Joshua Camacho

Professor Raheja

CS 260

7/25/17

# Bash vs Perl Scripting

For this report I will be comparing and contrasting scripting in Bourne Again Shell and using the Perl scripting language. The script used in both Bourne Again Shell and with Perl was a simple record lookup program. Upon runtime, a menu was displayed and provided a few utilities for manipulating a record file located in the same directory as the script. These utilities include displaying records in alphabetical order by first name or last name, displaying the records in reverse alphabetical order by first name or last name, searching for a record by the last name, and searching for a record by the birthday.

I'll begin my comparison by looking at the line count of each script. The Bourne Again Shell script came in at 106 lines and the Perl script was 61 lines. A big portion of the lines in the bash script came from the switch statements. Weirdly enough, Perl did not have a switch statement, which made menu logic a lot uglier. You can handle a menu decently enough with if(EXPR) {} elsif (EXPR) {} else {} but switch statements are so much nicer for handling such cases. Fall through and conditional switches allows you to easily add casing insensitivity, example case "a" | "b": or case a: case b: without a break in-between. Apparently there is a switch package for pearl which adds switch statements, but I didn't bother trying to install it. It should really come by default instead of having to download a third party library which may or may not have support in the future and then you also have to worry about compatibility on other machines. Perl might have been much shorter than bash, but bash menus were much more human readable and understandable.

I had a hard time figuring out how to make sure that a filename was passed in on command line in Perl. At first I tried if(undef($1)) but that was giving me unexpected results. I think that undef is supposed to return 0 if the argument passed in is undefined, which is backwards logic. I then tried using if(!(-f $1)) but that was throwing me an error that I was trying to edit a file without permissions. It was at that point that I realized that -f does not check if the file exists, like it does in every shell scripting. Turns out you use -e in order to accomplish that in Perl. I finally landed with using if (!$1) because an empty string will result in a 0. Needless to say this was a headache to figure out, not sure why they would choose a syntax so close to shell scripts, but arbitrarily choose a different letter to be able to check if a file exists.

Some of the syntax of Perl I liked. Coming from programming languages like java and C, I was used to the {} notation of scoping control and loop structures. I did have issues with semi

colons. I t I would constantly have that mistake in my code. With writing all these shell scripts, I got into the habit of omitting a semi colon.

I'll now take the time to explain how much I dislike the default variable $_ and the default array @_ in perl. Ambiguity is not good in a programming language. Code, especially high level code, should be human readable, and shouldn't have pointless learning curves like having to understand that there are hidden, implied variables. What might save you 10 seconds writing the code, will cause you, and others that read your code, confusion and headache. Pseudo code with perl-like syntax would not be transferrable to other languages if you used the default variable $_ or the default array @_. Using this invisible variable is a bad idea and promotes non-sharable, nonsensical code.

Now I'll discuss bash. Comparing bash with perl, Bash has more difficult syntax with control structures and loop structures. Even with the upgrade from sh using structures like if [ $1 -lt 5 ], the new bash way if (( $3 < $5 )) still has its quirks. The specific need to have spacing on either side feels like an unnecessary burden to pay attention to. Perl was much more lax with whitespace and had easier control structure notation altogether.

Bash really stood out anytime you had to use a Unix utility, as it had direct access to it. Perl required a clunky call to the System function, and required fiddling around with quotes, double and single, as well as trying to work in variable names into the System function. This took. Dealing with files was much easier with bash as well. Perl required an explicit call to open up a file, but bash could just access the files directly because it was using Unix utilities.

Perl did handle lists better than Bash. My favorite feature of perl is how easy it is to access and set lists. @list[0..10] quickly accesses the first 11 values in the list, and can be combined with a foreach $item @list[0…10] {} for easy iteration through a list of values. Values could also flexibly be integers, floating point numbers, or strings which is a nice feature that most higher level languages have. @list[0…$#list] is an easy way to iterate over the whole array, without explicitly knowing the length. Having quick access to string concatentation in Perl was really nice, along with the quick method of doing so too, $string1 .= $string2 allows you to quickly add string2 to string 1.

The repetition operator, "x", is a neat feature of Perl that I haven't come across in any programming language yet. For example you can quickly make a line by doing something like this $line = '-' x 80 which will insert into $line 80 '-'s, effectively forming a dotted line to be used for the terminal. It's neat, and I'm sure it comes in handy for a select parts of scripts, anytime you have to format something for the terminal in a specific way. It's not a huge core feature of Perl, but I appreciate this small little time saver which still keeps the code human readable.

Overall, while Perl offers some nice syntax, built in functions, and some easy handling of lists, I think I still prefer bash for handling small basic scripts. Perl might be nicer dealing with something slightly larger than basic scripts, but I'm not sure if I'm sold on it because it lacks

switch statements. Bash might have slightly worse control syntax, but being able to easily access Unix utilities allows for a much easier time trying to get stuff done.