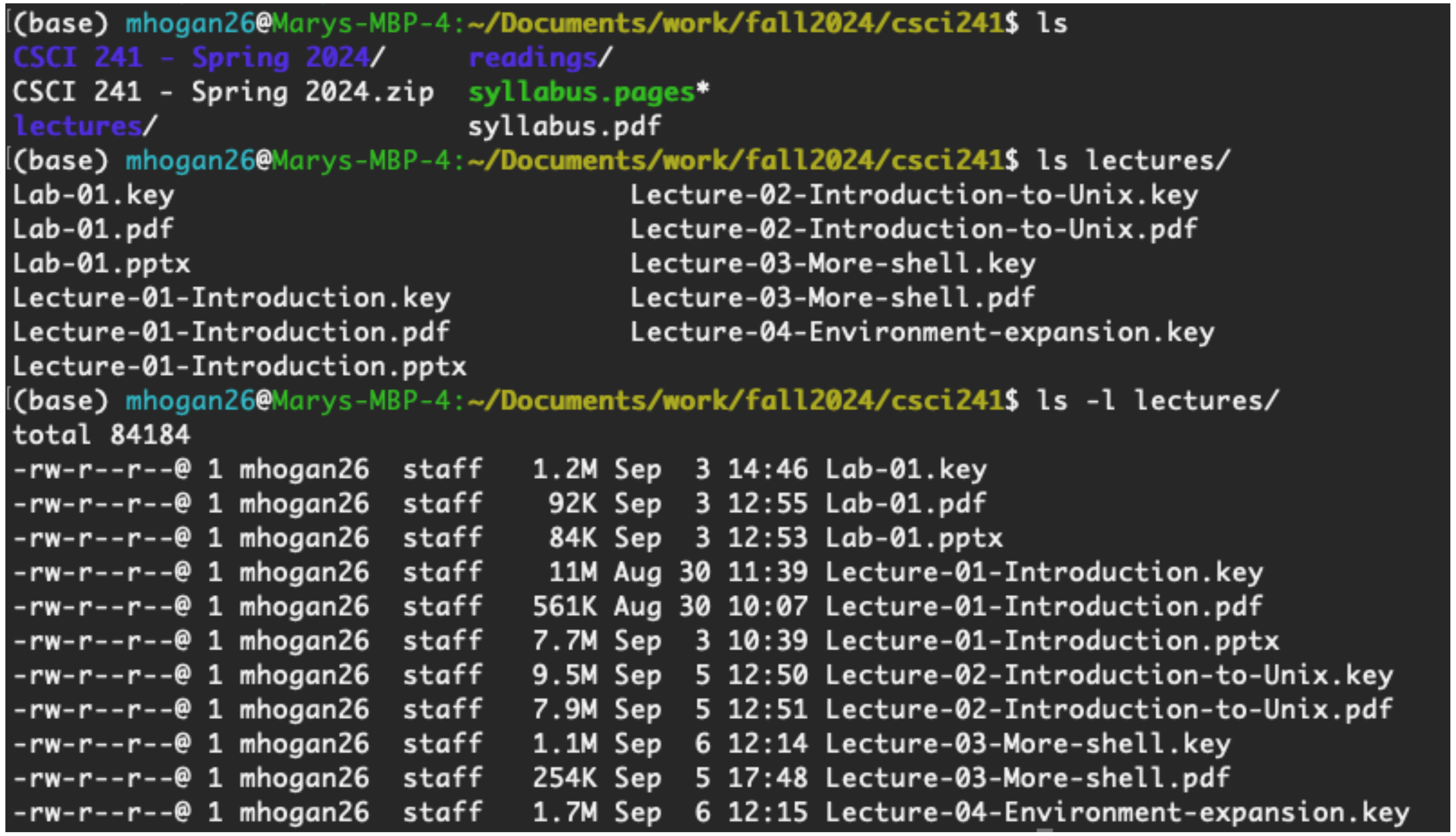
CS 241: Systems Programming Lecture 4. Environment and Expansion

Spring 2025

Prof. Stephen Checkoway

Program behavior

Most programs can have different behaviors when run with different arguments. E.g., the `ls` program can list the contents of different directories and can display the output in multiple formats



```
(base) mhogan26@Marys-MBP-4:~/Documents/work/fall2024/csci241$ ls
CSCI 241 - Spring 2024/     readings/
CSCI 241 - Spring 2024.zip  syllabus.pages*
lectures/                   syllabus.pdf
(base) mhogan26@Marys-MBP-4:~/Documents/work/fall2024/csci241$ ls lectures/
Lab-01.key                          Lecture-02-Introduction-to-Unix.key
Lab-01.pdf                          Lecture-02-Introduction-to-Unix.pdf
Lab-01.pptx                         Lecture-03-More-shell.key
Lecture-01-Introduction.key         Lecture-03-More-shell.pdf
Lecture-01-Introduction.pdf         Lecture-04-Environment-expansion.key
Lecture-01-Introduction.pptx
(base) mhogan26@Marys-MBP-4:~/Documents/work/fall2024/csci241$ ls -l lectures/
total 84184
-rw-r--r--@ 1 mhogan26  staff   1.2M Sep  3 14:46 Lab-01.key
-rw-r--r--@ 1 mhogan26  staff    92K Sep  3 12:55 Lab-01.pdf
-rw-r--r--@ 1 mhogan26  staff    84K Sep  3 12:53 Lab-01.pptx
-rw-r--r--@ 1 mhogan26  staff    11M Aug 30 11:39 Lecture-01-Introduction.key
-rw-r--r--@ 1 mhogan26  staff   561K Aug 30 10:07 Lecture-01-Introduction.pdf
-rw-r--r--@ 1 mhogan26  staff   7.7M Sep  3 10:39 Lecture-01-Introduction.pptx
-rw-r--r--@ 1 mhogan26  staff   9.5M Sep  5 12:50 Lecture-02-Introduction-to-Unix.key
-rw-r--r--@ 1 mhogan26  staff   7.9M Sep  5 12:51 Lecture-02-Introduction-to-Unix.pdf
-rw-r--r--@ 1 mhogan26  staff   1.1M Sep  6 12:14 Lecture-03-More-shell.key
-rw-r--r--@ 1 mhogan26  staff   254K Sep  5 17:48 Lecture-03-More-shell.pdf
-rw-r--r--@ 1 mhogan26  staff   1.7M Sep  6 12:15 Lecture-04-Environment-expansion.key
```

What controls program behavior?

Input arguments (e.g., file/directory paths, a URLs or command names)

Contents of the input files

Command line options

Configuration/preference files (or OS-specific configuration/preference databases)

User input (for interactive programs)

Environment variables!

Bash simple command revisited

Recall we said a simple command has the form:

⟨`command`⟩ ⟨`options`⟩ ⟨`arguments`⟩

The truth is more complicated

- ⟨variable assignments⟩ ⟨words and redirections⟩
- Variables and their assigned values are available to the command
- The first word is the command, the rest are arguments*
- `FOO=blah BAR=okay cmd aaa >out bbb 2>err ccc <in`
- `FOO=blah BAR=okay cmd aaa bbb ccc <in >out 2>err`
- Real example: $ `IFS= read -r var`

* Bash doesn't distinguish between options and arguments, that's up to each command

Environment variables

Another method for passing data to a program

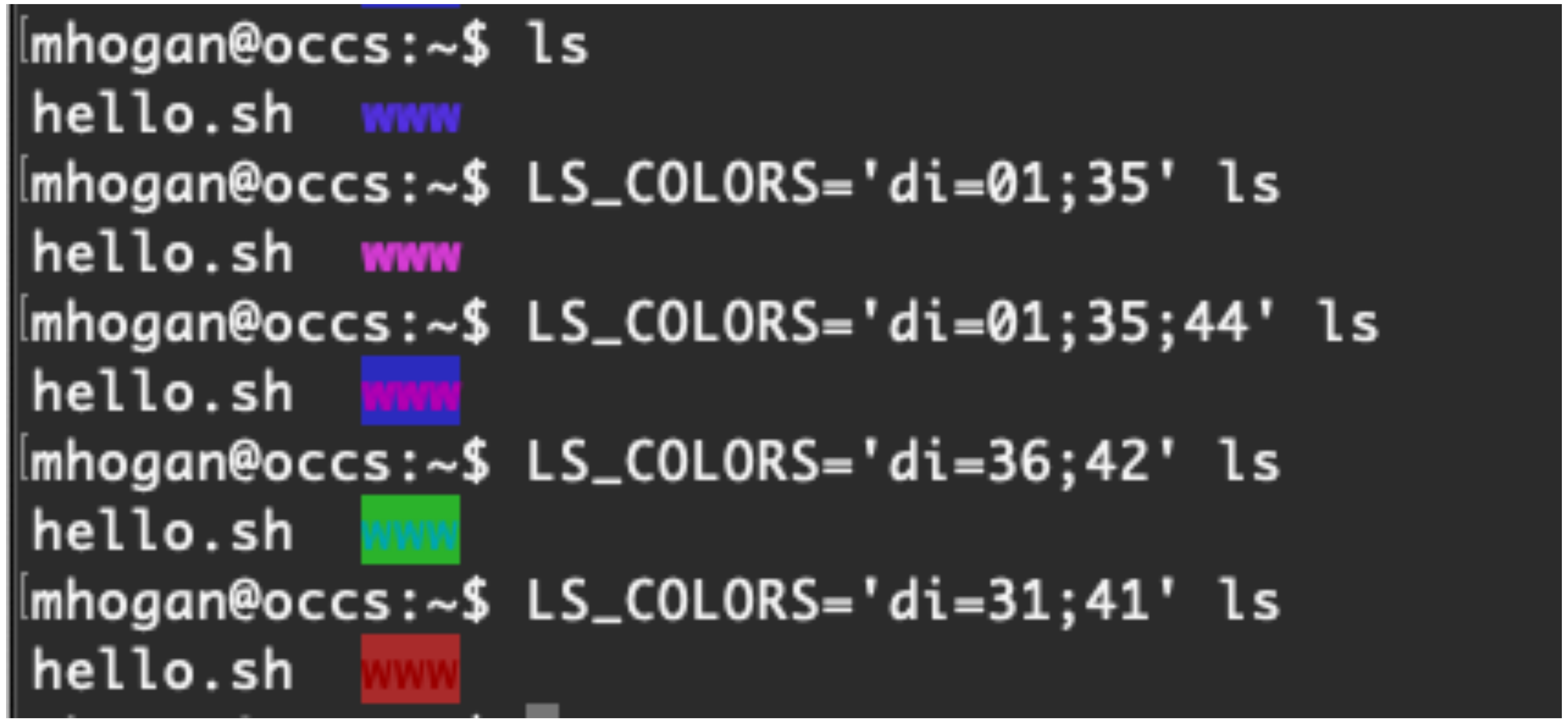
Essentially a key/value store (i.e., a hash map)

- `$ FOO=blah BAR=okay cmd aaa bbb ccc`
- `cmd` has access to the `FOO` and `BAR` environment variables plus args

Environment variables are inherited from the parent

- Every program started from the shell has access to a copy of the shell's environment

Example: color output from ls



```
mhogan@occs:~$ ls
hello.sh  www
mhogan@occs:~$ LS_COLORS='di=01;35' ls
hello.sh  www
mhogan@occs:~$ LS_COLORS='di=01;35;44' ls
hello.sh  www
mhogan@occs:~$ LS_COLORS='di=36;42' ls
hello.sh  www
mhogan@occs:~$ LS_COLORS='di=31;41' ls
hello.sh  www
```

Bash variables

Setting and using variables in bash

-
  ```
  $ place=Earth
  $ echo "Hello ${place}."
  Hello Earth.
  ```
- The {braces} aren't required
  ```
  $ echo "Hello $place."
  Hello Earth.
  ```

These are NOT environment variables!

Confusingly, Bash uses the same syntax to manipulate environment variables and normal variables

-
  ```
  $ echo "${LANG}"
  en_US.UTF-8
  ```

Bash variables

By default, variables set in bash aren't inherited by children

-
  ```
  $ place=Earth
  $ bash # Start a new shell
  $ echo "Hello ${place}."
  Hello . # ${place} expanded to the empty string
  ```

Exporting variables

We can export a variable which causes it to appear in the environment of children (essentially, turning a normal Bash variable into an environment variable)

```
$ place=World
$ export place
$ bash        # Starting a new shell
$ echo "Hello ${place}."
Hello World.
```

Equivalently, `$ export place=World`

Summarizing

```
$ FOO=bar cmd1
$ cmd2
```

- `FOO` available to `cmd1` but not `cmd2`

```
$ FOO=bar
$ cmd1
$ cmd2
```

- `FOO` not available to either `cmd1` or `cmd2`

```
$ export FOO=bar
$ cmd1
$ cmd2
```

- `FOO` available to both `cmd1` and `cmd2`

If bash is started via

```
$ W=foo bash
```

(so `W` is in bash's environment) and then following lines are executed,

```
$ X=bar
$ export Y=qux
$ Z=X some_program
```

which environment variables are available to `some_program`?

A. `W`, `X`, `Y`, and `Z`

B. `W`, `Y`, and `Z`

C. `X`, `Y`, and `Z`

D. `Y` and `Z`

E. `Z`

Useful environment variables

`EDITOR`	— Used when some commands need to launch an editor (e.g., git)
`HOME`	— Your home directory
`LANG`	— The language programs should use (this is complicated!)
`PAGER`	— A program like less that's used to display pages of text
`PATH`	— Colon-separated list of directories to search for commands
`PS1`	— The shell's prompt
`PWD`	— The current working directory
`SHELL`	— The shell you're using
`TERM`	— The terminal type, used to control things like color support
`UID`	— The real user ID number
`USER`	— User name

Adding directories to PATH

If you install software in `${HOME}/local/bin`, you can modify your `PATH` to access it

```
$ export PATH="${HOME}/local/bin:${PATH}"
```

This adds `${HOME}`/local/bin to the front of the PATH so it is searched first

```
$ export PATH="${PATH}:${HOME}/local/bin"
```

This adds `${HOME}/local/bin` to the end of the `PATH` so it is searched last

Environment variables are inherited

Environment variables are inherited by default by child processes

1. Bash starts up and sets some environment variables (from .bash_profile)
2. User runs git commit with no commit message
3. Git uses the EDITOR environment variable to open an editor for the user to enter the commit message

No need pass options to Git to select the editor, it can use the standard environment variable

Environment variables are inherited

Environment variables are inherited by default by child processes

1. Bash starts up and sets some environment variables (from .bash_profile)

2. User runs a script; the environment is inherited

3. The script runs git commit without a commit message; the environment is inherited

4. Git uses the EDITOR environment variable to open an editor for the user to enter the commit message

Bash expansion

Bash first splits lines into words by (unquoted) space or tab characters

```
$ echo 'quoted    string' unquoted    string
```

- Word 1: `echo`
- Word 2: `'quoted    string'`
- Word 3: `unquoted`
- Word 4: `string`

Most words then undergo **expansion**

- The values in variable assignment `var=value` (but not the names)
- The command and arguments
- The right side of redirections, e.g., 2>path

Expansion and then execution

Consider the example from before

```
$ place=Earth
$ echo "Hello ${place}."
```

Before the second line is executed, the whole line undergoes expansion

It becomes (essentially)

```
$ echo 'Hello Earth.'
```

and this gets executed

What is printed when I run this?

```
$ FOO=before
$ FOO=after echo "${FOO}"
```

A. before

B. after

C. beforeafter

D. Just a newline

E. Nothing, it's a syntax error

Variable expansion example

Most common expansions are variable expansion and globbing

```
base_dir=/tmp
if [[ $# -eq 1 ]]; then
    base_dir="$1"
fi

echo "Copying all Rust files to ${base_dir}/src"
mkdir -p "${base_dir}/src"
cp *.rs "${base_dir}/src"
```

Bash expansion

Order of expansion

- Brace expansion
- In left-to-right order, but at the same time
  - Tilde expansion
  - Variable expansion
  - Arithmetic expansion
  - Command expansion
  - Process substitution
- Word splitting (yes, this happens after the shell split the input into words!)
- Pathname expansion

And then each of the results undergoes quote removal

Brace expansion

Unquoted braces { } expand to multiple words

- `{foo,bar,baz}.txt` → `foo.txt bar.txt baz.txt`
- `foo{a,b,,c}bar → fooabar foobbar foobar foocbar`
- `'{a,b}' → '{a,b}'`
- `"{a,b}" → "{a,b}"`
- `{1..5} → 1 2 3 4 5`
- `{x..z} → x y z`
- `{1,2}{x..z} → 1x 1y 1z 2x 2y 2z`
- `{a,b{c,d}} → a bc bd`

Tilde expansion

Words starting with unquoted tildes expand to home directories

- `~` → `/usr/users/noquota/faculty/mhogan`
- `~mhogan` → `/usr/users/noquota/faculty/mhogan`
- `~steve` → `/usr/users/noquota/faculty/steve`
- `\~mhogan` → `~mhogan`
- `'~mhogan'` → `'~mhogan'`

Parameter/variable expansion

We can assign variables via `var=value` (e.g., `class='CS 241'`) the shell defines others like `HOME` and `PWD`

Words containing `${var}` or `$var` are expanded to their value, even in double quoted strings (**you almost always want to put them in quotes!**)

- `${HOME}` → `/usr/users/noquota/faculty/mhogan`
- `x${PWD}y  →  x/tmpy #` the current working directory
- `x$PWDy  →  x #` no `PWDy` variable so it expands to the empty string
- `'${class}'` → `'${class}'`
- `\${class}` → `${class}`
- `"${class}"` → `"CS 241"`

Command substitution

Replaces `$(command)` with its output (with the trailing newline stripped)

- `"Hello $(echo "${class}" | cut -c 4-)"` → `"Hello 241"`

These can be nested

You can also use `` `command` `` instead, but don't do that, use `$(…)`

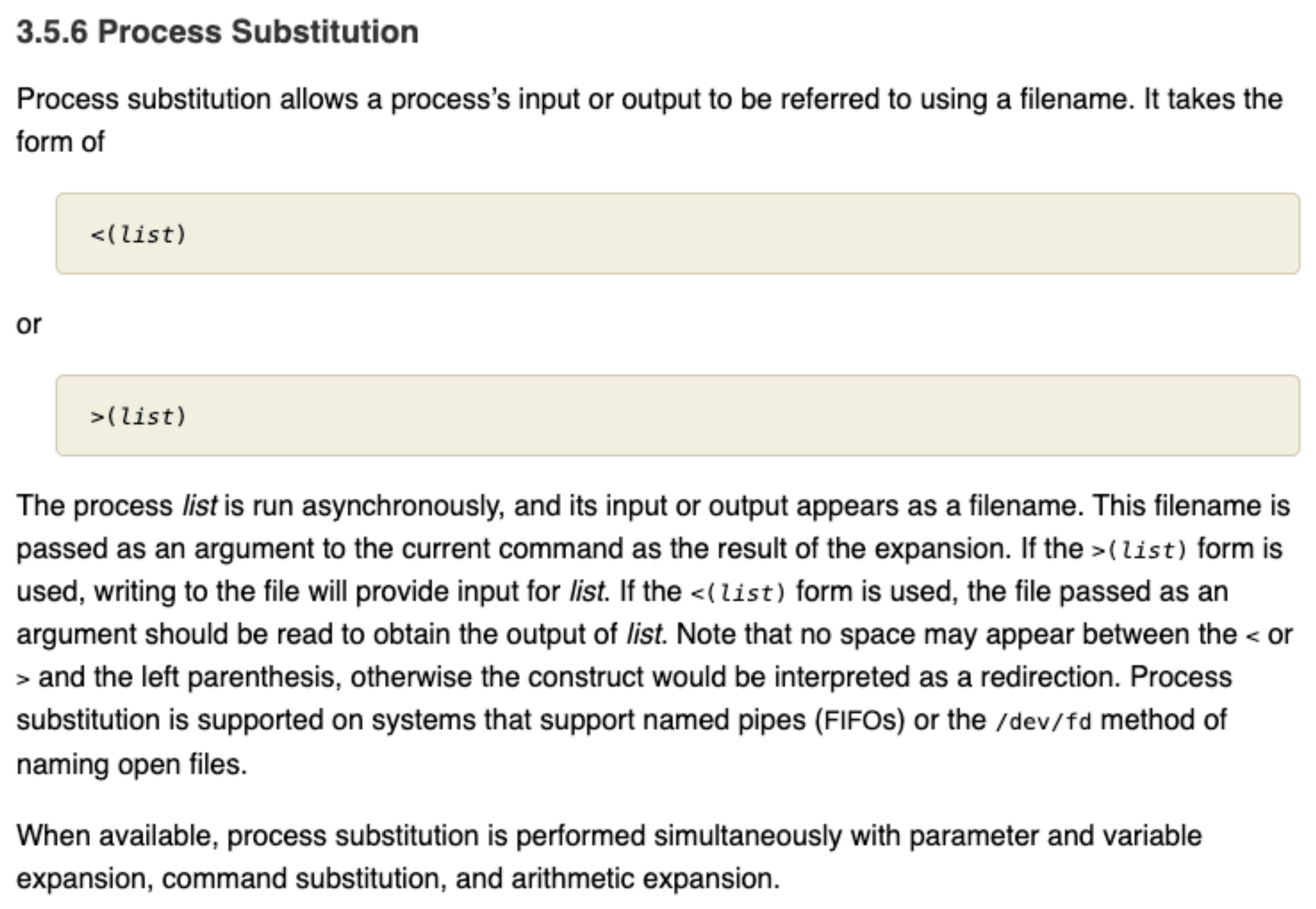
Arithmetic expansion

`$((arithmetic expression))` expands to the result, assume `x=10`

- `$((3+x*2 % 6))` → 5
- `\$((3+x*2 % 6))` → # syntax error
- `'$((3+x*2 % 6))'` → `'$((3+x*2 % 6))'`
- `"$((3+x*2 % 6))"` → `"5"`

Process substitution

Read the man page for bash if you want, we may come back to it



3.5.6 Process Substitution

Process substitution allows a process's input or output to be referred to using a filename. It takes the form of

```
<(list)
```

or

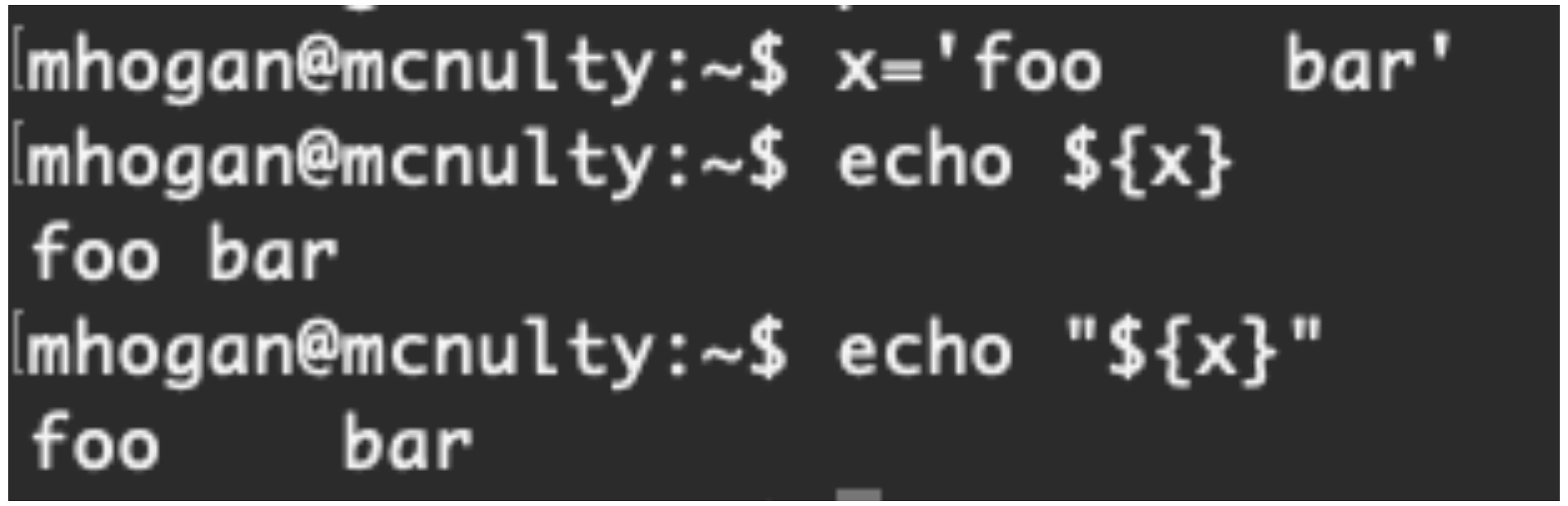
```
>(list)
```

The process *list* is run asynchronously, and its input or output appears as a filename. This filename is passed as an argument to the current command as the result of the expansion. If the `>(list)` form is used, writing to the file will provide input for *list*. If the `<(list)` form is used, the file passed as an argument should be read to obtain the output of *list*. Note that no space may appear between the `<` or `>` and the left parenthesis, otherwise the construct would be interpreted as a redirection. Process substitution is supported on systems that support named pipes (FIFOs) or the `/dev/fd` method of naming open files.

When available, process substitution is performed simultaneously with parameter and variable expansion, command substitution, and arithmetic expansion.

https://www.gnu.org/software/bash/manual/bash.html#Process-Substitution

Word splitting



```
mhogan@mcnulty:~$ x='foo     bar'
mhogan@mcnulty:~$ echo ${x}
foo bar
mhogan@mcnulty:~$ echo "${x}"
foo     bar
```

A misfeature in bash!

The results of

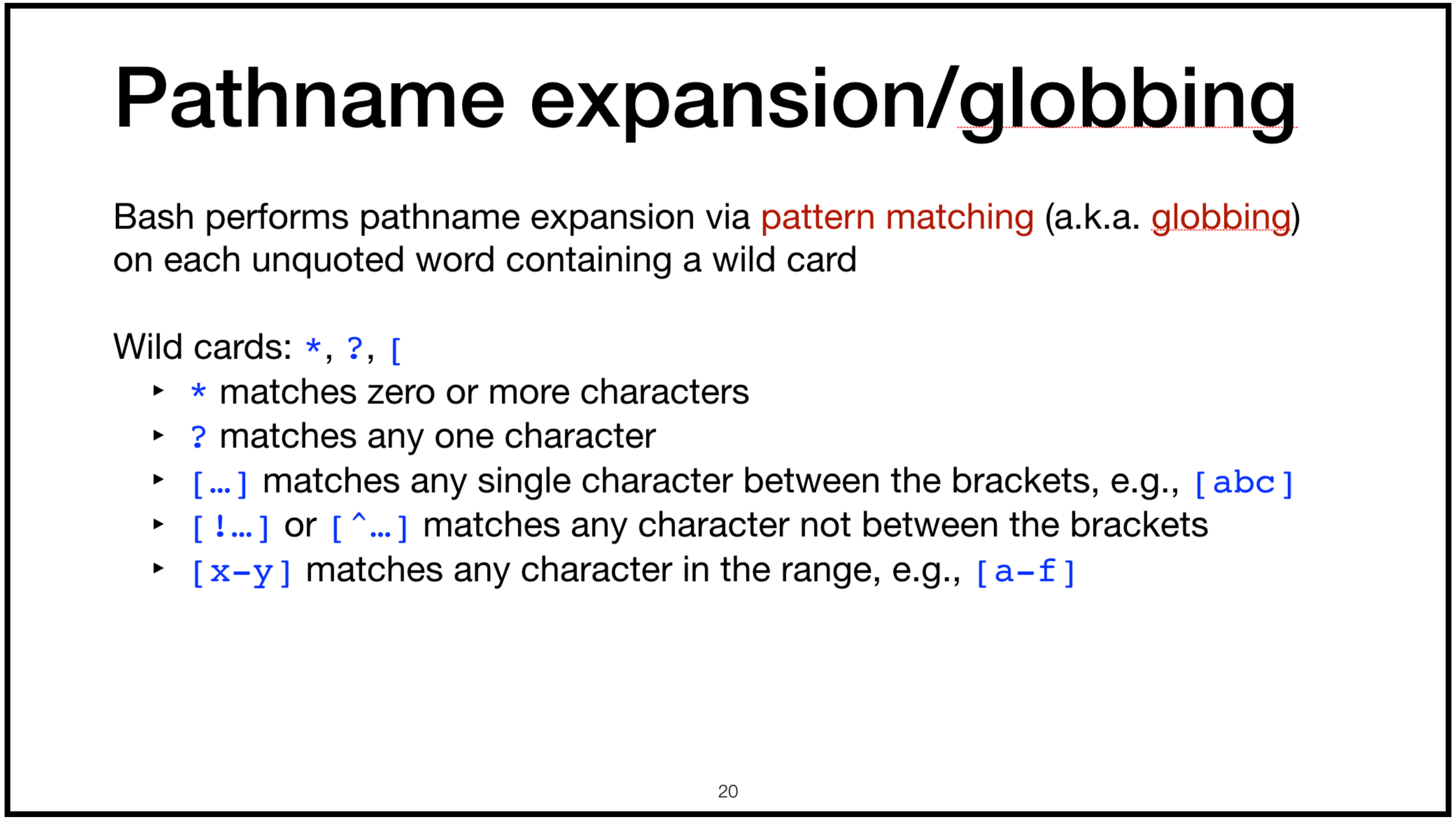
  parameter/variable expansion ${…},
  command substitution $(…), and
  arithmetic expansion $((…))

not in double quotes is split into words by splitting on (by default) space, tab, and newline

You never want word splitting! If you're using a $, put it in double quotes!

Pathname expansion

We saw this previously!



Pathname expansion/globbing

Bash performs pathname expansion via pattern matching (a.k.a. globbing) on each unquoted word containing a wild card

Wild cards: `*`, `?`, `[`

- `*` matches zero or more characters
- `?` matches any one character
- `[…]` matches any single character between the brackets, e.g., `[abc]`
- `[!…]` or `[^…]` matches any character not between the brackets
- `[x-y]` matches any character in the range, e.g., `[a-f]`

20

Quote removal

Unquoted ', ", and \ characters are removed in the final step

- `'foo  bar'` → `foo  bar (one word)`
- `"foo  bar"` → `foo  bar (one word)`
- `"${class}"` → `CS 241 (one word)`
- `"${class} is"' fun'` → `CS 241 is fun (one word)`

```
Upshot of quote removal:
$ program foo\ bar
$ program 'foo bar'
$ program "foo bar"
```

Program's first command line argument is `foo bar` with no quotes for all 3

Expansion summary

Braces form separate words \[{a,b,c}\] → [a] [b] [c]

Tildes give you home directories ~ → /home/mhogan

Variables expand to their values `"${class}"` → `"CS 241"`

Commands expand to their output `"$(ls *.txt | wc -l)"` → `"3"`

Wildcards expand to matching file names `*.txt` → `a.txt b.txt c.txt`

Put literal strings in `'single quotes'`

Put strings with variables/commands in `"${double} $(quotes)"`

If we have set a variable
`books='Good books'`
and we want to create a directory with that name, which command should we use?

A. `$ mkdir "${books}"`

B. `$ mkdir "$(books)"`

C. `$ mkdir ${books}`

D. `$ mkdir $(books)`

E. `$ mkdir $books`

Wed Preview - Version Control

- Creating git repositories
- Git commits
- Pulling from remote repositories