Week 5

Announcements

- HW3, ADV3 due by 11:59 PM on Feb 10
- HW4, ADV4 due by 11:59 PM on Feb 17

Random extras

- I'll be using a tool called `tmux` ("terminal multiplexer") that allows me to create multiple terminal windows inside of a terminal
  - An alternative terminal multiplexer is GNU Screen

Lecture 5: Unix++

```
:(){ :|:& };:
```

Do **NOT** run this

Overview

- *nix file descriptors
- Diving into Bash
- Regular expressions

Some more *nix

Remember that in *nix:

Some more *nix

Remember that in *nix:

- Everything is a **file**

Some more *nix

Remember that in *nix:

- Everything is a **file**
- A **file** is a *stream of bytes*

Some more *nix

Remember that in *nix:

- Everything is a **file**
- A **file** is a *stream of bytes*
- Each utility is narrow in scope and does its job well

Some more *nix

Remember that in *nix:

- Everything is a **file**
- A **file** is a *stream of bytes*
- Each utility is narrow in scope and does its job well
- Utilities can be stringed together to perform more complex tasks tying their outputs and inputs together using a pipe (`|`)

Some more *nix

Remember that in *nix:

- Everything is a **file**
- A **file** is a *stream of bytes*
- Each utility is narrow in scope and does its job well
- Utilities can be stringed together to perform more complex tasks tying their outputs and inputs together using a pipe (`|`)
  - `$ command1 | command2`
  - command1's output will go directly to command2's input

What is a file?

- To *nix processes, **files** are visible as **file descriptors**

What is a file?

- To *nix processes, **files** are visible as **file descriptors**
- Each *nix process has a **file descriptor table** containing handles to various resources
  - Such resources could be: an "actual" data file living on disk, a virtual file representing OS info, a network socket, a terminal input, a terminal output, another program's input etc.

What is a file?

- To *nix processes, **files** are visible as **file descriptors**
- Each *nix process has a **file descriptor table** containing handles to various resources
  - Such resources could be: an "actual" data file living on disk, a virtual file representing OS info, a network socket, a terminal input, a terminal output, another program's input etc.
- **File descriptors** are integers that index into this **file descriptor table**

What is a file?

- To *nix processes, **files** are visible as **file descriptors**
- Each *nix process has a **file descriptor table** containing handles to various resources
  - Such resources could be: an "actual" data file living on disk, a virtual file representing OS info, a network socket, a terminal input, a terminal output, another program's input etc.
- **File descriptors** are integers that index into this **file descriptor table**
- When a (terminal) shell creates a process the shell sets the terminal's input and output as the process's input and output

What is a file?

- To *nix processes, **files** are visible as **file descriptors**
- Each *nix process has a **file descriptor table** containing handles to various resources
  - Such resources could be: an "actual" data file living on disk, a virtual file representing OS info, a network socket, a terminal input, a terminal output, another program's input etc.
- **File descriptors** are integers that index into this **file descriptor table**
- When a (terminal) shell creates a process the shell sets the terminal's input and output as the process's input and output
- Reminder:
  - fd 0: `stdin`, `cin`
  - fd 1: `stdout`, `cout`
  - fd 2: `stderr`, `cerr`

What is a file?

- To *nix processes, **files** are visible as **file descriptors**
- Each *nix process has a **file descriptor table** containing handles to various resources
  - Such resources could be: an "actual" data file living on disk, a virtual file representing OS info, a network socket, a terminal input, a terminal output, another program's input etc.
- **File descriptors** are integers that index into this **file descriptor table**
- When a (terminal) shell creates a process the shell sets the terminal's input and output as the process's input and output
- Reminder:
  - fd 0: `stdin`, `cin`
  - fd 1: `stdout`, `cout`
  - fd 2: `stderr`, `cerr`
- Some related POSIX C API functions (**not to be confused with C standard library functions!**):
  - `open()`, analogous to `fopen()`
  - `close()`, analogous to `fclose()`
  - `read()`, analogous to `fread()`
  - `write()`, analogous to `fwrite()`
  - `dup2()`
  - `pipe()`

File redirection

Now that we have discussed file descriptors, we can look more in-depth into how we can manipulate these *streams of data*

Recall from the first *nix lecture:

- `<`: set file as standard input (fd 0)
- `>`: set file as standard output, overwrite (fd 1)
- `>>`: set file as standard output, append (fd 1)
- `|`: connect output of one process to input of another (command 1's fd 1 -> command 2's fd 0)

File redirection

Now that we have discussed file descriptors, we can look more in-depth into how we can manipulate these *streams of data*

Recall from the first *nix lecture:

- `<`: set file as standard input (fd 0)
- `>`: set file as standard output, overwrite (fd 1)
- `>>`: set file as standard output, append (fd 1)
- `|`: connect output of one process to input of another (command 1's fd 1 -> command 2's fd 0)

Let's look at them in a more general form (brackets mean optional):

- `[n]<`: set file as an input for fd *n* (fd 0 if unspecified)
  - "input" means that the process can `read()` from this fd
- `[n]>`: set file as an output for fd *n* (fd 1 if unspecified)
  - "output" means that the process can `write()` to this fd
  - `2>`: capture `stderr` to a file
- `[n]>>`: set file as an output for fd *n*, append mode (fd 1 if unspecified)

Advanced Bash file redirection

- `&>`: set file as fd 1 and fd 2, overwrite (`stdout` and `stderr` go to same file)
- `&>>`: set file as fd 1 and fd 2, append (`stdout` and `stderr` go to same file)
- `[n]<>`: set file as input and output on fd *n* (fd 0 if unspecified)
- `[n]<&digit[-]`: copies fd *digit* to fd *n* (0 if unspecified) for input; `-` closes *digit*
- `[n]>&digit[-]`: copies fd *digit* to fd *n* (1 if unspecified) for output; `-` closes *digit*
- (there's a few more like Here Documents; refer to the manual)

By piping and redirecting, we can put together larger and larger commands

```
# note: '\U' and friends are a GNU sed extension;
#       BSD sed might not have it
wget https://eecs.umich.edu/courses/eecs201/
  files/text/lorem-ipsum.txt
cat lorem-ipsum.txt | sed -e 's/./\U&/g' |
  sed -e 's/[.,]//g' | sed -e 's/U/V/g' -e 's/J/I/g' >
  LOREM-IPSVM.txt
```

- Useful utilities
  - `cat`
  - `head`
  - `tail`
  - `cut`
  - `sed`
  - `awk`

Diving into Bash

- Side note: `bash` != `sh`
- `bash` has a feature superset over `sh` (kinda like a `vim`/`vi` relationship)
  - Again, confounded by some systems linking/aliasing `sh` to `bash`
- The horse's mouth: GNU Bash manual
  - If you like the nitty gritty details it's a great read
  - These slides summarize major features of Bash
- You may have stumbled upon these while working on HW2

Diving into Bash

- Side note: `bash` != `sh`
- `bash` has a feature superset over `sh` (kinda like a `vim`/`vi` relationship)
  - Again, confounded by some systems linking/aliasing `sh` to `bash`
- The horse's mouth: GNU Bash manual
  - If you like the nitty gritty details it's a great read
  - These slides summarize major features of Bash
- You may have stumbled upon these while working on HW2

What Bash does

- Receive a command from a file or terminal input
- Splits it into tokens separated by **white-space**
  - Takes into account *"quoting"* rules
- Expands/substitutes special tokens
- Perform file redirections (and making sure they don't end up as command args)
- Execute command

Command grouping

- We discussed before that we can string commands together with `;`, `&&`, `||`
- We can also group commands together as a unit, with redirects staying local to them:
- `(commands)`: performs *commands* in a "subshell" (another shell instance: this means that variable assignments won't be visible to the parent shell)
- `{ commands; }`: performs *commands* in the calling shell instance
  - **Note**: There has to be spaces around the brackets and a semicolon (or newline or `&`) terminating the *commands*

Expansion and substitution

Bash has special characters that will indicate that it should *expand* or *substitute* to something in a command

Expansion and substitution

Bash has special characters that will indicate that it should *expand* or *substitute* to something in a command

Variable expansion

- `$varname` will expand to the value of `varname`
- `${varname}`: you can use curly brackets to explicitly draw the boundaries on the variable name
  - `$ echo ${varname}somestring` vs `$ echo $varnamesomestring`
- **Note**: expansions/substitutions will be further split into individual tokens by their white-space

Expansion and substitution

Bash has special characters that will indicate that it should *expand* or *substitute* to something in a command

Variable expansion

- `$varname` will expand to the value of `varname`
- `${varname}`: you can use curly brackets to explicitly draw the boundaries on the variable name
  - `$ echo ${varname}somestring` vs `$ echo $varnamesomestring`
- **Note**: expansions/substitutions will be further split into individual tokens by their white-space

Command substitution (via subshell)

- `$(command)` will substitute the output of a *command* in the brackets
  - `$(echo hello | rev)` will be substituted with "olleh"

Process substitution (ironically helpful on HW2)

- `<(command)` will substitute the *command* output as a filepath, with the output of *command* being **readable**
- `>(command)` will substitute the *command* input as a filepath, with the input of *command* being **writeable**
- `$ diff <(echo hello) <(echo olleh | rev)`
  - `diff` takes in two file names, but we're replacing them with command outputs

Process substitution (ironically helpful on HW2)

- `<(command)` will substitute the *command* output as a filepath, with the output of *command* being **readable**
- `>(command)` will substitute the *command* input as a filepath, with the input of *command* being **writeable**
- `$ diff <(echo hello) <(echo olleh | rev)`
  - `diff` takes in two file names, but we're replacing them with command outputs

Arithmetic expansion

- `$((expr))` will expand to an evaluated arithmetic expression *expr*

Process substitution (ironically helpful on HW2)

- `<(command)` will substitute the *command* output as a filepath, with the output of *command* being **readable**
- `>(command)` will substitute the *command* input as a filepath, with the input of *command* being **writeable**
- `$ diff <(echo hello) <(echo olleh | rev)`
  - `diff` takes in two file names, but we're replacing them with command outputs

Arithmetic expansion

- `$((expr))` will expand to an evaluated arithmetic expression *expr*

But wait, what if I actually wanted to not expand a variable?

What if I didn't want a variable to be split by white-space?

What if I'm lazy and don't want to escape spaces?

Quoting

- Allows you to retain certain characters without Bash expanding them and keep them one string
  - Common use case is to preserve spaces e.g. for filepaths that have spaces in them (spaces delimit tokens in a command)

Quoting

- Allows you to retain certain characters without Bash expanding them and keep them one string
  - Common use case is to preserve spaces e.g. for filepaths that have spaces in them (spaces delimit tokens in a command)
- Single quotes (`'`) preserves all of the characters between them
  - `$ echo '$HOME'` will output `$HOME`

Quoting

- Allows you to retain certain characters without Bash expanding them and keep them one string
  - Common use case is to preserve spaces e.g. for filepaths that have spaces in them (spaces delimit tokens in a command)
- Single quotes (`'`) preserves all of the characters between them
  - `$ echo '$HOME'` will output `$HOME`
- Double quotes (`"`) preserve all characters except: `$`, `\`, and backtick
  - `$ ls "$HOME/Evil Directory With Spaces"` will list the contents of a directory `/home/jdoe/Evil Directory With Spaces`
  - Variables expanded inside of double quotes retain their white-space

Quoting

- Allows you to retain certain characters without Bash expanding them and keep them one string
  - Common use case is to preserve spaces e.g. for filepaths that have spaces in them (spaces delimit tokens in a command)
- Single quotes (`'`) preserves all of the characters between them
  - `$ echo '$HOME'` will output `$HOME`
- Double quotes (`"`) preserve all characters except: `$`, `\`, and backtick
  - `$ ls "$HOME/Evil Directory With Spaces"` will list the contents of a directory `/home/jdoe/Evil Directory With Spaces`
  - Variables expanded inside of double quotes retain their white-space
- Note that when quoting, the quotes don't appear in the program's argument
  - `$ someutil 'imastring'`: `someutil`'s argv[1] will be `imastring`

Control flow

`if-elif-else`

```
# brackets indicate optional parts
if test-commands; then
  commands
[elif more-test-commands; then
  more-commands]
[else
  alt-commands]
fi
```

- *test-commands* is executed and its **return code** is used as the condition
  - *0* = success = "true", everything else is "false"

Commands for conditionals

You can use any commands for conditions, but these constructs should be familiar:

- `test expr`: `test` command
  - Shorthand: `[ expr ]` (remember your spaces! `[` is technically a utility name)
  - `test $a -eq $b`
  - `[ $a -eq $b ]`

Commands for conditionals

You can use any commands for conditions, but these constructs should be familiar:

- `test expr`: `test` command
  - Shorthand: `[ expr ]` (remember your spaces! `[` is technically a utility name)
  - `test $a -eq $b`
  - `[ $a -eq $b ]`

- `[[ expr ]]`: Bash conditional
  - Richer set of operators: `==`, `=`, `!=`, `<`, `>`, among others
  - **Note**: The symbol operators above operate on strings, thus `<` and `>` operators do lexicographic (i.e. dictionary) comparison; "100" is lexicographically less than "2" since for the first characters "1" comes before "2"
  - Use specific arithmetic binary operators (*a la* `test`) if you intend on comparing numeric values
  - `[[ $a == $b ]]`
  - `[[ $a < $b ]]`: this would evaluate to "true" if a=100, b=2

Commands for conditionals

You can use any commands for conditions, but these constructs should be familiar:

- `test expr`: `test` command
  - Shorthand: `[ expr ]` (remember your spaces! `[` is technically a utility name)
  - `test $a -eq $b`
  - `[ $a -eq $b ]`

- `[[ expr ]]`: Bash conditional
  - Richer set of operators: `==`, `=`, `!=`, `<`, `>`, among others
  - **Note**: The symbol operators above operate on strings, thus `<` and `>` operators do lexicographic (i.e. dictionary) comparison; "100" is lexicographically less than "2" since for the first characters "1" comes before "2"
  - Use specific arithmetic binary operators (*a la* `test`) if you intend on comparing numeric values
  - `[[ $a == $b ]]`
  - `[[ $a < $b ]]`: this would evaluate to "true" if a=100, b=2

- `(( expr ))`: Bash arithmetic conditional
  - Evaluates as an arithmetic expression
  - `(( $a < $b ))`: this would evaluate to "false" if a=100, b=2

`while`

```
while test-commands; do
  commands
done
```

- Similarly to **`if`**, the return code of *test-commands* is used as the conditional
- Repeats *commands* until the condition **fails**

`while`

```
while test-commands; do
  commands
done
```

- Similarly to **`if`**, the return code of *test-commands* is used as the conditional
- Repeats *commands* until the condition **fails**

`until`

```
until test-commands; do
  commands
done
```

- Repeats *commands* until the condition **succeeds**

`for`

```
for var in list; do
  commands
done
```

- *list* will be **expanded** and on each iteration *var* will be set to each member of the list
- **Note**: if there is no `in list`, it will implicitly iterate over the argument list (i.e. `$@`)

Functions

```
func-name () compound-command
# or
function func-name [()] compound-command # [] for optional parens
```

- A **compound command** is a **command group** (`()`, `{}`) or a control flow element (`if-elif-else`, `for`)
- Called by invoking them like any other utility, including passing arguments
  - Arguments can be accessed via `$n`, where *n* is the argument number
  - `$@`: list of arguments
  - `$#`: number of arguments

Examples

```
hello-world ()
{
  if echo "Hello world!"; then
    echo "This should print"
  fi
}
# calling
hello-world
```

```
function touch-dir for x in $(ls); do touch $x; done
# calling
touch-dir
```

```
echo-args ()
{
  for x in $@; do
    echo $x
  done
}
# calling
echo-args a b c d e f g
```

```
divide ()
{
  if (( $2 == 0 )); then
    echo "Error: divide by zero" 1>&2
    # the redirection copies stderr to stdout
    # so when echo outputs to its stdout, it's
    # really going to stderr
  else
    echo $(($1 / $2))
  fi
}
# calling
divide 10 2
divide 10 0
```

Scripts

- As was mentioned a few weeks ago, it's annoying to have to type things/go to the history to repeatedly run some commands
- Scripts are just plain-text files with commands in them
- There's no special syntax for scripts: if you can enter the commands in them line by line at the terminal it would work
- You can treat it as a simple programming language

Scripts

- As was mentioned a few weeks ago, it's annoying to have to type things/go to the history to repeatedly run some commands
- Scripts are just plain-text files with commands in them
- There's no special syntax for scripts: if you can enter the commands in them line by line at the terminal it would work
- You can treat it as a simple programming language
- First line specifies the interpreter ("shebang")
  - `#!/bin/bash`
  - `#!/usr/bin/env bash`

Scripts

- As was mentioned a few weeks ago, it's annoying to have to type things/go to the history to repeatedly run some commands
- Scripts are just plain-text files with commands in them
- **There's no special syntax for scripts: if you can enter the commands in them line by line at the terminal it would work**
- You can treat it as a simple programming language
- First line specifies the interpreter ("shebang")
  - `#!/bin/bash`
  - `#!/usr/bin/env bash`
- Arguments work like that of functions:
  - `$n` **Note**: $0 will refer to the script's name, as per *nix program argument convention
  - `$@`
  - `$#`

Reiterating *running* vs *sourcing*

- *Running* (executing) a script puts it into its own shell instance; variables set *won't* be visible to the parent shell
  - `./script.sh`
  - `bash script.sh`
- Sourcing a script makes your current shell instance run each command in it; variables set *will* be visible
  - `source script.sh`
  - `. script.sh`

Regular expressions (regexes)

- A pattern that matches a set of strings
- Provide a (relatively) standardized way to perform matches on text

Regular expressions (regexes)

- A pattern that matches a set of strings
- Provide a (relatively) standardized way to perform matches on text
- Important to know as *many* tools and utilities make use of them
  - `grep`, `sed`, `find` to name a scant few

Regular expressions (regexes)

- A pattern that matches a set of strings
- Provide a (relatively) standardized way to perform matches on text
- Important to know as *many* tools and utilities make use of them
  - `grep`, `sed`, `find` to name a scant few
- Lots of different flavors, but they all encapsulate similar ideas

Regular expressions (regexes)

- A pattern that matches a set of strings
- Provide a (relatively) standardized way to perform matches on text
- Important to know as *many* tools and utilities make use of them
  - `grep`, `sed`, `find` to name a scant few
- Lots of different flavors, but they all encapsulate similar ideas
- You provide a **pattern** that is matched on the text
- The **pattern** can be a simple unassuming string or contain special characters that perform more powerful matching

Regular expressions (regexes)

- A pattern that matches a set of strings
- Provide a (relatively) standardized way to perform matches on text
- Important to know as *many* tools and utilities make use of them
  - `grep`, `sed`, `find` to name a scant few
- Lots of different flavors, but they all encapsulate similar ideas
- You provide a **pattern** that is matched on the text
- The **pattern** can be a simple unassuming string or contain special characters that perform more powerful matching
- For this lecture, we'll be looking at POSIX BRE (basic regex) and ERE (extended regex)
  - `grep` is a utility that searches for patterns in a file via regexes
  - Defaults to BRE; `-E` flag (or `egrep`) for ERE
  - `ls /dev | grep tty`: list `/dev` directory, filtering by things that contain "tty"

Regular expressions (regexes)

- A pattern that matches a set of strings
- Provide a (relatively) standardized way to perform matches on text
- Important to know as *many* tools and utilities make use of them
  - `grep`, `sed`, `find` to name a scant few
- Lots of different flavors, but they all encapsulate similar ideas
- You provide a **pattern** that is matched on the text
- The **pattern** can be a simple unassuming string or contain special characters that perform more powerful matching
- For this lecture, we'll be looking at POSIX BRE (basic regex) and ERE (extended regex)
  - `grep` is a utility that searches for patterns in a file via regexes
  - Defaults to BRE; `-E` flag (or `egrep`) for ERE
  - `ls /dev | grep tty`: list `/dev` directory, filtering by things that contain "tty"

Resources

- Online regex tester: https://regex101.com/ (one among many)
- GNU `grep`'s manual on regular expressions
- Highly detailed website: https://www.regular-expressions.info/

Regex basics

- Patterns are composed of smaller regexes that are concatenated
- The atomic regexes are those that match single characters
- The alphanumeric characters (A-Z, a-z, 0-9) act like normal characters
    - `hello` is a simple pattern that matches strings that contain "hello"

Regex basics

- Patterns are composed of smaller regexes that are concatenated
- The atomic regexes are those that match single characters
- The alphanumeric characters (A-Z, a-z, 0-9) act like normal characters
  - `hello` is a simple pattern that matches strings that contain "hello"
- There are also special functions denoted by special characters
  - `.` for any single character
  - `|` for an OR
  - `\` for special expressions/escapes
  - Quantifiers: how many to match
  - Brackets: a set of characters to match
  - Anchors: for *positional* matching
  - Backreferences: for matching a previous match
  - `^tty[0-9]+$` is a less simple pattern that matches lines that exactly compose of only "tty" and some numeric digits after it

Misc special characters

- `.` matches *any* single character
  - `...` matches strings containing three characters

Misc special characters

- `.` matches *any* single character
  - `...` matches strings containing three characters
- `|` for an OR between regexes
  - `hello|world` matches a string containing "hello" or "world"

Misc special characters

- `.` matches *any* single character
  - `...` matches strings containing three characters
- `|` for an OR between regexes
  - `hello|world` matches a string containing "hello" or "world"
- `\` for special expressions/escapes
  - `\b` matches empty string at the edge of a word
  - There's more: check the GNU `grep` manual for the rest

Misc special characters

- `.` matches *any* single character
  - `...` matches strings containing three characters
- `|` for an OR between regexes
  - `hello|world` matches a string containing "hello" or "world"
- `\` for special expressions/escapes
  - `\b` matches empty string at the edge of a word
  - There's more: check the GNU `grep` manual for the rest
- `(`, `)` enclose a whole expression as a *subexpression*
  - `(Hello|Goodbye) (Brandon|Jiwon)` matches:
  - "Hello Brandon"
  - "Hello Jiwon"
  - "Goodbye Brandon"
  - "Goodbye Jiwon"

Quantifiers

- Specify how many of a preceding regex to match
- `?`: ≤1 time
- `*`: ≥0 times
- `+`: ≥1 times
- `{n}`: *n* times
- `{n,}`: ≥*n* times
- `{,m}`: ≤*m* times
- `{n,m}`: ≥*n* and ≤*m* times

Quantifiers

- Specify how many of a preceding regex to match
- `?`: ≤1 time
- `*`: ≥0 times
- `+`: ≥1 times
- `{n}`: *n* times
- `{n,}`: ≥*n* times
- `{,m}`: ≤*m* times
- `{n,m}`: ≥*n* and ≤*m* times

Examples

- `a{4}`: matches "aaaa"
- `ba+`: matches "ba", "baa", "baaa"...
- `(hello){3}`: matches "hellohellohello"

Brackets

- `[`, `]` enclose a set to match for one character
  - `[abc]` matches 'a', 'b', or 'c'

Special things you can put inside them:

- `-`: range
  - `[A-Za-z0-9]`: capital and lowercase numbers and digits
- `^`: not in set
  - `[^ab]`: everything not 'a' or 'b'
- Named classes
  - `[:alnum:]`: alphanumeric characters
  - `[:alpha:]`: alphabetic characters
  - `[:digit:]`: digit characters
  - `[:blank:]`: space and tab characters
  - ...and others (see the GNU `grep` manual)
  - Brackets are part of the class name: e.g. `[[:alnum:]]` to match alphanumerics

Anchors

- Perform *positional* matching
- `^`: match empty string at the beginning of a line
  - i.e. following regex must be at the beginning
  - `^hello`: "hello" must be at the beginning

Anchors

- Perform *positional* matching
- `^`: match empty string at the beginning of a line
  - i.e. following regex must be at the beginning
  - `^hello`: "hello" must be at the beginning
- `$`: match empty string at the end of a line
  - i.e. preceding regex must be at the end
  - `world$`: "world" must be at the end

Anchors

- Perform *positional* matching
- `^`: match empty string at the beginning of a line
  - i.e. following regex must be at the beginning
  - `^hello`: "hello" must be at the beginning
- `$`: match empty string at the end of a line
  - i.e. preceding regex must be at the end
  - `world$`: "world" must be at the end
- `^hello world$`: entire string must be "hello world"

Backreferences

- Match previous parenthesized `()` subexpression
- `\n`: match *n* th parenthesized subexpression
  - `(123)testing\1` matches "123testing123"

Backreferences

- Match previous parenthesized `()` subexpression
- `\n`: match *n* th parenthesized subexpression
  - `(123)testing\1` matches "123testing123"

Q: `<([[:alpha:]][[:alnum:]]*[^>])>.*</\1>`

Backreferences

- Match previous parenthesized `()` subexpression
- `\n`: match *n* th parenthesized subexpression
  - `(123)testing\1` matches "123testing123"

Q: `<([[:alpha:]][[:alnum:]]*[^>])>.*</\1>`

- Match (simple) HTML/XML tags

Caveats

- GNU `grep` defaults to BRE flavor
  - Use `-E` flag or use `egrep` for ERE flavor
  - In ERE mode, use `[{]` to capture literal '{' for portability
- Other flavors may require escaping certain characters

BRE vs ERE

- In BRE `?`, `+`, `{`, `|`, `(`, and `)` must be escaped with `\`

Questions?