The Shell & Scripting

CSC 510

Shells

- A shell is a command line interface (CLI) to the operating system
- Common Unix shells
  - Bourne – `sh`
  - Korn – `ksh`
  - C shell – `csh`
  - Debian Almquist Shell – `dash`
  - TC shell – `tcsh`
  - Z shell – `zsh`
- Here we will focus on the `bash` shell, which is one of the most widely used shells. `bash` syntax is similar to many other shells.

Command Editing

- Some shortcuts (by default emacs based):
    - `^E` – go to the end of the line
    - `^A` – go to the beginning of the line
    - `^P` – steps back through previous command history
    - `^R` – reverse search command history
- Note `^E` (also `Ctr-E` or `<C-E>`) indicates control + the 'e' key.
- If you like `vi` editing, you can set the shell to vi mode with

```
set -o vi
```

History Expansion

- `!!` – entire last command
- `!:^` – first argument from last command
- `!:$` – last argument from last command
- `!:n` – nth argument from last command
- `!:` – all arguments from last command
- `!:x-y` – arguments in a given range from last command
- `!<prefix>` – most recent previous command starting with `<prefix>`

Tab Completion

- Pressing the `[TAB]` key while typing a command will make bash inspect the input to find a relevant completion specification (`compspec`), for example filenames in the current working directory
- You can create custom `compspecs` with the `complete` builtin
- Example

```
$ complete -W "foo bar baz" echo
$ echo [TAB][TAB]
bar baz foo
$ echo f[TAB]
foo
```

Connecting Programs

- `>` – stdout redirection
- `>>` – stdout redirection, append
- `2>` – stderr redirection
- `>&` – redirect both stdout and stderr
- `<` – stdin redirection
- `|` – pipe; connect stdou of one program to stdin of another program
- `$( CMD )` – command substitution
- `<( CMD )` – process substitution

Variables and Quoting

- Variable assignment uses the = operator
- Variable reference uses $ prefixed to the variable name
- Strings can be single or double quoted, but have different semantics
    - single quotes: string literals
    - double quotes: interpolates variable values
- Example (note that # begins a line comment)

```
foo=bar
echo "$foo"  # prints bar
echo '$foo'  # prints $foo (literally)
```

Globbing

- Globbing refers to filename expansion with special characters
- Wildcards – `?` matches a single character and `*` matches zero or more characters in a filename
- Brace expansion – takes an optional preamble, a series of comma separated strings in braces, and an optional postscript and generates a sequence of new strings for each string in the braces
    - Example

```
$ echo a{b,c,d}e
abe ace ade
```

Environment Variables

- When a UNIX process starts, it receives a set of command line arguments and a set of “environment variables”
- The `printenv` command will display the currently set environment variables
- By convention environment variables are all written in all caps
- A variable can be promoted to an environment variable with the `export` keyword
- Example: make programs that use the `EDITOR` environment variable use `nano`

```
export EDITOR=nano
```

Scripting

- Shell commands can be written in a file and executed like a program
- The first line in a shell script should be the shebang which indicates which interpreter to run the script through (in our case bash)

  ```
  #!/bin/bash
  ```

- The script must also have execute permissions set

Special Variables

- Bash uses special variables to refer to arguments, error codes, etc.
- Some examples:
  - `$0` – name of the script
  - `$1` to `$9` – arguments to the script
  - `$@` – all the arguments
  - `$#` – number of arguments
  - `$?` – return code of previous command
  - `$$` – process id (PID) of the current script

Control Flow

- Bash has selection and iteration constructs (which rely on Boolean values)
- The shell uses the exit codes of programs for Boolean tests: an exit code of zero is true and non-zero is false
- The shell has short circuit operators for `||` (or) and `&&` (and)
- The semicolon is a sequencing operator

  Example:

  ```
  false || echo 'hello' # prints hello
  true || echo 'hello'  # nothing gets printed
  true && echo 'hello'  # prints hello
  false && echo 'hello' # nothing gets printed
  true ; echo 'hello'   # prints hello
  false ; echo 'hello'  # prints hello
  ```

Comparison Operators

String	Numeric	True when
x = y	x -eq y	x is equal to y
x != y	x -neq y	x is not equal to y
x < y	x -lt y	x is less than y
	x -leq y	x is less than or equal to y
x > y	x -gt y	x is greater than y
	x -geq y	x is greater than or equal to y
-n x		x is not null
-z x		x is null

Note: the < and > operators need to be backslash escaped or double bracketed to prevent interpretation of file redirection

File Operators

Operator	True when
-d file	file exists and is a directory
-e file	file exists
-f file	file exists and is a regular file
-r file	user has read permission on file
-s file	file exists and is not empty
-w file	user has write permission on file
file1 -nt file2	file1 is newer than file2
file1 -ot file2	file1 is older than file2

if ... else

- Syntax

  ```
  if [ <condition> ] then
      <commands>
  elif [ <condition> ]; then
      <commands>
  else
      <commands>
  fi
  ```

- Note the bracket syntax for conditions must have whitespace separating the brackets and the condition. The brackets are an alternative syntax to the `test` command

`for ... in`

- Syntax

  ```
  for <variable name> in <sequence>
  do
    <commands>
  done
  ```

- `<sequence>` can be a file expansion, a range with the syntax `{<start>..<stop>..<step>}` where `<step>` is optional, or an array.

`for` loop

- Syntax

```
for ((<initialization>; <condition>; <step>))
do
  <commands>
done
```

while loop

- Syntax:

```
while [ <condition> ]
do
  <commands>
done
```

`break` and `continue`

- Looping constructs support the `break` and `continue` keywords
- `break` – exit out of the loop
- `continue` – restart the loop body with the next iteration

Functions

- Function definition syntax

```
<function name> () {
    <commands>
    [return <8-bit integer>]
}
```

  - Note function arguments are accessed via the special variables, for example, $1 to $9.

- Function call syntax

```
<function name>
```

ShellCheck

- ShellCheck is a linter for shell scripts
- A linter is a program the performs static analysis of script files in order to find potential errors
- You should use ShellCheck when writing your scripts

Job Control

- `^Z` – suspend the currently running process
- `bg` – move a suspended process to the background
- `fg [n]` – move a process to the foreground by job number
- `jobs` – list current jobs

Aliases

- A shell alias is a custom (usually) shorter form for another command
- Syntax

  ```
  alias <alias name>="<command> [args...]"
  ```

- Example:

  ```
  alias ll="ls -lh"
  ```

Dotfiles

- Dotfiles are configuration files for programs
- Bash related dotfiles
  - `~/.bashrc`
  - `~/.bash_profile`

Regular Expressions

- Many commands and configure files accept regular expressions.
- A regular expression (regex) are a pattern matching mechanism for text (strings)
- You can think of a regular expression as a function that takes a text and a pattern and returns a Boolean value.

Common Special Characters in Regex

Symbol	Meaning
`.`	match any character
[*chars*]	match any character in the given set
[^*chars*]	match any character not in the given set
`^`	match the beginning of the line
`$`	match the end of the line
`\w`	match any “word” `[A-Za-z0-9_]`
`\s`	match any whitespace character
`\d`	match any digit
|	match the element on the left or right (alternation)
(expr)	groups elements

Common Special Characters in Regex

Symbol	Meaning
`?`	match zero or one of preceding element
`*`	match zero or more of the preceding element
`+`	match exactly one of the preceding element
`{` *n* `}`	match exactly *n* instances of the preceding element
`{` *n*, `}`	match at least *n* instances of the preceding element
`{` *n*,*m* `}`	match any number of instances from `n` to `m`