CS 35L: More Commands and Shell Scripting

Lab 6

Hengda Shi

Week 2 Lecture 1

Announcements

- Assignment 2 is hard, please start them early!
- Lab setup
  - $ export LC_ALL='C'
  - Please set your locale using above command
  - We would like the `sort` command to be ASCII character compliant

More Commands

tr (translate or delete characters)

- Translate, squeeze, and/or delete characters from standard input, writing to standard output.
- Usage: tr [OPTION]... SET1 [SET2]
- Examples
  - $ tr 'a' 'z' < file.txt # translate all ‘a’ in the file to ‘z’
  - $ tr [:lower:] [:upper:] or tr [a-Z] [A-Z] # translate all lower case characters to upper case
  - $ tr -c 'A-Za-z' 0
  - $ tr [:alnum:] '\t'
  - $ tr -d ' '
  - $ tr -cd [:digit:]

grep (global regular expression print)

- Usage: grep [OPTION...] PATTERNS [FILE...]
- As the name suggests, grep prints out lines that match patterns in the input using regular expression
- Example:
  - Put text into three files (demo_file demo_file2 demo_file3)
  - $ echo -e "this line is the 1st lower case line in this file.\n\nTwo lines above this line is empty.\n\nAnd this is the last line." > demo_file
  - $ grep "this" demo_file*

Regular Expressions (regexes)

- A text string with special notation for describing a search pattern
- Not compatible with Linux wildcard
- Basic regular expression (BRE) can be directly invoked by `grep`
- Extended regular expression (ERE) can be invoked by `grep -E`
- Fast grep can be invoked by `grep -F`, it only matches fixed strings instead of regular expressions
- regex online tester: https://regex101.com/

Quantifiers

Token	**Match**
.	any single character
.*	any sequence of character
a?	zero or one of a
a*	zero or more of a
a+	one or more of a
a{3}	exactly three of a
a{3,}	3 or more of a
a{3, 6}	between 3 and 6 of a

Anchors

Token	Match
^	start of string
$	end of string

Character Classes

Token	**Match**
[abc]	a single character of a, b or c
[^abc]	a character except: a, b or c
[a-z]	a character in the range: a-z
[^a-z]	a character not in the range: a-z
[a-zA-Z]	a character in the range: a-z or A-Z

Bracket Expressions

Token	Match
[:alnum:]	Alphanumeric characters: ‘[:alpha:]’ and ‘[:digit:]’; in the ‘C’ locale and ASCII character encoding, this is the same as ‘[0-9A-Za-z]’
[:alpha:]	Alphabetic characters: ‘[:lower:]’ and ‘[:upper:]’; in the ‘C’ locale and ASCII character encoding, this is the same as ‘[A-Za-z]’
[:digit:]	Digits: 0 1 2 3 4 5 6 7 8 9
[:blank:]	Blank characters: space and tab
[:graph:]	Graphical characters: ‘[:alnum:]’ and ‘[:punct:]’
[:print:]	Printable characters: ‘[:alnum:]’, ‘[:punct:]’, and space
[:lower:] / [:upper:]	Lower case / upper case characters
[:punct:]	Punctuation characters; in the ‘C’ locale and ASCII character encoding, this is ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

Capturing Groups

- Use parentheses to create a capturing group
- e.g. (abc){3}
- Backreference: matches the substring previously matched by the nth parenthesized subexpression of the regular expression.
    - Example: ‘(a)\1’ matches ‘aa’

Regular Expression Rules (cont’d)

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(...\) and \{...\}.
.	Both	Match any single character except NULL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, .* means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.
$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.

Regular Expression Rules (cont'd)

Character	BRE / ERE	Meaning in a pattern
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{n,m\}	BRE	Termed an interval expression, this matches a range of occurrences of the single character that immediately precedes it. \{n\} matches exactly n occurrences, \{n,\} matches at least n occurrences, and \{n,m\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	Both	Save the pattern enclosed between \(and \) in a special holding space. Up to nine sub-patterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, \(ab\).*\1 matches two occurrences of ab, with any number of characters in between.

BRE (Basic Regex) vs. ERE (Extended)

- In GNU sed, the only difference between basic and extended regular expressions is in the behavior of a few special characters: ‘?’, ‘+’, parentheses, braces (‘{}’), and ‘|’.
- With basic (BRE) syntax, these characters do not have special meaning unless prefixed with a backslash (‘\’); While with extended (ERE) syntax it is reversed: these characters are special unless they are prefixed with backslash (‘\’).

Desired pattern	**Basic (BRE) Syntax**	**Extended (ERE) Syntax**

literal ‘+’ (plus sign)	$ echo 'a+b=c' > foo
	$ grep 'a+b' foo
	a+b=c
		$ echo 'a+b=c' > foo
		$ grep -E 'a\+b' foo
		a+b=c

One or more ‘a’ characters followed by ‘b’ (plus sign as special meta-character)	$ echo aab > foo
	$ grep 'a\+b' foo
	aab
		$ echo aab > foo
		$ grep -E 'a+b' foo
		aab

Ref: https://www.gnu.org/software/sed/manual/html_node/BRE-vs-ERE.html

Regular Expression Example

regex	**matches**
tolstoy	The seven letters tolstoy, anywhere on a line
^tolstoy	The seven letters tolstoy, at the beginning of a line
tolstoy$	The seven letters tolstoy, at the end of a line
^tolstoy$	A line containing exactly the seven letters tolstoy, and nothing else
[Tt]olstoy	Either the seven letters Tolstoy, or the seven letters tolstoy, anywhere on a line
tol.toy	The three letters tol, any character, and the three letters toy, anywhere on a line
tol.*toy	The three letters tol, any sequence of zero or more characters, and the three letters toy, anywhere on a line (e.g., toltoy, tolstoy, tolWHOtoy, and so on)

More Regular Expression Example

- '^$' matches blank lines
- '[vV]ivek[0-9]' matches character v or V followed by ivek and then followed by one digit
- 'foo[0-9][0-9]' matches foo followed by two digits
- '<([a-zA-Z][a-zA-Z0-9]*)\b[^>]*>.*?</\1>' matches HTML tags

Read
https://www.gnu.org/software/grep/manual/html_node/Regular-Expressions.html
for more information on Regular Expression

Task 1

- locate starting and ending spaces and tabs for each line
  - Use extended regex
  - Hint 1: ^ indicates the start of line, $ indicates the end of line
  - Hint 2: \s indicates whitespace, \t indicates tab

Linux Wildcard vs. Regular Expressions

- They are not compatible with each other
- Symbols in wildcard do not mean the same in regex
- $ grep -E "t.*s" demo*

Glob	Regular Expression Equivalent	Description
?(patterns)	(regex)?	Match an optional regex
*(patterns)	(regex)*	Match zero or more occurrences of a regex
+(patterns)	(regex)+	Match one or more occurrences of a regex
@(patterns)	(regex)	Match the regex (one occurrence)

sed (stream editor)

stream editor for filtering and transforming text

Can be used for:

- Printing specific lines or address ranges (p is for print, d is delete)
  - $ sed –n '1p' file.txt (specific) [last line: $p]
  - $ sed –n '1,5p' file.txt (from-to)
  - $ sed –n '1~2p' file.txt (skip) (first~step)
  - $ sed –n '1p;3p' file.txt (mult-lines)
- Deleting text
  - $ sed '1~2d' file.txt
- Substituting text - s/regex/replacement/flags
  - $ sed 's/cat/dog/' file.txt
  - $ sed 's/cat/dog/g' file.txt

More sed commands

- $ sed -n 12,18p file.txt
- $ sed 12,18d file.txt
- $ sed '1~3d' file.txt
- $ sed '1,20 s/Johnson/White/g' file.txt # Replace Pattern1 with Pattern2 in first 20 lines
- $ sed –n ‘/pattern/p’ filename # Print lines having Pattern
- $ sed ‘s/pattern1/pattern2/’ filename # Replace Pattern1 with Pattern2 (First Occurrence)
- $ sed ‘s/pattern1/pattern2/g’ filename # Replace Pattern1 with Pattern2 (whole file)
- $ sed '/regexp/d' file.txt

Pipe (|)

- Pipe lets you feed the standard output from the program on the left as standard input to the program on the right.
- Examples
  - Suppose we have 8 files: barry.txt bob example.png f g q w z. What would be the output of `$ **ls | head -3**`?
  - What would be the output of `$ **ls | head -3 | tail -1**` given the same files?

Use Pipe and Redirection together

- List all content in the current working directory, remove all new lines, put them into a file
  - List all content using ‘ls’, remove using command ‘tr’, redirect using ‘>’
  - $ ls | tr -d ‘\n’ > file2

Task 2 (grep & sed)

- Checking for the given string in multiple files: $ grep "string" FILE_PATTERN
- Create a file f1.txt and copy f1.txt to f2.txt
- Content of f1.txt and f2.txt:
  UPPER CASE LINE
  Lower case line
  Break;
  empty LINE
  Last line
- Replace last line of f2.txt with ‘End line’ using sed and write to f3.txt
- Check for the given string ‘line/LINE’ in text files which start with ‘f’ and end with a number using egrep and regex

Task 3

- Create file with following text (singers.txt):

  1, Justin Timberlake
  2, Taylor Swift
  3, Mick Jagger
  4, Lady Gaga
  5, Johnny Trash
  6, Elvis Presley
  7, John Lennon

- Print all lines having 'John' using sed

Quoting

- To preserve literal meaning of special characters
- Escape Character \ - Literal value of following character
  - $ echo \|
- single quote - literal meaning of all things inside double quotation
  - $ hello=1
  - $ str='$hello' # saves value to str as literal string ‘$hello’
  - $ str="$hello" # saves value stored in variable hello to str
  - $ echo $str
- Double Quote - Literal meaning except for $, ` and \
  - $ hello=1
  - str="abc$hello"
  - echo $str -> abc1
- Backquote - execute the command
  - echo `ls` -> prints result after running ls

Fun command: cron (http://bit.ly/cron_)

- Linux Cron utility is an effective way to schedule a routine background job at a specific time and/or day on an on-going basis.
- Won't be going in detail, search for it
- View crontab entries
  - crontab –l
- Example: Scheduling a Job For a Specific Time
  - 30 08 10 06 * /home/mithal/script
  - 30 – 30th Minute
  - 08 – 08 AM
  - 10 – 10th Day
  - 06 – 6th Month (June)
  - * – Every day of the week

Shell Scripting

Compiled vs. Interpreted Languages

Compiled Languages

- Examples: C/C++, Golang
- Programs are translated from their original source code into object code, executed by hardware
- Work at low level, dealing with bytes, integers, floating points, etc

Advantages:

- Efficient
- Easier to debug

Disadvantages:

- Portability: compiled program relies on current machine's architecture, program might not be usable on other machine with different architecture

Interpreted Languages

- Examples: Python, Bash script, Javascript
- An interpreter reads a program, and executes it line by line

Advantages:

- Easier to develop
- Better portability

Disadvantages:

- Hard to debug
- Bad performance

Compiled vs. Interpreted Languages

- Some languages also combine the advantages of those type: Java
- The source code written in Java will be compiled into bytecode, and interpreted by jvm (Java Virtual Machine).
- It helps the program to achieve higher performance while maintaining portability among different platforms.

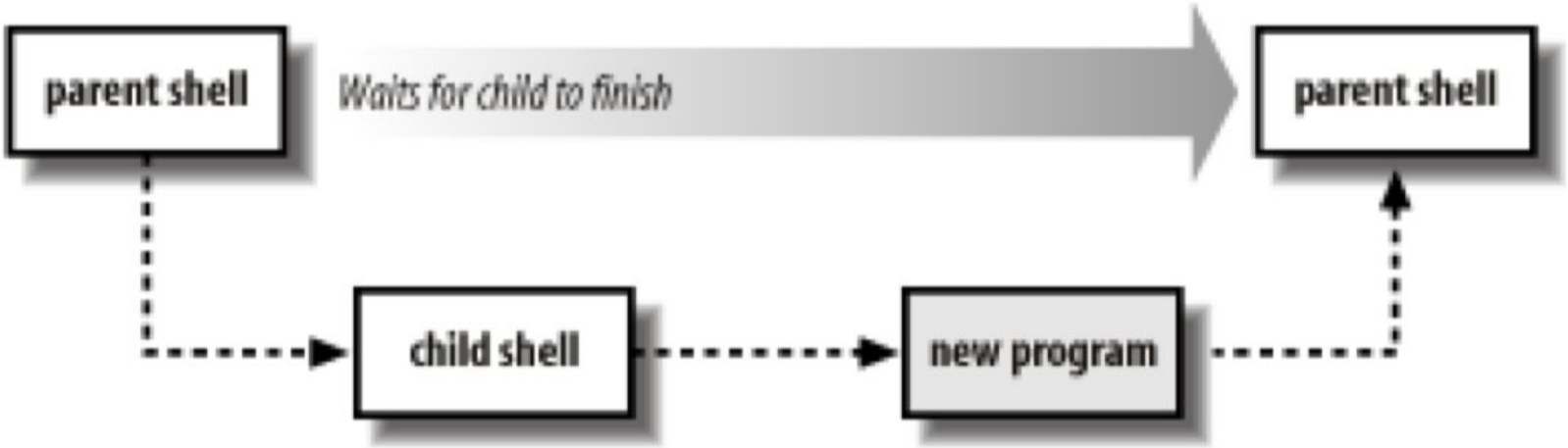
Shell Script

- A self-executable file that contains shell commands and functions
- Example
  - $ emacs test.sh
  - Content of file:
    #!/bin/bash
    who | grep "class"
  - $ chmod +x test.sh
  - $ ./test.sh # note that $ test.sh will not work cuz shell will search $PATH for this script
  - Or $ bash test.sh

Self-Contained Scripts: The #! First Line

- The #! First Line (shebang): a way to tell the kernel which shell to use for a script
- Some typical shebang lines:
  - $ #!/bin/sh # Execute the file using the Bourne shell assumed to be in the /bin directory
  - $ #!/bin/bash # Execute the file using the Bash shell
  - $ #!/usr/bin/env python # Execute with a Python interpreter, find with the program search path
- By including the shebang line, we could directly run the text script if the execute permission is set
- Without including the shebang line, we would need to run the script by invoking `$ bash script` or `python script`, etc.

How does shell run the script?



Variables

- Valid character string [a-zA-Z0-9_] to which a value is assigned
  var_name=var_value !!No spaces around =!!
- Special Variables: certain characters reserved as special variables
  - $: PID of current shell
  - #: number of arguments the script was invoked with
  - n: nth argument to the script
  - ?: exit status of the last command executed
  - echo $$; echo $#; echo $2; echo $?;
- scalar variable vs array variable:
- array_name[index]=value; echo ${array_name[index]}

Variables (cont’d)

- Command Substitution: store output of command evaluated in $(...)
  - current_dir=$(pwd)
  - dirs=$(find . -maxdepth 1)
- Arithmetic Expansion: store output of integer math evaluated in $((...))
  - a=3
  - b=4
  - c=$(($a + $b))
  - echo $c

Script Arguments

- We can also pass in arguments into script
- For historical reasons, enclose the number in braces if greater than 9

```
#!/bin/bash
# this is the argument test script
First=$1
Last=$2
tentharg=${10}
echo "My Name is ${First} ${Last}, and 10th argument is ${tentharg}"
echo "all arguments are $@"
```