CS 114 - Fall 2004

Lecture 9 - Monday, October 18, 2004

Shell scripts

Control flow

The Bourne shell provides several control-flow commands for use in shell scripts. Most of these control-flow commands test *exit status* of a command. If the exit status of a command is `0`, indicating success, then the test is **true**; otherwise, the test is **false**.

-
  ```
  if command1; then
      command2
  fi
  ```
  if *command1* is successful (i.e., its exit status is 0), then run *command2*.

-
  ```
  if command1; then
      command2
  else
      command3
  fi
  ```
  if *command1* is successful, then run *command2*; otherwise, run *command3*.

-
  ```
  if command1; then
      command2
  elif command3; then
      command4
  else
      command5
  fi
  ```
  if *command1* is successful, then run *command2*; otherwise, if *command3* is successful, run *command4*; otherwise, run *command5*.

-
  ```
  while command1; do
      command2
  done
  ```
  while *command1* is successful, run *command2*.

-
  ```
  until command1; do
      command2
  done
  ```
  until *command1* is successful, run *command2*.

For example, the following program echoes "`hello found`" if the file `file1` contains the string "`hello`".

```
if egrep -q hello file1
then
    echo hello found
fi
```

The `-q` option tells `egrep` to not output the matching lines; the program simply exits with `0` if the string is found, and non-zero otherwise.

Most programs do not have an option to suppress their output. However, `stdout` can be redirected to a special file called `/dev/null`. For example, the following behaves identically to the program above.

```
if egrep hello file1 > /dev/null
```

```
then
    echo hello found
fi
```

Conditionals

Each of the control-flow commands above can branch on the exit status of any command. The command `test` can be used to check specific conditions:

`test -f` *`file`*	**true** if *`file`* exists and is a normal file.
`test -d` *`dir`*	**true** if *`dir`* exists and is a directory.
`test -e` *`file`*	**true** if *`file`* exists (may be a file or directory)
`test -r` *`file`*	**true** if *`file`* is readable
`test -w` *`file`*	**true** if *`file`* is writeable
`test -x` *`file`*	**true** if *`file`* is executable
`test` *`num1`* `-eq` *`num2`*	**true** if *`num1`* equals *`num2`*
`test` *`num1`* `-ne` *`num2`*	**true** if *`num1`* does not equal *`num2`*
`test` *`num1`* `-gt` *`num2`*	**true** if *`num1`* is greater than *`num2`*
`test` *`num1`* `-lt` *`num2`*	**true** if *`num1`* is less than *`num2`*
`test` *`num1`* `-ge` *`num2`*	**true** if *`num1`* is greater than or equal to *`num2`*
`test` *`num1`* `-le` *`num2`*	**true** if *`num1`* is less than or equal to *`num2`*
`test` *`string1`* `=` *`string2`*	**true** if *`string1`* equals *`string2`*
`test` *`string1`* `!=` *`string2`*	**true** if *`string1`* does not equal *`string2`*
`test -n` *`string`*	**true** if *`string`* is non-empty
`test -z` *`string`*	**true** if *`string`* is empty

The `test` command can also be invoked with the name `[`. For example:

```
if test $x -gt $y; then
    max=$x
fi
```

can also be written:

```
if [ $x -gt $y ]; then
    max=$x
fi
```

`for` loops

Another control flow statement is the `for` loop. `for` takes the name of a variable and a *list*. For example,

```
for i in a b c; do
    echo $i
done
```

will echo

```
a
b
c
```

and

```
for i in "$@"; do
    echo $i
done
```

will echo each of the command-line arguments on a separate line.

for is also useful on the command line (if your shell is a Bourne shell like bash or ksh). For example, the following command will create backups of all files in the current directory:

```
$ ls
file1 file2 scratch
$ for i in *; do cp "$i" "$i.bak"; done
$ ls
file1     file2     scratch
file1.bak file2.bak scratch.bak
```

Command-line argument parsing

A script can take arguments on the command line. The first argument is in the special variable $1, the second in $2, etc.,

The special variable $0 expands to the name of the script, and the variable $# expands to the number of command-line arguments.

If $# is greater than 0, the shift command renames $2 to $1, $3 to $2, etc., unsets the last parameter variable, and decrements $#.

The variables $* and $@ both expand to all of the command-line arguments. These variables behave differently when enclosed in double quotes.

- "$*" is equivalent to "$1 $2 $3 ..."
- "$@" is equivalent to "$1" "$2" "$3" ...

In general, "$@" should be used rather than $*, since it correctly handles command-line arguments containing spaces and other special characters.

A small example: watch

The following script, called watch, runs the command given on the command line every two seconds, refreshing the display between each run. A similar program is found in many Linux distributions. I use watch mainly to help debug networking problems. For example watch ifconfig will run the ifconfig program every two seconds, which outputs, among other things, the IP address assigned to the host (or not assigned, as is usually the case when I'm experiencing network problems).

```
#!/bin/sh

# Height of window - 1.
height=39

while true; do
    clear
    (
        date
        echo
        "$@"
    ) | head -$height
    sleep 2
```

```
done
```

The program `true` simply returns 0, indicating success. A similar program `false` returns 1, indicating failure. The program loops forever. It can be interrupted by typing `Ctrl-C`.

Each iteration of the loop first clears the screen with the `clear` program. Then the date and a blank line are output, followed by the output of the command given to the script on the command line. The output of several commands can be grouped together with parentheses and redirected, or, in this case, piped to `head` in order to keep the output from scrolling off the screen. The commands in parentheses are actually run in a subshell, a separate process; therefore, local variables are in scope only within the subshell. The `sleep` program sleeps for the number of seconds given as its argument.

Adding command-line options to the thumbnail script

The following script adds command-line argument parsing to the `make-thumbnail` script from last time. The new script can generate thumbnails for several images specified on the command line, not just one image. It also adds an option `-xy` to specify the bounding box for the generating thumbnails (200x200 by default), and adds an option `-v` to turn on verbose output,

```
#!/bin/sh

# Create a thumbnail image from the file given on the command line.
# usage: make-thumbnail [-xy MxN] [-v] files

# Make sure the programs we need are in the path.
PATH=/usr/bin:/bin:/usr/local/graphics/jpeg-6b/bin:/usr/local/pbm


# The default bounding box size.
x=200
y=200

# Non-empty if verbose; default to non-verbose.
verbose=

# Get just the filename part of the script name.
prg=`basename $0`

# Process -xy and -v
while [ $# -gt 0 ]; do
    if [ "$1" = "-xy" ]; then
        shift
        # $1 now contains MxN; check if it's there
        if [ $# -eq 0 ]; then
            echo "$prg: missing argument to -xy"
            echo "usage: $prg [-xy MxN] [-v] files..."
            exit 1
        fi
        x=`echo $1 | sed 's/x[0-9]*//'` # get the number before the x
        y=`echo $1 | sed 's/[0-9]*x//'` # get the number after the x
        shift
    elif [ "$1" = "-v" ]; then
        verbose=yes
        shift
    else
        # Must be a file name; break out of the while loop
        break
    fi
done
```

```
# Remaining arguments are in $@.

# Complain if there are no files on the command line.
if [ $# -eq 0 ]; then
    echo "usage: $prg [-xy MxN] [-v] files..."
    exit 1
fi

# Compute the minimum of $x and $y
min=$x
if [ $x -gt $y ]; then
    min=$y
fi

# Make sure all the input files exist
for jpg in "$@"; do
    if [ ! -r "$jpg" ]; then
        echo "$prg: $jpg not found"
        exit 1
    fi
done

# Create the thumbnails.
for jpg in "$@"; do
    thumb=`echo "$jpg" | sed 's/\.jpg$/.thumbnail.jpg/'`

    if [ -n "$verbose" ]; then
        echo "producing thumbnail $thumb from $jpg"
    fi

    # Use nawk to get image size. Change to awk or gawk if nawk not
found.
    w=`rdjpgcom -verbose "$jpg" |
        nawk '/^JPEG image is/ { sub("w$", "", $4); print $4 }'`
    h=`rdjpgcom -verbose "$jpg" |
        nawk '/^JPEG image is/ { sub("h,$", "", $6); print $6 }'`

    if [ -n "$verbose" ]; then
        echo "$jpg is $w * $h"
    fi

    # Want a thumbnail that's at most $x*$y, but preserves
    # the $w*$h aspect ratio

    # If the image already fits, just copy it.
    if [ $w -le $x ] && [ $h -le $y ]; then
        echo "$jpg already fits in a $x * $y bounding box"
        cp "$jpg" "$thumb"
        continue
    fi

    # Compute the smaller of the two scaling factors.
    factor=`echo "scale=4; if ($x/$w < $y/$h) ($x/$w) else ($y/$h)" |
bc`

    if [ -n "$verbose" ]; then
     echo "scaling $jpg by $factor to fit in $x * $y bounding box"
    fi

    djpeg -pnm "$jpg" | pnmscale $factor | cjpeg > "$thumb"
done
```