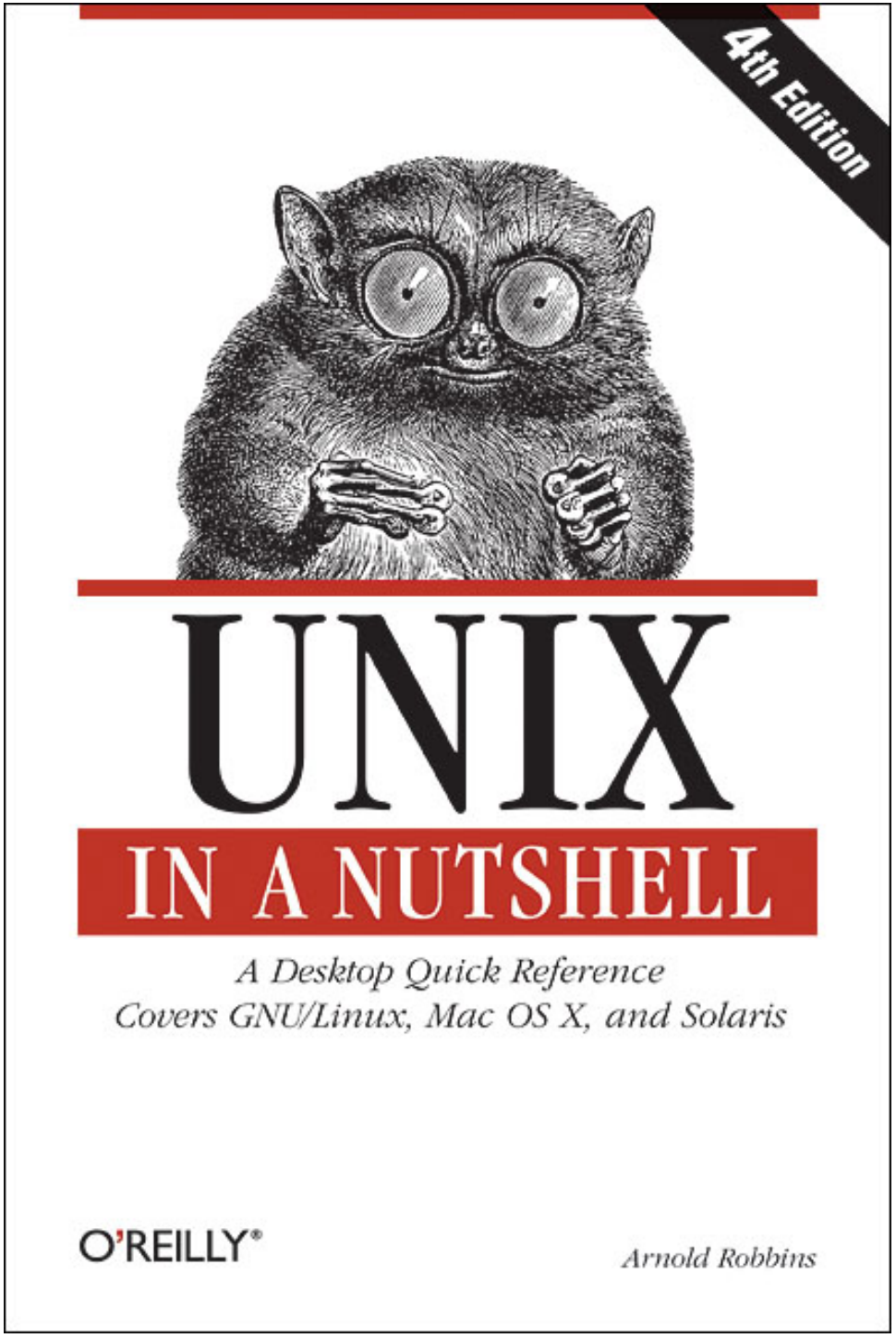
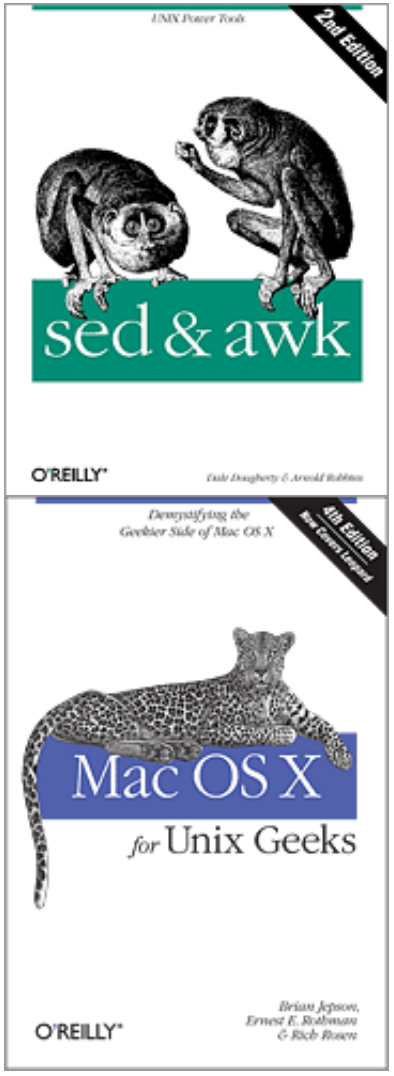
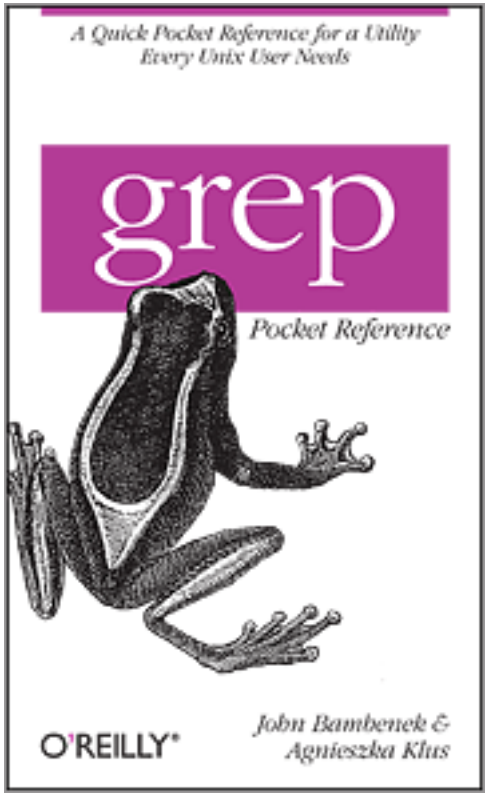
Unix Tools 1

Beyond the Mouse







Resources

- Unix and Linux
  - It comes with the package
- Macintosh
  - Really a Unix machine with fancy wrapping
  - Terminal.app
- Windows
  - VirtualBox: http://www.virtualbox.org/
  - Cygwin: http://www.cygwin.com/
  - .NET: https://www.microsoft.com/net/

Outline

- What is Unix, what is the shell?
  - Windows: Microsoft .NET framework
- Directories, files, wildcards
- Redirecting input/output, pipes
- Variables and control
- Power tools: GMT, etc. in future lectures
- Need more power? Upgrade to `perl`
  - `perl` = Practical Extraction and Reporting Language
  - `perl` = Pathologically Extensible Rubbish Lister
- Increasingly popular: `python`

Unix

- Unix is the most common operating system for "serious computers"
  - Developed in 1969 at Bell Labs (of the old AT&T)
  - At first, it could support two simultaneous users!
  - Rewritten in C in 1973 (before that, assembly language)
- From Wikipedia:
  - Unix was designed to be portable, multi-tasking and multi-user in a time-sharing configuration. Unix systems are characterized by various concepts: the **use of plain text for storing data**; a **hierarchical file system**; **treating devices** and certain types of inter-process communication (IPC) as **files**; and the use of a large number of software tools, **small programs that can be strung together through a command line interpreter using pipes, as opposed to using a single monolithic program that includes all of the same functionality**. These concepts are known as the Unix philosophy.

Unix, Unices, Linux

- Numerous Unix variants have sprung up over the years, some academic and some commercial.
  - BSD, Solaris, HP-UX, …
  - Linux is unix-like, not Unix
    - Started as a hobby project by Linus Torvalds
    - Made useful by existence of free, open source software tools (like the Gnu project)
- The OS consists of two major parts: the kernel and everything else
  - Kernel: master control program, starts and stops processes, handles low-level file/disk access, etc.
  - Many modular tools and programs
  - Every program runs within its own ***process***
  - User interacts through a command shell

Strengths and Weaknesses

- Strengths
  - Underlying philosophy has proven wildly successful
  - Unix is a ***very*** robust OS (Linux approximately so)
  - Simple tools can be linked together to do complex things; ***Unix makes this easy***
- Weaknesses
  - Names of many commands/programs are famously cryptic
  - Online help in the form of ***man pages***, which are really designed to remind experts of details they have forgotten, not teach novices how things work.

Programs = Little Black Boxes

- Most tools you use in Unix/Linux are really external programs (not a shell command or part of the kernel).
  - There is (almost) no distinction between “built-in” and “external” commands/programs
- Most tools communicate with the outside world in 4 ways
  - They get arguments on the command line
  - They receive input from standard input
  - They send output to standard output
  - (They also send error messages to standard error).
- Small, reusable pieces that you can assemble in any way you like to do complex tasks.

Examples of Tools

- `ls` print a listing of files in a directory
- `mv` move or rename files
  - Example:

    jeff% `ls SRTM`

    SRTM1/ SRTM3/
    - "jeff%" is a prompt from the shell, telling me it is ready for input
    - I typed "`ls SRTM`"
    - The ls program produced some output: "SRTM1/ SRTM3/"
- Note that Unix treats even directories, inputs and outputs as files.
- You might think these are "low level" functions of the operating system, but each exists as an independent program. They take parameters (given as command line arguments), and send output to a "standard output" file

What is the Shell?

- The shell is a user interface. It is a program that interprets the commands you type and executes them. It also provides output in some useful form (to a window on your screen)
  - It sends output to "**standard output**"
- The shell doesn't care whether its input comes from the keyboard or from a file
  - It takes input from "**standard input**"
  - As far as Unix is concerned, the keyboard is just another file.
- Many shells can be running at once, each with its own little world inside it.
  - You can start up one or more *sub-shells*, which do something and report output back to your shell.

Which Shell?

- There are many different shells
  - Bourne shell (sh)
  - C shell (csh) syntax is more like the C language
  - ***tcsh (tcsh) is really like the C shell, except it is free***
  - Bash (the Bourne Again SHell) popular with Linux
  - More shells: ksh, zsh, …
- Which shell is the best?
  - *Which is better, rock or jazz?*
  - I will use tcsh in my examples
- Which is your default shell?
  - Our Virtual Boxes: tcsh
  - My Linux cluster: tcsh
  - A standard MacOSX machine: bash
  - You can change your default shell

Some basics: directories

- Files are organized into directories, like in every other computer system. You might refer to a file like this:

  `/srtm/version2/SRTM3/Africa/S35E025.hgt.zip`

- ***Names are case-sensitive!*** `Jeff.dat ≠ jeff.dat`
- Unix has two ways of specifying files/directories:
  - Full pathnames

    `/home/jeff/junk_files/bork.dat`

    / is the very top of the directory tree
  - Relative pathnames

    `bork.dat`

    `junk_files/bork.dat`
    - Relative pathnames are relative to the current directory

Some basics: current directory

- The ***current directory*** is the directory you are sitting in right now
  - In a graphical system, this is kind of like the top window in your "Windows Explorer" or "Finder".
  - If you create a new file, it will be in this directory unless you tell the shell otherwise.
  - You saw the same thing in MATLAB.
- Some directory commands:

`cd junk_files`	(change current directory)
`pwd`	(print working directory)

- Special directory symbols:

`.`	(the current directory)
`..`	(one level up)
`~`	(your home directory)

Some basics: Rules for mv

- The mv command actually will do two things: move files to another directory, or rename files
  - Which one depends on what the last argument is
- Renaming a file:

  `mv file1 file2`

  Changes the name of file1 to file2
- Moving a file:

  `mv file1 dir`

  Move the file file1 to directory dir

  `mv file1 file2 dir`

  Move the files file1 and file2 to directory dir
- The mv command makes the decision based on whether the last argument is a directory or not. If you give it more than two filenames, then it will exit with an error if the last argument is NOT a directory.

Some basics: wildcards

- It is really useful to be able to match several filenames at once. For example

```
mv bork.dat fubar.* junk_files
```

  - The wildcard * matches any number of characters, so the line above would match these files:
    - `fubar.txt`, `fubar.job`, `fubar.1`
  - But ***not*** the file `fubar1.txt`
- Wildcards:

`*`	(match 0 or more characters)
`?`	(match 1 character)

- What does this match?

```
12*alaska*.??d
```

- Here is a fancier wildcard, which matches a range of characters:

```
clgo20090[1-6]??.dat
```

```
.[a-z]*
```

Wildcard Quiz

	foo1.dat	bork.dat	foobar.txt	My_files.txt
foo*				
*.txt				
_				
?o??.*				

Did you ace the quiz?
Extra Credit:

foo?.*

[a-z]*

*t

For some reason, evaluating wildcards is called “globbing”

Some basics: the path

- When you type something, the shell will try to execute it as a program. For example:
  - jeff% `rm bork.dat`
  - The shell breaks it down this way:
    - Program to run: `rm`
    - Argument(s) passed to program: `bork.dat`
  - This particular command removes (`rm`) the file named `bork.dat`
  - How does the shell know where to find the program `rm`?

The path: path or PATH

- It uses a special variable called the ***path***.
- Both csh and tcsh have two ways to set the path. The sh and bash shells do the same thing in a different way.

```
setenv PATH /gipsy/bin:${PATH}
set path = (/gipsy/bin $path)
```

- The shell internally maintains a list of all ***executable programs*** in these directories.
  - The command `rehash` refreshes this list.
- It looks in the first directory in the list first.
  - If you have two programs with the same name, you need to know the path to know which one will be executed!

Everything is Scriptable

- Why do repetitive tasks yourself? That's what you have a computer for.
- This is equally true for shell scripts and MATLAB programs (yes, the MATLAB command window is a kind of shell)
- A script doesn't have to be complicated to be useful, it just has to do something reliably and more easily than typing.
- Here's a script I use to update the online copy of my website from the master code on my own computer:

```
#!/bin/tcsh -f
cd ~/Sites/jeff
rsync --rsh=ssh -av * denali.gps.alaska.edu:/home/jeff/Web
```

The tilde (~) is a special character that means your home directory

Everything is Scriptable

- Don't
  - Type a long series of commands over and over again
  - Copy and paste a long series of commands
- Do!
  - Any set of commands you type at the prompt can be saved and made into a script for repeating later
  - Recording the commands you type can be a good way to get started at making a simple program.
  - Learn to use variables to make your series of commands more general and improve automation.

Redirecting Input and Output

- Many programs are designed to take input from **standard input**, and send output to **standard output**.
  - By default, these are the keyboard and screen
  - You can change that!
- This is called I/O redirection
  - > means redirect output
  - < means redirect input
  - You can use both at the same time
- Examples:
- Send the output of something to a file:
  - `ls *.dat > myfiles`
  - `psxy in.dat > map.ps`
- Take input from a file
  - `sort < myfiles`
- Both at the same time:
  - `myprogram < commands > output`

Unleashing the Shell: the Pipe

- A “Pipe” is a way for the output of one program to be sent as the input to another program.
- A vertical bar (|) indicates a pipe
- You can pipe together as many programs as you like, as long as each one reads from standard input and writes to standard output.

- Examples:
- Use a pager:
  - `ls *.dat | more`
- Count files of a given type:
  - `ls *.dat | wc -l`
- Sort listings:
  - `ls –l | sort –nr –k 5`
- More
  - `ls –l | sort –nr –k 5 | more`

Shell Variables

- Regular variables
  - Purely internal to each shell
  - Generally lowercase
- How to set a variable
  - `set counter = 3`
  - `@ counter = $counter + 1`
- How to access it
  - echo $counter
- A neat trick:
  - `set datfiles = `ls *.dat``

- Environment variables
  - Can be accessed from within programs started by shell
  - Generally uppercase
- How to set:
  - `setenv PRINTER blackburn`
- How to access
  - `echo $PRINTER`

Control Structures

- Conditionals
- if (test) then … endif
  - if (test) statement
  - if – then – endif
  - if – then – else – endif
  - if – then – elseif – endif
- Test can be a comparison to a numerical or string value, or special stuff:
  - `if ( $val == 0 ) echo "Zero!"`
  - `if ($count > 1) echo positive`
  - `if ( -d $file ) echo "$file is a directory"`

Loops

- foreach val (*) … end
- while (test) … end

```
if (! -e ${run_id}.qm) then
  echo MERG_QM died.
  exit
endif
```

```
foreach sat ($delsats)
  echo "Deleting satellite:  $sat"
  /bin/mv -f ${run_id}.qm tmp
  del_qm -i tmp –d $sat -o ${run_id}.qm
  /bin/rm -f tmp
end
```

Syntax: MATLAB vs. tcsh

MATLAB	tcsh
i = 20;	set i = 20
i = i + 1	@ i = $i + 1

if (a == b)
  i = i + 1;
  disp(num2str(i));
end
	if ($a == $b) then
	  @ i = $i + 1
	  echo $i
	endif

If (a ==b) disp("a and b are equal"); end	if ($a ==$b) echo "a and b are equal"

for i = 1:10
  disp(['The number is ' num2str(i)]);
end
	foreach i (1 2 3 4 5 6 7 8 9 10)
	  echo "The number is $i"
	end

The shell also has a “while” control structure, with the same syntax as MATLAB

If tests

- The shell has a different set of built-in tests than MATLAB does.
  - Extra: all sorts of tests on files
  - You can also make sophisticated tests by stringing together shell programs with pipes

Test	Function
-e $filename	True (1) if the file $filename exists
-z $filename	True (1) if the file $filename is zero size
-d $filename	True (1) if the file $filename is a directory

The amazing *foreach*

- The foreach command executes a loop once for each element of a list. The list could be a list of files, or a list of command line arguments, or a list of sites, …
- The list can be in a variable or from ‘globbing’:

```
foreach file ( [R,r]edoubt* )
   if ( -d $file ) echo "File $file is a directory"
   if ( -z $file ) echo "File $file is zero length"
   if ( $file == `basename $file .dat`.dat ) \
        echo "File $file is a data file"
end
```

The *"backtick"* and other quotes

- The previous example used a command surrounded by "backticks" (`). This command is evaluated and the whole thing is then replaced by the result of that command.
- Double quotes are used for strings. If there is a variable inside the string, the value of the variable will be used.
- Single quotes are like double quotes, except that variables are not de-referenced.
- A single quote inside double quotes, and a double quote inside single quotes, are just treated as ordinary characters.