Unix Tools

Ian Pratt

Michaelmas Term 2000

4 Lectures

Mondays,Wednesdays,Fridays

Heycock 11am

Unix Tools

- Widely available
  - even for Windows
  - “standardised”
- Often quicker/more powerful than GUI approaches
  - but, takes time to learn
- Good for ‘medium complexity’ operations
  - in between interactive tasks and real programming
- Course aims:
  - provides a basic introduction
  - let you know where to get further info

Outline

- Lecture 1 : The shell
  - tool philosophy
  - streams, redirections and pipes
  - use of regular expressions
- Lecture 2 : Development tools
  - Make
  - RCS
  - job control
- Lecture 3 : Data Manipulation Tools
  - sed, tr, diff, etc.
- Lecture 4 : Scripting languages
  - intro to perl

Unix Tools Philosophy

Suite of “simple” tools that can be combined to perform complex functions

- Individual tools:
  - address a limited class of problem
  - but, completely general within that class
  - small, “easy to learn”, well implemented
  - easily linked with other tools
- Typical syntax:
  - command <options> <file(s)>
  - take input from stdin or file(s)
  - options start with a `-' or `--‘
  - return an exit code

Getting help:

- man *command*
- *command --help* or -?
- info pages
- Books:
  - "Unix in a Nutshell", Daniel Gilly, O-Reilly
- reading other shell scripts:
  - `~/bin/scripts/`

The Unix Shell

- Ordinary user-mode application
  - user can choose/re-implement shell
- Many to choose from:
  - sh, bsh, csh, ksh, ash, msh, bash, ...
- Two main family lines
  - csh and bsh
- csh & its derivatives are NOT recommended
- GNU bash is ubiquitous and pretty good
- Interactive and shell script use
  - `#!/bin/sh`
  - `chmod +x script`

Streams, redirects and pipes:1

- `command < input.file > output.file`
- Equivalent to:
  - `cat input.file | command > output.file`
- Can also typically do:
  - `command input.file > output.file`
- Use >> to append to output file
  - `echo “append this” >> file`
- Use `tee` to get a copy
  - `command2 | tee log.file | more`

Streams, redirects and pipes:2

- Standard error stream
  - Programs typically send error output to stderr
  - `command > output.log 2> error.log`
  - Stderr normally doesn’t pass through pipe
- Merging stderr to stdout
  - `gcc myprog.c 2>&1 | more`
  - `echo “error!” 1>&2`
- `Here documents’
  - Embedding input in scripts

Command line expansion

- Shell performs expansion before execution
  - Expand wildcards to match filenames
    - `echo *` is equivalent to `ls`
    - List all but `hidden' file, including directories
    - `echo *foo*`
  - ? character matches any single character
    - `echo f??`
  - Backslash and quotes can be used to `escape' special characters

Environment variables

- variable_name = value; export value
  - `COMPILER = CC;  export COMPILER`
  - `CFLAGS   = -O3; export CFLAGS`
  - `COMP     = $COMPILER $FLAGS; export COMP`
- Expanded by shell before execution of cmd
  - `$foo` or `${foo}` syntax
  - `$COMP myfile.c`
- Use as a `shorthand'
  - `cd $SRC`
- Enables easy changing later
- `Printenv` lists all env variables defined

Arguments to shell scripts

- `$1, $2...$9` are set to arg1, arg2…arg9
- `$0` is the name used to invoke the script
- `$#` is the number of the last argument
- `$*` is a list of all arguments (excluding `$0`)

```
#!/bin/sh
echo Start of $0, called with $# args
echo All args: $*
echo Arg1 = $1
echo Arg2 = $2
```

Functions

- Can be defined interactively or in a script
  - `function m () { more }`
  - `function ll () { ls –al }`
- Used as shorthand
- Can make scripts more readable

```
killproc() {     # kill named process(es)
pid=‘/usr/bin/ps –e |
/usr/bin/grep $1 |
/usr/bin/sed –e ‘s/^  *//’ –e ‘s/ .*//’`
case “$pid” in [1-9]*) kill –TERM $pid;; esac
}
```

Regular expressions : grep

- `grep <options> <regexp> <file(s)>`
- Common options:
  - -i case insensitive
  - -v invert sense of matching
  - -c just produce a count of matching line
  - -l print name of file in which match occured
- egrep necessary for more complex regexps

`egrep` Regexps

- (A) Parenthesis of a sub-string
- A | B Alternation
- A* Accepts zero or more instances of A
- A+ Accepts one or more instances of A
- A? Accepts zero or one instances of A
- A{n,m} Accepts from n to m repetitions
- [a-z] Match single char e.g. [^0-9] [a-z,A-Z]
- ^ and $ Match start/end of line
- \< and \> Match start or end of words

• `escape' characters of significance to shell!

Finding files : `find`

- Find <pathnames> <options>
- options narrow the set of file that match:
  - `-name <simple regexp>` (like shell regexps)
  - `-type f,d` : Files, directories
  - `-mtime days, -size bytes, -perm 001`
  - `-, ,+` : at most, exactly, at least
  - `!, -o , -a` : NOT, OR, AND (AND usually implicit)
  - `-print` Print the results (also `–exec foo {} \;`)

```
find . –name '*.[ch]' –type f –print | grep foo
```

- Use of `xargs` to turn stdout into command line args
  - `find . –type f –print | xargs grep foobar`

Command exit codes

- Zero for success
- Non-zero implies some abnormal condition
- Accessed via `$?`
- `||` and && enable conditional execution
  - Shortcircuting evaluation
- `A || B`   B executed iff A has non-zero exit
- `A && B`   B executed iff A has zero exit code

```
ping –c1 lundy || ping –c1 bailey || echo “net broken”
```

Expression evaluation: `test`

- Comparing strings and integers (in bash)
  - `test $1 = “skip” ; echo $?`
  - `test $TWO*2 != 4`
  - `test 3 –gt 4 ; echo $?`
- Testing files for existence, access permissions, modification times, etc.

  `-r/w/x foo` :File exists and is readable/writeable/executable
- Abbreviated to `[ <exp> ]`
  - `[ –r foo –a ! –w foo –o $1 = “skip” ]; echo $?`

Conditional execution

- **if** [-r foo]
  then echo “readable”
  else echo “not readable” ; **fi**

- **while** [! –r foo] ; **do**
  echo “not readable”; sleep 1
  done

for and case statements

- **for** fi in `cat list` ; **do**

```
mv $fi /tmp/ ; echo $fi moved
done
```

- **case** `arch` **in**

```
*hpux*)
  PSOPT="-eafl"  # HPUX has strange ps
  ;;
*)
  PSOPT="auxw"   # OSF, Linux etc
  ;;
esac
```

Shell Job Control

`nice make &`	make in background at low pri
`fg`	bring to the foreground
`Ctrl-Z`	suspend it…
`bg`	…and put back into background
`jobs`	list the current background jobs in this shell
`kill %3`	kill job number 3
`ps aux`	list all processes on machine (aux Unix specific…)
`top`	show real-time machine status
`at/cron`	used to start tasks at future times

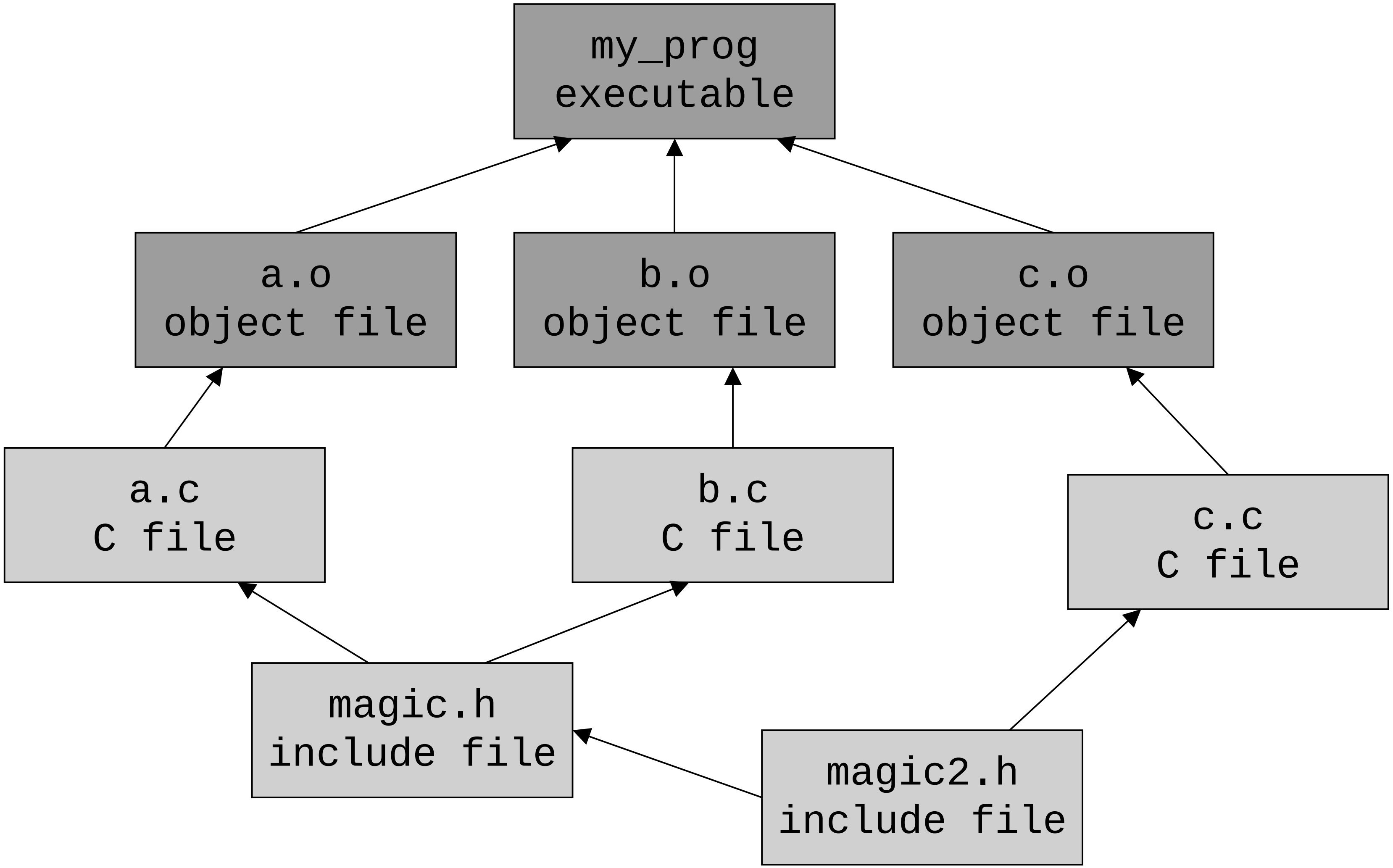
`make` and project building

- Automates the build process
  - 'Documents' build procedure
  - Can also use to invoke / regression test program, operate revision control system, install, tidy etc.
  - Useful for projects other than program sources
    - E.g. LaTeX documents
- Re-build only what needs updating
  - Exploit parallelism where available

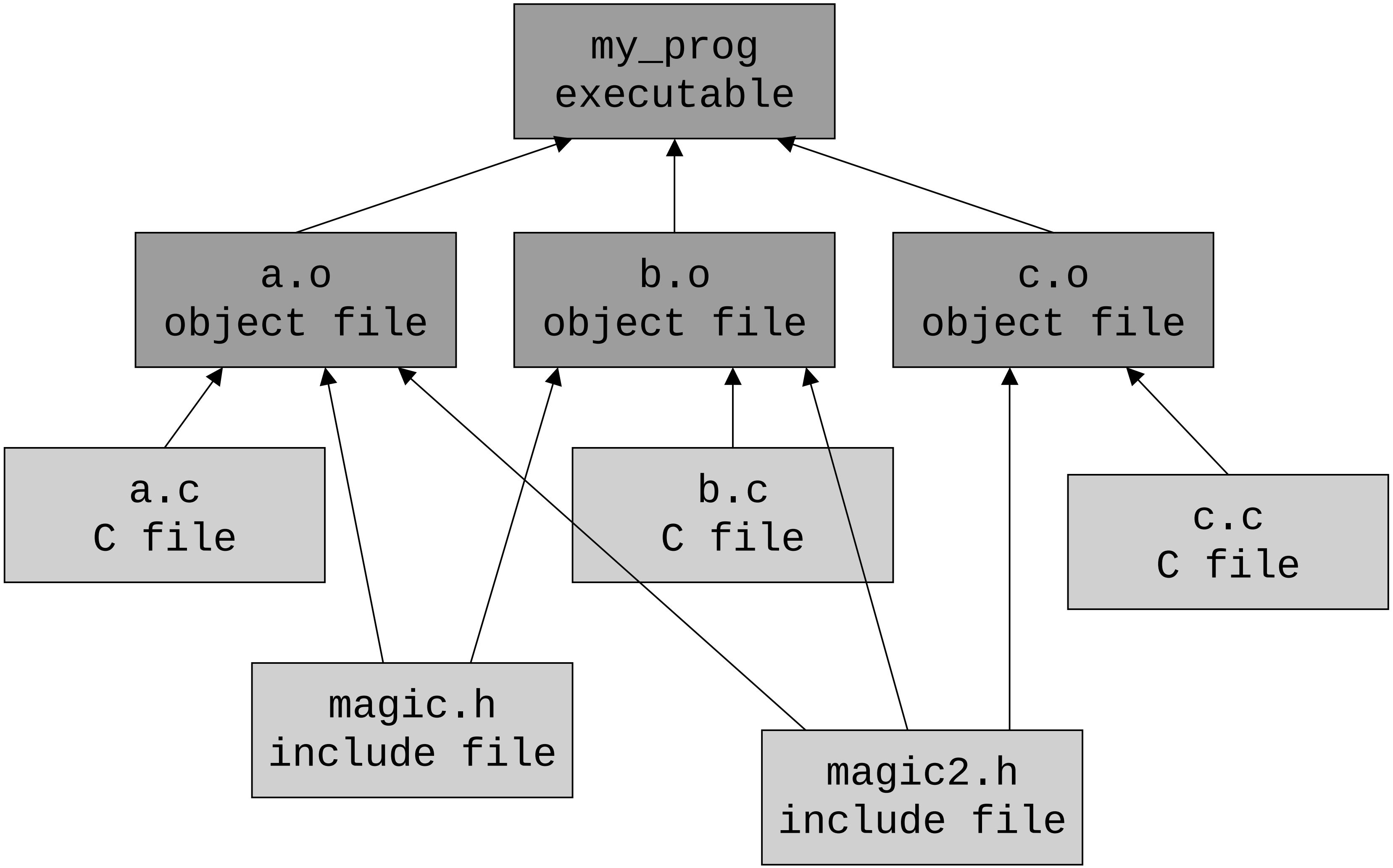
A simple `Makefile`

```
CC      = gcc
CFLAGS  = -O2
LD      = ld
LDFLAGS = -Bdynamic
LIBS    = -lm
RM      = rm -f

myprog: a.o b.o c.o
	$(LD) $(LDFLAGS) -o myprog a.o b.o c.o $(LIBS)

a.o: a.c magic.h magic2.h
	$(CC) $(CFLAGS) -c a.c -o a.o

b.o: b.c magic.h magic2.h
	$(CC) $(CFLAGS) -c b.c -o b.o

c.o: c.c magic2.h
	$(CC) $(CFLAGS) -c c.c -o c.o

clean:
	$(RM) *.o $(TARGET) core
```

Dependencies



Dependencies



```
TARGET   = myprog
SRCS     = a.c b.c c.c
OBJS     = $(SRCS:.c=.o)

CC       = gcc
CFLAGS   = -O2
LD       = ld
LDFLAGS  = -Bdynamic
LIBS     = -lm
RM       = rm -f

.c.o:
	$(CC) $(CFLAGS) -c $< -o $@

$(TARGET): $(OBJS)
	$(LD) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

clean:
	$(RM) *.o $(TARGET)

depend:
	makedepend $(SRCS)

# DO NOT DELETE THIS LINE -- make depend depends on it.

a.o: magic.h magic2.h
b.o: magic.h magic2.h
c.o: magic2.h
```

Revision Control System `rcs`

- Archival system for versions of files
  - File versions may be “checked in”, and retrieved later if necessary
  - Helps protects against accidental deletion, ill conceived editing etc.
  - Enables old versions of a project to be reconstructed
  - Also provides rudimentary “file locking” to assist in projects with multiple programmers

Checking files in/out

- Best to create an `RCS` directory
- `ci prog.c`
  - Checks file in to database, drops the lock
  - Enter description of the file
    - Further ci’s will request a log entry
- `co prog.c`
  - Defaults to checking out the latest revision
  - Doesn’t grab the lock, hence file is read-only

Locking files for editing

- `co –l prog.c`
  - Takes the lock (if available!) & leaves a read/write copy for editing
- `ci -l prog.c`
  - Checks a version in, but retains the lock
- Can view a prior version with:
  - `co –px.x prog.c`
- View rcs log with `rlog`
- Auto-annotate source files with `$Log$`

Other useful `rcs` tools

- View differences between versions
  - `rcsdiff –rx.x –ry.y`
    - Defaults to current version
- Create a ‘symbolic name’ for a set of source files
  - `rcsfreeze`
- See `cvs` or `prcs` for more demanding revision control applications

Comparing files : `diff`

- `diff [options] from-file to-file`
- `-c` generate easier to read ‘context diff’
  - Show lines of context
- `-u` GNU ‘unified diff’ : easiest to read & robust
- From-file / to-file can be directories

  `-r` recursive over sub-tree

  `-N` handle new/deleted files
- `-b` ignore whitespace changes
- `-i` ignore case distinctions

‘Packaging’ files : `tar`

- Creates an ‘archive’ containing sub-tree
- Preserves file system structure
- Typical usage:
  - create: `tar –cvf mytree-1.2.tar mytree-1.2`
  - extract: `tar –xvf mytree-1.2.tar`
- GNU `–z` extension, equivalent to e.g.:
  - `gunzip –c mytree-1.2.tgz | tar –xvf -`

`patch`

- Takes output from diff to ‘patch’ a file/tree
- Patch files best created with:

```
diff –ruN old-tree new-tree  > old2new.patch
```

- Apply patch with:

```
cd tree; patch –p1 < old2new.patch
```

- `-p1` strips off the first pathname component
- Use `–R` to reverse patch

Data manipulation tools : `tr`

- `tr` is used to translate one character to another
- Character substitution:

  convert to lower case: `tr ’A-Z’ ’a-z’`

  “encrypt”: `tr ’a-zA-Z’ ’b-zA-Za’`
- Deleting particular characters, e.g. CR, Ctrl-Z

  `tr –d '\015\032' < dos.file > unix.file`

  (note use of octal to specify Ctrl-M & Ctrl-Z)
- Translate and “squeeze repeats”

  Convert all numeric strings to a single ‘#’:

  `tr –s '0-9' '##########'`

  Removing repeated spaces, tabs and NL’s:

  `tr –s ' \011\012' ' \011\012'`

Data manipulation tools : `sort`

- Sort lines in files based on key field list (primary 1st)
- By default, fields delimited by whitespace
  - Specify separator with: `-t` e.g. `–t :`
- By default, sort key is entire line
  - Key fields specified with: `-k pos1[,pos2]`
  - E.g. `–k2,4` specifies fields 2 through 4
- Treat field as numeric with `–n`

```
sort –t : -k2n
/bin/ls –l | sort –k5n
```

- Reverse sort order with `–r`
- Remove duplicate lines in sorted output with `–u`

Data manipulation tools : `cut`

- Extracts sections from lines in its input
- Field extraction:

```
cut –d : –f 5 < /etc/passwd
```

  - Multiple fields (can't change order)

    ```
    cut –d : –f 1,5,7 < /etc/passwd
    ```

- Extract by character column:

```
cut –c 1-10,20-
/bin/ls –l | sort –k5n | cut –c 56-
```

Data manipulation tools : `sed`

- Typ. used for regexp search/replace and delete
- Takes options on cmd line or in file with:
  `sed –f cmd.sed <input >output`
- Search/replace (**S**ubstitute **G**lobal)
  `sed ’s/apple/pear/g; s/orange/kiwi/g’`
  - Without 'g' only 1st match on each line is replaced
- Commands can preceded with line addresses:
  `1,5s/apple/pear/` or `12,$s/apple/pear/`
  - $ is last line of file

Data manipulation tools : `sed`

- Delete command : `1,5d` `10,$d`
- Addresses can be regexps too! E.g.
  - Delete lines containing match
    `/banana/d delete all line`
  - Substitute lines defined by region
    `/orange/,/pear/s/0/X/g`
- Text variables:
  - Use **`\( \)`** to capture strings into vars **`\1 \2 .. \9`**
  - E.g. re-order 3 space-separated columns **`ABC`** into **`BACA :`**
    `s/ *\([^ ]*\) *\([^ ]*\) *\(.*\)/\2 \1 \3 \1/`

Selecting the right tool for the job

- Other scripting languages offer a more integrated & efficient environment than shell scripts
- `awk` (and GNU `gawk`)
  - Good for data manipulation & processing
- `perl, python`
  - At least as powerful as C/C++
    - Integrates well with other programming languages
    - Many high-level features and add-on modules
  - Auto-compiled to interpreted byte code
    - Respectable performance

(g)awk

- Use info pages to find out more
- Simple one-liners:

```
awk -F: '{print $5 "*" $1 "*" $6 }' </etc/passwd
awk '{ if (length($0) > max) max = length($0) }
   END { print NR " lines, longest = " max }'
awk -F: '/apple/ { print "apple=" $2 }
   /^b(an)+a/ { print "banana=" $2*100 }'
```

- Execute commands in a script with `-f cmd.file`
  - Also, `#!/bin/awk -f` mechanism to create executable scripts

awk scripts

```
#!/bin/awk –f

# dupword --- find duplicate words in text
{
  $0 = tolower($0)
  gsub(/[^A-Za-z0-9 \t]/, "");
  if ($1 == prev)
    printf("%s:%d: duplicate %s\n",
      FILENAME, FNR, $1)
  for (i = 2; i <= NF; i++)
    if ($i == $(i-1))
      printf("%s:%d: duplicate %s\n",
        FILENAME, FNR, $i)
  prev = $NF
}
```

- gsub   substitute global (old, new)
- FNR   number of records read in this file
- NF   number of fields in current record
- $(i-1)   using an index to access fields

awk scripts

```
#!/bin/awk -f

BEGIN { binsize = 0.1;
}

$0 !~ /^#/ { arr[ NR%32, int($1 / binsize) ]++; }

END {
  for ( seek=0; seek<32; seek++) {
    filename = "conner100." seek ".dat";
    for ( bin = 0; bin < int(20 / binsize) ; bin++) {
      print bin * binsize, arr[seek, bin]+0 > filename;
    }
  }
}
```

- !~ 'does not match' operator
- arr[x,y] 2D sparse array with auto-initialisation
- >filename file creation and redirection
- arr[x,y]+0 fixing the type of initialised elements

perl

```
#!/usr/bin/perl -s
$f=$d?-1:1;$D=pack('C*',33..86);$p=shift;
$p=~y/a-z/A-Z/;$U='$D=~s/(.*)U$/U$1/;
$D=~s/U(.)/$1U/;';($V=$U)=~s/U/V/g;
$p=~s/[A-Z]/$k=ord($&)-64,&e/eg;$k=0;
while(<>){y/a-z/A-Z/;y/A-Z//dc;$o.=$_}$o.='X'
while length ($o)%5&&!$d;
$o=~s/./chr(($f*&e+ord($&)-13)%26+65)/eg;
$o=~s/X*$// if $d;$o=~s/.{5}/$& /g;
print"$o\n";sub v{$v=ord(substr($D,$_[0]))-32;
$v>53?53:$v}
sub w{$D=~s/(.{%_[o]})(.*)(.)/$2$1$3/}
sub e{eval"$U$V$V";$D=~s/(.*)([UV].*[UV])(.*)/$3$2$1/;
&w(&v(53));$k?(&w($k)):($c=&v(&v(0)),$c>52?&e:$c)}
```

- Compact representation…
- …but, lots of cryptic operators, significant punctuation, strange default behaviour etc…
- However, `perl` is widely installed and used
- `cgi-bin` scripts are `perl` scripts run on web servers

`Perl` scripts

```
#!/usr/bin/perl -w

while (<>) {
   if (/regexp/) {
      print $_;
   }
}
```

- `-w` option enables some warnings (recommended!)
- <> angle operator reads next line of input to `$_`
- Regexps enclosed in `/ /`
- Curly braces following if are not optional
- `print $_` could be abbreviated to just `print`
- Run `perl` scripts on the command line with `–e`
  - Iterate over all line is input with `-n`

```
perl –en 'split; print "$_[1] $_[0] $_[2] $_[0]\n"'
```

`perl` arrays & associative arrays

```
#!/usr/bin/perl -w

while (<>) {
    chomp;
    @words = split;

    foreach $word (@words) {
      $count{$word}++;
     }
}

foreach $w (keys %count) {
    print "$w was seen $count{$w} times\n";
}
```

- chomp removes `CR/LF` at ends of lines
- split creates an array of whitespace separated words
- `@var` denotes entire array (# elts in array in a scalar context)
- `$var{elt}` denotes an element of an associative array
- `%var` denotes entire associative array

`perl` ’default behaviour’

```
#!/usr/bin/perl –w

while (<STDIN>) {
  chomp;
  $gcos = (split /:/)[4];              #line 1
  ($real) = split(/,/, $gcos);         #line 2
  ($first) = split(/\s+/, $real);      #line 3
  print "$first * $real * $gcos\n";
}
```

- `(split /:/)` returns an array of strings separated by ’:’
  - Defaults to operating on $_
  - Call to split has one argument, hence no need for brackets
  - 1st elt of array is [0] (not 1!)
- Line2 assigns the strip result array to the array `($real,`…)
- `\s` is a shorthand for whitespace characters (not ws is `\S`)
  - One on many

More `perl`

```
#!/usr/bin/perl -w
chomp($data = ‘uptime‘);
#  9:34am  up 12 days, 19:09, 26 users,  load average: 0.08...
#  9:38am  up 16:23,  6 users,  load average: 0.00, 0.00, 0.00

if ($data =~ /up\s+((\d+) days?,\s+)?(\d+):(\d+)/) {
    $days = defined($2)? $2 : 0;
    print "uptime: ${days}days $3hours $4mins\n";
} else {
    print "uptime produced weird output: ’$data’\n";
}
```

- Use of back quotes to exec command and capture output to scalar (could have used an array for multi line output)
- `=~` is the match operator (`$data` unaltered in this case)
- Note use of brackets for grouping and field capture
- `\s` matches whitespace chars, `\d` decimal digits
- Note use of `days?` To match *day* and *days*

python

- A little more verbose than `perl`, but arguably more readable and less prone to bugs
- Relatively clean object-oriented design
- Many add-on classes available (like perl)
  – GTK, OpenGL etc
- Whitespace is significant!
  – Uses indentation instead of curly braces
- A nice prototyping language for significant programs…