# Data Analysis in Geophysics
# ESCI 7205

## Class 6

Bob Smalley

## Basics of UNIX commands

Some more useful commands

# Basics of the Unix/Linux Environment

# Additional useful commands

## wc: word count

```
%wc suma1.hrdpicks
37753   253998 3561084 suma1.hrdpicks
```

Reports number of lines, words (separator=space), and characters in the file.

# Additional useful commands

**cmp**: compare files and report equal or not.

```
496:> cmp hw1.txt hw1a.txt
hw1.txt hw1a.txt differ: char 175, line 12
497:>
```

No output if the same, else reports byte and line numbers at which the first difference occurred (starts at 1).

# Additional useful commands

## `diff`: show differences between two files

```
498:> diff hw1.txt hw1a.txt
12c12
< 2) [2] Create a directory in your account for this course - you
might call it something like ESCI7205.
---
> 2) [2*] Create a directory in your account for this course -
you might call it something like ESCI7205.
14c14
```

Sometimes useful (if files completely different is mess).
Less than sign, suck, for to left, file 1, greater than sign, spit, for file to right, file 2.
(if have extra lines, will re-synch, afterwards.)

# Additional useful commands

## sort: alphabetical or numeric sort function

### sort alphabetically

```
525:> more samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE
COGO -31.15343 -70.97526 CAP [3] 1993 1996 2002 CHILE OKRT
MORA -30.20823 -70.78971 CAP [3] 1993 1996 2002 CHILE OKRT
MOR2 -30.20823 -70.78971 CAP [?] CHILE OKRT
TOFO -29.45939 -71.23842 CAP [4] 1993 1996 2001 2002 CHILE OKRT
SILA -29.24037 -70.74956 CAP [3] 1993 1996 2002 CHILE OKRT
HUAS -28.47848 -71.22235 CAP [3] 1993 1996 2002 CHILE OKRT
. . .
526:> sort samgps.dat
ABAC -24.433 -66.217 SAGA [-] ARGENTINA NORT
ABEL -25.667 -65.483 SAGA [-] ARGENTINA NORT
ACOL -30.78337 -66.21338 CAP [3] 1993 1997 2000 ARGENTINA OKRT
ACPM -33.447181 -70.537434 CAP2 [c] continuous (2005-) CHILE
ADLS -26.08449 -67.4191 CAP [2] 1993 1997 ARGENTINA OKRT
AGAL -24.317 -66.467 SAGA [-] ARGENTINA NORT
. . .
```

```
506:> more flong.dat
2
1
20
9
11
10
```

sort default is alphabetical

```
513:> sort flong.dat
1
10
11
2
20
9
```

Alphabetically "10" comes before "2 " (note the space) because is is sorting on the first character, then the second…

```
alpaca.ceri.memphis.edu506:> more flong.dat
2
1
20
9
11
10
```

# Sort numerically

```
514:> sort -n flong.dat
1
2
9
10
11
20
```

# Sort numerically on second column or position

```
530:> sort -n -k 2 samgps.dat | head -3
W01A -87.41565 -149.43328 WAGN [2] 2002 2005 OKRT
W01B -87.41518 -149.44311 WAGN [2] 2002 2005 OKRT
W02A -85.61192 -68.55633 WAGN [3] 2002 2005 2008 OKRT
```

# Sort numerically and remove duplicates

```
% more dirList
812
799
812
825
799
825
% sort –n -u dirList
799
812
825
```

# Sort using a different separator (default is white space)

```
% sort -n -t"," –k3 SUMA.NEW.loc.csv
1,1,1918,9,22,9,54,49.29,,,-1.698,,,98.298,,,15.0,,0.0,,0.0,ehb
10,10,1935,11,25,10,3,7.39,,,,5.886,,,93.737,,,35.0,,0.0,,0.0,ehb
100,100,1964,1,7,0,50,7.03,,,,1.801,,,99.483,,,15.0,,5.0,,0.0,ehb
```

# More sort fun

```
# cat tsort.dat
b 1 1 3
a 1 1 3
c 1 2 5
b 2 1 4
c 1 1 5
# sort -k 2 tsort.dat
a 1 1 3
b 1 1 3
c 1 1 5
c 1 2 5
b 2 1 4
# sort -k 2,3 tsort.dat
a 1 1 3
b 1 1 3
c 1 1 5
c 1 2 5
b 2 1 4
# sort -k 2,3 -u tsort.dat
b 1 1 3
c 1 2 5
b 2 1 4
#
```

# To figure all this out

# Read the man page to see what else it will do.

```
NAME
     sort - sort, merge, or sequence check text files

SYNOPSIS
     /usr/bin/sort  [-bcdfimMnru]  [-k keydef]   [-o output]   [-
S kmem]  [-t char] [-T directory] [ -y  [kmem]] [-z recsz] [+pos1
[-pos2]] [file...]

     /usr/xpg4/bin/sort [-bcdfimMnru] [-k keydef] [-o output]  [-
S kmem]  [-t char] [-T directory] [ -y  [kmem]] [-z recsz] [+pos1
[-pos2]] [file...]
```

From now one, you will be expected to read the man pages for all the commands we have used or will use to see how to use them and what they will do.

You have to read the man pages for all the commands

and then think of all the side effects each can have to figure out what you can do.

# Time

`cal`: displays a calendar

Default is current month

Will also display the year

Good way to figure out day of year (doy - often incorrectly called julian day) using the `-j` flag

```
$ cal
    September 2009
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

```
$ cal -j
      September 2009
 Su  Mo  Tu  We  Th  Fr  Sa
         244 245 246 247 248
249 250 251 252 253 254 255
256 257 258 259 260 261 262
263 264 265 266 267 268 269
270 271 272 273
```

# Time

## date: displays date and time

```
%date
Wed Aug 27 17:12:01 CDT 2008


%date –u –r 10
Thu Jan  1 00:00:10 UTC 1970
```

# Basic Math

**bc**: basic math calculator

`+, -, *, /, %, ^, sqrt`

also test Boolean expressions and
`>,<,==,!=`, etc.

quit or CNTL-D to exit

**expr**: evaluate the expression
more powerful, command line calculator for integer math and string comparison

**units**: unit conversion

# Job Control

**top**: lists all processes currently running

**ps**: process status, another way to display process identification numbers (PID)

```
585:> ps -aef
     UID   PID  PPID  C    STIME TTY      TIME CMD
    root     0     0  0   Jun 13 ?       0:04 sched
    root     1     0  0   Jun 13 ?       0:10 /etc/init -
. . .
rsmalley  9790  9580  1 23:17:45 pts/12 0:00 ps -aef
rsmalley  9578  9575  1 18:50:33 ?       0:04 /usr/lib/ssh/sshd
```

**kill**: allows you to hard kill processes by their PID (get from **ps**). You can only kill your own jobs (unelss you are root).

```
586:> kill -9 9578
```

`CNTL-Z`: suspends the current job (use to end man program).

`fg`: resume job and runs it in the foreground (grabs screen).

`bg`: resume the job in the background (initially set on the command line by adding the `&` to the end of the command/script).

`jobs`: lists all jobs running in the background, including their PIDs.

# Finding/Searching

`find`: search for files

Syntax: `find path expressions`

Read the (confusing) man page.

Is powerful UNIX tool.

Starts where we are (.), looks there and below in the directory structure.

```
508:> find . -name cap_ice\* -print
./dem/cap_icezooms_.5v2.ps
. . . .
/from_midtown/dem/cap_ice_.5v2.ps
```

Don't need the "-print" anymore (but you may see it).

In old days, found the files, but needed instructions on what to do with them (did not automatically send to standard out, kept it a secret).

Finding files in the working directory and below which have been modified in the last n (=-1 here) days:

```
find . -mtime -1
```

List all the files and subdirectories from the subdirectories starting at root, with 777 permissions?

```
find / -perm 777
```

Use option "`-iname`" for a case insensitive search in working directory and below, by default find searches are case sensitive.

```
find . –iname "error"
```

Delete temporary files in working directory and below using `find` and `xargs` commands together with a pipe

```
find . -name "*.tmp" | xargs rm –f
```

`xargs` can be used to do whatever you want to each file found by the find command.

`xargs` along with `find` gives you immense power to do whatever you want (including stuff you don't want [and/or did not anticipate] – big opportunities for disaster) with each search result.

# Find all text files in working directory and below which contain the word Exception using `find` command

## Two ways to do it

```
find . –name "*.txt" –print | xargs grep “Exception”

find . –name "*.txt" –exec grep “Exception” ‘{}‘ \;
```

```
find . –name "*.txt" –exec grep “Exception” ‘{}‘ \;
```

The **-exec** action takes a Unix command (along with its options) as an argument. The arguments should contain {} (usually quoted), which is replaced in the command with the name of the currently found file. The command is terminated by a semicolon, which must be quoted (escaped) so the shell will pass it literally to the find command.

The **-exec** action in find is very useful, but since it runs the command listed for every found file it isn't very efficient. On a large system this makes a difference!

# Finding files only in current directory not searching on sub directories:

```
find . -maxdepth 1 -type f -newer first_file
```

Or

```
find . -type f -newer first_file -prune
```

Find all files in current directory and subdirectory, greater than some size using find command in Unix (and then do and `ls -l` on them using `-exec`)

```
find . -size +1000c -exec ls -l {} \;
```

Use the `c` after the number to specify the size in bytes, otherwise you will get (confused because `find -size` reports) results based on size of the file in disk blocks not bytes.

To find files using a range of file sizes, a minus or plus sign can be specified before the number.

The minus sign means "less than," and the plus sign means "greater than."

Suppose if you want to find all the files within a range you can use find command as below

```
find . -size +10000c -size -50000c -print
```

This find command example lists all files that are greater than 10,000 bytes, but less than 50,000 bytes:

Find files which are some days old and greater than some size in Unix.

Very common scenario where you want to delete some large old files to free some space in your machine. You can use combination of "**-mtime**" and "**-size**" to achieve this (and then do and `ls -l` on them using **-exec**).

```
find . -mtime +10 -size +50000c -exec ls -l {} \;
```

This command will find which are more than 10 days old and size greater than 50K.

# Common Find Gotcha:

If the given expression to find does not contain any of the action primaries `-exec`, `-ok`, or `-print`, the given expression is effectively replaced by:

```
find \( expression \) –print.
```

The implied parenthesis can cause unexpected results.

# i.e., consider these similar commands:

```
$ find -name tmp -prune -o -name \*.txt
./bin/data/secret.txt
./tmp
./missingEOL.txt
$ find -name tmp -prune -o -name \*.txt –print
./bin/data/secret.txt
./missingEOL.txt
```

The lack of an action in the first command means it is equivalent to:

```
find . \( -name tmp -prune -o -name \*.txt \) –print
```

This causes tmp to be included in the output.

The implies parenthesis are important.

# i.e., consider these similar commands:

```
$ find -name tmp -prune -o -name \*.txt –print
./bin/data/secret.txt
./missingEOL.txt
$ find -name tmp –prune -print -o -name \*.txt -print
./bin/data/secret.txt
./missingEOL.txt
./tmp
```

For the second find command (top one here) the normal rules of Boolean operator precedence apply, so the pruned directory does not appear in the output.

Compare with bottom command with action (-print) for both arguments to –o (boolean or).

You can use "awk" (will do it next) in combination of find to print a formatted output e.g. next command will find all of the symbolic links in your home directory, and print the files your symbolic links points to:

```
find . -type l | xargs ls -ld | awk '{print $10}'
```

"." says starts from current directory and include all sub directories, "-type l" says list all links

# Finding/Searching

`find`: search for files

To make this really useful (as if what we have seen already is not dangerous enough), we need a way to search for patterns in the filenames (or within files).

use regular expressions (not shell wildcards).

So now we are going to have two kinds of special characters, or metacharacters.

Those that mean something special to the shell (such as the "$" on a shell or environment variable or the "/" in a path).

Those that are used to specify a pattern as a regular expression.

And will need a way to "turn off", or escape, the special meaning as either a shell or regular expression metacharacter.

These rules are global.

Say I want to look for all files that start with a "v" or "V", and have any "extension" (the "`.dat`", part of the file name).

```
514:> find . -name *olcanoes*
find: bad option Volcanoes.dat
find: path-list predicate-list
```

This does not work for some reason.

The `find` command is not "using" the "*" properly.

(This is because the shell recognized it as a shell wildcard and got hold of it first and did something with it instead of passing it on to `find`.)

# Metacharacter Escaping

# We have to "escape" the shell's interpretation of the "*", so it gets passed to find to be used as a regular expression there. Use \

```
-bash 520 dem # find . -name Volcanoes.dat
./Volcanoes.dat
-bash 521 dem # find . -name *olcanoes*
find: volcanoes: unknown option
-bash 522 dem # find . -name \*olcanoes*
./volcanoes
./Volcanoes.dat
./volcanoes.f
-bash 523 dem # find . -name \*olcanoes\*
./volcanoes
./Volcanoes.dat
./dem/volcanoes.f
-bash 535 dem # cd ..
-bash 536 unixside # find . -name \*olcanoes\*
./dem/volcanoes
./dem/Volcanoes.dat
./dem/volcanoes.f
```

There are three ways to escape metacharacter interpretation.

Backslash "\", escapes the next character from interpretation [the first time \ is encountered], i.e. the next character is treated as a regular character.

```
\*olcanoes\*
'*olcanoes*'
"*olcanoes*"
```

Works for all programs (the shell is just another program).

```
\*olcanoes\*
```

So the splat is not used as a wildcard by the shell (all the files in the directory), the first program to encounter it, and it is passed as a * to the program find where it is (finally) used as a wildcard (any combo of characters).

The backslash “\”, is the strongest method to escape a character.

It works everywhere.

If you want to place text on two or more lines for readability, but the program expects one line, you need a line continuation character. Just use the backslash as the last character on the line:

```
% echo This could be \
a very \
long line\!
This could be a very long line!
%
```

This *escapes* or *quotes* the end of line (eol,<CR>) character, so it no longer has a special meaning.

(In the above example, the backslash before the exclamation point is necessary if you are using the C shell, which treats the "!" as a special character.)

Another example of the thought processes involved in taking advantage of the power of Unix.

What would you enter if you were looking for a file named "*olcanoes"?

(rhetorical question).

Next two methods.

Protect metacharacters from interpretation by the shell only.

Single quotes.

‘quote’, ‘escape’, or ‘protect’ everything inside the quotes them from the shell.

```
'*olcanoes*'
```

Next two methods.

Protect metacharacters from interpretation by the shell only.

Double quotes.

```
"*olcanoes*"
```

"quote", "escape", or "protect" everything inside the quotes from the shell

except variables and backquoted expressions (`)

(we will get to that soon),

which are expanded by the shell and replaced with their value.

# Quote syntax

What happens if you forget the quotes depends on the shell and which quotes you forgot/used incorrectly.

In csh/tcsh, the variable b below would be set to Hello, and the shell would ignore the string world.

```
545:> echo $0
-tcsh
546:> set b=hello world
547:> echo $b
hello
```

In sh/bash, the variable b below would be set to hello, and the shell would try to run the command world producing an error message.

```
alpaca.ceri.memphis.edu501:> /bin/sh
$ b='hello world'
$ echo $b
hello world
$ b=hello world
world: not found
$
```

# Single vs. double quotes example.

```
503:> set a = A
504:> echo $a
A
505:> set b = 'letter $a'
506:> echo $b
letter $a
```

Shell did not expand variable a to its value. Treated string $a literally as the characters $a.

```
507:> set c = "letter $a"
508:> echo $c
letter A
509:>
```

Now shell expanded variable a to its value, A, and passed value on.

Works same in csh/tcsh and sh/bash.

# Back to commands

`grep`: search for a pattern inside files (or standard in, or <<).

(general regular expression,
general regular expression processor,
. . . )
(name created using UNIX naming philosophy)

highly useful and it is worth your time to sit down with the man page.

# Simple examples

Find the string PELD in the file samgps.dat.

`grep` sends all lines in input (standard in, file [don't need redirect, but can use it], or pipe) that contain the string "PELD" to the standard out.

```
533:> grep PELD samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE OKRT
```

Takes standard Unix "regular expressions", of which we have seen a few.

This finds all the lines that start with a "P" ("^" is the metacharacter for the beginning of a line) and sends them to standard out.

```
534:> grep ^P samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE OKRT
PSTO -28.17157 -69.79377 CAP [3] 1993 1996 2002 CHILE OKRT
PNAZ -26.14822 -70.65368 CAP [3] 1993 1996 2001 CHILE OKRT
```

# Finds all the lines with "ARGEN" and sends them to standard out.

```
535:> grep ARGEN samgps.dat
TND2 -37.3 -59.2167 CAP|C1960 [0] ARGENTINA NORT dropped
ZAPX -38.82775 -70.02394 CAP|C1960 [?] ARGENTINA OKRT
```

# Finds all the lines with "3 ARGEN" and sends them to standard out.

```
510:> grep "3 ARGEN" samgps.dat
ZAPL -38.82775 -70.02394 CAP|C1960 [4] 1993 1997 1997 2003 ARGENTINA
BSON -42.01391 -71.20485 CAP|C1960 [3] 1993 1997 2003 ARGENTINA OKRT
```

# What does this do?

```
grep CAP.*ARGENTINA samgps.dat
```

```
-bash 538 geolfigs # grep CAP.*ARGENTINA samgps.dat
TNDL -37.32423 -59.08637 US|CAP|C1960 [7] 1993 1997 1997 1998 1998 2001
2003 ARGENTINA OKRT Tandil
TND2 -37.3 -59.2167 US|CAP|C1960 [0] ARGENTINA NORT dropped
ZAPL -38.82775 -70.02394 US|CAP|C1960|C2010 [5] 1993 1997 1997 2003 2010
ARGENTINA OKRT Zapala
```

Finds all lines with the string `CAP` followed by 0 or more characters then the string `ARGENTINA`.

(like find but for file contents rather than file names. Combined with regular expressions is very powerful.)

I probably use grep every time I'm on a Unix system!

# Command Substitution

# Command substitution

Invoked by using the back or grave quotes (actually French grave accent, `). (ex in bash)

```
a=`echo "hello world." | wc`
```

What does this do?

Command substitution tells the shell to run what is inside the back quotes and substitute the output of that command for what is inside the quotes.

So the shell runs the commmand

```
echo hello world. | wc
```

Producing (which you don't see)

```
1  2  13
```

takes the output (the "1 2 13" above), substitutes it for what is in the back quotes (`echo "hello world | wc.`), and sets the shell variable equal to it

```
a=`echo "hello world." | wc`
```

does this (is as if you typed)

```
a='1  2  13'
```

Compare/contrast this to what the pipe ("|") does.

# Trivial example of substituting into a switch. As a variable (why do I need the quotes?)

```
509:> set options='-la'
510:> ls $options
total 2203891
-rw-rw-rw-  1 rsmalley user 54847 Mar  7  2009 CHARGE-2002-107
drwxr-xr-x 94 rsmalley user 31232 Sep 19 17:46 .
dr-xr-xr-x  3 root     root     3 Sep 19 17:51 ..
drwxr-xr-x 2 rsmalley user    512 Oct  1  2004 .acrobat
-rw-r--r-- 1 rsmalley user    237 Oct  1  2004 .acrosrch
```

## Directly

```
511:> ls `echo $options`
total 2203891
-rw-rw-rw-  1 rsmalley user 54847 Mar  7  2009 CHARGE-2002-107
drwxr-xr-x 94 rsmalley user 31232 Sep 19 17:46 .
dr-xr-xr-x  3 root     root     3 Sep 19 17:51 ..
drwxr-xr-x  2 rsmalley user    512 Oct  1  2004 .acrobat
-rw-r--r--  1 rsmalley user    237 Oct  1  2004 .acrosrch
```

# Variables are set to the final output of all commands within the back single quotes. In csh and bash.

```
509:> set options=‘-la’
513:> set ops = `echo $options`
514:> echo $ops
-la
515:>

-bash 662 dem # options='-la'
-bash 663 dem # echo $options
-la
-bash 664 dem # ops=`echo $options`
-bash 665 dem # echo $ops
-la
-bash 6668 dem #
```

This is a very useful and powerful feature.

# Standard Error

# What does >&! mean?

We have already seen the > (it means redirect output) and ! (it means clobber any existing files with the same name).

So far we have discussed standard-in and standard-out.

But there is another standard output stream - introducing

standard-error.

```
carpincho:ESCI7205 smalley$ ls nonexitantfile
ls: nonexitantfile: No such file or directory
```

The message above shows up on the screen, but in this case it is actually standard-error, not standard-out.

```
$ ls nonexitantfile > filelist
ls: nonexitantfile: No such file or directory
$ ls -l filelist
-rw-r--r--  1 smalley  staff  0 Sep 21 16:01 filelist
$
```

Can see this by redirecting standard-out into a file. The error message still shows up and the file with the redirected output is empty.

(it has 0 bytes, our standard Unix output, ready for the next command in pipe.)

>& is the csh/tcsh syntax for redirecting standard-error to standard-out.

(else standard-error it goes to the screen)

Append standard-error to standard-out

>>&

You can't handle standard-error alone.

(With what we have seen so far, in csh/tcsh.)

The sh/bash syntax uses 1 to [optionally] identify standard-out and 2 to identify standard-error.

To redirect standard-error in sh/bash use

2>

To redirect standard-error to standard-out

>&

(! has usual meaning – clobber)
To pipe standard-out and standard-error

>& |

# Redirect standard-error to file

```
$ls nonexitantfile > filelist 2> errreport
$cat errreport
ls: nonexitantfile: No such file or directory
$
```

Redirect standard-error to standard-out into a file. Can't do second redirect to a file. Use subshell command format, redirect output subshell to file. combofile has both standard-out and standard-error.

```
$(ls a.out nonexistantfile >&)>combofile
$more combofile
nonexistantfile: No such file or directory
a.out
$
```

# Subshells

# Combining stdout and stderr In tcsh

the best you can do is (Unix think)

```
( command > stdout_file ) >& stderr_file
```

which runs "command" in a subshell.

stdout is redirected inside the subshell to stdout_file.
both stdout and stderr from the subshell are redirected to stderr_file, but by this point stdout has already been redirected to a file, so only stderr actually winds up in stderr_file.

Subshells can be used to group outputs together into a single pipe.

sh/bash

(when a program starts another program [more exactly, when a process starts another process], the new process runs as a subprocess or child process. When a shell starts another shell, the new shell is called a *subshell*.)

# So in our earlier example using command substitution we could have done

```
(my_convert_program input file1; cat input file2) |\
my_processing_program << END
```

Where we are using the \ to continue the command on the second line.

The semi-colon "**;**", allows us to enter multiple commands, to be executed in order, in the sub-shell

(In typical Unix fashion, the ";" works in the shell and shell scripts also. Try it.) .

# Misc commands and stuff

`finger` - find out information about users

`who` - who is currently logged in and

`w` - who is currently logged in what are they doing

`whoami` – reports your username

`id` - tells you who you are (username, uid, group, gid)

`uname` – reports basic system information

`whois` – reports information about hosts

`which/whereis` - locates commands/files

`whatis` - gives brief summary of a command

`talk` - chat with other users on the system
(instant messaging of the 60's! There is nothing new under the sun.)

`write/wall` - send messages to all users

# Types of commands

Built-in commands: commands that the shell itself executes.

Shell functions: self-contained chunks of code, written in the shell language, that are invoked in the same way as a command.

External commands: commands that the shell runs by creating a separate process.

# Special shell variables

$< : special BSD Unix csh command that essentially acts as read except it is not white space delimited

```
set name = "$<"
```

instead of

```
read firstname lastname
```

# Special shell variables

$# : the number of arguments passed to the shell.

Useful when checking calling arguments (did you enter the correct number of them?), writing if:then:else blocks and loops.

We will cover this more later.

# Special shell variables

"$@": (need quotes) represents all command line arguments at once, maintaining separation, same as

"$1" "$2" "$3"

"$*" : (should have quotes) represents all command line arguments as one, same as

"$1 $2 $3 $4"

Without quotes, $* equals "$@"

```
$ arglist.sh first second\ third

Listing args with "$@":
Arg #1 = first
Arg #2 = second
Arg #3 = third
Arg list seen as separate words.

Listing args with "$*":
Arg #1 = first second third
Entire arg list seen as single word.

Listing args with $* (unquoted):
Arg #1 = first
Arg #2 = second
Arg #3 = third
Arg list seen as separate words.
$
```

# Special shell variables

`$-` : Options given to shell on invocation.

`$?` : Exit status of previous command.

`$$` : Process ID of shell process.

`$!` : Process ID of last background command. Use this to save process ID numbers for later use with the wait command.

# Special shell variables

`$IFS` : Internal field separator
the list of characters that act as word separators.
Normally set to space and newline (maybe tab) (is a bash, not tcsh variable).

```
$ echo $IFS

$ echo $IFS | od -x
0000000 0a00
0000001
$ echo $IFS | od -c
0000000  \n
0000001
$
```

# Special files

`/dev/null` : null device is a special file that discards all data written to it (but reports that the write operation succeeded), and provides no data to any process that reads from it (yielding EOF immediately).

Also know as the bit bucket, black hole, or a WOM (write only memory).

# Special files
# /dev/tty: redirects script's stnd-in to the terminal

## Script

```
#!/bin/sh
printf "Hello.  My name is hdmacpro.  What is yours?\n"
read name < /dev/tty
printf "Nice to meet you %s.\n" $name
printf "Hello.  My name is hdmacpro.  What is yours?\n?"
read name
printf "Nice to meet you %s.\n?" $name
```

## Run it

```
517:> tsttty.sh
Hello.  My name is hdmacpro.  What is yours?
bob
Nice to meet you bob.
Hello.  My name is hdmacpro.  What is yours?
?Bob
Nice to meet you Bob.
? 518:>
```

# AWK/NAWK

Quick intro for HW

awk :
[Aho, Weinberger, Kernighan]
new-awk = nawk

Powerful pattern-directed scanning and processing language.

So powerful that we will devote a lot of time to it.

One of the most used Unix tools.

For now we will present the bare basics that will allow us to start processing data.

`nawk` reads a file and processes it a line at a time.

The input line is parsed into fields separated by spaces or tabs.

The fields are addressed as `$1, $2,` etc.

`$0` is the while line.

Here is the basic syntax for performing simple nawk processing from the command line.

```
nawk '[/regex/...] {print $n, $m, …}' file
```

Where /regex/ is an optional (the [ ]s) regular expression contained within forward slashes "/"s.
(There can be more than one {the …}, combined logically &&, ||, !).

`$n, $m`, etc. are the columns of the file to print out

file specifies the input file

# A basic, and useful, nawk example.

Print out a number of columns of a file (say the lat and long contained in columns 6 and 7) for plotting by GMT).

Here is the file mydatafile.dat

```
CAT     YEAR  MO DA ORIG TIME    LAT    LONG    DEP  MAGNITUDE
PDE     1973  01 05 123556.50   46.47 -112.73
PDE     1973  01 07 225606.10   37.44  -87.30   15  3.2
PDE     1973  01 08 091136.80   33.78  -90.62    7  3.5
```

What do we do about the first line (a "header", useful to humans, confuses programming.)

If a regular expression is specified, nawk will only print out the lines which contain a match to it. All the other lines in the example start with `PDE` – so use that to find data)

```
% nawk '/PDE/ {print $7, $6}' mydatafile.dat
-112.73 46.47
-87.30 37.44
-90.62 33.78
```

Prints out the seventh and sixth column (x,y order, the guys who wrote GMT came from mathematical plotting where x is usually first, not geographical processing where lat is usually first) for all lines in the file mydatafile.dat containing the character string "good data".

If we did not have the search for lines with /PDE/ it would print out the seventh and sixth column for all lines in the file mydatafile.dat.

```
% nawk ' {print $7, $6}' mydatafile.dat
LONG LAT
-112.73 46.47
-87.30 37.44
-90.62 33.78
```

Which would make GMT very unhappy.

# The input to nawk can also be piped or redirected from stdin.

```
% cat mydatafile.dat | nawk ' {print $7, $6}'
```

or

```
% nawk ' {print $7, $6}' << END
`cat mydatafile.dat`
END
```

(Although one would never do either of the above in practice!! Why?)

This is enough awk/nawk knowledge to do the homework.

# Shell Scripting

Basic scripting

What is a shell script?

It is a program that is written using shell commands

(the same commands you type to do things in the shell).

# When to use a shell script?

Shell scripts are used most often for combining existing programs to accomplish some small, specific job, typically one you want to run often/ multiple times.

Once you've figured out how to get the job done, you put the commands into a file, or script, which you can then run directly.

Why use shell scripts?

- Repeatability -

why bother retyping a series of common commands?

# Why use shell scripts?

## - Portability -

Once you have a useful tool, you can move you shell script from one machine/flavor Unix to another.

POSIX standard – formal standard describing a portable operating environment.

IEEE Std 1003.2 current POSIX standard.

# Why use shell scripts?

Simplicity

# Simplest shell script

You have to do the same N commands every day.

Put them in a shell script and just type the file/shell script name (to run it).

# Next simplest shell script

You have to do the same N commands every day but on a different input file each day.

Same as before, but now you have to pass the name of the file when you run it and have to refer to that file somewhere in the shell script.

# Next simplest shell script

You have to do the same N commands every day but on different input files each day.
You may also have to vary what needs to be done based on some properties of the files, etc.

Same as before, but now you also have to be able to test conditions and make decisions about what to do next based on the results of those decisions (if-then-else).

# Next simplest shell script

You have to do the same N commands every day but on different input files each day.
You may also have to vary what needs to be done based on some properties of the files, etc.

You will also need some way to repeat the process for many files (loops).

Some programming features that do not exist in shell scripts.

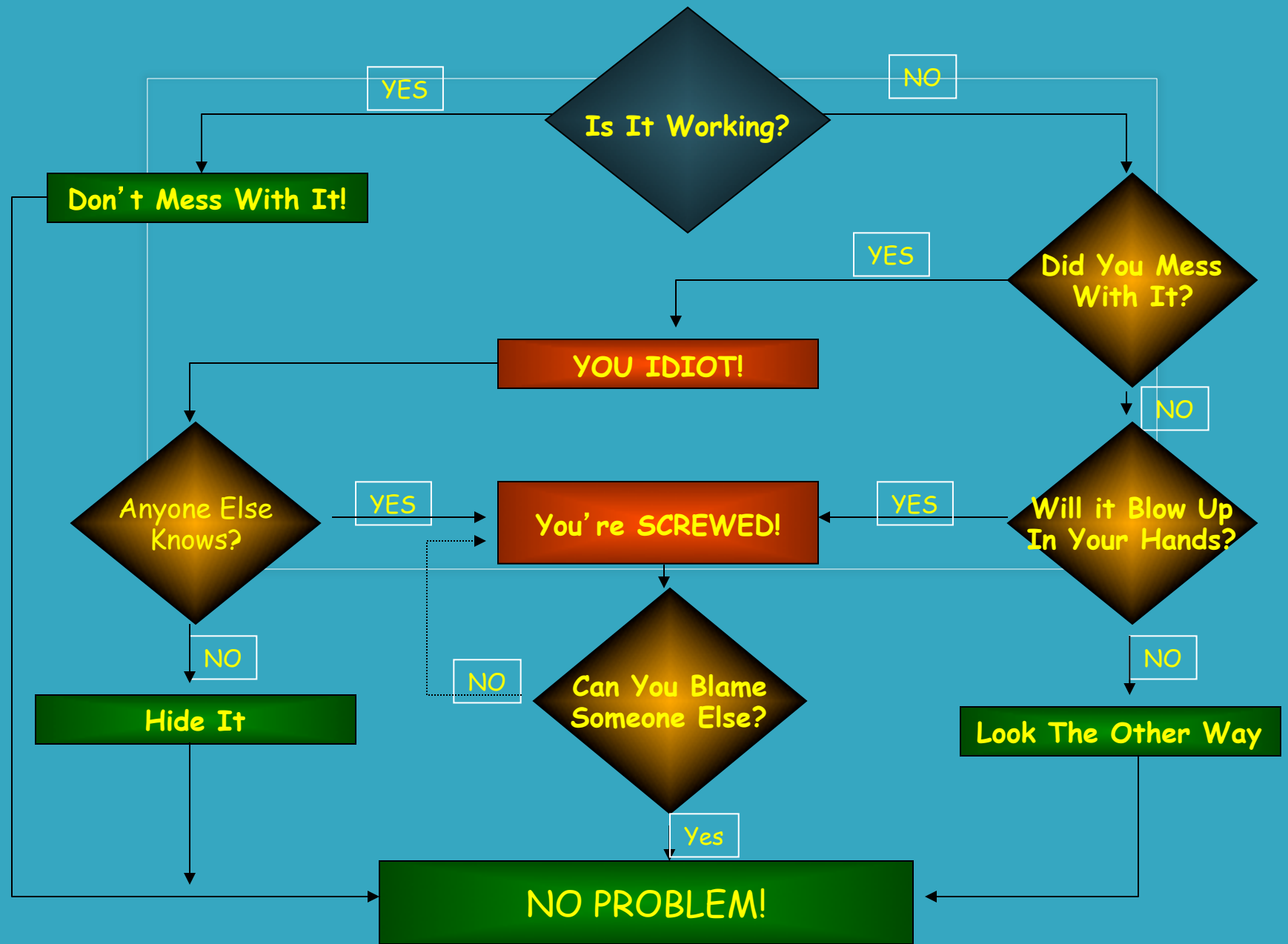
# Subroutines and Functions.

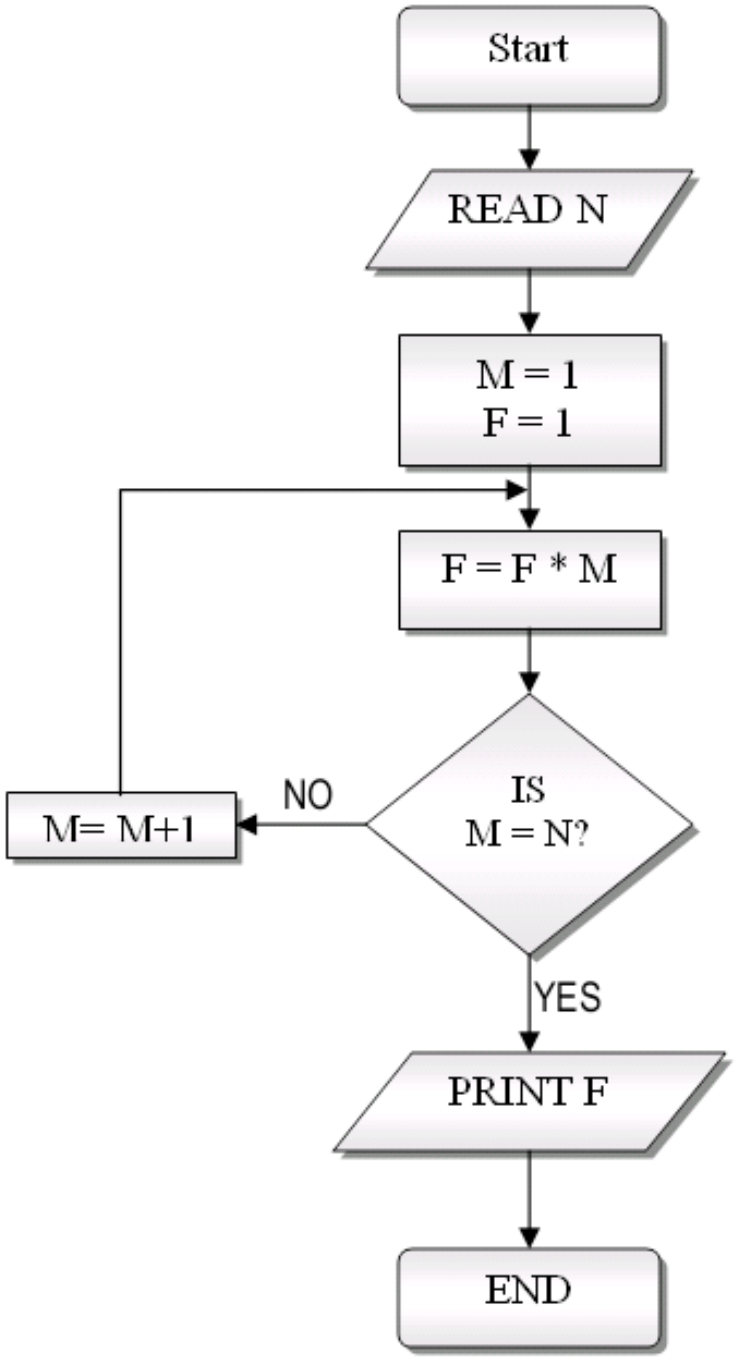
You have some tasks that need to be repeated in many of your programs.

You can write a general program to process these tasks and then use, or call them, them within other programs.

Subroutines and Functions are like little standalone programs that take input arguments and produce a result.
The main difference between them is how they give you the result.

# Flowchart For Problem Resolution






# Flowchart for computing N!

Has

- tests/decisions
- loop

# Interpreted vs. Compiled Languages

End members of methods for changing what *you* write in a "high-level language" (variables, mathematical operations, subroutines, etc. FORTRAN, C, C++, PEARL, etc.) into the individual machine instructions the computer's CPU executes (put numbers into CPU register, multiply them, store result in memory).

# Compiler

The name "compiler" is primarily used for programs that translate source code written in a high-level programming language to a lower level language such as assembly language or machine code (everything has to end up as machine code eventually).

Typically produces most efficient (in terms of run time) implementation of a program.

Source to executable with compiled language.

Several steps.

-Edit the source code.

-Compile the source code into machine code.

-Link the machine code with libraries, etc. (oftentimes done in one command together with compile, but not necessarily and you can control it and do it separately).

- Run the program (which is itself another, executable, file).

During development you will probably be doing this kind of cycle.

Edit the Fortran source code.

Compile it, link it (here done automatically - combine it libraries such as the library of VAX/VMS extensions to Fortran, and files with extra code, but you have to tell it what to use) and make an executable file.

Then run it.

```
$ vi chinn2gmt_fm.f
…
$ f77 -W132 -lU77 chinn2gmt_fm.f ../opens.f -o chinn2gmt_fm
$ chinn2gmt_fm
```

Examples of some languages that are typically compiled.

FORTRAN

C

C++

ALGOL

PASCAL

BASIC/Visual BASIC

# Interpreter

The name "interpreter" is primarily used for programs that translate source code written in a high-level programming language to machine code a line at a time at the time of execution.

Typically produces less efficient (in terms of run time) implementation of a program (for example, in a loop it has to interpret the instructions each time through the loop).

Source to executable with interpreted language.

Two steps.

-Edit the source code.

-Run the program

(It does the "compiling" and "linking", or translating to machine code, steps automatically, but repeatedly.)

It does not produce an "executable" file. To run it again it has to be interpreted again. (So there is actually some interpreter program running and your program is really input to it.)

# Examples of some languages that are typically interpreted

Shell scripts (always)
awk/nawk/gawk
BASIC
MATLAB
HTML
LISP (which stands for LISt Processor or (Lots of ((Irritating, Spurious) (Parentheses)))). Favorite language for AI.

- Compiled languages are compiled.

- Modern interpreted languages are typically hybrids (they will compile the code in a loop for instance, instead of interpreting it each pass).

MATLAB is an example of the modern hybrid. It has interpreted parts, compiled as needed parts, and you can compile your code.

When you run a compiled program (again and again) you skip the compile/link steps.

(this is fast)

(the compile/link process produces an executable file, which is the file that is run/executed – not the source file.)

When you run an interpreted program (again and again) the computer has to redo the interpretation each time.

(this is slow)

(So while modern interpreters may internally take shortcuts such as compiling a loop, it is local to each running of the program. Each time you run/execute the interpreted program it returns to the source file and starts from scratch.)

Shell scripts are strictly interpreted.

The philosophy of shell scripting is to

- develop a tool with the shell and

- then write the final, efficient, implementation of the tool in C (or other high level language).

The second step is typically (universally?) skipped.

# Interpreting vs. Compiling.

Compiling: good for medium, large-scale, complicated problems, number crunching, when you need the efficiency.

Interpreting: good for smaller scale, simpler problems, when not number crunching, when your efficiency is more important than the CPU's.

(It is not worth spending an hour of your time to save a microsecond of execution time on a program that will run once – unless saving that microsecond is the point of a homework problem.)

-Olden days –

Computer was expensive, limited, resource.
Programmer – relatively less expensive.
Lots of effort went into writing small, efficient programs.

-Today –

Abundance of inexpensive computer resources.
Programmer – very expensive.
(Buy a faster computer with another gigabyte of memory!)

## ~ Take home lesson ~

Use the appropriate tool.