Lecture 7

Part I
Shell Scripting (continued)
Parsing and Quoting
Shell Quoting

• Quoting causes characters to lose special meaning.

• \ Unless quoted, \ causes next character to be quoted. In front of new-line causes lines to be joined.

• '...' Literal quotes. Cannot contain '

• "..." Removes special meaning of all characters except $, ", \ and \. The \ is only special before one of these characters and new-line.
Simple Commands

• A simple command consists of three types of tokens:
  – Assignments (must come first)
  – Command word tokens
  – Redirections: `redirection-op + word-op`
  – The first token must not be a reserved word
  – Command terminated by new-line or ;

• Example:
  – `foo=bar` `z=``date``
    `echo $HOME`
    `x=foobar > q$$ $xyz z=3`
Word Splitting

• After parameter expansion, command substitution, and arithmetic expansion, the characters that are generated as a result of these expansions that are not inside double quotes are checked for split characters.
• Default split character is space or tab.
• Split characters are defined by the value of the IFS variable (IFS="" disables).
Word Splitting Examples

FILES="file1 file2"
cat $FILES
  
a
  b

IFS=
cat $FILES
  cat: file1 file2: cannot open

IFS=x v=exit
echo exit $v "$v"
exit e it exit
Pathname Expansion

• **After** word splitting, each field that contains pattern characters is replaced by the pathnames that match

• Quoting prevents expansion

• `set -o noglob` disables
  – Not in original Bourne shell, but in POSIX
Parsing Example

\[ \text{DATE}=`\text{date}` \text{ echo } \$\text{foo} > \backslash \] 
\text{/dev/null}
The eval built-in

• `eval arg ...`
  – Causes all the tokenizing and expansions to be performed again
• **trap** specifies command that should be *eval*ed when the shell receives a signal of a particular value.

• **trap** `[ [command] {signal}+ ]`
  – If *command* is omitted, signals are ignored

• Especially useful for cleaning up temporary files

  ```
  trap 'echo "please, dont interrupt!"' SIGINT
  trap 'rm /tmp/tmpfile' EXIT
  ```
Reading Lines

- `read` is used to read a line from a file and to store the result into shell variables
  - `read -r` prevents special processing
  - Uses `IFS` to split into words
  - If no variable specified, uses `REPLY`

```
read
read -r NAME
read FIRSTNAME LASTNAME
```
Script Examples

• Rename files to lower case
• Strip CR from files
• Emit HTML for directory contents
#!/bin/sh

for file in *
do
    lfile=`echo $file | tr A-Z a-z`
    if [ $file != $lfile ]
    then
        mv $file $lfile
    fi
done
Remove DOS Carriage Returns

#!/bin/sh

TMPFILE=/tmp/file$$

if [ "$1" = "" ]
then
    tr -d \r
    exit 0
fi

trap 'rm -f $TMPFILE' 1 2 3 6 15

for file in "$@"
do
    if tr -d \r < $file > $TMPFILE
then
    mv $TMPFILE $file
fi
done
Generate HTML

$ dir2html.sh > dir.html

Directory listing for /home/jlk/nyu/scripts
The Script

#!/bin/sh

[ "$1" != "" ] && cd "$1"

cat <<HUP
<html>
<h1> Directory listing for $PWD </h1>
<table border=1>
<tr>
HUP
num=0
for file in *
do
do
genhtml $file    # this function is on next page
done
cat <<HUP
</tr>
</table>
</html>
HUP
Function genhtml

genhtml()
{
  file=$1
  echo "<td><tt>"
  if [ -f $file ]
  then    echo "<font color=blue>$file</font>"
  elif [ -d $file ]
  then    echo "<font color=red>$file</font>"
  else    echo "$file"
  fi
  echo "</tt></td>"
  num=`expr $num + 1`
  if [ $num -gt 4 ]
  then
    echo "</tr><tr>"
    num=0
  fi
}
Korn Shell / bash Features
Command Substitution

• Better syntax with $(command)
  – Allows nesting
  – x=$(cat $(generate_file_list))
• Backward compatible with `…` notation
Expressions

• Expressions are built-in with the [[ ]] operator
  
  if [[ $var = "" ]] ...

• Gets around parsing quirks of /bin/test, allows checking strings against patterns

• Operations:
  – string == pattern
  – string != pattern
  – string1 < string2
  – file1 -nt file2
  – file1 -ot file2
  – file1 -ef file2
  – &&, ||
Patterns

- Can be used to do string matching:
  ```
  if [[ $foo = *a* ]]
  if [[ $foo = [abc]* ]]
  ```
- Similar to regular expressions, but different syntax
Additional Parameter Expansion

- \${\#param} – Length of \param
- \${param\#pattern} – Left strip min \pattern
- \${param\#\#pattern} – Left strip max \pattern
- \${param\%pattern} – Right strip min \pattern
- \${param\%\%pattern} – Right strip max \pattern
- \${param\=value} – Default value if \param not set
Variables

- Variables can be arrays
  - foo[3]=test
  - echo ${foo[3]}
- Indexed by number
- ${#arr} is length of the array
- Multiple array elements can be set at once:
  - set -A foo a b c d
  - echo ${foo[1]}
  - Set command can also be used for positional params:
    set a b c d; print $2
Printing

- Built-in `print` command to replace `echo`
- Much faster
- Allows options:
  - `-u#` print to specific file descriptor
Functions

• Alternative function syntax:
  function name {
    commands
  }

• Allows for local variables
• $0$ is set to the name of the function
Additional Features

• Built-in arithmetic: Using `$((expression ))`
  – e.g., `print $(( 1 + 1 * 8 / x ))`

• Tilde file expansion
  `~` $HOME
  `~user` home directory of user
  `~+` $PWD
  `~` $OLDPWD
KornShell 93
Variable Attributes

- By default attributes hold strings of unlimited length
- Attributes can be set with typeset:
  - readonly (-r) – cannot be changed
  - export (-x) – value will be exported to env
  - upper (-u) – letters will be converted to upper case
  - lower (-l) – letters will be converted to lower case
  - ljust (-L width) – left justify to given width
  - rjust (-R width) – right justify to given width
  - zfill (-Z width) – justify, fill with leading zeros
  - integer (-I [base]) – value stored as integer
  - float (-E [prec]) – value stored as C double
  - nameref (-n) – a name reference
Name References

• A name reference is a type of variable that references another variable.

• `nameref` is an alias for `typeset -n`
  
  – Example:

```
user1="mehryar"
user2="adam"
typeset -n name="user1"
print $name
mehryar
```
New Parameter Expansion

- \$\{param/pattern/str\} – Replace first pattern with str
- \$\{param//pattern/str\} – Replace all patterns with str
- \$\{param:offset:len\} – Substring with offset
Patterns Extended

- Additional pattern types so that shell patterns are equally expressive as regular expressions
- Used for:
  - file expansion
  - [[]]
  - case statements
  - parameter expansion

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>.</td>
</tr>
<tr>
<td>*</td>
<td>.*</td>
</tr>
<tr>
<td>[ . . . ]</td>
<td>[ . . . ]</td>
</tr>
<tr>
<td>[! . . . ]</td>
<td>[ ^ . . . ]</td>
</tr>
<tr>
<td>? (. . .)</td>
<td>(. . .)?</td>
</tr>
<tr>
<td>* (. . .)</td>
<td>(. . .)*</td>
</tr>
<tr>
<td>+ (. . .)</td>
<td>(. . .)+</td>
</tr>
<tr>
<td>@ (. . .)</td>
<td>(. . .)</td>
</tr>
<tr>
<td>! (. . .)</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a &amp; b</td>
<td></td>
</tr>
<tr>
<td>{n} (. . .)</td>
<td>(. . .){n}</td>
</tr>
<tr>
<td>{m, n} (. . .)</td>
<td>(. . .){m, n}</td>
</tr>
<tr>
<td>\d</td>
<td>\d</td>
</tr>
</tbody>
</table>
ANSI C Quoting

- $\ldots$ Uses C escape sequences
  - $'\t'$   $'Hello\nthere'$
- `printf` added that supports C like printing:
  - `printf "You have %d apples" $x`
- Extensions
  - `%b` – ANSI escape sequences
  - `%q` – Quote argument for reinput
  - `\E` – Escape character (033)
  - `%P` – convert ERE to shell pattern
  - `%H` – convert using HTML conventions
  - `%T` – date conversions using date formats
**Associative Arrays**

- Arrays can be indexed by string
- Declared with `typeset -A`
- Set: `name["foo"]="bar"`
- Reference `$\{name["foo"]\}$`
- Subscripts: `$\{!name[@]\}$`
Corresponding Shell Features

• Standard input, output, error
  – Redirection
  – Here documents
  – Pipelines
  – Command substitution
• Exit status
  – $?
  – &&, ||, if, while
• Environment
  – export, variables
• Arguments
  – Command substitution
  – Variables
  – Wildcards
Lecture 7

Part II
Networking, HTTP, CGI
Network Application

- Client application and server application communicate via a network protocol.
- A protocol is a set of rules on how the client and server communicate.

```
web client  HTTP  web server
```
TCP/IP Suite

User

- client
- server

Kernel

- TCP/UDP
- IP
- drivers/hardware

Network access layer

(ethernet)
Data Encapsulation

Application Layer

Transport Layer

Internet Layer

Network Access Layer

Data

H1

H2

H3

H1

H2

H1

Data

Data

Data

Data
Network Access/Internet Layers

- Network Access Layer
  - Deliver data to devices on the same physical network
  - Ethernet

- Internet Layer
  - Internet Protocol (IP)
  - Determines routing of datagram
  - IPv4 uses 32-bit addresses (e.g. 128.122.20.15)
  - Datagram fragmentation and reassembly
Transport Layer

• Transport Layer
  – Host-host layer
  – Provides error-free, point-to-point connection between hosts

• User Datagram Protocol (UDP)
  – Unreliable, connectionless

• Transmission Control Protocol (TCP)
  – Reliable, connection-oriented
  – Acknowledgements, sequencing, retransmission
Ports

- Both TCP and UDP use 16-bit *port numbers*
- A server application listens to a specific *port* for connections
- Ports used by popular applications are well-defined
  - SSH (22), SMTP (25), HTTP (80)
  - 1-1023 are reserved (*well-known*)
  - 1024-49151 are user level
  - 49152-65535 are private to the machine
- Clients use *ephemeral* ports
Name Service

• Every node on the network normally has a hostname in addition to an IP address
• Domain Name System (DNS) maps IP addresses to names
  – e.g. 128.122.20.15 is sparky.cs.nyu.edu
• DNS lookup utilities: nslookup, dig
• Local name address mappings stored in /etc/hosts
Sockets

- Sockets provide access to TCP/IP on UNIX systems
- Invented in Berkeley UNIX
- Allows a network connection to be opened as a file (returns a file descriptor)
Major Network Services

• Telnet (Port 23)
  – Provides virtual terminal for remote user
  – The telnet program can also be used to connect to other ports

• FTP (Port 20/21)
  – Used to transfer files from one machine to another
  – Uses port 20 for data, 21 for control

• SSH (Port 22)
  – For logging in and executing commands on remote machines
  – Data is encrypted
Major Network Services \textit{cont.}

- **SMTP (Port 25)**
  - Host-to-host mail transport
  - Used by mail transfer agents (MTAs)
- **IMAP (Port 143)**
  - Allow clients to access and manipulate emails on the server
- **HTTP (Port 80)**
  - Protocol for WWW
Files in the form
/dev/tcp/hostname/port result in a socket connection to the given service:


eexec 3<>/dev/tcp/smtp.cs.nyu.edu/25 #SMTP
print -u3 "EHLO cs.nyu.edu"
print -u3 "QUIT"
while IFS= read -u3
do
    print -r "$REPLY"
done
HTTP

- Hypertext Transfer Protocol
  - Use port 80

- Language used by web browsers (IE, Netscape, Firefox) to communicate with web servers (Apache, IIS)

HTTP request:
Get me this document

HTTP response:
Here is your document
Resources

- Web servers host web resources, including HTML files, PDF files, GIF files, MPEG movies, etc.
- Each web object has an associated MIME type
  - HTML document has type text/html
  - JPEG image has type image/jpeg
- Web resource is accessed using a Uniform Resource Locator (URL)
HTTP Transactions

• HTTP request to web server
  
  GET /v40images/nyu.gif HTTP/1.1
  Host: www.nyu.edu

• HTTP response to web client
  
  HTTP/1.1 200 OK
  Content-type: image/gif
  Content-length: 3210

يزيد من جامعة نيوyawك University
Sample HTTP Session

GET / HTTP/1.1
HOST: www.cs.nyu.edu

HTTP/1.1 200 OK
Date: Wed, 19 Oct 2005 06:59:49 GMT
Server: Apache/2.0.49 (Unix) mod_perl/1.99_14 Perl/v5.8.4
         mod_ssl/2.0.49 OpenSSL/0.9.7e mod_auth_kerb/4.13 PHP/5.0.0RC3
Last-Modified: Thu, 12 Sep 2002 17:09:03 GMT
Content-Length: 163
Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
<title></title>
<meta HTTP-EQUIV="Refresh" CONTENT="0; URL=csweb/index.html">
<body>
</body>
</html>
Status Codes

- Status code in the HTTP response indicates if a request is successful
- Some typical status codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>302</td>
<td>Found; Resource in different URI</td>
</tr>
<tr>
<td>401</td>
<td>Authorization required</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
</tr>
</tbody>
</table>
Gateways

- Interface between resource and a web server
**CGI**

- **Common Gateway Interface** is a standard interface for running helper applications to generate dynamic contents
  - Specify the encoding of data passed to programs
- Allow HTML documents to be created on the fly
- Transparent to clients
  - Client sends regular HTTP request
  - Web server receives HTTP request, runs CGI program, and sends contents back in HTTP responses
- CGI programs can be written in any language
CGI Diagram

HTTP request

Web Server

HTTP response

spawn process

Document

Script
• Document format used on the web

<html>
<head>
<title>Some Document</title>
</head>
<body>
<h2>Some Topics</h2>
This is an HTML document
<p>
This is another paragraph
</p>
</body>
</html>
• HTML is a file format that describes a web page.
• These files can be made by hand, or generated by a program
• A good way to generate an HTML file is by writing a shell script
Forms

- HTML forms are used to collect user input
- Data sent via HTTP request
- Server launches CGI script to process data

```html
<form method=POST action="http://www.cs.nyu.edu/~unixtool/cgi-bin/search.cgi">
Enter your query: <input type=text name=Search>
<input type=submit>
</form>
```
Input Types

- **Text Field**
  ```html
  <input type=text name=zipcode>
  ```

- **Radio Buttons**
  ```html
  <input type=radio name=size value="S" > Small
  <input type=radio name=size value="M" > Medium
  <input type=radio name=size value="L" > Large
  ```

- **Checkboxes**
  ```html
  <input type=checkbox name=extras value="lettuce" > Lettuce
  <input type=checkbox name=extras value="tomato" > Tomato
  ```

- **Text Area**
  ```html
  <textarea name=address cols=50 rows=4>
  ...
  </textarea>
  ```
Submit Button

• Submits the form for processing by the CGI script specified in the `form` tag

```html
<input type=submit value="Submit Order"> 
```
HTTP Methods

• Determine how form data are sent to web server

• Two methods:
  – **GET**
    • Form variables stored in URL
  – **POST**
    • Form variables sent as content of HTTP request
Encoding Form Values

- Browser sends form variable as name-value pairs
  - name1=value1&name2=value2&name3=value3
- Names are defined in form elements
  - <input type=text name=ssn maxlength=9>
- Special characters are replaced with %## (2-digit hex number), spaces replaced with +
  - e.g. “10/20 Wed” is encoded as “10%2F20+Wed”
GET/POST examples

GET:
GET /cgi-bin/myscript.pl?name=Bill%20Gates&company=Microsoft HTTP/1.1
HOST: www.cs.nyu.edu

POST:
POST /cgi-bin/myscript.pl HTTP/1.1
HOST: www.cs.nyu.edu
   ...other headers...

name=Bill%20Gates&company=Microsoft
GET or POST?

- GET method is useful for
  - Retrieving information, e.g. from a database
  - Embedding data in URL without form element
- POST method should be used for forms with
  - Many fields or long fields
  - Sensitive information
  - Data for updating database
- GET requests may be cached by clients browsers or proxies, but not POST requests
Parsing Form Input

• Method stored in HTTP_METHOD
• GET: Data encoded into QUERY_STRING
• POST: Data in standard input (from body of request)
• Most scripts parse input into an associative array
  – You can parse it yourself
  – Or use available libraries (better)
CGI Environment Variables

- DOCUMENT_ROOT
- HTTP_HOST
- HTTP_REFERER
- HTTP_USER_AGENT
- HTTP_COOKIE
- REMOTE_ADDR
- REMOTE_HOST
- REMOTE_USER
- REQUEST_METHOD
- SERVER_NAME
- SERVER_PORT
Anonymous Comment Submission

Please enter your comment below which will be sent anonymously to korn@cs.nyu.edu. If you want to be extra cautious, access this page through Anonymizer or SafeWeb.
Part 1: HTML Form

<html>
<center>
<H1>Anonymous Comment Submission</H1>
</center>
Please enter your comment below which will be sent anonymously to <tt>mohri@cs.nyu.edu</tt>. If you want to be extra cautious, access this page through <a href="http://www.anonymizer.com">Anonymizer</a>. 
<p>
<form action="cgi-bin/comment.cgi" method="post">
<textarea name="comment" rows="20" cols="80">
</textarea>
<input type="submit" value="Submit Comment">
</form>
</html>
Part 2: CGI Script (ksh)

#!/home/unixtool/bin/ksh

. cgi-lib.ksh  # Read special functions to help parse
ReadParse
PrintHeader

print -r -- "${Cgi.comment}" | /bin/mailx -s "COMMENT" mohri

print "<H2>You submitted the comment</H2>"
print "<pre>"
print -r -- "${Cgi.comment}"
print "</pre>"
Debugging

- Debugging can be tricky, since error messages don't always print well as HTML
- One method: run interactively

```bash
$ QUERY_STRING='birthday=10/15/03'
$ ./birthday.cgi
Content-type: text/html

<html>
  Your birthday is <tt>10/15/02</tt>.
</html>
```
How to get your script run

- This can vary by web server type
  
  http://www.cims.nyu.edu/systems/resources/webhosting/index.html

- Typically, you give your script a name that ends with `.cgi`

- Give the script execute permission

- Specify the location of that script in the URL
CGI Security Risks

- Sometimes CGI scripts run as owner of the scripts
- Never trust user input - sanity-check everything
- If a shell command contains user input, run without shell escapes
- Always encode sensitive information, e.g. passwords
  - Also use HTTPS
- Clean up - don’t leave sensitive data around
CGI Benefits

• Simple
• Language independent
• UNIX tools are good for this because
  – Work well with text
  – Integrate programs well
  – Easy to prototype
  – No compilation (CGI scripts)
Example: Dump Some Info

#!/home/unixtool/bin/ksh

. ./cgi-lib.ksh
PrintHeader
ReadParse

print "<h1> Date </h1>"
print "<pre>"
date
print "</pre>"

print "<h1> Form Variables </h1>"
print "<pre>"
set -s -- ${!Cgi.*}
for var
do
    nameref r=$var
    print "${var#Cgi.} = $r"
    unset r
done
print "</pre>"

print "<h1> Environment </h1>"
print "<pre>"
env | sort
print "</pre>"
Example: Find words in Dictionary

<form action=dict.cgi>
Regular expression: <input type=entry name=re value=".*">
<input type=submit>
</form>
Example: Find words in Dictionary

#!/home/unixtool/bin/ksh

PATH=$PATH:
. cgi-lib.ksh
ReadParse
PrintHeader

print "<H1> Words matching <tt>${Cgi.re}</tt> in the dictionary</H1>
"
print "<OL>
grep "${Cgi.re}" /usr/dict/words | while read word
do
    print "<LI> $word"
done
print "</OL>"