

CS108 Software Systems: Unix

Spring 2008

Roadmap

- gnuplot
- xfig
- gimp
- Shell scripting

Also, happy Leap Day!

Gnuplot commands

The primary gnuplot commands are `set` and `plot`/`splot`.

- `set option parameters...`
- `plot function with linestyle`
- `plot "data_file" using entry with linestyle`
- `splot function with linestyle`
- `splot "data_file" using entry with linestyle`

entry specifies a column of data, e.g. `1:2` – column 2 versus column 1, `1:($2+$3)` – the sum of column 2 and column 3 versus column 1.

The Gnuplot `set` command

```
set title "text"
```

- Create (or clear) a title for a graph.

```
set xrange [min:max]
```

- Set the range for the x-axis.

```
set xlabel "text"
```

- Create a label for the x-axis.

```
set xtics value1, value2, value3, ...
```

- Set the major tick mark positions on the x-axis.

```
set autoscale axes
```

- Let Gnuplot manage the range for one or more axes.

More set commands

`set contour`

- Include a contour graph for 3D data.

`set key`

- Enables/disables inclusion of a key or legend in a graph.

`set output "file"`

- Create a graph in a file rather than on a display.

`set terminal type`

- Often used with `set output` to specify the format of the file, e.g. `pbm`, `png`, `postscript`, `latex`, etc.

Gnuplot Scripts

Gnuplot can execute commands stored within a file.

```
set terminal png
set output "myplot.png"
set logscale z
set nokey
set hidden3d
set view 25,25
splot [-2:2] [-2:2] (100 * (y - x**2)**2 + (1 - x)**2)
```

```
odin:~> gnuplot myscript.gnuplot
```

xfig

xfig can be used for

1. simple drawings
2. simple picture modifications

gimp

GNU Image Manipulation Program

Started in 1995 and still actively being developed.

Think of Adobe Photoshop, but free, a wonkey menu system, and lots of free plugins.

Scripts

A *script* is a program written in a *scripting language*. Scripting languages typically do not include facilities for complex I/O, mathematics, or data structures, and are therefore most useful for “gluing” together other software components. Some common scripting languages:

- AWK
- Perl
- Python

We will be studying *shell scripting*, specifically, the scripting language provided by the `bash` shell.

A Script Example

Suppose I wish to generate a list of “friends” currently logged in to my machine:

```
#!/bin/bash

FriendList="mtaylor nate nkj kuhlmann serita"
UserList=$(who | cut -d " " -f 1)
for Friend in $FriendList; do
    for User in $UserList; do
        if [ $User = $Friend ]; then
            echo "My friend $Friend is online."
            break
        fi
    done
done
```

Script Execution

Scripts can be invoked by name, like any program, if the script file is *readable* and *executable*. If the previous script were named `online.sh`:

```
odin:~> online.sh
```

Scripts invoked in the above manner execute in a ***subshell***, a child shell process. In bash, Shell scripts may also be ***sourced***, executed by the current shell process.

```
odin:~> source online.sh
```

A sourced script need not be *executable*.

Script Contents

Scripts should begin with a line specifying which shell will execute the script:

```
#! path-of-shell-executable
```

Shell scripts may contain any command or syntax usable at the prompt, and vice versa.

- Commands, redirection, quoting.
- Variable assignment/expansion.
- Flow control (`if`, `for`, `while`, etc).
- Arithmetic operations.

Shell Variables

There are several types of variables, all manipulated with the familiar syntax:

```
FriendList="mtaylor nate nkj kuhlmann serita"  
echo "My friend $Friend is online."
```

- *User-defined variables* exist only in executing shell. (Default)
- *Environment variables* are copied to subshells.
- *Local variables* exist only in shell functions.

We saw environmental variables earlier (`setenv` to set the PATH). Environment variables in Bash are created with the `declare` or `export` commands:

```
export FriendList  
declare -x FriendList
```

Script Parameters

<code>\$0</code>	The name of the executing script.
<code>\$1 ... \$9</code> <code>\${10} ... \${99}</code>	Command line parameters (positional parameters).
<code>\$#</code>	The number of positional parameters.
<code>\$*</code> or <code>\$@</code>	Equivalent to <code>\$1 \$2 \$3 ...</code>
<code>"\$*"</code>	Equivalent to <code>"\$1 \$2 \$3 ..."</code>
<code>"\$@"</code>	Equivalent to <code>"\$1" "\$2" "\$3" ...</code>

Script Parameter Examples

A script to check what's running on a different machine:

```
#!/bin/bash  
  
rsh $1 /lusr/bin/ps -e
```

A script displaying information about a logged in user:

```
#!/bin/bash  
  
finger -m "$1" | head -1 | cut -c 32-  
echo "-----"  
who | grep "^$1" | cut -c 36- | sort | uniq  
echo "-----"  
ps aux | grep "^$1" | cut -c 64-
```

Flow control - Conditionals

```
if condition; then commands  
elif condition; then commands  
else commands  
fi
```

condition must be an expression producing an ***exit status*** value, an integer generated when a command terminates. If the exist status value equals zero the *condition* is **true**, otherwise the the *condition* is **false**.

Conditional Tests

The `test` and `[]` commands performs basic file and value comparisons.

- `[str1 = str2]` – Tests if string *str1* equals string *str2*.
- `[-z string]` – Tests if the length of *string* is 0.
- `[int1 -eq int2]` – Tests if number *int1* equals *int2*.
- `[int1 -le int2]` – Tests if number *int1* is less than *int2*.
- `[! test]` – Negate a test.
- `[test1 -a test2]` – Logical AND of two tests.
- `[-e path]` – Tests if *path* refers to an existing file.
- `[-f path]` – Tests if *path* refers to a plain file.
- `[-d path]` – Tests if *path* refers to a directory.

if Example (1/2)

```
#!/bin/bash

if [ ! $# -eq 2 ]; then
    echo "Error - I need two parameters"
elif [ $1 -eq 0 ]; then
    echo "arg1 is 0!"
elif [ $2 -le 5 ]; then
    echo "arg2 is less than 5 (in fact, it's $2)"
else
    echo "I give up. 42?"
fi
```

if Example (2/2)

```
#!/bin/bash

if [ $UID -eq 0 ]; then
    echo "I am ROOT, fear my power..."
elif [ "$(date +%a)" = "Fri" ]; then
    echo "The weekend is sooooo cloooooose..."
elif [ -e ~/mailbox ]; then
    echo "I should check my mail."
fi
```

Flow control – Conditionals

When comparing a string against several values, a `case` statement is often more concise than sequential `if` statements. Plus, strings can be matched via patterns.

```
case expression in
    pattern1) commands ;;
    pattern2) commands ;;
    ...
esac
```

case Example

Here we use a `case` statement to check a command line argument and take appropriate action.

```
#!/bin/bash

case "$1" in
  --help) echo "Usage: view.sh FILE" ;;
  *.html) netscape "$1" ;;
  *.pdf)  acroread "$1" ;;
  *.jpg)  gimp "$1" ;;
  *)      echo "Unable to view file $1" ;;
esac
```

Flow control – Loops

```
for variable in list; do commands
done
```

variable is set to each word in *list* and *commands* are executed. For example, a simple script that displays each of its command line arguments, one per line:

```
#!/bin/bash

for arg in "$@"; do
    echo $arg
    gzip $arg
    mv $arg.gz ~/backup
done
```

Loops may contain the **break** command to prematurely terminate the loop, and the **continue** command to jump to the next loop iteration.

Flow control – Loops

Count-based for loops use a slightly different, C-style syntax:

```
for ((init; test; update)); do commands
done
```

```
#!/bin/bash

echo "An automated sobriety test... please count backwards from 100 by 7s..."
for ((i=100; i>0; i-=7)); do
    sleep 1
    echo $i
done
```

Flow control - More Loops

```
while condition; do commands  
done
```

While *condition* is true execute *commands*.

```
until condition; do commands  
done
```

Until *condition* is true execute *commands*.

File Traversal

for loops allow easy traversal of lists of paths. If a script is given a list of files on the command line:

```
for file in "$@"; do
    ...
done
```

Traversal with Type Guarding

Like any program, a well written script should perform safety checks on its input. For example, if we wish to traverse only “plain” files:

```
for file in "$@"; do
    if [ -f "$file" ]; then
        ...
    fi
done
```

Generalizing Our Traversal

The previous examples dealt solely with a list of files, but suppose we'd like to recursively traverse any directories in that list (assume this script is named `traverse.sh`):

```
for file in "$@"; do
    if [ -d "$file" ]; then
        traverse.sh "$file"/*
    elif [ -f "$file" ]; then
        ...
    fi
done
```

Functions in Scripts

The previous script was recursive, self-calling. Each call requires a new subshell, thus system resources are wasted. To solve this inefficiency, we can define a recursive ***function***, which doesn't require additional subshells.

```
function-name () {  
    commands  
}
```

A function must be defined before it is used, and may accept arguments in the form of positional parameters.

A Caveat

A FAQ Matt read said:

Does bash permit recursion?

Well, yes, but...

It's so slow that you gotta have rocks in your head to try it.

and later, in an example script:

```
if [ "$1" -gt $MAX_ARG ]
then
  echo "Out of range (5 is maximum)."
```

By the way, note the `exit` command which terminates the script and passes back and (optional) return value to the caller.

A Better Traversal

```
helper () {
  for file in "$@"; do
    if [ -d "$file" ]; then
      helper "$file"/*
    elif [ -f "$file" ]; then
      ...
    fi
  done
}

helper "$@"
```

A Canonical Recursion Example

```
#!/bin/bash

factorial(){
    if [ $1 -eq 1 ]; then
        return 1
    else
        factorial $(( $1 - 1 ))
        return $(( $1 * $?))
    fi
}

factorial "$1"
echo "Factorial of $1 is $?."
```

Notes:

`$?` is the return value for the last function.

`$((var simbol var))` does arithmetic (more on this next week).