Lecture 6

Shell Part II:
sh, bash, ksh
Parsing and Quoting
How the Shell Parses

• Part 1: Read the command:
  – Read one or more lines a needed
  – Separate into tokens using space/tabs
  – Form commands based on token types

• Part 2: Evaluate a command:
  – Expand word tokens (command substitution, parameter expansion)
  – Split words into fields
  – Setup redirections, environment
  – Run command with arguments
Useful Program for Testing

/include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i < argc; i++) {
        printf("Arg %d: %s
", i, argv[i]);
    }
    return(0);
}
Shell Comments

• Comments begin with an unquoted #
• Comments end at the end of the line
• Comments can begin whenever a token begins
• Examples
  # This is a comment
  # and so is this
  grep foo bar # this is a comment
  grep foo bar# this is not a comment
Special Characters

• The shell processes the following characters specially unless quoted:
  `| & ( ) < > ; " ' $ \` space tab newline`

• The following are special whenever patterns are processed:
  `* ? [ ]`  (turn off with `set -o noglob`)

• The following are special at the beginning of a word:
  `# ~`

• The following are special when processing assignments:
  `= [ ]`
Token Types

- The shell uses spaces and tabs to split the line or lines into the following types of tokens:
  - Control operators (|, ||)
  - Redirection operators (<, >, >>)
  - Reserved words (while, if)
  - Assignment tokens (foo=bar)
  - Word tokens (everything else)
Operator Tokens

• Operator tokens are recognized everywhere unless quoted. Spaces are optional before and after operator tokens.

• I/O Redirection Operators:
  >  >>  >|  >&  <  <<  <<-  <&
  – Each I/O operator can be immediately preceded by a single digit

• Control Operators:
  |  &  ;  (  )  ||  &&  ; ;
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.
Quoting Examples

$ cat file*
  a
  b

$ cat "file*"
cat: file* not found

$ cat file1 > /dev/null
$ cat file1 ">" /dev/null
  a
cat: >: cannot open

FILES="file1 file2"
$ cat "FILES"
cat: file1 file2 not found
Simple Commands

- A simple command consists of three types of tokens:
  - Assignments (must come first)
  - Command word tokens (name and args)
  - 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``
  - `print $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 (if 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
print 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

```
DATE=`date` echo $foo > /dev/null
```

- **DATE** = `date`: assignment
- **echo**: word
- **$foo**: param
- **>**: redirection

```
/bin/echo hello there
```

- **PATH expansion**
- **split by IFS**
The eval built-in

- `eval arg ...`
  - Causes all the tokenizing and expansions to be performed again
Input/Output Shell Features

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

• The shell is a language that lets you use programs as you would use procedures in other languages
  – Called with command line arguments
  – If used in if, while statements programs behave like functions returning a boolean value
    • /bin/true: Program that just does exit(0)
    • /bin/false: Program that just does exit(1)
  – If used in command substitution, programs behave like functions returning a string
  – Environment behaves like global variables
test Summary

• String based tests
  -z string  
  -n string  
  string1 = string2  
  string1 != string2  
  string
  Length of string is 0
  Length of string is not 0
  Strings are identical
  Strings differ
  String is not NULL

• Numeric tests
  int1 -eq int2  
  int1 -ne int2  
  -gt, -ge, -lt, -le
  First int equal to second
  First int not equal to second
  greater, greater/equal, less, less/equal

• File tests
  -r file  
  -w file  
  -f file  
  -d file  
  -s file
  File exists and is readable
  File exists and is writable
  File is regular file
  File is directory
  file exists and is not empty

• Logic
  !
  -a, -o
  ( expr )
  Negate result of expression
  and operator, or operator
  groups an expression
Example

#!/bin/sh

if test -f /tmp/stuff && \
[ `wc -l < /tmp/stuff` -gt 10 ]
then
    echo "The file has more than 10 lines"
else
    echo "The file is nonexistent or small"
fi
Arithmetic

• No arithmetic built in to /bin/sh
• Use external command /bin/expr

  • `expr` expression
    – Evaluates expression and sends the result to standard output
    – Yields a numeric or string result

  ```
  expr 4 "*" 12
  expr \( 4 + 3 \) \* 2
  ```
for loops

• Different than C:

```bash
for var in list
do
  command
done
```

• Typically used with positional params or a list of files:

```bash
sum=0
for var in "$@
  do
    sum=`expr $sum + $var`
  done

for file in *.c ; do echo "We have $file"
done
```
Case statement

• Like a C switch statement for strings:
  - case $var in
    opt1) command1
        command2
        ;;
    opt2) command
        ;;
    *) command
        ;;
  esac

• * is a catch all condition
#!/bin/sh

echo "Say something."
while true
do
  read INPUT_STRING
  case $INPUT_STRING in
    hello)
      echo "Hello there."
      ;;
    bye)
      echo "See ya later."
      ;;
    *)
      echo "I'm sorry?"
      ;;
  esac
done
echo "Take care."
Case Options

- **opt** can be a shell pattern, or a list of shell patterns delimited by `|`

- **Example:**

  ```bash
  case $name in
    *[0-9]*)
      echo "That doesn't seem like a name."
    ;;
    J*|K*)
      echo "Your name starts with J or K, cool."
    ;;
  *)
      echo "You're not special."
    ;;
  esac
  ```
Types of Commands

All behave the same way

• Programs
  – Most that are part of the OS in /bin
• Built-in commands
• Functions
• Aliases
Built-in Commands

• Built-in commands are internal to the shell and do not create a separate process.

Commands are built-in because:

– They are intrinsic to the language (**exit**)
– They produce side effects on the process (**cd**)
– They perform much better
  • No fork/exec
Important Built-in Commands

exec : replaces shell with program
cd : change working directory
shift : rearrange positional parameters
(un)set : set positional parameters
wait : wait for background proc. to exit
umask : change default file permissions
exit : quit the shell
eval : parse and execute string
time : run command and print times
export : put variable into environment
trap : set signal handlers
Important Built-in Commands

**continue** : continue in loop
**break** : break in loop
**return** : return from function
**:** : true
*. : read file of commands into current shell; like `#include`
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**

```sh
read
read –r NAME
read FIRSTNAME LASTNAME
```
trap command

• **trap** specifies command that should be executed 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

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

Functions are similar to scripts and other commands except that they can produce side effects in the callers script. The positional parameters are saved and restored when invoking a function. Variables are shared between caller and callee.

Syntax:

```plaintext
ame ()
{
    commands
}
```
Aliases

• Like macros (#define in C)
• Shorter to define than functions, but more limited
• Not recommended for scripts
• Example:
  alias rm='rm -i'
Search Rules

• Special built-ins
• Functions
  – *command* bypasses search for functions
• Built-ins not associated with PATH
• PATH search
• Built-ins associated with PATH
• Executable images
Script Examples

• Rename files to lower case
• Strip CR from files
• Emit HTML for directory contents
Rename files

#!/bin/sh

for file in *
do
   lfile=`echo $file | tr A-Z a-z`
   if [ $file != $lfile ]
      then
         mv $file $lfile
      fi
done
#!/bin/sh

TMPFILE=/tmp/file$$

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

trap 'rm -f $TMPFILE' EXIT

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

<table>
<thead>
<tr>
<th>arctoc.sh</th>
<th>dir.html</th>
<th>dir2html.sh</th>
<th>foo</th>
<th>foo.tar</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>stripcr.sh</td>
<td>tolower.sh</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#!/bin/sh

if test -n "$1"
then
    cd "$1"
fi

cat <<HUP
<html>
<h1> Directory listing for $PWD </h1>
<table border=1>
<tr>
    num=0  # global variable counting file number
    for file in *
    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>"
    # Check if this is the end of the row
    num=`expr $num + 1`
    if [ $num -gt 4 ]
    then
        echo "</tr><tr>"
        num=0
    fi
}
Korn Shell / bash Features
Command Substitution Syntax

- 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 issues when using `/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]* ]]```
Additonal 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`
Functions

- Alternative function syntax:
  ```
  function name { 
      commands 
  }
  ```
- Allows for local variables (with typeset)
- $\theta$ 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
Printing (ksh only)

• Built-in **print** command to replace echo
• Not subject to variations in echo
• Much faster
• Allows options:
  - **-u#**    print to specific file descriptor
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:

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

- \${param/pattern/str} – Replace first pattern with str
- \${param//pattern/str} – Replace all patterns with str
- \${param:offset:len} – Substring
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><code>?</code></td>
<td><code>( ... )?</code></td>
</tr>
<tr>
<td><code>*</code></td>
<td><code>( ... )*</code></td>
</tr>
<tr>
<td><code>[ ... ]</code></td>
<td><code>( ... )</code></td>
</tr>
<tr>
<td><code>[^ ... ]</code></td>
<td><code>[^ ... ]</code></td>
</tr>
<tr>
<td><code>@ ( ... )</code></td>
<td><code>( ... )</code></td>
</tr>
<tr>
<td><code>! ( ... )</code></td>
<td><code>! ( ... )</code></td>
</tr>
<tr>
<td>`a</td>
<td>b`</td>
</tr>
<tr>
<td><code>a &amp; b</code></td>
<td><code>a &amp; b</code></td>
</tr>
<tr>
<td><code>{n} ( ... )</code></td>
<td><code>( ... ) {n}</code></td>
</tr>
<tr>
<td><code>{m, n} ( ... )</code></td>
<td><code>( ... ) {m, n}</code></td>
</tr>
<tr>
<td><code>\d</code></td>
<td><code>\d</code></td>
</tr>
</tbody>
</table>
ANSI C Quoting

• $'…' 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, like awk
• Declared with `typeset -A`
• Set: `name["foo"]="bar"`
• Reference `$\{name["foo"]\}$`
• Subscripts: `$\{!\text{name[@]}\}$`
Coprocesses

- `|&` operator supports a simple form of concurrent processing
- `cmd |& cmd` runs as a background process whose standard input and output channels are connected to the original parent shell via a two way pipe.
- Can read and write from process with
  - `read -p`
  - `print -p`
- Note that `echo` couldn’t be used. Why?
C Expressions

• We have already seen built-in expressions with the `[[ ]]` operator:
  - `[[ $var = *foo* ]] && print "contains foo"

• New operator `(( ))` for C-like numeric expressions:
  - `(( x > 10 )) && print "x=$x, greater than 10"
  - `(( x ++ ))`
  - Note variables don't have to be used with `$` inside parens

• Value of `(( ))` expression can be used with `$(())`
  - `y=$(( x + 1 ))`
  - `print $(( x * y - sin(y) ))`
Compound Variables

• Variables can contain subfields (like structures or classes)
• Syntax: variable name containing .
• Example:
cust=(name=Jeff zip=10003)
cust.state=NY
print ${cust.name}
print ${!cust.*}
New for loop syntax

- Regular syntax:
  ```
  for var in list
    do
      ...
    done
  ```

- Additional syntax like C:
  ```
  for (( initialization; condition; increment ))
    do
      ...
    done
  ```

- Example: `for (( i=0; i < $VAR; i++ ))`
Example: Word Count

#!/home/unixtool/bin/ksh

integer l=0 w=0 c=0
while read –r LINE
do
    (( l++ ))
    set -- $LINE
    (( w += $# ))
    (( c += ${#LINE}+1 ))
done < $1

print "$l lines, $w words, $c characters"
Example: Word Count

```bash
integer l=0 w=0 c=0
while read -r LINE
do
    (( l++ ))
    set -- $LINE
    (( w += $# ))
    (( c += $#LINE+1 ))
done < $1

print "$l lines, $w words, $c characters"
```

- **integer** tag indicates variables will be used as integers
- **while** loop is a command, so redirection works
Example: Word Count

integer l=0 w=0 c=0
while read -r LINE
do
  (( l++ ))
  set -- $LINE
  (( w += $# ))
  (( c += ${#LINE}+1 ))
done < $1

print "$l lines, $w words, $c characters"

• set -- $LINE turns LINE into positional parameters ($1, …), splitting up the value with IFS
• $# is the number of positional parameters
Example: Word Count

```bash
integer l=0 w=0 c=0
while read -r LINE
do
    (( l++ ))
    set -- $LINE
    (( w += $# ))
    (( c += $#LINE+1 ))
done < $1

print "$l lines, $w words, $c characters"
```

- `${#LINE}` returns the length of the value of LINE
- We add 1 because the newline character is not part of LINE
Example: Spell a Phone Number

Given a number, finds possible words that the number spells on a telephone.

Example:

$ phonespell 8643
void
Algorithm

- Create function `combo` that prints all combinations of words. Check those against the dictionary.
- function `combo` is *recursive*:
  - Pass in part of number, part of word spelled

```
combo 8643 ""
```

```
combo 643 t  combo 643 u  combo 643 v
```

```
combo 43 vm  combo 43 vn  combo 43 vo
```

```
combo "" void
```
Example: Spell a Phone Number

```bash
function combo {
    typeset num=$1 word=$2
    if [[ $num == '' ]]; then
        print $word
    else
        typeset -L1 digit=$num
        for letter in ${get_letter[digit]};
        do
            combo "${num#?}" "$word$letter"
        done
    fi
}
```

- functions defined in ksh take arguments as positional parameters, like commands
- `typeset` makes a variable local
Example: Spell a Phone Number

```bash
function combo {
    typeset num=$1 word=$2
    if [[ $num = '' ]]; then
        print $word
    else
        typeset -L1 digit=$num
        for letter in ${get_letter[digit]};
        do
            combo "${num#?}" "$word$letter"
        done
    fi
}
```

• End of recursion: If number is empty, just print the given word. Should end up happening for every combination
Example: Spell a Phone Number

```bash
function combo {
    typeset num=$1 word=$2
    if [[ $num = '' ]]
    then
        print $word
    else
        typeset -L1 digit=$num
        for letter in ${get_letter[digit]}
        do
            combo "${num#?}" "$word$letter"
        done
    fi
}
```

- Extract leftmost digit from `num`
Example: Spell a Phone Number

function combo {
    typeset num=$1 word=$2
    if [[ $num == '' ]]
    then print $word
    else typeset -L1 digit=$num
        for letter in ${get_letter[digit]}
            do combo "${num#?}" "$word$letter"
        done
    fi
}

- **for** loop goes through all letters that correspond to the number (stored in get_letter array, shown next slide)
- Recursively calls itself for each letter, taking off one character from the left (using the # operator with pattern ?)
Spell a Phone Number (cont’)

set -A get_letter o i "a b c" "d e f" "g h i" "j k l" \ "m n o" "p r s" "t u v" "w x y"

# method 1
combo $1 | comm -12 /usr/dict/words -

# method 2
trap 'rm -f /tmp/full$$' EXIT
combo $1 > /tmp/full$$
spell < /tmp/full$$ | comm -13 - /tmp/full$$

- set –A arrayname value value ...
  - sets elements of an array all at once
Spell a Phone Number (cont’)

set -A get_letter o i "a b c" "d e f" "g h i" "j k l" \ 
"m n o" "p r s" "t u v" "w x y"

# method 1
combo $1 | comm -12 /usr/dict/words -

# method 2
trap 'rm -f /tmp/full$$' EXIT
combo $1 > /tmp/full$$
spell < /tmp/full$$ | comm -13 - /tmp/full$$

- Call function **combo** with first argument, pipe to **comm**
  - suppress fields 1 and 2 (show only matching lines)
  - **combo** emits sorted lines, and dictionary is sorted so **comm** works well
Spell a Phone Number (cont’)

- Another method: use spell command
  - Create temporary file storing combos
  - Run through spell, generating list of misspelled words
  - Pipe to comm, suppressing fields 1 and 3 (show correct words)

```bash
set -A get_letter o i "a b c" "d e f" "g h i" "j k l" \ "m n o" "p r s" "t u v" "w x y"

# method 1
combo $1 | comm -12 /usr/dict/words -

# method 2
trap 'rm -f /tmp/full$$' EXIT
combo $1 > /tmp/full$$
spell < /tmp/full$$ | comm -13 - /tmp/full$$
```
Example: Mortgage Calculator

```bash
float rate=$1 principle=$2 payment
integer months years=$3

[[ $1 ]] || read -r 'rate?rate in per cent: '
[[ $2 ]] || read -r 'principle?principle: '
[[ $3 ]] || read -r 'years?years to amortization: '

print "\n\n\nprinciple\n\nrate\n\namortization\n\n\n\n\n\nprinciple\nrate\nyears" 

(( months = years*12 ))
(( rate /= 1200. ))
(( payment = (principle*rate)/(1.-pow(1.+rate,-months)) ))
```

- Declare variables
- Read in unspecified inputs
Example: Mortgage Calculator

```bash
float rate=$1 principle=$2 payment
integer months years=$3

[[ $1 ]] || read -r 'rate?rate in per cent: '
[[ $2 ]] || read -r 'principle?principle: '
[[ $3 ]] || read -r 'years?years to amortization: '

print "\n\n\nprinciple\t$principle"
print "\t\trate		$rate"
print "\t\tamortization	$years"

(( months = years*12 ))
(( rate /= 1200. ))
(( payment = (principle*rate)/(1.-pow(1.+rate,-months)) ))
```

- Initialize values
- Uses built-in arithmetic (**pow**, floating point /)
Example: Mortgage Calculator

```plaintext
printf "\tmonthly payment\t%8.2f\n\n" "$payment"
print '\tYears   Balance'
print '\t======   ======='

for (( months=0; principle > 0; months++))
do   (( principle *= (1.+rate) ))
     (( principle -= payment ))
if      (( ((months+1)%12) == 0 ))
then    printf "\t%d\t%8.2f\n" months/12 "$principle"
fi
done
```

• Print table header
  – Uses `printf` to format floating point number
Example: Mortgage Calculator

```
printf "\tmonthly payment\t%8.2f\n\n" "$payment"
print '\tYears	Balance'
print '\t======	======'

for (( months=0; principle > 0; months++))
do  (( principle *= (1.+rate) ))
    (( principle -= payment ))
if      (( ((months+1)%12) == 0 ))
then    printf "\t%d\t%8.2f\n" months/12 "$principle"
fi
done
```

• C-style for loop with numerical calculations
Documentation

• Web version of *Learning the KornShell* documents ksh93. Good for learning ksh.
• Glass documents ksh88 and bash
• UNIX in a Nutshell has a chapter that is a great ksh93 reference. Documents:
  – Bourne shell compatible features
  – ksh88 compatible features
  – ksh93 features