CSE 374
Programming Concepts & Tools

Hal Perkins
Winter 2012
Lecture 4 – Shell Variables, More Shell Scripts
Where we are

- We understand most of the bash shell and its “programming language”. Final pieces we’ll consider:
  - Shell variables
    - Defining your own
    - Built-in meanings
    - Exporting
  - Arithmetic
  - For loops
- End with:
  - A long list of gotchas (some bash-specific; some common to shells)
  - Why long shell scripts are a bad idea, etc.
Shell variables

• We already know a shell has state: current working directory, streams, users, aliases, history.
• Its state also includes shell variables that hold strings.
  – Always strings even if they are “123” – but you can do math
• Features:
  – Change variables’ values: foo=blah
  – Add new variables: foo=blah or foo=
  – Use variable: ${foo}   (braces sometimes optional)
  – Remove variables: unset foo
  – See what variables “are set”: set
• Omitted feature: Functions and local variables   (see manual)
• Roughly “all variables are global (visible everywhere)”
• Only assignment is similar to mainstream “real” programming languages
Why variables?

• Variables are useful in scripts, just like in “normal” programming.
• “Special” variables affect shell operation. 3 most (?) common:
  – PATH
  – PS1
  – HOME
• Some variables make sense only when the shell is reading from a script:
  – $#, $n (where n is an integer), $@, $*, $?
Export

• If a shell runs another program (perhaps a bash script), does the other program “see the current variables that are set”?  
  – i.e., are the shell variables part of the initial environment of the new program?
• It depends.  
  – export foo – yes it will see value of foo  
  – export -n foo – no it will not see value of foo  
  – Default is no.
• If the other program sets an exported variable, does the outer shell see the change?
• No.  
  – Somewhat like “call by value” parameters in conventional languages  
  – Remember, each new program (and shell) is launched as a separate process with its own state, environment, etc.
Arithmetic

- Variables are strings, so \( k = i + j \) is not addition.
- But \((k = i + j)\) is (and in fact the \$\ is optional here).
- So is \( \text{let } k = "i + j" \).
- The shell converts the strings to numbers, silently using 0 as necessary.
For loops

- Syntax:
  ```
  for v in w_1 w_2 ... w_n
  do
    body
  done
  ```
- Execute body n times, with v set to $w_i$ on $i^{th}$ one. (Afterwards, v=$w_n$).
- Why so convenient?
  - Use a filename pattern after in
  - Use list of argument strings after in: "$@"
Quoting

• Does \texttt{x=*} set \texttt{x} to string-holding-asterisk or string-holding-all-filenames?
• If \texttt{$x} is \texttt{*}, does \texttt{ls $x} list all-files or file named asterisk?
• Are variables expanded in double-quotes? single-quotes?
• Could consult the manual, but honestly it’s easier to start a shell and experiment. For example:
  \begin{verbatim}
  x="*"
  echo x
  echo $x
  echo "$x"  \hspace{1em}  \text{(Double quotes suppress some substitutions)}
  echo '$_x' \hspace{1em}  \text{(Single quotes suppress all substitutions)}
  ...
  \end{verbatim}
Gotchas: A very partial list

1. Typo in variable name on left: create new variable
   oops=7
2. Typo in variable use: get empty string – ls $oops
3. Use same variable name again: clobber other use
   HISTFILE=uhoh
4. Spaces in variables: use double-quotes if you mean
   “one word”
5. Non-number used as number: end up with 0
6. set f=blah: apparently does nothing (assignment in
   csh)
7. Many, many more…
Shell programming revisited

• How do Java programming and shell programming compare?
• The shell:
  – “shorter”
  – convenient file-access, file-tests, program-execution, pipes
  – crazy quoting rules and syntax
  – also interactive
• Java:
  – none of the previous gotchas
  – local variables, modularity, typechecking, array-checking, . . .
  – real data structures, libraries, regular syntax
• Rough rule of thumb: Don’t write shell scripts over 200 lines?
Treatment of strings

• Suppose foo is a variable that holds the string hello

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Bash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use variable (get “hello”)</td>
<td>foo</td>
<td>$foo</td>
</tr>
<tr>
<td>The string foo</td>
<td>“foo”</td>
<td>foo</td>
</tr>
<tr>
<td>Assign variable</td>
<td>foo = hi</td>
<td>foo=hi</td>
</tr>
<tr>
<td>Concatenation</td>
<td>foo + “oo”</td>
<td>${foo}oo</td>
</tr>
<tr>
<td>Convert to number</td>
<td>library call</td>
<td>silent and implicit</td>
</tr>
</tbody>
</table>

• Moral: In Java, variable-uses are easier than string-constants.
• Opposite in Bash.
• Both biased toward common use.
More on shell programming

• Metapoint: Computer scientists automate and end up accidentally inventing (bad) programming languages. It’s like using a screwdriver as a pry bar.
• HW3 in part, will be near the limits of what seems reasonable to do with a shell script (and we’ll end up cutting corners as a result)
• There are plenty of attempts to get “the best of both worlds” in a scripting language: Perl, Python, Ruby, . . .
• Personal opinion: it raises the limit to 1000 or 10000 lines? Gets you hooked on short programs.
• Picking the bash shell was a conscious decision to emphasize the interactive side and see “how bad programming can get”.
• Next: Regular expressions, grep, sed, others.
Bottom line

• Never do something manually if writing a script would save you time.
• Never write a script if you need a large, robust piece of software.
• Some programming languages try to give the “best of both worlds” – you now have seen two extremes that don’t (Java and bash).