COMP 161 - Lecture Notes - 03 - Shells, part 2

Spring 2014

In these notes we'll dig deeper into the CLI and see just how working with *bash* gets us ready for the bigger changes in mindset we need to work with C++.

Paths

Paths tell you where in the file system a file or folder can be found and the come in two flavors: *relative* and *absolute*. Understand the differences between these path variants and how to use, or spot, one vs the other is pretty important.

Absolute Paths

Absolute paths always begin from the beginning, *root* or /. For that reason they're easy to spot:

> When a path begins with /, then it's an absolute path.

For example, everyone has a home directory on the system and all the home directories are found within the *home* directory. The home director is in turn, housed within /. So the absolute path to the home director for user *jdoe* is:

the ending / on the path is optional. I like it because it makes the fact that we're explicitly dealing with a directory

```
/home/jdoe/
```

Absolute paths are great because they unambiguously specify a file and folder on the system. Dr. James Logan Mayfield, IV is my full, absolute, name. You're unlikely to get me confused with anyone else if you use it. On the other hand, they are often long, unwieldy to type[1], and require some big picture understanding of the file system's organizational structure.

[1] TAB autocomplete solves this problem!

Relative Paths

Relative paths specify a path relative to your current working director. Put another way, the absolute path to your working directory is an assumed prefix to the absolute path of file or folder in question. There are a few ways to recognize and write relative paths. The first is, in my feeling, more explicit and therefore less prone to ambiguity. The path to your current working directory can always be invoked with the shortcut **./**[2]. This leads to your first indicator of a relative path:

[2] read ./ as "here"

> When a path begins with **./**, then it's a relative path.

So if my current directory is */home/* then we can form the relative path to jdoe's home directory like this:

```
./jdoe/
```

It turns out that the ./ is optional and this leads to the other typical way of picking out a relative path.

> When something shows up where a file/folder path specification is supposed to be and theres no leading /, then it's a relative path.

If we are once again in */home/* then *jdoe/* is a valid relative path to jdoe's home directory. In general, I like using ./ because it's clear you're providing a path. Leaving off ./ leaves it to the reader to decide the thing they're about to read is a path[3].

[3] thankfully when the reader is the OS, you tend not to have problems

Shortcuts

In addition to ./, there are a two path shortcuts you should memorize.

1. **../**

   This shortcut always refers to the parent of ./. If you're in your home directory, then ../ is */home/*. If you're in */home/*, then ../ is /. The odd directory out is /. It has no parent, so the system treats it as its own parent. That means that relative to root, ../ is still root. So, when you're not in /, *cd ../* is like hitting the up button[4] in your GUI.

[4] note necessarily the back button

2.

   This shortcut is your home directory. So *cd* is the command to "go home".

It's worth noting that adding *-a* to *ls* adds **.**[5] and **..**[6] to the list of directory contents.

[5] here
[6] parent of here

Racket Functions and CLI commands

Let's begin by leveraging what we know, Racket Functions, in order to better understand the world of CLI commands. What do we know about Racket functions:

- They use *prefix* notation in which the operator[7] comes before the operands[8].
- Function invocations are surrounded by parenthesis.
- Racket functions have one or more parameters and the number of parameters for a given function is fixed. Additionally, the order in which you pass parameters matters.

[7] command name, function name, etc
[8] arguments/inputs

- Racket functions take data values as input and return them as output, always. Given the same input, a Racket function will always produce the same output.

The question we now ask is, in what ways are CLI commands similar to and different from Racket functions?

Some highly utilized commands

To ground our inquiry, lets look at some highly utilized CLI commands.

1. *cd directory*

   The change directory command clearly highlights that, like Racket functions, CLI commands utilize prefix notation, but the do so without parenthesis. The big change here is that *cd produces not output*. Typically, the prompt changes, but there's not apparent output form cd, instead it seems to have an **effect**. That being said, if we think in terms of absolute paths, it always[9] produces the same effect for its inputs.

2. *ls*, *ls -l*, and *ls -la*

   The command *ls* shows us that CLI commands can accept zero arguments and optional arguments. What you probably don't know is that *ls -la* and *ls -al* are both allowable and equivalent. This means the order in which we chain together short optional arguments seems to not matter. The biggie here though is that in terms of the actual files and folders listed, the output produced by *ls* is independent of the inputs! Instead, something else drives the behavior of this command. That something else is the current **state** of the system, namely your current working directory.

3. *rm -v file* and *rm file -v*

   The *-v*[10] argument forces *rm* to output a record of what it deletes. What you don't know is that you can usually put optional arguments after required arguments. Yet another blow to *order of inputs matters.*[11]

So while there are some similarities to Racket functions, the rules for CLI commands seem to be pretty loose in comparison. The really big changes though are the introduction of notions of STATE and EFFECT.

[9] unless the file system has changed!

[10] verbose mode

[11] the preferred style here is *cmd options arguments*, so options first. stick to it!

Variables

State is a big term in computing. You're probably more familiar with it as a geographic term, i.e. the 50 states of the US. Let's build off this. What would you expect to happen if you walked up to a police officer with a half ounce of marijuana? Well, in Colorado, I'd expect very little to happen. Here in Illinois, I'd expect to get arrested. Why? Well in Colorado it's now[12] legal to carry up to one ounce on your person. In Illinois, possession of any amount is illegal. What we're seeing is that your surroundings, the state you're currently occupying and your **environment**, can cause different actions to have different effects.

[12] as of 2014

In computing, state typically refers to a VARIABLE[13]. This is not the variable as you know it from Racket or Math. A STATE VARIABLE is, for the most part, a named abstract representation of a piece of memory; it's a *named* location where we save some information about the state of the system[14]. The value that we store in the the variable are changed by operations we call MUTATORS. On the other hand, if an operation just inspects the contents/value in the variable, then we call it an ACCESSOR. There's a third kind of operation that will be more important to our programming than to our work at the CLI. An INITIALIZER operation is used to assign initial values to a variable. This is really important stuff; let's recap:

[13] aka STATE VARIABLE

[14] or program!

> A VARIABLE is memory with an associated name. It stores information about the current STATE of the computer or program. They are first assigned values by an INITIALIZER operation. Subsequently, their values may be changed by MUTATOR operations or retrieved by various ACCESSOR operations.

The ENVIRONMENT is the collection of STATE VARIABLES under which your commands are currently executing. Type the following command:

```
printenv | less
```

What you see is your current ENVIRONMENT, a list of all your VARIABLES and their current values. There are a lot of variables there, but you should see *PWD* on the list somewhere[15] Let's talk about this variable and some of the operations related to it.

[15] *printenv | grep "PWD"* will get all the lines with PWD in it.

First, let's look at *pwd*. This command takes zero inputs but outputs the absolute path of your current working directly. Maybe a better way to say this is, *it's an* ACCESSOR *for the PWD variable*. Now consider *cd*. The basic form of this command takes in a single input, a path, and produces no output. Instead, it has an EFFECT; its mutates the value of the *PWD* to the path argument, i.e. *cd is a* MUTATOR *for the variable PWD*. To round things out, notice that when you login, you're always in your home directory regardless of where you were when you last logged out. The system clearly runs an INITIALIZER on PWD that sets PWD to your home directory.

This is our new reality:

1. Systems[16] have VARIABLES which capture their current state. Operations execute in the ENVIRONMENT defined by this state.
2. VARIABLES require us to think not just in terms of input and output but EFFECT.
3. Operations involving variables can be categorized as INITIALIZERS, MUTATORS, or *accessors*.

[16] programs

I/O

When we talk about the output of a command on the CLI, we really talking about something different than the output of a Racket function. This distinction can be tough to navigate at first but we'll get a lot of practice and it's much easier to manage in C++ than with bash.

Let's return to *pwd*. Previously, we described something like this:

```
pwd
purpose: determine the current working directory
input: none
output: the value stored by PWD
effect: none
```

This isn't technically accurate. Instead we should document *pwd* as follows.

```
purpose: determine the current working directory
input: none
output: none
effect: write the value of PWD to the standard output
```

You see the output produced by *pwd* is really the result of a new *effect*. When we introduced variables we had to allow for a change of value effect[17]. Our new effect is causes change to what we see on the STANDARD OUTPUT[18]. What we've really discovered a specific example of I/O[19].

[17] mutation

[18] aka stdout, command-line out

[19] input/output

When we talk about I/O we're talking about the READ and WRITE effect taking place on a device attached to the system. With the CLI, there are three big places where I/O takes place:

1. STDOUT & STDIN I/O on the standard input output device, the command-line itself[20]
2. FILES input or output to a file[21]
3. STDERR the place that error messages are typically read from and written to

[20] keyboard and monitor

[21] hard drive

The STDERR is the odd one. There isn't really a piece of hardware associated with it that we can think about. Instead we have to imagine it as an agreed upon CHANNEL where communication about errors can occur between processes on the system. In practice, writing to STDERR typically results in output on the screen as systems users typically need to be made aware of errors.

Redirects and Expansion

Now that we know about I/O devices and effects, we can look at bash redirection and expansion in a new light.

1. | redirects a *write* effect to STDOUT to a READ effect from STDIN
2. $>$ redirects a WRITE effect from STDOUT to a WRITE effect to aFILE
3. $>>$ like $>$ but with a different spin or writing (append vs overwrite)

The $<$ redirect is a bit different as you're not really redirecting an effect as much as you're causing one. That is, the name of a file is not an implicit command to write to stdout, and so using $<$ is probably best though of as a compound effect: READ from FILE and WRITE to *stdin*.

> Redirects allow us to associate I/O EFFECTS with an I/O DEVICE other than the one for which it was originally intended.

This is all probably a better description of the redirection we've seen, not all possibilities or bash redirection

Now what about all the expansion stuff? In general, they're special rules for how to interpret some special forms and assign a value to them. The key here is it's about the VALUE of the expression being expanded. For example, for user jdoe,   has the value as */home/jdoe/* and so the two can be used interchangeably. In the case of wildcard and brace expansion, the expansions allow us write a pattern that will be expanded in to all things things[22] that match that pattern.

[22] often paths

Parameter expansion and command expansion are, for us, significant. *They let us recapture the functional input and output we know from Racket.* In Racket we'd write things like $(f(g5))$ and expect the VALUE output by $(g5)$ to be fed for f as an input. To get the same thing in bash, we use command expansion:

```
f $(g 5)
```

Now stop. How is that different than this:

```
g 5 | f
```

The difference is very important to understand and can be a first tricky to navigate.

The bash command $f\$(g5)$ is not a redirection. It takes the value of the output of $g5$ and uses it as the input to f. On the other hand,

$g5|f$ takes what $g5$ writes to STDOUT and instead causes f to read it from STDIN. The result in this case might be the exact same thing, but the first route avoids notions of I/O and instead uses *functional computing*.

Now parameter expansion. Try this:

```
echo PWD
```

What you should see is *PWD*. You might have expected to see the same thing as the command *pwd*, why? In Racket, feeding a variable to a function meant "use the value associated with name". The PWD variable is a different beast and so we have to be more specific. Try this:

```
echo $PWD
```

Now, we see the same thing as *pwd* because the parameter expansion invoked by $ effectively retrieves the value associated with the variable *PWD*.

The key with all of this is that we've returned to computation with functions and can there fore compose and nest a command like we did with Racket functions. The alternative is to chain together effects through redirects.[23].

[23] This is subtle and very very important. Give it serious thought

> Parameter and command expansion allow us to compute functionally and thereby avoid EFFECT-based command composition.

Big Picture

Something fundamental has changed. We now talk about things like actions involving hardware devices and modifying contents of the memory system. Much of this change is captured in the following terms:

- EFFECT
- VARIABLE
- STATE
- ENVIRONMENT
- ACCESSOR
- MUTATOR
- INITIALIZER
- I/O
- STDIN

- stdout
- file
- read
- write