CS 241: Systems Programming
Lecture 4. Environment and expansion

Fall 2019

Prof. Stephen Checkoway

Announcement

If you are not a CS major and you would like to be, please declare ASAP

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⟩⟨control operator⟩
- 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

A (second) 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

Bash variables

Setting and using variables in bash

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

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

‣ ```
$ 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

```
$ 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`

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 `~/local/bin`, you can modify your `PATH` to access it

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

This adds ~/local/bin to the front of the PATH so it is searched first

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

This adds `~/local/bin` to the end of the `PATH` so it is searched last

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 command`
which environment variables are available to `command`?

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

D. `Y` and `Z`

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

E. `Z`

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

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`

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

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
- `${HOME}` → `/usr/users/noquota/faculty/steve`
- `x${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

Word splitting

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!


```
steve@clyde:~$ x='foo     bar'
steve@clyde:~$ echo ${x}
foo bar
steve@clyde:~$ echo "${x}"
foo     bar
```

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)

Expansion summary

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

Tildes give you home directories ~ → /home/steve

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`

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

In-class exercise

https://checkoway.net/teaching/cs241/2019-fall/exercises/Lecture-04.html

Grab a laptop and a partner and try to get as much of that done as you can!