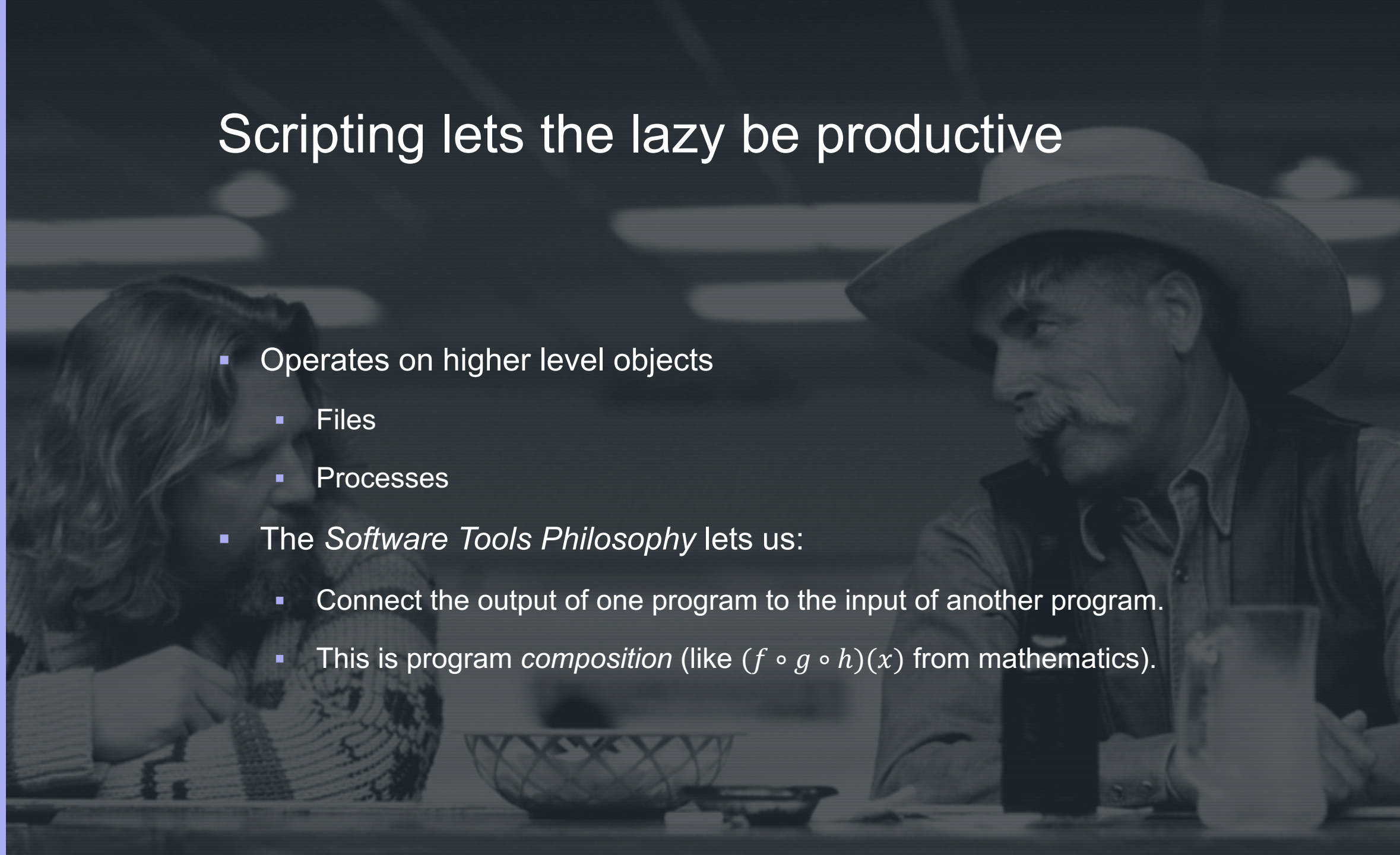
Prof. Darrell Long

CSE 13S

Scripting

5/19/21

Scripting lets the lazy be productive



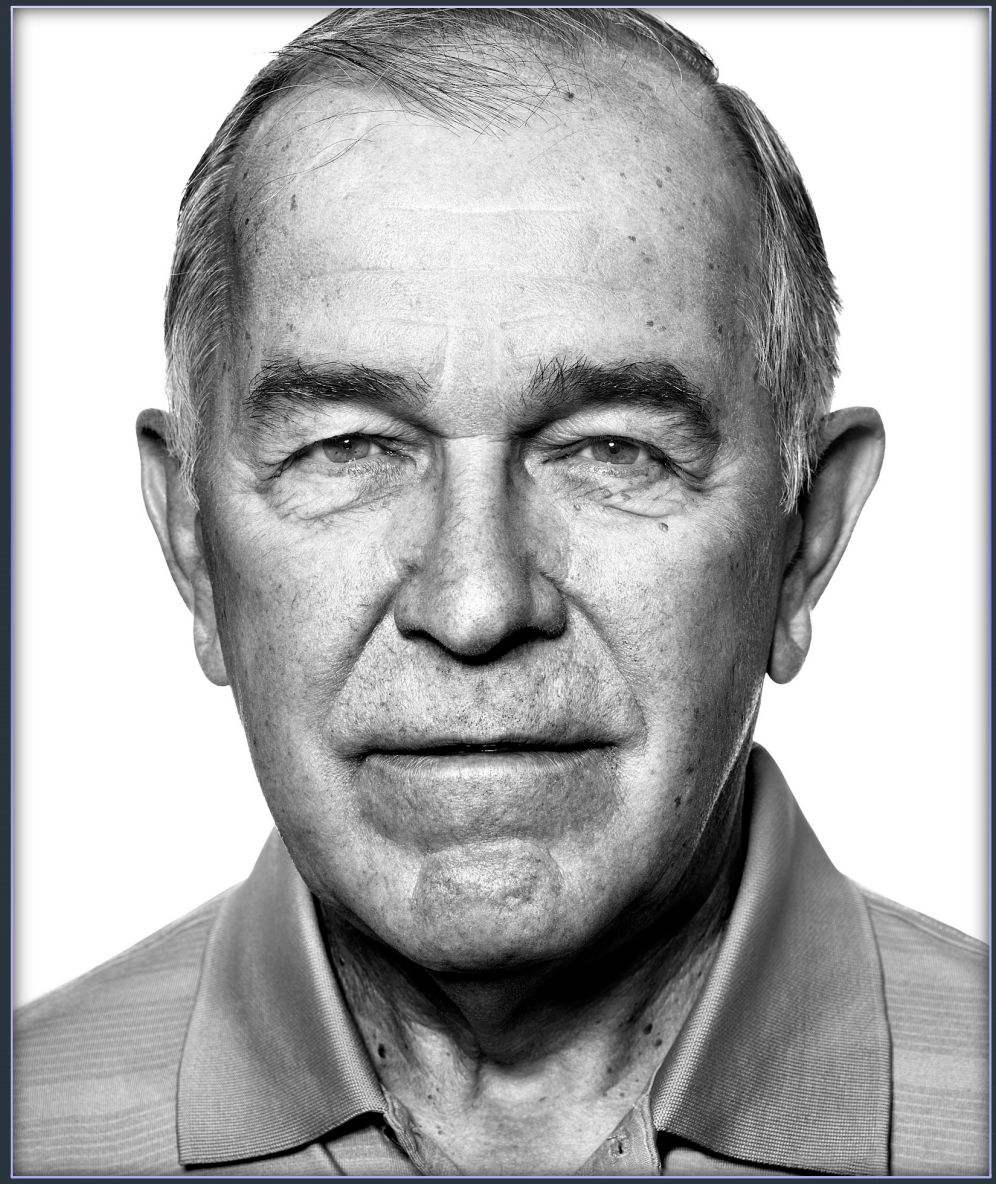
- Operates on higher level objects
  - Files
  - Processes
- The *Software Tools Philosophy* lets us:
  - Connect the output of one program to the input of another program.
  - This is program *composition* (like $(f \circ g \circ h)(x)$ from mathematics).

Who are you calling lazy?

- Smart programmers reuse code, programs, components.
- If I can use `sort` to sort a file, why should I rewrite it?
- If I can use `sed` to make edits in a file, why should I write an editor?
- If I can use `grep` to find all regular expressions in a file, …
- If I can use `awk` to parse a file, …

From `sh` to `zsh`

- `sh` was the original UNIX shell:
  - Called the Bourne shell, after Stephen Bourne
- There are many shells
  - `csh`, `tcsh`, `bash`, `zsh`, …
  - You can write your own!



BASH

- The **B**ourne **A**gain **Sh**ell
  - Written by Brian Fox as a replacement for the Bourne shell.
- Shells:
  - Act as *command interpreters.*
  - Allows users to give commands to the OS either…
    - Interactively: as a prompt or command line (typically indicated with a `$`)
    - Batched: as a *script* (a file containing a sequence of commands)

Commands

- Bash reads some input (usually a terminal or a file) line-by-line.
  - Each line is treated as an entire *command*.
  - Each line is split into different tokens, or *words*, by whitespace.
  - The first word on a line is the *command* to execute, with the following words the *arguments* for the command.
- Example:
  - `$ ls -a`
    - `ls` is the command to list files.
    - `-a` is an argument to `ls` that tell it to list files prefixed with a `.` (dot).

Types of commands

- **Aliases**
  - Only used in interactive shells.
  - Aliases are replaced with what they are aliasing before a command is executed
    - E.g. if `ll` is an alias for `ls -l`, then `ll` is replaced with `ls -l` before being executed interactively.
- **Builtins**
  - Bash-provided commands, such as `printf`, `echo`, and `cd`.
- **Functions**
  - A sequence of commands that perform some task.
  - Bash functions can accept arguments.
- **Executables**
  - Executable programs are specified and executed by file path.
    - Or just by their respective names if the directory they're contained in are in the `$PATH` environment variable (more on this soon).

$PATH

- A colon-delimited list of directory names.
  - These directories indicate where executable programs are located.
- Directories typically included in `$PATH`:
  - `/bin` – For commands required by system for repairs and booting.
  - `/usr/bin` – Primary directory of executables on the system.
- The current directory `.` should **not** be included in your `$PATH` for security.
  - Which is why executing your own programs typically requires a relative path.
    - `$ ./hello`

Writing a script

- A script is a sequence of commands in a file (usually made executable).
- The first line of a Bash script should be the *interpreter directive*.
  - Also referred to as a *hashbang* or a *shebang*.
  - Indicates which interpreter to interpret the script with.
  - `#!/bin/bash` – Interpret script using `/bin/bash`.

```
#!/bin/bash

echo "Hello World!"
echo "Goodbye World!"
```

Basic dotfiles

- Dotfiles are files prefixed with a dot (.) that sit in your home directory.
- There are 2 that directly relate with Bash (there are variants for different shells):
  1. `.bashrc`
  2. `.bash_profile`
- `.bashrc` is a script that is executed whenever Bash is started interactively.
  - Commands that should be run every time a new interactive shell is started go here.
  - Such as customized shell prompts, aliases, etc.
- `.bash_profile` is also a script, but is executed only at the start of a new login shell.
  - Commands that should only be run once should be put in here.
  - Example: modifying/exporting the `$PATH` environment variable.

Parameters & Variables

- Used to store data.
- A variable is a kind of parameter.
  - The fact that a variable has a name is the distinguishing factor.
- Variable names can only consist of letters, digits, and underscores, and only start with either a letter or an underscore.
- Assign data using the assignment operator: =.
  - Cannot use spaces around = in an assignment.
- Variables can be strings, integers, or arrays.

```
# String.
i="Hello World!"

# Integer.
j=0

# Array (doesn't need to be homogeneous).
k=("Hello" 6 7 "Goodbye")
```

Expansion

- Can expand parameters using the expansion operator: `$`.
- Variable expansion substitutes the variable with its value.

```
msg="This is a message."

# Echoes out "This is a message."
echo $msg
```

Arithmetic

- Arithmetic expressions are performed in `(( … ))`.
- The result of these expressions can be expanded with `$(( … ))`.
- Some arithmetic operators (non-exhaustive):
  - `var++, var--`
  - `++var, --var`
  - `+, -, *, /, %`
  - `<, <=, =, !=, >=, >`
  - `&, ^, |, <<, >>`
  - `+=, -=, *=, /=, %=`

```
i=5
((i-=2))

j=$((i+3))

echo $i # 3.
echo $j # 6.
```

Special Parameters

- Variables are named parameters.
  - *Special parameters* are parameters that are not variables.
- Examples:
  - `$n` - The `n`-th argument passed to the current script or function (these are positional parameters).
    - Use curly braces if `n` has more than one digit: `${10}`.
  - `$#` - The number of positional parameters.
  - `$$` - the process ID (PID) of the current shell.
  - `$_` - The last argument of the last executed command.
  - `$?` - Exit status of last executed command.

Functions

```bash
# Greets the first argument.
function greet() {
    echo "Hello $1!"
}

# Inelegant function to count files ending with ".txt".
function count_txts() {
    # Local variable to store count.
    local count=0

    # Loop through all files in directory.
    for f in *; do
        # Increase count if filename matches pattern.
        # More on globbing later.
        if [[ $f = *.txt ]]; then
            ((count+=1))
        fi
    done

    echo "Files ending with \".txt\": $count"
}

greet "CSE13S"
count_txts
```

- Sequences of commands.
- Can have multiple functions within one script.
- Allows for *local* variables.
  - Prevents issue of overwriting variables from the function caller's scope.

Arrays

- An array is a list of strings (yes, it maps integers to strings).
- Most simply created using the `arr=(…)`.
- Bash allows for arrays created using explicit indices for more flexibility.
  - Arrays with gaps between indices are *sparse arrays*.

```
# Array of fruits (known set size).
fruits=("apple" "banana" "cranberry")

# Array of treats using explicit indices.
treats=([0]="skittles" [1]="gummy bears" [4]="twix")
```

More on arrays

- Expanding specific array indices:
  - `${arr[n]}`
- Getting the number of elements of an array:
  - `${#arr[@]}`
- Expanding array to *list* of words:
  - `"${arr[@]}"`
- Expanding array as a *single* string:
  - `"${arr[*]}"`
- Appending array with another array:
  - `+= ( … )`
- Expanding n-th index of array:
  - `${arr[n]}`

```
# Array of fruits.
fruits=("apple" "banana" "cranberry")

# Echoes 3, the number of elements in the array.
echo ${#fruits[@]}

# Loop to echo out each fruit.
for fruit in "${fruits[@]}"; do
  echo $fruit
done

# Add grape and kiwi.
fruits+=("grape" "kiwi")

# Echo out all fruits as a single string.
echo "${fruits[*]}"

# Echoes "banana".
echo ${fruits[1]}
```

Globs & Patterns

- **Globs:** patterns used to match strings.
  - `*` - matches any string including the null (empty) string.
    - `$ ls *.c` (matches anything that ends with "`.c`")
  - `?` - matches any single character.
  - `[chars]` - matches any character in chars
  - `[a-z]` - matches any character between `a` and `z`
- Echo names of text files, JPEGs, and PDFs:
  - `echo *.{txt,jpeg,pdf}`

Testing & Conditionals

- The logical operators that appear in **C** appear here in Bash as well:
  - `&&` – logical AND
  - `||` – logical OR
  - `!` – logical NOT
- Equality is tested using = (not == as is used in **C**).
- There are also `if`, `elif`, and `else` blocks.
  - Conditional blocks must end with `fi`.
- Tests are done within `[[ … ]]`.
  - Can also use `[ … ]`, but it's older and lacks features such as pattern matching.
  - Use `[ … ]` if the script may be run in an environment without Bash.

```
a="hello"
b="goodbye"

if [[ $a = $b ]]; then
  echo "a matches b"
elif [[ $a ≠ $b ]]; then
  echo "a doesn't match b"
else
  echo "what happened?"
fi
```

Testing (strings)

- Tests on strings:
  - `$s1 = $s2` (s1 match s2?)
  - `$s1 != $s2` (s1 doesn't match s2?)
  - `$s1 < $s2` (s1 lexicographically less than s2?)
  - `$s1 > $s2` (s2 lexicographically greater than s2?)
  - `-z $s` (is s an empty string?)
  - `-n $s` (is s not an empty string?)

```
s1="abc"
s2="abcd"

if [[ $s1 = $s2 ]]; then
  echo "s1 matches s2"
elif [[ $s1 < $s2 ]]; then
  echo "s1 < s2"
elif [[ $s1 > $s2 ]]; then
  echo "s1 > s2"
fi

s3=""
if [[ -z $s3 ]]; then
  echo "s3 is empty string"
else
  echo "s3 is not empty string"
fi
```

Testing (integers)

- Tests on integers:
  - `$x -lt $y` (`$x < $y`)
  - `$x -le $y` (`$x <= $y`)
  - `$x -eq $y` (`$x = $y`)
  - `$x -ne $y` (`$x != $y`)
  - `$x -ge $y` (`$x >= $y`)
  - `$x -gt $y` (`$x > $y`)
- Can also test integers in arithmetic contexts.
  - Between `(( … ))`

```
s1=7
s2=11

# In Bash test [[ ... ]].
if [[ $s1 -lt $s2 ]]; then
  echo "s1 < s2"
fi

# In arithmetic context.
if (( $s1 < $s2 )); then
  echo "s1 < s2"
fi
```

Testing (files)

- Tests pertaining to files:
  - `-f $file` (is `$file` a regular file?)
  - `-d $file` (is `$file` a directory?)
  - `-e $file` (does `$file` exist?)
  - `-r $file` (can user read `$file`?)
  - `-w $file` (can user write to `$file`?)
  - `-x $file` (can user execute `$file`?)

```
f="hello"

if [[ -e $f ]]; then
    if [[ -f $f ]]; then
        echo "$f exists and is a regular file"
    elif [[ -d "$f" ]]; then
        echo "$f exists and is a directory"
    fi
else
    echo "$f doesn't exist"
fi

if [[ -e $f ]]; then
    if [[ -r $f ]]; then
        echo "Can read $f"
    fi
    if [[ -w $f ]]; then
        echo "Can write to $f"
    fi
    if [[ -x $f ]]; then
        echo "Can execute $f"
    fi
fi
```

Loops

- Four different loop syntaxes in Bash:
  - `While`
    - Loop *while* condition is true.
  - `Until`
    - Loop *until* condition is true.
  - `for win word`
    - Loop for each word of `words`, setting `w` as each word.
  - `for ((init; cond; incr))`
    - Loop starting with the `init` expression, while the `cond` expression is true, and incrementing with the `incr` expression after each loop body execution.

```
# While loop.
i=0
while [[ $i ≠ 5 ]]; do
  ((i+=1))
done

# Until loop.
i=0
until [[ $i = 5 ]]; do
  ((i+=1))
done

# For loop (first version).
# {0..4} is brace expansion for 0 1 2 3 4.
# txt is set to each of the expanded words.
for i in {0..4}; do
  echo $i
done

# For loop (C-like version).
for ((i=0; $i≠5; i+=1)); do
  echo $i
done
```

File redirection

- Each process on UNIX has access to the following file descriptors:
  - `stdin` - standard input
    - File descriptor 0
  - `stdout` - standard output
    - File descriptor 1
  - `stderr` - standard error
    - File descriptor 2
- These file descriptors can be redirected from or to files.

```
# Redirects stdout of echo to hello.txt
echo "Hello!" > hello.txt

# Redirects hello.txt into stdin of cat.
cat < hello.txt

# Redirects stderr of program into err.txt.
# The 2 indicates the file descriptor to redirect.
./program 2> err.txt

# First, redirect program's stderr to its stdout.
# Then, redirect stdout to all.txt.
# This means that both stderr and stdout are redirected.
./program 2>&1 > all.txt

# Can also redirect stderr and stdout like so.
./program &> all.txt
```

Pipes

- Connects `stdout` of one process to `stdin` of another process.
- Pipes are denoted with the pipe operator: "|".
- Pipes can be chained together, creating a pipeline.
- Programs can be written such that they can be chained together to get interesting results.
  - The whole philosophy behind the software tools that drives UNIX.
- Example pipeline: list the IDs of the user's processes (limit to 10).
  - `ps aux | grep $USER | awk '{ print $2 }' | head`
    - What's `awk`?

Why is called `awk`?

- `awk` is a program for simple parsing of text.
  - It follows the *Software Tools Philosophy*.
- It is named after:
  - Albert Aho (`a`)
  - Peter Weinberger (`w`)
  - Brian Kernighan (`k`)

My friend PJW

awk

- Developed by Alfred **A**ho, Peter J. **W**einberger, and Brian **K**ernighan.
- Designed as a text processor that scans input files, splitting input lines into fields automatically.
- Output from `awk` can be processed even further with other tools.
  - Such as with `sort`, or `uniq`.
  - All thanks to the UNIX software philosophy.

```
eugene at icicleinn in ~/bash
$ cat sea.txt
Spongebob 1
Patrick 2
Sandy 3
Squidward 4
eugene at icicleinn in ~/bash
$ awk '{ print $1 }' sea.txt
Spongebob
Patrick
Sandy
Squidward
```

A simple `awk` program

- A simple `awk` program is a sequence of pattern-action statements:
  - Written as:
    - `awk 'pattern { action }'`
  - Patterns are optional.
- Suppose we have a file of workers containing their names, the hours they worked for the week, and their hourly wage.
  - How would we process the file to only display the names of the workers who worked more than 10 hours during the week?

```
eugene at icicleinn in ~/bash
$ cat wages.txt
Spongebob 20 15.45
Patrick 8 20.00
Sandy 9 11.45
Squidward 40 16.45
eugene at icicleinn in ~/bash
$ awk '$2 > 10 { print $1 }' wages.txt
Spongebob
Squidward
```

More advanced patterns

- Patterns in `awk` can be combined (or even constructed using *regular expressions*).
  - Combined using parentheses and logical operators:
    - `&&` – logical AND
    - `||` – logical OR
    - `!` – logical NOT
  - Using the same wage file as before, we want the names of the workers who worked at least 10 hours *and* make at least $16 an hour.

```
eugene at icicleinn in ~/bash
$ cat wages.txt
Spongebob 20 15.45
Patrick 8 20.00
Sandy 9 11.45
Squidward 40 16.45
eugene at icicleinn in ~/bash
$ awk '($2 > 10) && ($3 > 16.00) { print $1 }' wages.txt
Squidward
```

Larger `awk` programs

- `awk` is capable of much more than the simple text processing seen so far.
  - Can write full-fledged programs in `awk`.
- Here's a simple program that counts the number of workers that worked more than 10 hours and less than 10 hours:
  - Uses `BEGIN`, and `END`, as well as `if-else` conditions.
  - Also makes use of variables.
    - User-created variables in `awk` aren't declared.
- Run as:
  - `$ awk -f worker.awk wages.txt`

```
# worker.awk - Counts hard, average, and total workers.

# Performed before first line of first input file.
BEGIN {
    print "Workers and the hours they worked:"
}

# Performed on each line of input.
{
    if ($2 > 10)
        hardworker += 1
    else
        avgworker += 1
    print $1 ":", $2, "hours"
}

# Performed after last line of last input file.
END {
    print "Number of hard workers:", hardworker
    print "Number of average workers:", avgworker
    print "Number of workers:", hardworker + avgworker
}
```

Another `awk` example

- Another sample awk program to find and print out the longest line in some input.
- Makes use of:
  - **Arrays**
  - **Built-in functions**:
    - length(x): calculates length in characters of x.
  - **Built-in variables**:
    - NR: holds the number of lines currently read.

```
# longest.awk - Find longest line.

{
    # Save each input line into array.
    # $0 refers to the entire line.
    lines[NR] = $0

    # Update longest line and its line number.
    if (length($0) > len)
        len = length($0)
        linenr = NR
}

END {
    print "Longest line:", lines[linenr]
    print "Characters:", len
}
```

One last `awk` example

- Sorts input lines by increasing length.
- Makes use of
  - **Arrays**
  - **Functions**
    - Can accept parameters
    - Can return values
  - **Loops** (`while`/`for`)
    - Syntax of loops matches **C**-style loops.

```
# sortlines.awk - Sorts input lines by length.

# Inserts a line in the array lines.
# nlines refers to the number of lines in the array.
function insert(lines, nlines, line) {
    slot = nlines
    while (slot != 0 && length(lines[slot - 1]) > length(line)) {
        lines[slot] = lines[slot - 1]
        slot -= 1
    }
    lines[slot] = line
}

{
    # NR - 1 since NR starts the first line as 1.
    # Passing NR - 1 (0) means the lines array is empty.
    insert(lines, NR - 1, $0)
}

END {
    for (i = 0; i < NR; i += 1) {
        print lines[i]
    }
}
```

Useful commands

- `diff` – compare files line by line
- `tee` – copies `stdin` to `stdout`, copying the read contents to 0 or more files
- `sort` – sort or merge lines of files
- `uniq` – report or filter out duplicate lines in a file
- `wc` – report word, line, character, and/or byte count
- `mv` – move (or renames) a file
- `cp` – copy a file
- `rm` – remove a file