Regular expressions

Regular expressions

- Key to powerful, efficient, and flexible text processing
- Defined as a string composed of letters, numbers, and special symbols, that defines one or more strings
- You have already used them in selecting files when you used asterisk (*) and question mark characters to select filenames
- Used by several Unix utilities such as ed, vi, emacs, grep, sed, and awk to search for and replace strings
  - Checking the author, subject, and date of each message in a given mail folder
    \[egrep "^\(From|Subject|Date\): " <folder>\]
  - The quotes above are not a part of the regular expression but are needed by the command shell
- A regular expression is composed of characters, delimiters, simple strings, special characters, and other metacharacters defined below
- Characters
  - A character is any character on the keyboard except the newline character `\n`
  - Most characters represent themselves within a regular expression
  - All the characters that represent themselves are called literals
  - A special character is one that does not represent itself (such as a metacharacter) and needs to be quoted
    * The metacharacters in the example above (with egrep) are ".", ``, `, `1, and `)`
  - We can treat the regular expressions as a language in which the literal characters are the words and the metacharacters are the grammar
- Delimiters
  - A delimiter is a character to mark the beginning and end of a regular expression
  - Delimiter is always a special character for the regular expression being delimited
  - The delimiter does not represent itself but marks the beginning and end of the regular expression
  - Any character can be used as a delimiter as long as it (the same character) appears at both ends of the regular expression
  - More often than not, people use forward slash `/` as the delimiter (guess why)
  - If the second delimiter is to be immediately followed by a carriage return, it may be omitted
  - Delimiters are not used with the grep family of utilities
- The metacharacters in the regular expressions are
  ^ $ . * [ ] \{ \} \ \ ( )
  - In addition, the following metacharacters have been added to the above for extended regular expressions (such as the one used by egrep)
    + ? | ( )
  - The dash (-) is considered to be a metacharacter only within the square brackets to indicate a range; otherwise, it is treated as a literal
    * Even in this case, the dash cannot be the first character and must be enclosed between the beginning and the end of range characters
• The regular expression search is not done on a word basis but utilities like `egrep` display the entire line in which the regular expression matches.

• Simple strings
  – The most basic regular expression
  – Matches only itself
  – Examples

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>Matches</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/ring/</code></td>
<td>ring</td>
<td>ring</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spring</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ringing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stringing</td>
</tr>
<tr>
<td><code>/Thursday/</code></td>
<td>Thursday</td>
<td>Thursday</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thursday's</td>
</tr>
<tr>
<td><code>/or not/</code></td>
<td>or not</td>
<td>or not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>poor nothing</td>
</tr>
</tbody>
</table>

• Special characters
  – Cause a regular expression to match more than one string
  – Period
    * Matches any character
    * Examples

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>Matches</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/ .alk/</code></td>
<td>All strings that contain a space followed by any character</td>
<td>will talk</td>
</tr>
<tr>
<td></td>
<td>followed by alk</td>
<td>may balk</td>
</tr>
<tr>
<td><code>/ .ing/</code></td>
<td>All strings with any character preceding ing</td>
<td>singing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>before inglenook</td>
</tr>
<tr>
<td><code>/09.17.98/</code></td>
<td>Date with any separator</td>
<td>09/17/98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>09-17-98</td>
</tr>
</tbody>
</table>

• Square brackets
  * Define a class of characters that matches any single character within the brackets
  * If the first character immediately following the left square bracket is a caret `^`, the square brackets define a character class that match any single character not within the brackets
  * A hyphen can be used to indicate a range of characters
  * Within a character class definition, the special characters (backslash, asterisk, and dollar signs) lose their special meaning
  * A right square bracket appearing as a member of the character class can only appear as the first character following the square bracket
  * A caret is special only if it is the first character following the square bracket
  * A dot within square brackets will not be a metacharacter
    - `/07[-]17[-]98/` will not match 07/17/98 but will match 07-17-98
  * Examples
### Regular Expressions

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>Matches</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/[bB]ill/</td>
<td>Member of the character class b and B followed by ill</td>
<td>bill, Bill, billed</td>
</tr>
<tr>
<td>/t[aeiou].k/</td>
<td>t followed by a lowercase vowel, any character, and a k</td>
<td>talkative, stink, teak, tanker</td>
</tr>
<tr>
<td>/number [6-9]/</td>
<td>number followed by a space and a member of the character class 6 through 9</td>
<td>number 60, number 8, get number 9</td>
</tr>
<tr>
<td>/[^a-zA-Z]/</td>
<td>any character that is not a letter</td>
<td>1, 7, @, ., ) Stop!</td>
</tr>
</tbody>
</table>

- Asterisk
  - Can follow a regular expression that represents a single character
  - Represents zero or more occurrences of a match of the regular expression
  - An asterisk following a period matches any string of characters
  - A character class definition followed by an asterisk matches any string of characters that are members of the character class
  - A regular expression that includes a special character always matches the longest possible string, starting as far toward the beginning (left) of the line as possible
  - Examples

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>Matches</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ab*c/</td>
<td>a followed by zero or more b’s followed by a c</td>
<td>ac, abc, abbc, debbcaabbbc</td>
</tr>
<tr>
<td>/ab.*c/</td>
<td>ab followed by zero or more other characters followed by a c</td>
<td>abc, abxc, ab45c, xab 756.345 x cat</td>
</tr>
<tr>
<td>/t.*ing/</td>
<td>t followed by zero or more characters followed by ing</td>
<td>thing, ting, I thought of going</td>
</tr>
<tr>
<td>/([a-zA-Z ]+)/</td>
<td>a string composed only of letters and spaces</td>
<td>1. any string without numbers or punctuation!</td>
</tr>
<tr>
<td>/(.* )/</td>
<td>as long a string as possible between ( and )</td>
<td>Get (this) and (that);</td>
</tr>
<tr>
<td>/([^)])]*/</td>
<td>the shortest string possible that starts with ( and ends with )</td>
<td>(this) Get (this and that)</td>
</tr>
</tbody>
</table>

- Caret and dollar sign
  - A regular expression beginning with a caret `^` can match a string only at the beginning of a line
    - The regular expression cat finds the string cat anywhere on the line but `^cat` matches only if the string cat occurs at the beginning of the line
    - `^` is used to anchor the match to the start of the line
  - A dollar sign `$` at the end of a regular expression matches the end of a line
The regular expression cat finds the string cat anywhere on the line but cat$ matches only if the string cat occurs at the end of the line, it cannot be followed by any character but newline (not even space)

* Examples

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>Matches</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/^T/</td>
<td>a T at the beginning of a line</td>
<td>This line ... That time...</td>
</tr>
<tr>
<td>/^[0-9]</td>
<td>a plus sign followed by a number at the beginning of a line</td>
<td>+5 + 45.72</td>
</tr>
<tr>
<td>/:$/</td>
<td>a colon that ends a line</td>
<td>...below:</td>
</tr>
</tbody>
</table>

- Quoting special characters
  * Any special character, except a digit or a parenthesis, can be quoted by preceding it with a backslash
  * Quoting a special character makes it represent itself
  * Examples

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>Matches</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/end./</td>
<td>all strings that contain end followed by a period</td>
<td>The end. send. pretend.mail</td>
</tr>
<tr>
<td>/\</td>
<td>a single backslash</td>
<td>\</td>
</tr>
<tr>
<td>/*</td>
<td>an asterisk</td>
<td><em>.c an asterisk (</em>)</td>
</tr>
<tr>
<td>[5]</td>
<td>[5]</td>
<td>it was five [5]</td>
</tr>
<tr>
<td>/and/or/</td>
<td>and/or</td>
<td>and/or</td>
</tr>
</tbody>
</table>

- Rules

  - Longest match possible
    * A regular expression always matches the longest possible string, starting as far towards the beginning of the line as possible
  - Empty regular expressions
    * An empty regular expression always represents the last regular expression used
    * Let us give the following command to vi
      
      :s/mike/robert/

      * If you want to make the same substitution again, the following is sufficient
      
      :s//robert/

      * You can also do the following
      
      /mike/  
      :s//robert

- Bracketing expressions

  - Regular expressions can be bracketed by quoted parentheses () and \)
  - The string matching the bracketed regular expression can be subsequently used as quoted digits
  - The regular expression does not attempt to match quoted parentheses
  - A regular expression within the quoted parentheses matches exactly with what the regular expression without the quoted parentheses will match
  - The expressions /\(rexp\)/ and /rexp/ match the same patterns
  - Quoted digits
    * Within the regular expression, a quoted digit (\n) takes on the value of the string that the regular expression beginning with the nth \ matched
* Assume a list of people in the format  
  last-name, first-name initial  
* It can be changed to the format  
  first-name initial last-name  
by the following vi command  
  :%s/\([^-]*\), \(.*\)/\s \t dMNORx7 1 u\n
- Quoted parentheses can be nested  
  * There is no ambiguity in identifying the nested quoted parentheses as they are identified by the opening  
    \ \  
  * Example  
    /\([a-z]\)([A-Z]\)*x\)/  
    matches  
    3 t dMNORx7 1 u

  • Replacement string  
  - vi and sed use regular expressions as search strings with the substitute command  
  - Ampersands (&) and quoted digits (\n) can be used to match the replacement strings within the replacement  
    string  
  - An ampersand takes on the value of the string that the search string matched  
  - Example  
    :s/[0-9][0-9]*/Number &/

  • Word boundaries  
  - The word boundaries in the regular expressions are denoted by any whitespace character, period, end-of-line,  
    or beginning of line  
  - Expressed by  
    \< \t beginning of word  
    \> \t end of word

  • Regular expressions cannot be used for the newline character

sed

  • Stream editor
  • Derivative of ed  
  - Takes a sequence of editor commands  
  - Goes over the data line by line and performs the commands on each line

  • Basic syntax  
    sed 'list of ed commands' filename[s] ...

  • The commands are applied from the list in order to each line and the edited form is written to stdout
  • Changing a pattern in the file  
    sed 's/pat_1/pat_2/g' in_file > out_file
• sed does not alter the contents of the input file
• Quotes around the list of commands are necessary as the sed metacharacters should not be translated by the shell
• Selecting range of lines
• Command to remove the mail header from a saved mail message

\[ \text{sed '1,~/^$/d' in_file > out_file} \]

• Removing the information from the output of the finger command to get only the user id and login time

\[ \text{finger | sed 's/([a-zA-Z][a-zA-Z]*\//.*\([0-9][0-9][0-9][0-9]\) .*/1 \2/'} \]

• Problem: The first line should have been removed as well

\[ \text{finger | sed 's/([a-zA-Z][a-zA-Z]*\//.*\([0-9][0-9][0-9][0-9]\) .*/1 \2/'} | sed '1d' \]

• Indenting a file one tab stop

\[ \text{sed 's/~/>/ file} \]

• The above matches all the lines (including empty lines)
• Problem can be solved by

\[ \text{sed '/./s/~/>/ file} \]

• Another way to do it

\[ \text{sed '*/$/s/~/>/ file} \]

• Multiple commands in the same invocation of sed

\[ \$ \text{finger | sed 's/([a-zA-Z][a-zA-Z]*\//.*\([0-9][0-9][0-9][0-9]\) .*/1 \2/} > 1d' \]

• The commands must be on separate lines
• sed scripts
  – The sed commands can be put into script files and can be executed by

\[ \text{sed -f cmdfile in_file} \]
• Lines containing a pattern can be deleted by

\[ \text{sed '/regexp/'} \]

• Automatic printing
  – By default, sed prints each line on the stdout
  – This can be inhibited by using the -n option as follows

\[ \text{sed -n '/pattern/p'} \]

• Matching conditions can be inverted by the !

\[ \text{sed -n '/pattern/!p'} \]
- The last achieves the same effect as `grep -v`

- Inserting newlines
  - Converting a document from single space to double space
    
    ```
    $ sed 's/$/\n'
    > /
    ```
  - Creating a list of words used in the document
    
    ```
    $ sed 's/[ - ]*[ - ]*/\'
    > /g' file
    ```
  - Counting the unique words used in the document
    
    ```
    $ sed 's/[ - ]*[ - ]*/\'
    > /g' file | sort | uniq | wc -l
    ```

- Writing on multiple files
  
  ```
  $ sed -n '/pat/w file1
  > /pat!w file2' filename
  ```

- Line numbering
  - Line numbers can be used to select a range of lines over which the commands will operate
  - Examples
    
    ```
    $ sed -n '20,30p'
    $ sed '1,10d'
    $ sed '1,/^$/d'
    $ sed -n '/^$/,'"end'/p'
    ```
  - `sed` does not support relative line numbers (difference with respect to `ed`)

**awk**

- Acronym for the last names of its designers – Aho, Weinberger, Kernighan

- Not as good as `sed` but includes arithmetic, variables, built-in functions, and a programming language like C; on the other hand, it is a more general processing model than a text editor

- Looks more like a programming language rather than a text editor

- Mostly used for formatting reports, data entry, and data retrieval to generate reports

- `awk` is easier to use than `sed` but is slower

- Usage is

  ```
  awk 'awk_script' files
  ```

- The `awk_script` looks like

  ```
  pattern { action }
  ```

  ```
  pattern { action }
  ```
Table 1: Summary of sed commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a\</td>
<td>append lines to output until one not ending in \</td>
</tr>
<tr>
<td>b label</td>
<td>branch to command : label</td>
</tr>
<tr>
<td>c\</td>
<td>change lines to following text (as in a)</td>
</tr>
<tr>
<td>d</td>
<td>delete lines</td>
</tr>
<tr>
<td>i\</td>
<td>insert following text before next output</td>
</tr>
<tr>
<td>l</td>
<td>list line, making all non-printing characters visible (tabs appear as &gt;; lines broken with )</td>
</tr>
<tr>
<td>p</td>
<td>print line</td>
</tr>
<tr>
<td>q</td>
<td>quit (for scripts)</td>
</tr>
<tr>
<td>r file</td>
<td>read file, copy contents to stdout</td>
</tr>
<tr>
<td>s/pat1/pat2/f</td>
<td>substitute pat2 for pat1</td>
</tr>
<tr>
<td>f = g</td>
<td>replace all occurrences</td>
</tr>
<tr>
<td>f = p</td>
<td>print</td>
</tr>
<tr>
<td>f = w file, write to file</td>
<td></td>
</tr>
<tr>
<td>t label</td>
<td>test: branch to label if substitution made to current line</td>
</tr>
<tr>
<td>w file</td>
<td>write line(s) to file</td>
</tr>
<tr>
<td>y/str1/str2/</td>
<td>replace each character from str1 with corresponding character from str2 (no ranges allowed)</td>
</tr>
<tr>
<td>=</td>
<td>print current input line number</td>
</tr>
<tr>
<td>!cmd</td>
<td>do sed cmd if line is not selected</td>
</tr>
<tr>
<td>: label</td>
<td>set label for b and t commands</td>
</tr>
<tr>
<td>{</td>
<td>treat commands up to the matching } as a group</td>
</tr>
</tbody>
</table>

• awk reads one line in the file at a time, compares with each pattern, and performs the corresponding action if the pattern matches

• Just like sed, awk does not alter its input files

• The patterns in awk can be regular expressions, or C-like conditions

• grep can be written in awk as

```
awk '/regular expression/ { print }' filename
```

• Either of pattern or action is optional and can be omitted
  – Omitting pattern performs the action on every line
    ```
    awk '{ print }' filename
    ```
  – Omitting action prints matched lines
    ```
    awk '/regular expression/' filename
    ```

• Just like sed, the awk_script can be presented to awk from a file by using

```
awk -f awk_script_file filename
```

• Fields
  – A field is a string of non-blank characters
  – awk splits each input line into fields, separated by blanks or tabs
  – The output of who has six fields as follows
Regular Expressions

sanjiv  console Nov 18 13:26
sanjiv  tty0   Nov 18 13:26 (:0.0)
sanjiv  ttypc  Nov 19 13:27 (:0.0)
vlad    tty7   Nov 19 16:46 (arrak13.umsl.edu)

- The fields are called $1, $2, .... $NF
  * NF is a variable whose value is set to the number of fields
  * NF and $NF are not the same
    - NF is the number of fields
    - $NF is the contents (string) of the last field
- The field separator is white space by default but can be changed by a command line option
  * Changing the field separator to colon (;)
    
    awk -F: '/regular expression/ { action }' file
  * To print the user names and real names in the passwd file
    
    awk -F: '{print $1"\n"$5}' /etc/passwd

- Printing
  - The current input line (or record) is tracked by the built-in variable NR
  - The entire input record is contained in the variable $0
  - To add line numbers to each line, you can use the following
    
    awk '{print NR, $0}' filename
  - Fields separated by comma are printed separated by the field separator – a blank space character by default
  - Complete control of the output format can be achieved by using printf instead of print as follows
    
    awk '{ printf "%4d %s\n", NR, $0 }' filename
  - printf in awk is almost identical to the corresponding C function

- Patterns
  - Checking for people who do not have a password entry in the file /etc/passwd
    
    awk -F: '$_2 == ""' /etc/passwd
  - Checking for people who have a locked password entry
    
    awk -F: '$_2 == "*"' /etc/passwd
  - Other ways to check for empty string

| $2 == ""       | 2nd field is empty |
| $2 == /"$/    | 2nd field matches empty string |
| $2 != /./      | 2nd field does not match any character |
| length($2) == 0 | length of 2nd field is zero |

- The symbol - indicates a regular expression match while !- indicates a regular expression non-match
- length is a built-in function to count the number of characters in the string (or field)
- Any pattern match can be preceded by ! to negate its match as follows
  
  awk -F: '!( $2 == "" )' filename

- Data validation using the number of fields as criterion – line valid if the number of fields is odd
  
  echo $LINE | awk 'NF % 2 != 0'

- Printing excessively long lines (> 72 characters)
awk 'length($0) > 72' filename

- Above problem with more informative solution

    awk '(length($0) > 72) { print "Line", NR, "too long: ", substr($0,1,50)}' filename

- The function substr( s, m, n ) produces the substring of s beginning at position m and with a length of n characters; if n is omitted, it continues to the end of string

- Extracting information with substr

    $ date
    Wed Nov 20 14:27:33 CST 1996
    $ date | awk '{ print substr ( $4, 1, 5 )}'
    14:27

- The BEGIN and END patterns

    - Special patterns used in awk scripts
    - BEGIN actions are performed before the first input line has been read (used to initialize variables, print headings, and like)
      * Setting the field separator within the script
        $ awk 'BEGIN {FS = "":} 
            $2 == "" ' /etc/passwd

    - END actions are done after the last line has been processed
      * Printing the number of lines in the input
        awk 'END { printf NR }' ...

- Arithmetic and variables

    - awk allows you to do more sophisticated arithmetic compared to the shell
    - Adding the numbers in a column (first column), and printing the sum and average
      { s = s + $1 }
      END { print s, s/NR }

    - Variables can be created by users and are initialized to zero by default
    - awk also allows for shorthand arithmetic operators like C
      { s += $1 }
      END { print s, s/NR }

    - Implementing wc in all its generality
      $ awk '{ nc += length ( $0 ) + 1     # number of chars, 1 for \n           nw += NF    # number of words
      END { print NR, nw, nc }' filename

- Variables can also store string of characters and the interpretation is based on context
- awk maintains a number of built-in variables of both types

Developing man pages with [nt]off

- nroff and troff
  - Native Unix programs to format text
  - Based on requests within the documents that start with a period in the first column
- Commonly used requests are

. I  Italicize following line
. B  Following line in bold
. R  Following line in Roman
. br  Break the line
. ce  Center the following line
. fi  Fill lines (Align right margins)
. ft  Set font
. na  No right alignment
. nf  Do not fill lines (Preferable to .na)
. sp  One vertical line

- The manual page

  - Stored in a subdirectory in the directory /usr/man
  - The subdirectory is called man x where x is a digit or character to indicate the section of the manual
  - The sections are numbered 1 to 8 and n and l
    1  User commands
    2  System calls
    3  C Library functions
    4  Devices and network interfaces
    5  File formats
    6  Games and demos
    7  Environments, tables, and troff macros
    8  Maintenance commands
    1  Misc. reference manual pages (Locally developed and installed)
    n  Misc. reference manual pages (New commands)

  - Printed with the man(1) command
    * A shellscript that runs nroff -man but may be compiled on newer machines
    * The locally developed man pages can be tested for printing with nroff -man command
    * The man pages in a given section can be printed by specifying the section number, for example, the man page for the system call umask can be printed by typing the command

      \texttt{man 2 umask}

    If the section number is not specified, the output will be for the user command from section 1

  - The macros for man are discussed in section 7 of the manual and can be invoked by

    man 7 man

- Layout of a Unix manual page

  - The manual page is laid out as per the specifications in the man macro of troff
    * Any text argument may be zero to six words
    * Quotes can be used to include the space character in a "word"
    * Some native nroff conventions are followed, for example, if text for a command is empty, the command is applied to the next line
    * A line starting with .I and with no other inputs italicizes the next line
    * The prevailing indentation distance is remembered between successive paragraphs but not across sections

  - The basic layout of a man page is described by

    .TH COMMAND <section-number>
    .SH NAME
    command \- brief description of function
.B command
options
.SH DESCRIPTION
Detailed explanation of programs and options.
Paragraphs are introduced by .PP
.PP
This is a new paragraph.
.SH FILES
Files used by the command, e.g., passwd(1) mentions /etc/passwd
.SH "SEE ALSO"
References to related documents, including other manual pages
.SH DIAGNOSTICS
Description of any unusual output (e.g., see cmp(1))
.SH BUGS
Surprising features (not always bugs)
- If any section is empty, its header is omitted
- The .TH line and the NAME, SYNOPSIS, and DESCRIPTION sections are mandatory
- The .TH line
  * Begins a reference page
  * The full macro is described by
    .TH command section date_last_changed left_page_footer center_header
  * Sets prevailing indent and tabs to 0.5"
- The .SH lines
  * Section headers
  * Identify sections of the manual page
  * NAME and SYNOPSIS sections are special; other sections contain ordinary prose
  * NAME section
    - Names the command (in lower case)
    - Provides a one-line description of it
  * SYNOPSIS section
    - Names the options, but does not describe them
    - The input is free form
    - Font changes can be described with the .B, .I, and .R macros
    - The name and options are bold while the rest of the information is in roman
  * DESCRIPTION section
    - Describes the commands and its options
    - It tells the usage of the command
    - The man page for cc(1) describes how to invoke the compiler, optimizer, where the output is, but
      does not provide a reference page for the manual
    - The reference page can be cited in the SEE ALSO section
    - However, man(7) is the description of the language of manual macros
    - Command names and tags for options are printed in italics, using the macros .I (print first argument
      in italics) and .IR (print first argument in italic, second in roman)
  * FILES section
    - Mentions any files implicitly used by the commands
  * DIAGNOSTICS section
    - Optional section and generally not present
    - Reports any unusual output produced by the command
- May contain diagnostic messages, exit statuses, or surprising variations of the command's normal behavior

* BUGS section
  - Could be called LIMITATIONS
  - Reports shortcomings in the program that may need to be fixed in a future release

- Other requests and macros for man

  .IP x  Indented paragraph with a tag x
  .LP    Left-aligned paragraph
  .PP    Same as .LP
  .SS    Section subheading