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

January 9, 2015

In these notes we'll dig deeper into the CLI and see 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. They come in two flavors: *relative* and *absolute*. Understanding the differences between these path variants and how to use, or spot, one versus the other is an important part of life at the CLI.

Absolute Paths

Absolute paths always begin from the main directory of the file system, *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 directory is in turn, housed within /. So the absolute path to the home directory 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 or 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, that name, like absolute paths are often long and unwieldy. This makes them hard to type[1] and often requires some big picture understanding of the file system's overall organizational structure.

[1] TAB autocomplete solves this problem!

Relative Paths

Relative paths specify a path relative to your current working directory. Put another way, the absolute path to your working directory is an assumed prefix to the absolute path of the 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 there's no leading /, then it's a relative path.

If we are once again working out of */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 personal 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] not 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

The Power of Paths

Because you can specify paths as parts of commands, it is possible to run commands on files that are not in your current working directory. Compare this to working in the GUI. You typically have to navigate to the folder containing the file you wish to work with, and then select that file. In a CLI environment you can run a command on any file on the system from any directory on the system[7]. This kind of capability is necessary for copying files. If you want to copy file *a* to your current working directory and *a* is in your current directory's parent directory then you can run

[7] assuming you have the proper file permissions

```
cp ../a ./
```

. What you might not realize is that you can also do things like this

```
cp ../a ./sub/folder/over/here/
```

. Notice that neither the target file nor the destination is the current working directory!

If you're going to repeatedly work on a one or more files in a specific directory, then it makes sense to be in that directory. If, however, you're running a one off command involving files not in your working directory, then resist the urge to first change your working directory before running the command. When you tap into the power of paths, you can save a lot of time at the CLI.

Racket Functions and CLI commands

What do we know about Racket functions:

- They use *prefix* notation in which the operator[8] comes before the operands[9] and everything is separated by white space.
- 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.
- 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.

[8] command name, function name, etc

[9] arguments/inputs

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

Here's what we'll learn:

- Bash commands also use prefix notation and white space to separate the name and operands
- Bash commands are not surrounded by parenthesis
- Bash commands can have zero or more parameters and many commands have optional parameters. This means that one command can take a variable number of parameters
- Bash commands don't always produce output at the CLI. Sometimes they produce a *side-effect* on the system that we can't see unless we look for it. Some commands will produce different results on the same input, or they have different behaviors based on the *state* of the system.

Bash examples

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 they do so without parenthesis. The big change here is that *cd produces no CLI output*. Typically, the prompt changes, but there's no apparent output from cd. Instead the *cd* command is run for its **effect**, namely changing the current working directory. If we run *cd* with absolute paths, then it always[10] produces the same effect for its inputs. From that perspective its effects are as predictable as a Racket function's output.

   [10] unless the file system has changed!

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

   Here we see three variations of *ls* with zero, one, and two arguments respectively. What you probably don't know is that *ls -la* and *ls -al* are both allowable and equivalent. So not only can we have a variable number of arguments, but order doesn't necessarily matter! You probably didn't have this kind of flexibility in your Racket functions. You could write functions that would allow this, but it takes a lot of extra work. The real biggie here is that the output of each of the commands, in terms of the actual files and folders listed, is independent of the inputs! Instead, something else drives the code behavior of this command. That something else is the current **state** of the system, namely your current working directory. Different working directories will cause *ls* to produce different outputs despite the fact that the command itself is exactly the same.

So while there are some similarities to Racket functions, the rules for CLI commands seem to be pretty loose in comparison. That can be good and bad. Flexibility is good, but can also give you enough rope to hang yourself with; it's easy to learn just one way of doing something but limiting to have just that one method. The really big changes though are the introduction of STATE and EFFECT to our computing world.

State and Variables

STATE is a big deal 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[11] 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**[12], can cause different actions to have different effects.

[11] as of 2014

[12] collective state

In computing, state typically refers to a VARIABLE[13]. This is not the variable as you know it from Racket or Mathematics. 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].

[13] aka STATE VARIABLE

[14] or program!

Functions and commands that act on variables are often used for effect. In the CLI world, commands often act on one or more variables implicitly. So while it seems the command takes no arguments, we could also imagine a world where we passed in the implicit variable as an argument[15]. For example, the *ls* command is effectively called implicitly on a state variable with explicit CLI arguments. When the value of that state variable is different, then *ls* produces different behavior.

[15] You'll see this in C++

When thinking about functions that act on state, we can categorize them into three groups. Functions that change the value of a variable are called MUTATORS. The *cd* command is clearly a mutator for the current working directory state. Functions that inspect the contents/value in the variable but leave them unchanged are called ACCESSORS. The *ls* command fits into the category. Finally, when variables are first created we use INITIALIZERS to set their initial value and introduce them to the system environment. We haven't seen an initializer, but we imagine the operating system uses one when we first log in to set our current working directory to our personal home directory.

This is all really important stuff; let's recap:

> A VARIABLE is named memory. It stores information about the current STATE of the computing system 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 presence and usage of initializers, accessors, and mutators is not always obvious or explicit. As programers we must be mindful of when we're dealing with state and what kind of operations we're trying to carryout on that state.

An ENVIRONMENT is a collection of STATE VARIABLES. When working at the CLI you are working in an environment that is a combination of personal state variables and system side state variables. To see the contents of your environment, your state variables and

their associated values, type the following command:

```
printenv | less
```

There are a lot of variables there. Look for the variable *PWD* on the list somewhere[16] It is the variable that stores your current working directory path; the variable upon which *ls* and *cd* act. We can add to that list the command *pwd*, which is an acceessor which retrieves and outputs the contents of *PWD*.

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

Our new, stateful world

This is our new reality:

1. Systems[17] have VARIABLES which capture their current state. Operations execute in the ENVIRONMENT defined by this state. Some operations act upon or according to those variables and some do not.
2. VARIABLES require us to think not just in terms of basic input and output but EFFECT. Some operators change state and some behave differently for different state values.
3. Operations involving variables can be categorized as INITIALIZERS, MUTATORS, or *accessors* based on how they interact with state.

[17] 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[18].

[18] it's explicit in C++

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 kind of *effect*. When we introduced variables we had to allow for a change of value effect[19]. Our new effect is to send data to the system's input/output (I/O) devices.

[19] mutation

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 that takes place on the default devices, the command-line itself[20]
2. FILES input or output to or from 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 device is the odd one. There isn't really a piece of hardware associated with it. 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

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

1. | redirects text *write*n on STDOUT so that it is READ from STDIN
2. > redirects text WRITEn on STDOUT so that it is WRITEn to aFILE instead
3. >> like > but with a variation on the WRITE effect (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*.

This is really just a description of the < redirection uses we've seen. Others exist.

We can now restate the nature of redirects in our new vocabulary.

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

By using redirects we can start to build COMPOUND OPERATIONS. Just like we can nest function calls in Racket so that the output of one function is the input to another, we can redirect the output of one Bash command to another. Hello programming.

Expansions

Expansions tap into the core ideas of functional input and output in that they allow you to substitute one value for another within a particular command just like we can substitute the return value of a function for it's function call when evaluating a Racket expression. The key here is that we're thinking about VALUE SUBSTITUTION as opposed to some kind of effect redirection.

Shortcuts represent a basic form of expansion. In some cases they're as simple as named values. For example, for the user jdoe, the expression has the value of */home/jdoe/* and we can pretty much use the former anywhere we want the later. The short cut for here, *./*, and parent, *../*, are a bit more complex as their value is dependent on the PWD state variable.

One of the most useful expansions is wildcard, * expansion. Here the expressions expands to all values which match a specific pattern. The pattern *./*.pdf* expands to the path to all of the pdfs in the current directory. If you have 32 pdfs in your current directory, then you get 32 paths! This pays off huge for things like copying. The command

```
cp *.pdf foo/
```

will copy every pdf in the current directory to the sub-directory foo. Wildcards are so powerful that you should be careful using them at first. They can cause commands to do way more than you expected if you and the computer don't interpret the pattern in the same way. Using wildcards along with *rm* is a really good way to accidentally erase some files.

Brace expansion, like wildcard expansion, expands to everything that matches the pattern. The pattern {*lab1,lab2*}*.cpp* will expand to *lab1.cpp lab2.cpp*. Combine this with wildcards and you can create some pretty powerful patterns. For example, *.{*cpp,h*} expands to all the cpp and h files in the current directory. This kind of pattern is likely to pop up a lot when we start C++ programming as these two file types are used in C++ programs.

Parameter expansion and command expansion are, given our background, significant CLI tools. *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 and think, How is that different than this command?

```
g 5 | f
```

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 seemingly avoids notions of I/O and instead uses *functional computing*.

Basic parameter expansion let's us easily substitute the value of a variable for its name. Put another way, it's an accessor shortcut. 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*.

All of these expansions let you recapture some of the functional feel of programming in Racket. The alternative is to chain together effects through redirects.[22]. More practically, they're the gateway to some serious commandline-fu.

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

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

- READ
- WRITE

Go back over these notes if you're not sure what they mean and how they show up when working at the CLI.