SET2.

7.3 grep

UNIX permits combining of the options for a utility. For example

```
$grep -n -r -v tcp dir
            ||
$grep -nrv tcp dir
```

An option with parameter can also be combined, but it must be at the end.

```
$grep -nrvf words.grep dir
```

Here the words.grep parameter corresponds to -f option of grep utility.

7.4 Practice Session

7.4.1 tr

Use:- translate or delete characters

%simple character translation
```
$tr 'oo' 'OO' good_bad.txt
```

%long sequence of characters can be given to tr.
```
$tr 'abcdefghijklmnopqrstuvwxyz' 'ABCDEFGHIJKLMNOPRQRSTUVWXYZ' <good_bad.txt
|| $tr 'a-z' 'A-Z' <good_bad.txt
```

%squeeze out the repeated characters in the output
```
$tr -s 'oo 'u' good_bad.txt
```

%demonstrate the complement selection option the command outputs a sequence of alphabets with no special characters
```
$tr -c 'a-z' '\0' < good_bad.txt
```

%a way to delete the lowercase characters is through '-d' option
```
$tr -d 'a-z' < good_bad.txt
|| $tr 'a-z' '\0' <good_bad.txt
```

%delete CR (̊) from windows text files
```
$tr -d '̊ <line_end_windows.txt
```

%in the above example, SET1 is greater than SET2. In this situation, the last character of SET2 extends to the size of SET1.

%If SET1 is smaller than SET2, the extra characters in SET2 are discarded.
```
$tr 'adcd' 'A-Z' <good_bad.txt
```

%illusion of word translation through 'tr'
```
$tr 'good' 'Good' <good_bad.txt
```
%if you think word translation works in tr, try this
```
$tr 'good' 'excellent' <good_bad.txt
```

%demonstrate POSIX character classes

%shows graphical locations of the non-letter and non-digit characters in the /etc/passwd file
```
$tr '[:alnum:]'  '  '  </etc/passwd
```

%a variation of the above command is to show all the non-alphanumerical characters of a line in serial order
```
$tr -d '[:alnum:]'  < /etc/passwd
|| $tr '[:alnum:]'  '\0' < /etc/passwd
```

%convert all lowercase characters to upper case characters
```
$tr '[:lower:]'  '[:upper:]'  <good_bad.txt
```

%generates words from the file
```
$tr -cs 'a-zA-Z' '\n' <good_bad.txt
```

7.4.2 grep

Use:- global regular expression print - prints lines matching a pattern

egrep = grep -E
fgrep = grep -F <file_name>
rgrep = grep -r

```
$nl virtue.txt
$grep 'virtue' virtue.txt
```
%show lines containing word 'virtue'

```
$grep -n 'virtue' virtue.txt
```
%show line numbers of the selected lines

```
$grep -n -v 'virtue' virtue.txt
|| $grep -nv 'virtue' virtue.txt
```
%show lines not containing the word 'virtue'

```
$grep -ni 'virtue' virtue.txt
```
%ignore the case of the word while matching

```
$grep -ni 'the' virtue.txt
```
%prove that grep searches for patterns, not words by showing

```
$grep -niw 'the' virtue.txt
```
%force grep to search for whole words, not patterns

```
$grep -niw -m 3 'the' virtue.txt
```
%search for the first lines containing the word 'the'

```
$grep -nHT -m 3 'virtue' virtue.txt
```
%tab align the output and show file names

%print the context. Along with the matched line, print two lines before and two lines after.
```
$grep -A 2 -B 2 'compassion' virtue.txt
```

%print equal number of lines before and after (context lines) the matched line
`$grep -C 2 'compassion' virtue.txt`

`$grep -inHTr 'know' tao` %search for pattern in all the file of a directory

%search for patten in all the files of a directory; exclude files whose names have the pattern 'know*'. wild cards (*,?,[...]) are allowed in the filenames.
`$grep -inHTr 'know' --exclude=know* tao`

%search for patterns in all the file in directory tree whose names match the pattern 'know*'
`$grep -inHTr 'know' --include=know* tao`

%take patterns from file and interpret them as basic regular expressions; search for patterns in the file
`$grep -inf words.grep virtue.txt`

%take patterns from file and interpret them as extended regular expressions; search for the patterns in the file
`$grep -inEf words.grep virtue.txt`

%take patterns from file and treat them as plain characters; search for the patterns in the file
`$grep -inFf words.grep virtue.txt`

%list the names of the files belonging to directory tree starting at 'tao' that contain the lines having the pattern
`$grep -lwir 'virtue' tao`

7.4.3 grep and RegEx

`$grep -E -e '^The' virtue.txt` %search for lines that start with the letters 'The'

`$grep -E -e ';$' virtue.txt` %search for lines that end with semi-colon

%match all the words that have 'the' pattern in them only print the words, not the entire line `$grep -noiwE -e '[^]*the[^]*' virtue.txt`

%search for regex patterns of the file, regex.grep in the file 'virtue.txt' `$grep -Ef regex.grep virtue.txt`

7.5 References

The [Bam09] book is the most authoritative and thorough reference for the topics covered in this class, namely regular expressions and grep-family of commands.
The *Chapters 1-3* of [Fri97] are a thorough introduction to the concept of regular expressions. The chapters also cover the meaning of regular expressions in the context of **grep** command.
The *Chapter 4* of [Koc03] provides an example-led explanation of the basic regular expressions. The explanation of *grep* command options is good in this chapter.
The *Chapter 3* of [Dou97] provides good explanation of the regular expressions concepts.
The *Chapter 9* of [For03] provides introductory discussion on regular expressions. The explanation can be used to supplement the material given in [Ker84, Gla03, Koc03, Bam09, Dou97] books.
The *[Chapter 10]* of [For03] covers the basic usage of the grep-family of utilities.
The *Chapter 3* of [Gla03] explores the popular options of *grep* and lists the regular expressions in a table in the Appendix.
The *Chapter 4* of [Ker84] covers text processing utilities. This chapter is still authoritative reference for text processing utilities / filter utilities. A recommended read.
The *Appendix 1* of [Ker84] provides a good overview of the **Editor (ed)**. Since the ed editor forms the basis for grep, sed and awk commands, a good knowledge of editor commands is helpful.
The UNIX Programmer's Manual is the best resource for grep command syntax and usage.