

# Week 2

# Shell Scripting, RegEx, and Streams

14 January 2019

CS 35L Lab 4

Jeremy Rotman

# Announcements

- Assignment #1 **was** due January 12 by 11:55pm
  - ◆ You can still submit the assignment
  - ◆ If you submit before 11:55pm tonight, it's only a 2% penalty!
- Assignment #2 is due January 23 (Wednesday) by 11:55pm

Questions?

# Outline

- Locales and environmental variables
- Shell scripting
  - ◆ Writing scripts
  - ◆ Conditional statements
- Regular Expressions
- Lab 2 Tips

# Locale

→ What is a locale?

# Locale

→ What is a locale?

- ◆ A set of parameters that define a user's cultural preferences
  - E.g. Language or Country

→ The `locale` command

- ◆ Prints information about the current locale environment to stdout

# Environment Variables

- Variables that can affect how processes run
- Common ones:
  - ◆ **HOME**: path to user's home directory
  - ◆ **PATH**: list of directories to search in for command to execute
    - Thus, why you prepended a path in assignment #1
- Changing an environment variable's value

```
export VARIABLE=...
```

# LC\_\* Environment Variables

→ `locale` gets its data from the LC\_\* environment variables

→ Example

- ◆ LC\_TIME

- Date and time formats

- ◆ LC\_CTYPE

- Character classification and case conversion



# Locale matters!

For example, if you ran `sort`:

- `LC_COLLATE='C'`
  - ◆ Sorting is in ASCII order
- `LC_COLLATE='en_US'`
  - ◆ Sorting is case insensitive

It's important to change your locale for assignment #2

# More useful commands

→ `sort`

◆ Sorts lines of text files

→ `comm`

◆ Compare two sorted files line by line

→ `tr`

◆ Translate or delete characters

# Shell Scripting

# What is a shell script?

# What is a shell script?

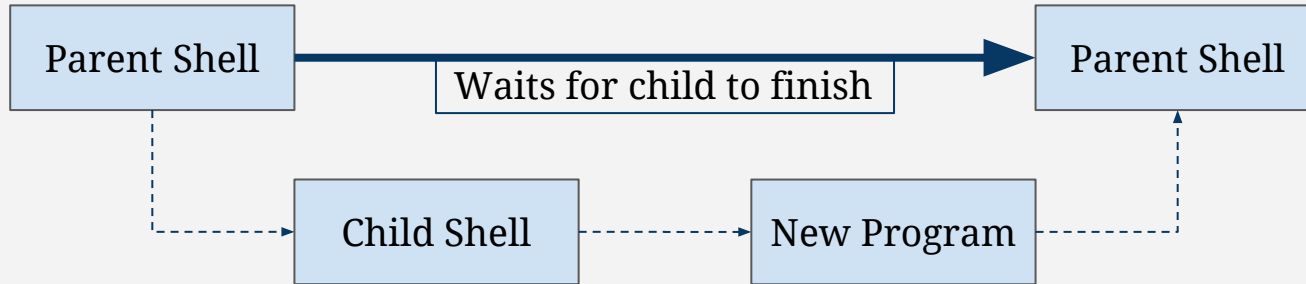
- It is a file that holds one or more shell command(s)
- Why is this useful?

# What is a shell script?

- It is a file that holds one or more shell command(s)
- Why is this useful?
  - ◆ Commands run in succession can be contained in one file
  - ◆ You can introduce conditionals to your shell commands
  - ◆ Clearer variable declarations

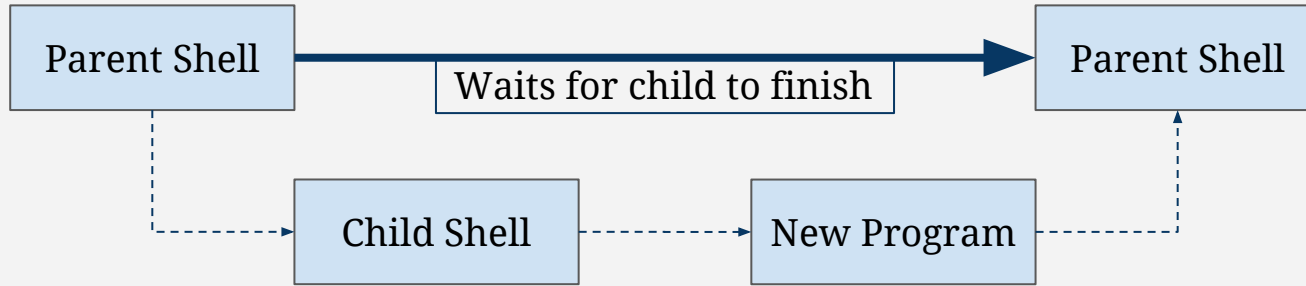
## The first line

→ A shell script spawns a “shell” process to run it



# The first line

→ A shell script spawns a “shell” process to run it



- ◆ Reminder: a shell is a command-line interpreter
- ◆ There are many types of shells
  - E.g. Bash, Bourne shell, C shell, KornShell
- ◆ How do we make sure the correct shell is used in our process?



SHEBANG!



# The first line

- The shebang (“#!”) opens the first line
- The line includes the absolute path to the interpreter
- E.g.
  - ◆ `#!/bin/bash`
  - ◆ `#!/bin/sh`

# Executing scripts

→ How do you run your newly created script?

◆ `./script.sh`

# Executing scripts

- How do you run your newly created script?
  - ◆ `./script.sh`
- Why might that not work?
  - ◆ Newly created files are not automatically given execution permission
  - ◆ How to fix?

# Executing scripts

- How do you run your newly created script?
  - ◆ `./script.sh`
- Why might that not work?
  - ◆ Newly created files are not automatically given execution permission
  - ◆ How to fix?
    - `chmod u+x script.sh`

# Execution tracing

- Shell can print out commands as it runs them
- `set -x`
  - ◆ turns tracing on
- `set +x`
  - ◆ Turns tracing off
- If you want it to trace within the script, this command must be within your script

# Variables

→ Within a shell script you can declare

- ◆ `myvar="aloha"`

- ◆ Note that there are no spaces!

→ And reference variables

- ◆ `echo $myvar`

- ◆ You can use curly brackets to denote which part is the variable

- `echo "${myvar}_Jeremy"`

# POSIX built-in shell variables

- There are a number of potentially useful built in variables
  - ◆ E.g. PATH
- You should avoid naming your own variables the same names
- Some useful variables
  - ◆ #
    - Number of arguments given to current process
  - ◆ ?
    - Exit status of previous command
  - ◆ IFS
    - Internal Field Separator



# Exit Status

- When a command finishes, it provides an exit status code
- You can check this by referencing the `?` variable
- 0
  - ◆ The command exited successfully
- 1-125
  - ◆ The command exited unsuccessfully
  - ◆ Each has their own meaning
    - E.g. try running `ls` on an imaginary directory, or `cd` into a text file
- 127
  - ◆ Command not found

## More built in variables

- If your script takes in arguments, they are automatically saved as variables
- The first argument is “1”, the second “2”, etc.
- You can then reference these like variables
  - ◆ E.g. `echo "${1}"`

# If statements

Can be used with the `test` command (Hint: `man test`)

```
if [ ${1} -ge 0 ]  
then  
    echo "Nonnegative"  
else  
    echo "Negative"  
fi
```

# While

While loops work similarly

```
count=1
while [${count} -le 10]
do
    echo "${count}"
    let count=count+1
done
```

# For

For loops also work similarly

```
phrase="hello world"  
for c in $phrase  
do  
    echo "${c}"  
done
```

# Quotes

- Different quotes mean different things
- Single Quotes: ‘ ’
  - ◆ Literal meaning, do not expand
- Double Quotes: “ ”
  - ◆ Expand only backticks and \$
- Backticks: ` `
  - ◆ Expand as shell commands

# Redirection Refresher

→ Programs have 3 streams

- ◆ `stdin (0)`
  - Contains data going to program
- ◆ `stdout (1)`
  - Program writes output here
- ◆ `stderr (2)`
  - Program writes error messages here

# Redirection Refresher

- `program < file_in`
  - ◆ Redirects `file_in` to `stdin`
- `program > file_out`
  - ◆ Redirects `stdout` to `file_out`
- `program 2> file_err`
  - ◆ Redirects `stderr` to `file_err`
- `program >> file_out`
  - ◆ Appends `stdout` to `file_out`
- `program1 | program2`
  - ◆ Redirects `stdout` from `program1` to `stdin` of `program2`



# Regular Expressions

# RegEx

- Allows you to search for patterns rather than a direct search
- Similar to wildcards
  - ◆ Note however, that they function slightly differently

# Anchors

➔ ^

- ◆ Match the following regular expression with the beginning of a line or string

➔ \$

- ◆ Match the preceding regular expression with the end of a line or string

# Quantifiers

- .
  - ◆ Match any single character
- \*
  - ◆ Match 0 or more of the preceding character
- +
  - ◆ Match 1 or more of the preceding character
- ?
  - ◆ Match 0 or 1 of the preceding character

# Quantifiers

→  **$\{n\}$**

◆ Match exactly  $n$  of the preceding character

→  **$\{n,\}$**

◆ Match  $n$  or more of the preceding character

→  **$\{n,m\}$**

◆ Match  $n$  to  $m$  of the preceding character

# Bracket

→ [...]

- ◆ Allows a match to any one of the enclosed characters

→ -

- ◆ A hyphen within a bracket designates a range, like A-Z

# POSIX Bracket expressions

Expression	Meaning
<code>[:alnum:]</code>	Alphanumeric Characters
<code>[:alpha:]</code>	Alphabetic Characters
<code>[:blank:]</code>	Space and Tab Characters
<code>[:cntrl:]</code>	Control Characters
<code>[:digit:]</code>	Numeric Characters
<code>[:graph:]</code>	Nonspace Characters

# POSIX Bracket expressions

Expression	Meaning
<code>[:lower:]</code>	Lowercase Characters
<code>[:print:]</code>	Printable Characters
<code>[:punct:]</code>	Punctuation Characters
<code>[:space:]</code>	Whitespace Characters
<code>[:upper:]</code>	Uppercase Characters
<code>[:xdigit:]</code>	Hexadecimal Digits



# Parentheses

- Parentheses allow you to apply quantifiers to sequences of characters
  - ◆ E.g. `(ab)*`
- Additionally, parentheses form capturing groups
  - ◆ These can be backreferenced later in the regular expression
  - ◆ E.g. `(ab)c\1c`
  - ◆ You can only store 9 capturing groups (`\1-\9`)

## BRE vs ERE

- Basic Regular Expression (BRE) is the standard mode for sed and grep
- Extended Regular Expression (ERE) is an optional flag you can use with the commands
- What's the difference?
  - ◆ BRE tends to take things more literally

# BRE vs ERE

- In BRE ‘?’, ‘+’, ‘{’, ‘}’, ‘(’, and ‘)’ lose their special meanings
  - ◆ They are treated literally
- To use the special meanings, you will either need to use the ERE option, or use ‘\’
  - ◆ E.g. `\(ab\) \+`
- For characters with special meaning, ‘\’ can be used to turn off special meanings of characters
  - ◆ Yes, it's a bit confusing

# Searching text with RegEx

- ➔ `grep [OPTIONS] PATTERN [FILE...]`
  - ◆ Search either FILE(s) or STDIN for the given RegEx PATTERN
  - ◆ It will return matching lines
- ➔ Useful grep options
  - ◆ `-E`
    - Uses extended regular expressions
  - ◆ `-F`
    - Matches fixed strings

# Replacing text with RegEx

→ Sed SCRIPT [FILE]

◆ Stream Editor

→ How to construct SCRIPT?

◆ Theoretically many ways, but we will focus on text replacement

- `'s/regexp/replacement/flags'`

◆ The 's' signifies substitution

◆ Useful flags

- g - global, replace all matches to *regexp*, not just the first
- I - case-insensitive matching

→ E.g.

◆ `sed 's/[Jj]eremy/John/g'`

# Assignment #2

## Lab #2

### → Build a file with Hawaiian words

- ◆ Download a copy of the linked webpage that contains a basic English-to-Hawaiian dictionary
- ◆ Extract the Hawaiian words
  - Treat upper-case letters as if they were lower-case
  - Treat “<u>a<\u>” as “a”, also for other letters like this
    - Kahako, or a macron, which extends the vowel
  - Treat ` as '
    - Okina, or a glottal stop
  - Treat spaces and commas as breaks between separate words
  - Some entries may be incorrect, so you will need to reject entries that include letters not in the Hawaiian alphabet

## Lab #2

- Automate your site scrape in a script
  - ◆ You should not have to do anything by hand
    - You can download the html outside of the script
  - ◆ It should read the file **from stdin** and write a sorted list of unique words **to stdout**
    - `cat foo.html bar.html | ./buildwords | less`
      - This command should work with your submitted script
- Modify the given spell checker to work for Hawaiian
- Test your spell checker on the hwords file you build, as well as the assignment webpage



# Hints

- `sed 's/<[^>]*>//g' a.html`
  - ◆ This will remove all html tags
- `sed '/pattern1/,/pattern2/d' input.txt`
  - ◆ This will delete from pattern1 to pattern2, inclusive
- `sed -n 'first_line~step_sizep'`
  - ◆ `p` commands sed to print the specified line(s)
  - ◆ `-n` suppresses other output
  - ◆ `first_line` is the line number you wish to begin from
  - ◆ `step_size` is the number of lines to step to reach next printed line

# Hints

## → Useful tr options

◆ -d

- Deletes options in the field, does not translate

◆ -s

- Squeeze repeats, repeated characters reduced to 1 character

→ Remember that your spell checker won't work on upper-case letters if your dictionary is all lower-case