bash, part 2

Prof. Chris GauthierDickey

COMP 2400 - Unix Tools

UNIVERSITY OF DENVER

Quoting and Strings

- bash will interpret variables on their own
- We can also surround them by quotes: ' or ""
  - '$foo' is the literal string $foo, without interpretation
  - "$foo" will become the string that contains the value of $foo. The following prints 42:
    - foo=42; echo "$foo"

Integer comparisons in conditions

- Bash can distinguish between strings and integers, but we use special comparison operators
  - <, >, !=, =, are for strings
  - -lt, -gt, -eq, -ge, -ne, -le are for numbers
    - foo=5; if [$foo -lt 6]; then echo "less than 6"; fi

bash loops

- The first type of for-loop can only iterate through list elements
- If 'in list' isn't specified, bash will iterate through the arguments to the script or function
- By default, lists are separated by a blank space

```
for name [in list]
do
  statements using $name
done
```

```
IFS=:
for p in $PATH
do
  echo $p
done
```

UNIVERSITY OF DENVER

A word about lists

- Lists are separated by a blank space, as noted
- bash uses the first character of environment variable IFS to determine what separates lists
- We can set it temporarily as long as it isn't needed by something else

```
# this at the start of
# a script will print
# out the arguments
# separated by a comma
# note that $@ isn't
# affected by IFS
OLD_IFS=$IFS

IFS=,

echo $*

IFS=$OLD_IFS
```

Constructing a list

- Lists are really just strings separated by some element: typically a space
- We construct them using quoting

```
mylist=25
mylist="30 $mylist"
mylist="35 $mylist"

# the following will
# print 35 30 25
echo $mylist
```

More on lists

- We've seen how to add, but how do we remove from a list?
  - What's that first line doing?
- Note the space in the second line after *
  - Why is that needed?

```
# this at the start of
stack="$1 ${stack:-eos' '}

# we can remove from a
# list using pattern
# matching as follows:
stack=${stack#* }
```

UNIVERSITY OF DENVER

Patterns Matching

- ${var#pattern}
  - if pattern matches beginning, delete the shortest match and return the rest
    - Try p=$(pwd); echo ${p#/*/}
- ${var##pattern}
  - if pattern matches beginning, delete the longest match, and return the rest
    - Try p=$(pwd); echo ${p##/* /}

- ${var%pattern}
  - Matches the shortest part at the end of var, deletes it and returns it
    - p=$(pwd); echo ${p%/*}
- ${var%%pattern}
  - Matches the longest part at the end of var, deletes it and returns it
    - p=$(pwd); echo $(p%%/*}

UNIVERSITY OF DENVER

- ${var/pattern/str}
  - The longest match to pattern in var is replaced by str
    - p=$(pwd); echo ${p/home/myhome}

- ${var//pattern/str}
  - Replaces all occurrences of pattern in var with str
    - p=$(pwd); echo ${p//\//:}

A word on bash patterns

- bash patterns are **NOT** regular expressions:
  - ? matches zero or one characters
  - * matches any character
  - [ ] is a set (as with regexes), so [a-f] matches a through f
  - [!a-f] matches anything that is not a to f.
  - {1..4} expands to 1 2 3 4
    - try echo g{em,ift,oodie}s

More bash patterns

- You can expand the bash patterns by using:
  - shopt -s extglob, which gives you a bit more power
  - +(pattern) matches one or more copies of pattern
  - (pat1|pat2) matches pat1 or pat2
  - ?(pattern) gives you 0 or 1 of the pattern
  - *(pattern) gives you 0 or more of the pattern
  - @(pattern) gives exactly 1 match of the pattern
  - !(pattern) matches anything **NOT** the pattern

bash Arrays

- bash also has arrays with the following syntax:
  - foo[0]="hello"; foo[1]="world"
  - foo=(hello world)
  - foo=([1]=world [0]=hello)
  - foo=(hello [5]=world)
  - echo "${foo[5]}"
  - We can also use "${foo[@]}" and "${foo[*]}"

More on Arrays

- Wonder what indices are used?
  - echo "${!foo[@]}"
- How can we iterate through the array?
  - for i in "${foo[@]}"; do echo $i; done
  - ${#foo[5]} returns the length of element 5
  - ${#foo[@]} returns now many elements are in foo

UNIVERSITY OF DENVER

bash functions

- We define a bash function using the 'function' keyword

- Arguments to functions are accessed just like script arguments: $1 to $n, where n is an integer

```
function printargs
{
  echo "printargs: $*"
  echo "$0: $1 $2 $3 $4"
  echo "$# arguments"
}
```

functions...

- New variables in a script, outside a function are global to the script
- New variables in a function are global to the script
  - We can add 'local' before the declaration to keep them in function scope

```
function printargs
{
  local var1="hello"
  echo "printargs: $*"
  echo "$0: $1 $2 $3 $4"
  echo "$# arguments"
  echo "$var1"
}

echo "\$var1 is ${var1:-null}"
```

functions...

- We call functions just by using their name
- When we source them, they become global, like they've been exported
- We can use recursion if we'd like

```
function foo
{
  for i in "$@"; do
    echo "foo: $1"
  done
}

foo bar{1..5}
```

The bash case

- 'case' is like switch in other languages, but does pattern matching on the arguments

- patterns can be separated by the pipe '|'

```
function casecheck
{
  for i in "$@"; do
    case $i in
      hello )
        echo "hi!" ;;
      world )
        echo "goodbye!" ;;
      a | b | c )
        echo "x y z!" ;;
      * )
        echo "default" ;;
    esac
  done
}
```

while/until

- bash also has the two common loop constructs: while and until

- while may or may not execute, depending on the command or condition

- until always executes at least once

```
while condition
do
  statements ...
done
```

```
until condition
do
  statements ...
done
```

bash math

- Arithmetic can be done in bash using $(( )), which signifies an arithmetic expression

```
# the following
# echos a 4
v=$(( 1 + 6 / 2 ))
echo $v
```

- Old-school: expr was used
- We don't have to escape special characters or even use $ in front of variables (though it's not a bug to do so)

```
if [ $(((5+6)/11)) = 1 ]
  then
    echo "1"
fi
```

math...

- bash arithmetic can also use logicals: &&, ||, but the truth value is 1, not zero!

- We can declare a variable as an integer using declare -i var

- We can declare and assign using let as shown

```
# the following creates
# x and assigns 6 to it
let x=5+1; echo $x
```

math operators

- ++: increment by 1
- --: decrement by 1
- +: plus
- -: minus
- *: multiplication
- /: divide
- %: remainder
- **: exponentiation
- <<: bit-shift left
- >>: bit-shift right
- &: bitwise and
- |: bitwise or

- ~: bitwise not
- !: logical not
- ^: bitwise exclusive or
- ,: sequential evaluation
- <: less than
- >: greater than
- <=: less than or equal
- >=: greater than or equal
- ==: equal
- !=: not equal
- &&: logical and
- ||: logical or

loop arithmetic

- We can use (( )) for arithmetic in our loops, or test conditions with them in while and until loops

```
# for loop
for (( init ; end ; update ))
do
   statements
done
```

```
for (( i=1; i <= 5; i++ ))
do
  echo $i
done
```