Shell Scripting

and

Regular Expressions

CS 35L

Slide Set 2.1

Spring 2020 - Section 7

Lab setup - *Locale* for Assignment 2

- Please set your Locale:
  - `export LC_ALL='C'`

- Important because we want the ‘sort’ shell command to be ASCII character complainant
  - Otherwise your output for ‘sort’ is unknown and not deterministic, and your assignment results will not be as expected

Regular Expressions

Basic I/O Redirection

- Most programs read from stdin
- Write to stdout
- Send error messages to stderr

```
$ cat                                   With no arguments,
   read standard
                                                  input, write
   standard output
now is the time                         Typed by the user
now is the time                         Echoed back by cat
for all good men
for all good men
to come to the aid of their country
to come to the aid of their country
^D                                      Ctrl-D, End of file
```

Linux Pipes and Filters

- A **pipe** allows a chain of commands where the output of one command becomes the input of another. The symbol "|" denotes a pipe
  - Ex: cat filename # displays the full text of a file
  - cat filename | less # pipes the output into the less command, which in turn displays the first page of the text.

- A filter takes the output of a command, performs some processing (filtering) then gives an output. Examples are grep and sort
  - Ex: cat filename | grep –v 'a' | sort

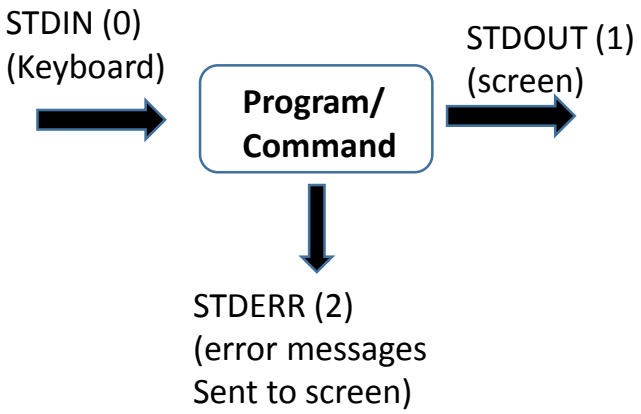
    # Takes the output of filename and pipes it into grep.

    # Applies a filter to extract all lines that do not contain the lowercase a

    # Applies another filter to sort the output alphabetically.

Linux Redirection

- In Linux, everything is a file and each file has a file descriptor (FD).
- When you execute a program/command on a terminal, 3 files are open:
  - Standard Input (STDIN) with FD = 0
  - Standard Output (STDOUT) with FD = 1
  - Standard Error (STDERR) with FD = 2



- Redirection: When executing a command we can redirect the STDIN, STDOUT and STDRERR

- Redirection can be IN (<) or OUT (> or >>)

  # >> will append to the file if it exists

Examples Redirection (1)

- ls -l > myfile # redirects the output from the screen to myfile

- ls doc1 doc2 > myfile # if doc2 does not exist, this will generate an error

  doc1

  ls: cannot access 'doc2': No such file or directory ‘doc2’

  # The screen output (doc1) is redirected to myfile, however the error message is still displayed on the screen

Examples Redirection (2)

- ls doc1 doc2 > myfile   2> error.log

  # This will redirect the screen output (doc1) to myfile

  and will redirect the error message to error.log;

  # 2> means redirect STDERR (2 is the FD of STDERR)

- ls doc1 doc2 > myfile   2>&1

  # redirect the screen output (doc1) and the error message to myfile

  # &1 means STDOUT; 2>&1 redirect the error to the screen which is being redirected to myfile

Regular Expressions

- Notation that lets you search for text that fits a particular criterion, such as “starts with the letter a” (pattern matching)
- Comes in two main flavors (in linux):
  - Basic Regular Expressions (BRE)
  - Extended Regular Expressions (ERE)
- Try http://regex101.com to test your regex
- Simple regex tutorial: https://www.icewarp.com/support/online_help/203030104.htm

Examples

Expression	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)

Examples Regular Expressions

- Match a string that has abc
  - abc (abc, habc, abcdefg, etc.)

- Match a string that has ab followed by one or more c

  abc+ (abc, abcc, abccc, etc.)

- Match a string that has ab followed by zero or more c
  - abc* (ab, abc, abccc, etc.)

- Match a string that starts with letter a
  - ^a (apple, android, avatar, etc.)

- Match a string that has ab followed by zero or more characters then cd
  - ab.*cd (abcd, abacd, abbzzcd, abbzzcdefg, etc)

Examples Regular Expressions

- Match a string that has one a or b c
  - [abc] (a, fang, cocoon, etc.)

- Match a string that has one alphabet character followed by 12 (range is dependent on locale)

  [a-zA-Z]12 (d12, P12, 45A123, etc.)

- Match a string that doesn't have alphabet characters or a dot or a hyphen or an underscore
  - [^-._a-zA-Z] (1, joe23, file#2, etc.)

- Match a string starting at the beginning of the line with a length more than 14 characters
  - ^.{15,} (IdontKnowWhatToType, etc.)

- Match a string with one pattern **or** another
  - ba|fa (banana, fanana, etc)

Regular expressions

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 NUL. 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.

Regular Expressions (cont'd)

$	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.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) 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.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine subpatterns 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.

Regular Expressions (cont'd)

\n	BRE	Replay the nth subpattern enclosed in \(and \) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{n,m}	ERE	Just like the BRE \{n,m\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
( )	ERE	Apply a match to the enclosed group of regular expressions.

POSIX Bracket Expressions

Class	**Matching characters**	**Class**	**Matching characters**
[:alnum:]	Alphanumeric characters	[:lower:]	Lowercase characters
[:alpha:]	Alphabetic characters	[:print:]	Printable characters
[:blank:]	Space and tab characters	[:punct:]	Punctuation characters
[:cntrl:]	Control characters	[:space:]	Whitespace characters
[:digit:]	Numeric characters	[:upper:]	Uppercase characters
[:graph:]	Nonspace characters	[:xdigit:]	Hexadecimal digits

Backreferences

- Match whatever an earlier part of the regular expression matched
  - Enclose a subexpression with \(and \).
  - There may be up to 9 enclosed subexpressions and may be nested
  - Use \digit, where digit is a number between 1 and 9, in a later part of the same pattern.

Pattern	**Matches**
\(ab\)\(cd\)[def]*\2\1	abcdcdab, abcdeeecdab, abcdddeeffcdab, ...
\(why\).*\1	A line with two occurrences of why

Matching Multiple Characters with One Expression

*	Match zero or more of the preceding character
\{*n*\}	Exactly n occurrences of the preceding regular expression
\{*n*,\}	At least n occurrences of the preceding regular expression
\{*n*,*m*\}	Between n and m occurrences of the preceding regular expression

Anchoring text matches

Pattern	Text matched (in bold) / Reason match fails
ABC	Characters 4, 5, and 6, in the middle: abc**ABC**defDEF
^ABC	Match is restricted to beginning of string
def	Characters 7, 8, and 9, in the middle: abcABC**def**DEF
def$	Match is restricted to end of string
[[:upper:]]\{3\}	Characters 4, 5, and 6, in the middle: abc**ABC**defDEF
[[:upper:]]\{3\}$	Characters 10, 11, and 12, at the end: abcDEFdef**DEF**
^[[:alpha:]]\{3\}	Characters 1, 2, and 3, at the beginning: **abc**ABCdefDEF

Operator Precedence (High to Low)

Operator	Meaning
[. .] [= =] [: :]	Bracket symbols for character collation
metacharacter	Escaped metacharacters
[ ]	Bracket expressions
\(\) *digit*	Subexpressions and backreferences
* \{ \}	Repetition of the preceding single-character regular expression
no symbol	Concatenation
^ $	Anchors

Sorting words

- Investigate the ‘sort’ command
- man sort

- sort –u (unique, no duplicates)
- sort –d (dictionary sort)
- sort –f (ignore case)

tr command (translate)

- Translate, squeeze, and/or delete characters from standard input, writing to standard output.
- tr [OPTION] set1 [set2]

EX1:

- echo abcd123 | tr –d [:digit:] → abcd

EX2:

- tr a-z A-Z
- Input: abtfy → output ABTFY

EX3:

- tr -s '\n’ ' ‘ < file.txt
- translate all <new line> into spaces and squeeze the spaces

tr command (translate)

EX4:

- tr abc def

EX5:

- tr ‘a-zA-Z’ ‘A[$*]Z’

EX6:

- tr –cs ‘A-Za-z’ ‘[\n*]’

“grep” command

- grep - searches the named input files for lines containing a match to a given pattern
  - grep <pattern> <file>
  - grep uses basic regular expressions
  - grep –E uses extended regular expressions
  - grep –F matches fixed strings (not a regular expression)

Searching for Text

- grep: Uses basic regular expressions (BRE)
- egrep: Grep that uses extended regular expressions (ERE)
  - grep -E
  - Egrep

- Fgrep: grep matching fixed strings instead of BRE or ERE.
  - grep -F
  - fgrep

grep -F

```
$ who                                                    Who is
   logged on
tolstoy tty1 Feb 26 10:53
tolstoy pts/0 Feb 29 10:59
tolstoy pts/1 Feb 29 10:59
tolstoy pts/2 Feb 29 11:00
tolstoy pts/3 Feb 29 11:00
tolstoy pts/4 Feb 29 11:00
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)

$ who | grep -F austen              Where is austen logged
   on?
austen pts/5 Feb 29 15:39 (mansfield-park.example.com)
austen pts/6 Feb 29 15:39 (mansfield-park.example.com)
```

“grep” command

- ls –a . | grep –E ‘^[^a-zA-Z]’ # list files that start with a non ASCII letter

- grep –E ‘<td>.+</td>’ $@ # find all lines in passed file arguments ($@) that have characters in between tags

sed (stream editor)

- Now you can extract, but what if you want to replace parts of text?
- Use sed!

```
sed 's/regExpr/replText/'
```

sed (stream editor)

- sed ‘s/[abc]123/john’ file # replace first instance of regexp to john
- sed ‘s/hi/lol/2’ file # replace 2nd instance of hi to lol
- sed ‘s/hi/lol/g’ file # replace all instances of hi to lol
- sed –E ‘s/\?|<u>//g’ file # remove all instances of ‘?’ or <u>
- sed ‘1~2d’ file # delete every other line (skip 2), starting from line 1

The Shell and OS

The Shell and OS

- The shell is the user's interface to the OS
- From it you run programs.
- Common shells
  - bash, zsh, csh, sh, tcsh
- Allow more complex functionality than interacting with OS directly
  - Tab complete, easy redirection

Scripting Languages Versus Compiled Languages

- Compiled Languages
  - Ex: C/C++, Swift
  - Programs are translated from their original source code into object code that is executed by hardware
  - Efficient
  - Work at low level, dealing with bytes, integers, floating points, etc
  - Not portable
- Scripting languages (Interpreted Languages)
  - Ex: Bash, Python, Javascript
  - Interpreted by program
  - Interpreter reads script code “line by line”, translates it into internal form, and execute programs
  - Portable and easier to develop (bad performance!)

Scripting Languages Versus Compiled Languages

- In between languages
  - Ex: Java
  - Compiled to bytecode, which is then interpreted by Java Virtual Machine
  - Sometimes chunks of bytecode get further compiled during runtime for better performance by Just-In-Time compiler.

Idea

- Build a script that searches for a name
  - i.e. $who | grep userWeAreLookingFor
- Check if userWeAreLookingFor is logged in
- Let's create it!
  - create a file called `finduser`

finduser

Script:

```
#! /bin/bash
# finduser --- see named by first argument is
  logged in
who | grep $1
```

Run it:

```
$ chmod +x finduser          Make it executable


$ ./finduser littenek
```

The #! First Line

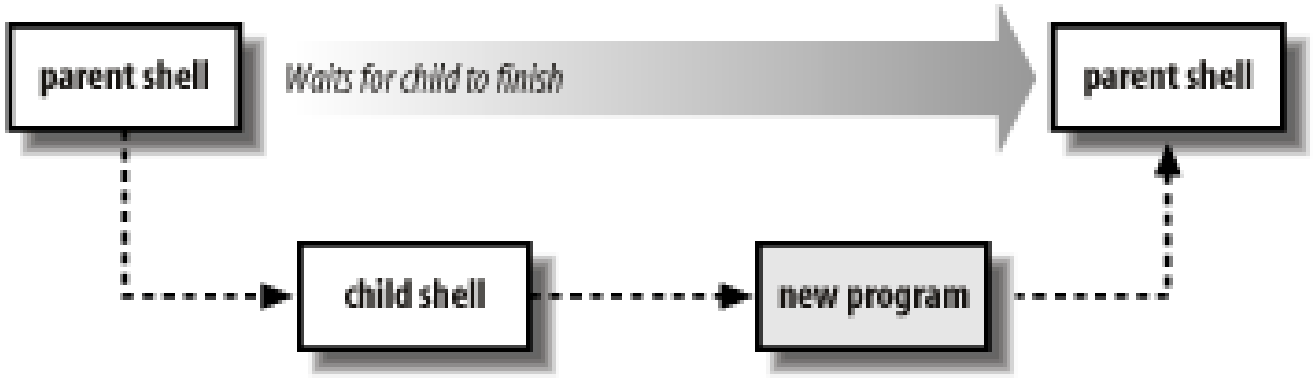
- A shell script is just a file with shell commands.
- When the shell runs a program (e.g finduser), it asks the kernel to start a new “child process” and run the given program in that process.
- First line is used to state which “child shell” to use:

  #! /bin/csh –f

  #! /bin/awk –f

  #! /bin/sh

  #! /bin/env bash



Variables

- Start with a letter or underscore and may contain any number of following letters, digits, or underscores
- Declared using = (no space)
  - `Var='helloworld'`
- Referenced with $
  - `echo $Var (${Var} preferred to avoid ambiguity e.g. $ab vs ${a}b )`
- X=`pwd`
  - Back ticks executes pwd command and stores output in variable X

Variables

- Command substitution: Assign the output of a command to a variable: $(command)
- E.g. x=$(pwd)
- E.g. y="Working directory is $(pwd)"
- Command substitution can be nested $(… $(…))

Variables

- Please see: https://www.tldp.org/LDP/abs/html/variables.html
- C-style Arithmetic Operations/Comparisons (only if digits stored)

- ((a++))
- ((a = 23))
- ((b = a * 2))

- https://www.tldp.org/LDP/abs/html/dblparens.html

Adding Variables to script files

```
#! /bin/bash
STRING=”HELLO WORLD”    #assign variable
echo $STRING                          #prints
                                    the value
```

Accessing Shell Script Arguments

- Positional parameters represent a shell script's command line arguments
- For historical reasons, enclose the number in braces if greater than 9

```
#! /bin/bash
#test script
echo first arg is $1
echo tenth arg is ${10}
echo all args is $@

> ./argtest 1 2 3 4 5 6 7 8 9 10
```

Single vs Double Quotes

- Single quotes preserve the literal value of each character within the single quotes
- Double quotes allow for parameter substitution

- E.g echo “$(pwd)” → /path/to/working/dir
- E.g echo ‘$(pwd)’ → $(pwd)