Review: Unix Commands

- What are regular expressions?
  - What can we represent with regular expressions?
  - How do we represent those things?
- What commands can we use for searching?
  - What are the differences between the commands?

Today

- grep
- bash

grep Family Syntax

```
grep [-hilnv] [-e expression] [filename]
egrep [-hilnv] [-e expression] [-f filename] [expression]
   [filename]
fgrep [-hilnxv] [-e string] [-f filename] [string]
   [filename]
```

Option	Meaning
`-h`	Do not display filenames
`-i`	Ignore case
`-l`	List only filenames containing matching lines
`-n`	Precede matching line with its line number
`-v`	Select non-matching lines
`-x`	Match whole line only
`-e expression`	Specify expression as option
`-f filename`	Take regular expression (egrep) or a list of strings (fgrep) from filename

grep: Backreferences

- ***Backreferences*** allow us to refer to a match that was made earlier in a regex
  - `\n` is the *backreference specifier*, where n is a number
  - Looks for n^{th} subexpression
- Example: HTML Tags
  - `<h[1-6]>.*</h[1-6]>` is not good enough to match html headers, since it matches `<h1>Hello world</h3>`
  - `<h\([1-6]\)>.*</h\1>` matches what we were trying to match before.

grep: Backreference Examples

- `^\([[:alpha:]]\{1,\}\) .* \1$`

egrep doesn't support backreferences, so need to use grep

- Another example:
  - `"Mr \(dog\|cat\) came home to Mrs \1 and they went to visit Mr \(dog\|cat \) and Mrs \2 to discuss the meaning of life"`

What text should this match?

grep: Backreference Examples

- To find if the first word of a line is the same as the last:
  - `^\([[:alpha:]]\{1,\}\) .* \1$`
  - `\([[:alpha:]]\{1,\}\)` matches 1 or more letters
- Another example:
  - `"Mr \(dog\|cat\) came home to Mrs \1 and they went to visit Mr \(dog\|cat \) and Mrs \2 to discuss the meaning of life"`

What text should this match?

Fun with the Dictionary

- **/usr/share/dict/words** contains over 400,000 words
  - **egrep hh /usr/share/dict/words**
    - aarrghh
    - Ahhiyawa
    - archhead
    - archheart
    - …
- **egrep** as a simple spelling checker: Specify plausible alternatives you know

```
egrep "n(ie|ei)ther" /usr/share/dict/words
Neither
```

- How many words have 3 a's one letter apart? 3 u's?

Fun with the Dictionary

- How many words have 3 a's one letter apart?
  - **egrep a.a.a /usr/share/dict/words | wc –l**
    - 1632
- How many words have 3 u's one letter apart?
  - **egrep u.u.u /usr/share/dict/words | wc -l**
    - 84

grep Examples

- `grep 'men' greptest`
- `grep 'fo*' greptest`
- `egrep 'fo+' greptest`
- `egrep –v 'fo+' greptest`
- `egrep -n '[Tt]he' greptest`
- `fgrep 'The' greptest`

SHELL SCRIPTING

Shell Scripts

- ***Script***: a shell program
- Tool for building applications by "gluing together" system calls, tools, utilities, and compiled binaries
- Just about everything we've done so far is available for use in a script
  - Adds even more
- Good for repetitive tasks that don't require a more structured programming language

Shell Scripting vs. [C/Python/Java] Programming

Advantages
Easy to work with/use other programs
Easy to work with directories, files
Easy to work with strings (easier than C, at least)
Good for prototyping

Shell Scripting vs. [C/Python/Java] Programming

Advantages	Disadvantages
Easy to work with/use other programs	Slower
Easy to work with directories, files	Not well-suited for algorithms and data structures
Easy to work with strings (easier than C, at least)	
Good for prototyping	

Scripts tend not to be long

In some ways, we'll love it;
in some ways, we'll hate it.

Shell Scripts

- A shell script is a regular **text** file that contains shell or UNIX commands
- Kernel uses the *first line* of script to determine which shell script to use
  - `#!pathname-of-shell`
    - Kernel invokes pathname and sends the script as an argument to be interpreted
  - If `#!` is not specified, the current shell assumes it is a script in its own language
    - Can lead to problems

Simple Example

```
#!/bin/bash

echo "Hello World"
```

Which shell to use

Command to execute

`echo` – like a print statement

Look at the available shells by executing

```
ls -l /bin/*sh
```

What do you notice about the shells?

Invoking a Script

- A script can be invoked as:
  - `sh scr_name [ arg … ]`
  - `sh < scr_name [ args … ]`
  - `path/scr_name [ arg …]`

Where `sh` is whatever shell you want

  - Before running it, it must have execute permission:
    - `chmod +x scr_name`

We'll typically use either the 1st or 3rd execution option and we'll use the `bash` shell

Your First Script

- Write a script that
  - Shows the time and date
  - Lists all logged-in users
  - Saves the output into a logfile
- Build in pieces
- Execute and test your script
  - Verify the output in the logfile

Types of Commands

All behave the same way

- Programs
  - Most that are part of the OS in **`/bin`**
- Built-in commands
- Functions
- Aliases

Built-in Commands

- Built-in commands are internal to the shell and do not create a separate process
- Commands are built-in because:
  - They are intrinsic to the language (`exit`)
  - They produce side effects on the current process (`cd`)
  - They perform faster
    - No fork/exec
- Special built-ins
  - `: . break continue eval exec export exit readonly return set shift trap unset`

Important Built-in Commands

`exec`	Replaces shell with program
`cd`	Change working directory
`shift`	Rearrange positional parameters
`set`	Set positional parameters
`wait`	Wait for background process to exit
`umask`	Change default file permissions
`exit`	Quit the shell
`eval`	Parse and execute string

Check out `cd`:

```
1. which cd
2. more `which cd`
```

Important Built-in Commands

`time`	Run command and print times
`export`	Put variable into environment
`trap`	Set signal handlers
`continue`	Continue in loop
`break`	Break in loop
`return`	Return from function
`:`	True
`.`	Read file of commands into current shell

Comments

- Comments begin with an #
- Comments end at the end of the line
- Comments can begin whenever a token begins
- Our text editors should help you with syntax highlighting
- Examples:

```
# This is a comment
# and so is this
grep foo bar # this is a comment
grep foo bar# this is not a comment
```

Add a comment at 2nd line in your script that lists you as author

Variables

- To set:
  `name=value` ← Notice no spaces around =
  - Variables are *untyped*
- Read: `$var`
- Variables can be local or environment
  - Environment variables are part of UNIX and can be accessed by child processes
- Turn local variable into environment var:
  `export variable`

Variable Example

```
#!/bin/sh

MESSAGE="Hello World"
echo $MESSAGE
echo '$MESSAGE'
echo "$MESSAGE"
```

Prints variable
Prints literally
Prints variable

variable.sh

Using Environment Variables

```
#!/bin/bash

echo I am $USER
echo "I live at $HOME"
```

Both statements would work either with or without quotes

env_var.sh

Assign 2

- Filters practice
  - Do what is requested first and record your commands in a file
    - Record your analysis in that file.
  - Then, demonstrate it all in one `script` file