UNIX Power Tools — Overview

AM	Shell 1	UNIX and the Shell
	Commands 1	Listing Information
	Commands 2	Creating and Destroying
	Shell 2	Command I/O
	Commands 3	Splitting and Joining
	Shell 3	Linking Commands
	Shell 4	Variables and Quoting
Lunch		
PM	Commands 4	Looking Inside
	Shell 5	Scripts and Arguments
	Commands 5	Numbers and Values
	Shell 6	Control Structures
	Commands 6	Scriptable Programs

UNIX Power Tools — Reading

- The Unix V Environment,
  Stephen R. Bourne,
  Wiley, 1987, ISBN 0 201 18484 2

  The author of the Borne Shell! A 'classic' which deals not only with the shell, but also other aspects of UNIX.

- Unix Shell Programming (3rd Ed.),
  Lowel Jay Arthur & Ted Burns,
  Addison-Wesley, 1994, ISBN 0 471 59941 7

  Covers Bourne, C and Korn shells in detail.

- UNIX manual pages:

  `man sh` *etc.*

  Most commands are in section 1 of the manual. The shell is usually well covered, but you need to read the awk documentation (perhaps in troff form in /usr/doc) as well as the manual entry.

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Shell 1

UNIX and the Shell

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

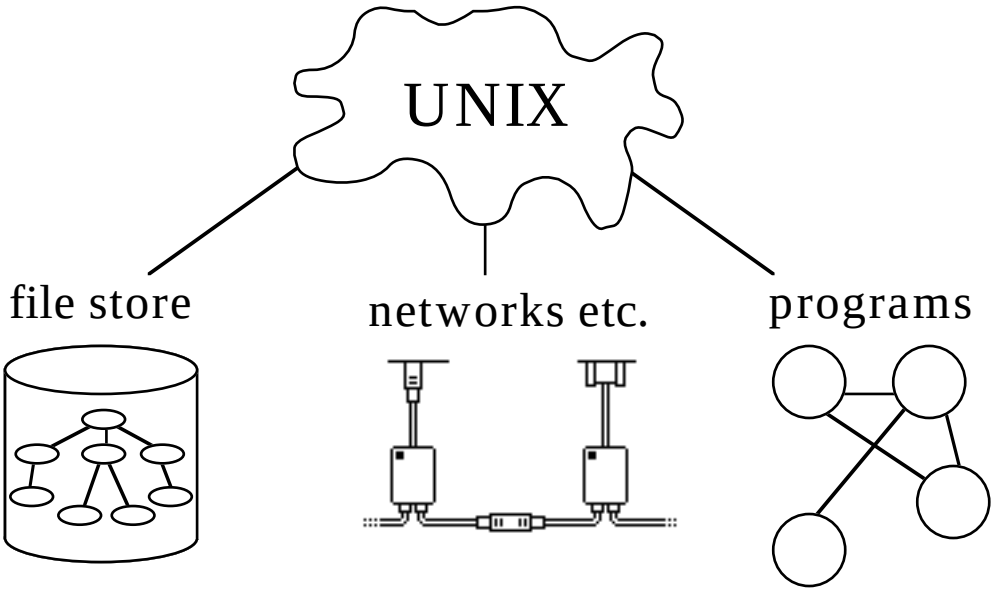
Shell 1

UNIX and the Shell

- the nature of UNIX
- what is the Shell?
- shell programming
- shell commands
- UNIX file system
- wildcards for file names

UNIX

UNIX is an operating system



It manages:

- files and data
- running programs
- networks and other resources

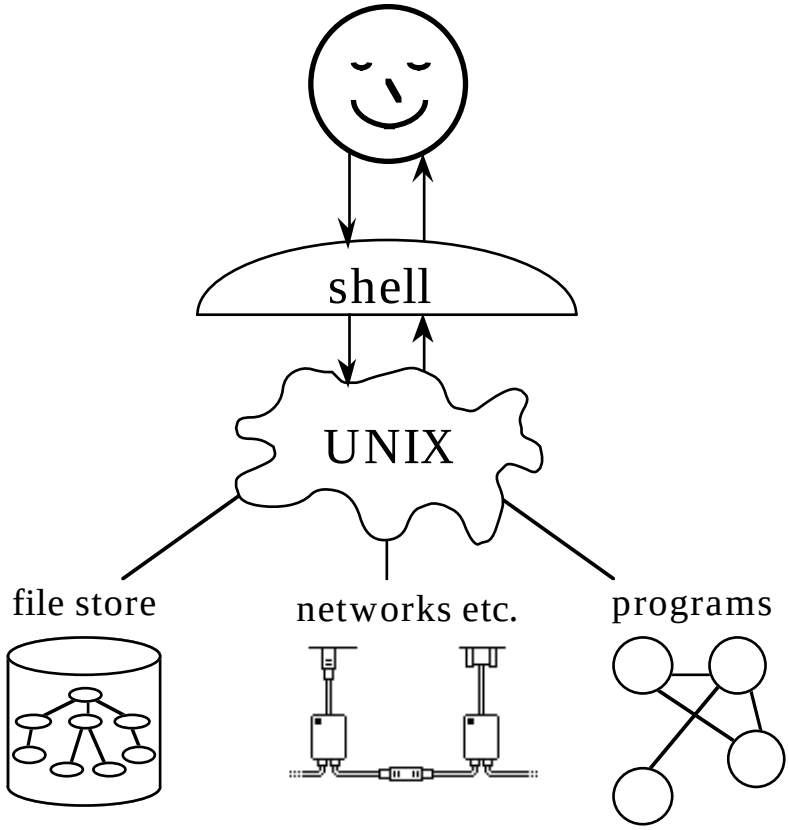
It is also:

- a collection of programs and utilities
- glued together by the shell

Shell

What is the shell?

- just a user interface for UNIX
- a rather poor user interface !
- a powerful user environment
- a flexible programming environment



Windows vs. the Shell

MS Windows or Mac/OS

supports

- exploratory actions
- with immediate feedback
- on single objects

Shell

supports

- planned actions
- with feedback on request
- on multiple objects

Different Shells

Bourne Shell

- also known as '`/bin/sh`' (where it is on most systems)
- available on all systems (so good for scripts)
- used throughout the course

C Shell

- more C like syntax
- process control
- command history for repeating past commands
- aliases for commands

Korn Shell

- incorporates features of both
- better handling of integers

Various graphical window front-ends

- usually only support limited functionality
- for complex actions have terminal windows . . .
  . . . and a shell

Shell as a programming language

programming language manages:

- data:
  - integers
  - character strings
  - records
- control flow:
  - choice – if-then-else
  - loops of repeated actions
- procedures:
  - packages frequent actions together

UNIX shell:

- data:
  - environment variables (character strings)
  - whole files
- control flow:
  - similar + special features
- procedures:
  - shell scripts

Shell “commands”

- some built into the shell
- some separate programs

Typical command:

```
command  options  filename1  filename2  ...
```

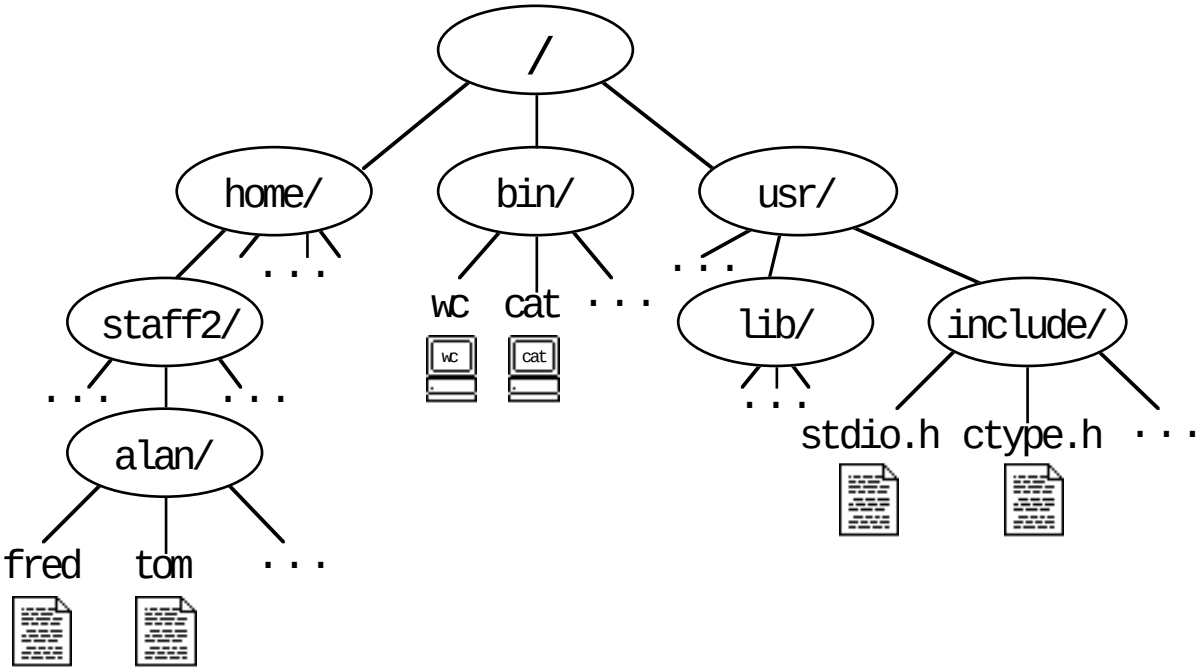
options either: `-abc`
or: `-a -b -c`

e.g.
```
ls -rlt  jane
cat fred tom
od -c tom
```

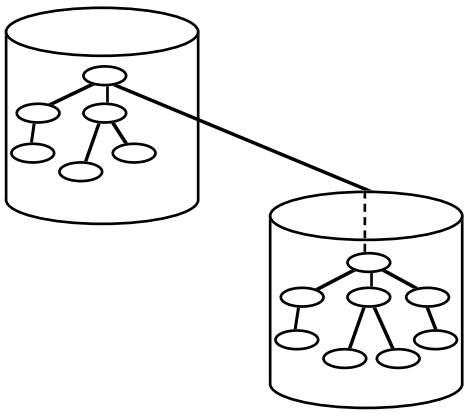
N.B. ① not all commands like this
② different versions of UNIX

UNIX file system

- organised as hierarchy



- other disks linked into hierarchy



all contents in same name space as other each other

(c.f. DOS A:, B: etc.)

Wildcards

- Shorthand to save typing!
- Refer to several files with similar names

File name contains one or more special characters:

?	–	any single character
*	–	any sequence of characters (poss. empty)
[abxy]	–	a single character from ‘abxy’
[h-m]	–	any character between ‘h’ and ‘m’

Directory:

fred.doc	fred3.doc	fred5.doc	tom.c
fred2.c	fred4.dat	harry.dat	tom.doc
fred2.dat	fred4.doc	harry5.doc	

Examples:

*.doc	–	fred.doc fred3.doc fred4.doc fred5.doc harry5.doc, tom.doc
fred?.doc	–	fred3.doc fred4.doc fred5.doc
[ht]*	–	harry.doc, harry5.doc, tom.c, tom.dat
*5.doc	–	fred5.doc harry5.doc

(N.B. different in DOS!!!)

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Commands 1

Listing Information

UNIX **Power Tools**

UNIX **Power Tools**

Commands 1
Listing Information

- fixed text
  - **echo**
- text files
  - **cat**
  - **cat -n**
  - **more**
- non-text files
  - **cat -v**
  - **od**
- directories
  - **ls**
- internal UNIX information
  - **ps**
  - **lpq**

echo

echo {-n} text

- print fixed text

```
$ echo hello
hello
$
```

- -n option - no new line after

```
$ echo -n hello
hello$
```

- to try things out

```
$ echo *5.doc
fred5.doc harry5.doc
$
```

- or print messages from 'scripts'

```
echo "sending output to $PRINTER"
```

text files

cat file1 file2 file3

– prints out the files to the terminal

```
$ cat fred
this is the contents
of the file called fred
$
```

cat -n file

– prints out the file with line numbers

```
$ cat -n fred
1  this is the contents
2  of the file called fred
$
```

more file

– prints the file screen by screen

you type: return key – for another line
space – for the next screen full

non-text files

files with non-printable characters

e.g. program data files

files copied from DOS or Mac platforms

(different end of line mark, accented characters etc.)

cat -v file

– not all version of UNIX!
uses \ sequences for special characters

```
$ cat -v my-dos-file
This is a PC file. DOS ends lines with\r
carriage return line feed, not just\r
line feed\r
$
```

od file

– prints out the file in octal (base 8)

options:

od -x file – hexadecimal instead of octal

od -c file – where possible uses ordinary chars

```
$ od -c small-pc-file
0000    L    i    n    e    1   \r   \n    L
0008    i    n    e    2   \r   \n
$
```

directories

ls

– list files in the current directory

ls dir

– list particular directory

ls -l

– long listing (size, dates etc.)

ls -l file

– details of single file

ls file1 file2 dir1 dir2 file3

– lists all the given files and directories

ls -rlt *.c

– list details of all files ending in '.c'
in reverse time order

the guts

ps

– list your running programs (processes)

ps uax

– list all running programs

lpq

– list files waiting to be printed

lpq -Psparc

– list files waiting to be printed on the printer called 'sparc'

N.B. options for lpq very system specific

++ PLUS ++

lots of other system administration information !!!

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Commands 2

Creating and Destroying

UNIX **Power Tools**

UNIX **Power Tools**

UNIX Power Tools

Commands 2
Creating and Destroying

Files

- creation
  ```
  >newfile
  cat >newfile
  ```
- deletion
  ```
  rm
  ```

Directories

- creation
  ```
  mkdir
  ```
- deletion
  ```
  rmdir
  rm -rf
  ```

Duplicating files

```
cp
ln
ln -s
```

Files

- file creation
  - created as the result of many commands
  - quick ways – to make an empty file:

```
$ >newfile
$
```

  - type it in from the keyboard
    (end file with control-D – echoed as '^D')

```
$ cat >newfile
text for the new file
^D
$
```

  - or use '`cat >>file`' to add to the end of a file

- file deletion

  rm file1 file2 file3

  - '`rm -i`' option prompts for confirmation
  - be very careful with wildcards!

Directories

- creation

  mkdir newname

  - creates a new sub-directory of the current directory called 'newname'

- deletion

  rmdir oldname

  - deletes the directory called 'oldname'
  - will only remove empty directories to protect you

  rm -rf old-dir

  - special options for `rm`
  - removes old-dir and all enclosing directories and files!!!

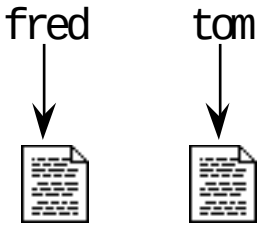
    N.B. '`rm -rf *`' – the UNIX nightmare

Duplicating

- UNIX filenames are pointers to the file
- there may be more than one pointer

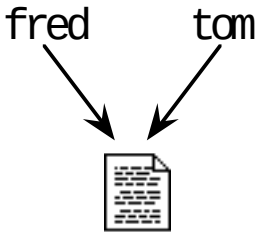
cp tom fred

- duplicates the file pointed to by tom
  fred points to the new copy



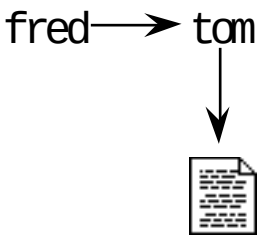
ln tom fred

- fred points to the same file as tom

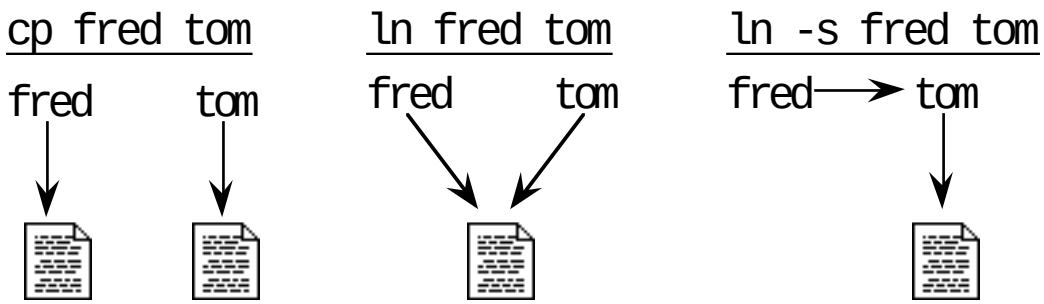


ln -s tom fred

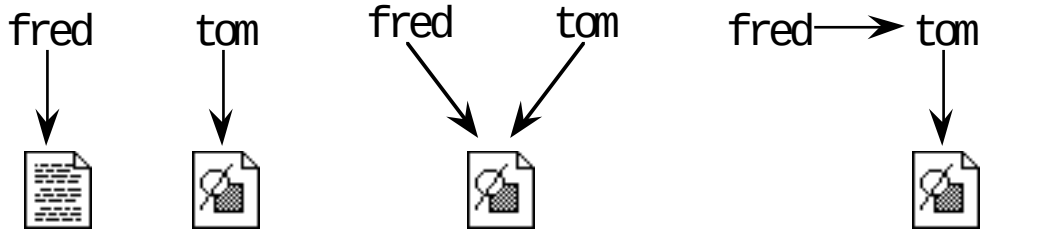
- fred points to the name 'tom' – an alias



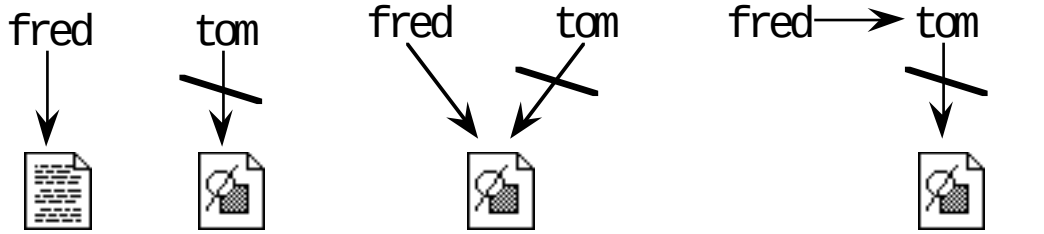
Links and updates



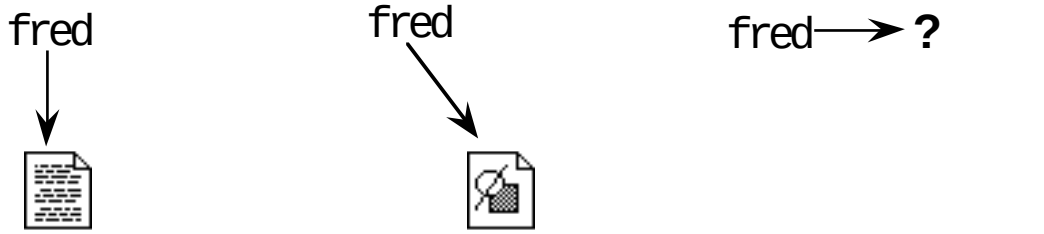
- update file tom



- delete file tom (rm tom)



- what is in fred?



☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ go to the directory 'tools/many-files'

```
$ cd tools/many-files
```

experiment with wildcards, e.g.:

```
$ echo [a-m]*.???
```

☞ go back to your home directory

```
$ cd
```

☞ create a file 'tom' using 'cat'

```
$ cat >tom
```

remember control-D to finish

☞ link fred to tom

```
$ ln tom fred
```

☞ print out fred and tom using 'cat'

```
$ cat tom
$ cat fred
```

☞ update tom using 'cat >> tom'
print out fred and tom again using 'cat'

☞ delete tom using '`rm tom`'
print out fred – what happens

☞ repeat using '`cp tom fred`' and '`ln -s tom fred`'
N.B. you will have to 'rm fred' each time before starting

☞ try '`ln -s fred fred`'
what happens when you do '`cat fred`'

Shell 2

Command I/O

Shell 2
Command I/O

- programs and processes
  - the nature of shell commands
- input and output
  - standard input
  - standard output
  - standard error
- redirection
  - sending input and output to files

Programs and processes

Processes

- UNIX can run many programs at the same time
- Also many copies of the same program
  (c.f. Windows and Mac)
- strictly UNIX runs many processes
  each of which executes the code of a program

The Shell

- the shell is just a program
- possibly different kinds of shell for different users
- often more than one copy of the shell for each user

Commands

- UNIX is profligate with processes!
- created and destroyed all the time
- one for most commands executed

Input and Output

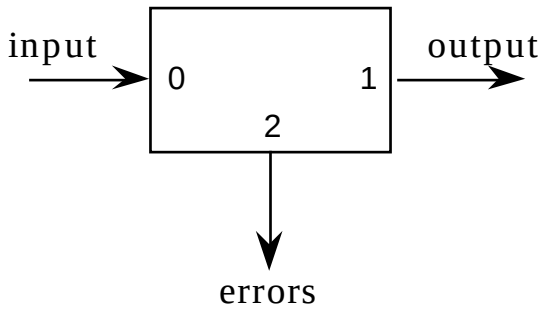
Each running program has numbered input/outputs:

0 – standard input
- often used as input if no file is given
- default input from the user terminal

1 – standard output
- simple program's output goes here
- default output to the user terminal

2 – standard error
- error messages from user
- default output to the user terminal

Other numbers are rarely used from the shell

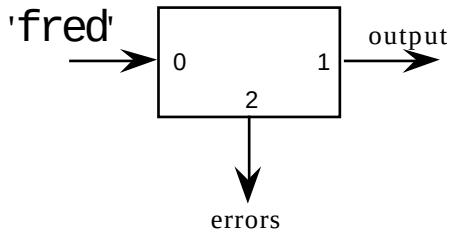


Redirection

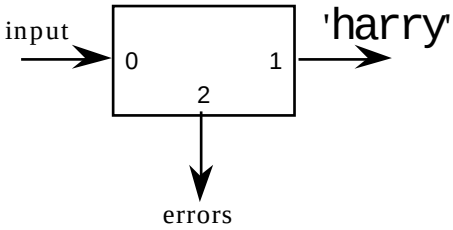
Default input/output is user's terminal

Redirection to or from files:

- command <fred
  - standard input from file 'fred'



- command >harry
  - standard output goes to file 'harry'



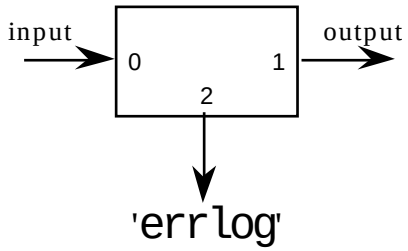
  - file is created if it doesn't exist

N.B. C shell prevents overwriting

- command >>harry
  - similar, but appends to end of 'harry'

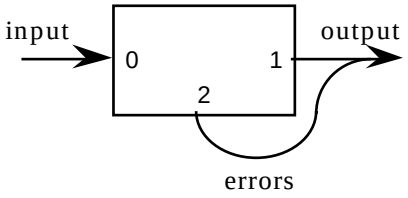
Redirection of standard error

- command **2>**errlog
  - standard error goes to file 'errlog'



- command **2>>**errlog
  - standard error appends to end of 'errlog'

- command **2>&1**
  - standard error goes to current destination of standard output



How it works ...

Quick file creation

`cat >fred`

- no files given – so `cat` reads standard input
- standard input defaults to user terminal
- `cat` copies to standard output . . .
  . . . which goes to the file `fred`

⇒ the users typing goes into `fred`

Empty file creation

`>fred`

- standard output goes to `fred`
- if it doesn't exist it is created
- no command given – the empty command
- the empty command generates no output

⇒ the file is empty

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Commands 3 UNIX **Power Tools**

Splitting and Joining

UNIX **Power Tools**

UNIX Power Tools

Commands 3
Splitting and Joining

Commands which break files apart . . .

. . . and ones to put them together again!

Horizontal

- splitting
  head
  tail
- joining
  cat

Vertical

- splitting
  cut
- joining
  paste

Horizontal split and join

Splitting

head -20 fred

'head -*n*' – first *n* lines

tail -50 fred

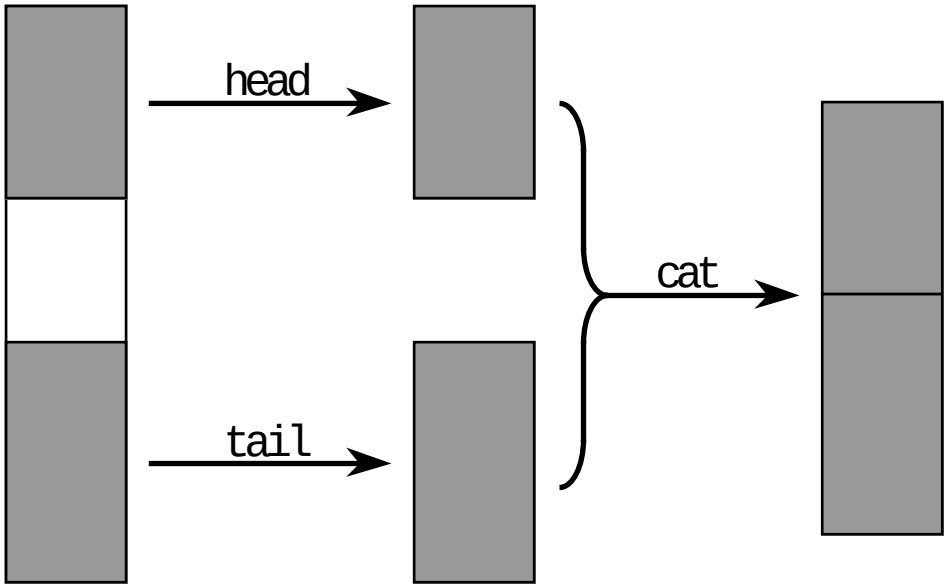
'tail -*n*' – last *n* lines

'tail +*n*' – from line *n* onwards

Joining

cat tom dick >harry

(N.B. use of redirection)



Vertical split

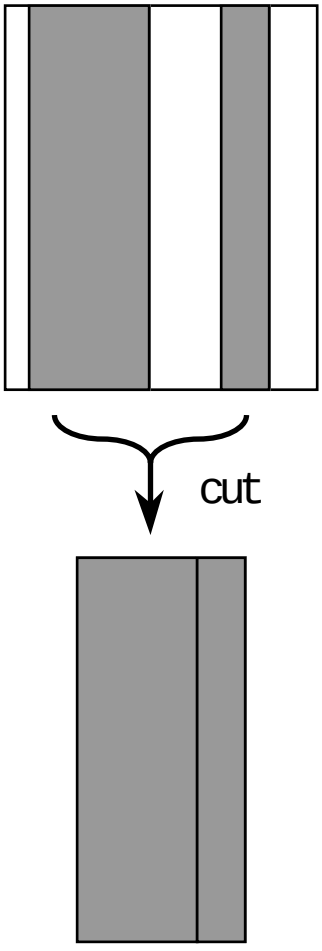
cut -c10-20,30-35 fred

'cut -c*n-m*' – chars *n* to *m* of each line

cut -d: -f1,7,8 fred

'cut -f*n,m*' – fields *n*, *m* of each line (tab delimited)

'cut -d*x*' – field delimiter is *x*



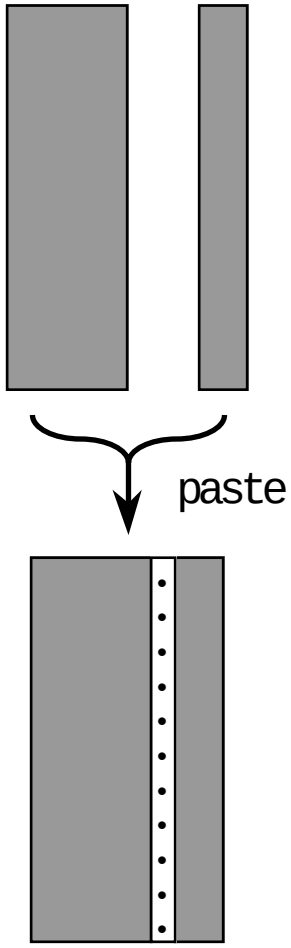
Vertical join

paste tom dick

- corresponding lines concatenated

'paste -d*list* tom dick'

- use characters in *list* as column separators



UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Shell 3
Linking Commands

UNIX **Power Tools**

UNIX **Power Tools**

Shell 3
Linking Commands

- pipes
  - linking the output and input

    ```
    a | b
    ```

- filters
  - commands made for piping

- sequences of commands

  ```
  a ; b
  ```

- conditional sequences

  ```
  a && b
  a || b
  ```

Putting them together – PIPES

Temporary files to build up complex sequences

e.g. the first 10 characters of the first 5 lines of fred

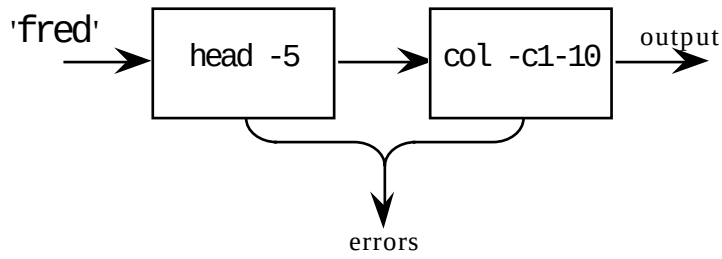
```
$ head -5 fred >tmpfile
$ cut -c1-10 tmpfile
$ rm tmpfile
```

- commands run one after the other

UNIX pipes join the standard output of one command to the standard input of another

```
$ head -5 fred | cut -c1-10
```

- commands run at the same time
- standard error from both are mixed together (!)



DOS has pipes too . . .

. . . but just a shorthand for hidden temporary files!

Filters

- Some commands only work on named files:

  e.g. copying – `cp` *`from-file to-file`*

- Many take standard input as default

  `cat, head, tail, cut, paste,` etc.

- These are called filters
  - very useful as part of pipes

- Filter pipes may be very long

```
$ head -50 | cut -c1-10 | tail +40 | more
```

- Also special filename '–'

  e.g. `cat header - footer`

  – this adds the files '`header`' and '`footer`' to the beginning and end of the standard input

  N.B. not all commands recognise '–'

More ways to put things together

Several ways to run commands one after the other:

Simple sequence using ';'

```
$ echo hello; cat fred
hello
this is the contents
of the file called fred
$
```

Conditional and '&&'

second command only runs if first is successful

```
$ echo -n hello && echo bye bye
hellobye bye
$
```

N.B. notice use of '-n' option for echo

Conditional or '||'

second command only runs if first is not successful

```
$ echo hello || echo bye bye
hello
$
```

Hands on

- copy the file `long-file` from the `tools` directory

```
$ cp tools/long-file .
```

- construct a command that lists lines 100-105 of it

  try first with a temporary file
  and then with a single command line

- do it again, but with the lines numbered:

  ① first number them 100, 101, ... 105

  ② then 1, 2, ... 6

- look at the output of `ls -l`

- construct a command line which lists all the files in the current directory, but is of the form:

```
date    filename
```

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Shell 4
Variables and Quoting

Shell 4
Variables and Quoting

- setting variables

```
name=value
```

- displaying the environment

```
set
```

- using variables

```
$name
${name-default}
```

- quoting

```
"hello $name"
'hello $name'
`echo hello`
```

Environment Variables

- set of `name=value` mappings
- most created during start-up (.profile, .login etc.)

Setting a variable:

```
myvar=hello
var2=" a value with spaces needs to be quoted"
export myvar
```

- no spaces before or after '=' sign
- variables need to be exported to be seen by other programs run from the shell
- in C shell: "`set name=val`" and no export

Listing all variables

```
$ set
HOME=/home/staff2/alan
myvar=hello
PATH=/local/bin:/bin:/local/X/bin
USER=alan
  . . .
$
```

Using Variables

- variables referred to by: $name

```
$ echo $myvar
hello
$ echo $var2
a value with spaces needs to be quoted
$
```

  - note that resulting value does not have quotes

- to avoid ambiguity can use: ${name}

```
$ echo a${myvar}bc
ahellobc
$
```

  - without the { } the shell would try to look for an variable called 'myvarbc'

- various sorts of default:

  ${name-default} – if name is not set use default

  ${name=default} – if unset, sets name to default

Quoting

- shell expands variables and wildcards ...

```
$ echo $myvar *5.doc
hello fred5.doc harry5.doc
```

... but not always wanted

- quoting gives you control over expansion
  - double quotes "$myvar *5.doc"
    variables – yes
    wildcards – no

```
$ echo "$myvar *5.doc"
hello *5.doc
```

  - single quotes '$myvar *5.doc'
    variables – no
    wildcards – no

```
$ echo '$myvar *5.doc'
$myvar *5.doc
```

  - backslash \$myvar
    protects single characters
    also used for special characters
    \n, \", \\, etc.

```
$ echo \$myvar "\\ *5.doc\""
$myvar \ *5.doc"
$
```

Back Quotes

- back quotes `` `echo hello` `` are very different
- they do not prevent variable or wildcard expansion
- the text in the quotes is:
  1. interpreted as a command
  2. executed
  3. the output is substituted back

```
$ cat `echo fred`
this is the contents
of the file called fred
$
```

- the command in the quotes may contain:
  - shell variables `` `echo $myvar` ``
  - wildcards `` `cat *.doc` ``
  - quotes `` `echo "$myvar"` ``
  - escapes `` `echo \*` ``
  - more backquotes `` `cat \`echo fred\`` ``

- example use: a file containing a list of file names

```
$ echo *.doc >my-docs
$ wc `cat my-docs`
```

UNIX Power Tools

UNIX Power Tools

UNIX Power Tools

UNIX Power Tools

UNIX Power Tools

UNIX Power Tools

Commands 4 Looking Inside

UNIX Power Tools

Commands 4
Looking Inside

Commands which work on file contents

Translating characters

tr *list1 list2 file*

tr -d *list file*

Sorting files

sort *file1 file2 file3*

Word count

wc *file*

Finding things

fgrep *string file1 file2*

find *dir* **-name** *name* **-print**

Translating characters

tr *list1 list2 file*

– changes characters from *list1*
to the corresponding character from *list2*

```
$ tr abcdef uvwxyz fred
hyllo
this is thy contynts
oz thy zily cullyd fryd
```

tr -d *list file*

– deletes characters from *list*

```
$ tr -d 'rst ' fred
hello
hiisheconen
ofhefilecalledfred
```

some useful translations:

`tr -d '\015'` – DOS → UNIX conversion
strips carriage returns
(UNIX → DOS is harder!)

`tr '\012' '\015'` – UNIX → MAC conversion
newline to carriage return

- both using octal character codes

`tr '[A-Z]' '[a-z]'` – upper to lower case conversion

Sorting files

sort *file1 file2 file3*

– sorts lines from the files in ascending order

```
$ cat a-file
this is the first line
and this is the second
the third is here
four is the last
$ sort a-file
and this is the second
four is the last
the third is here
this is the first line
$
```

options:

+n	numeric order
+r	reverse order
+u	'unique', remove duplicate lines
+*n* **-***m*	sort on fields *n* to *m*-1 inclusive
-t*c*	use the character *c* as the field separator default is any number of spaces or tabs

N.B. conventions different from **cut**
in particular **cut** numbers fields from 1
but **sort** numbers from 0 (!!!)

Word count

wc *file1 file2 file3*

- counts the number of characters, words and lines in each file
- also outputs total for all files (when more than one)

options:

-c	character count only
-w	word count only
-l	line count only

any combination can be given – default is '**-lwc**'

Examples

```
$ ls | wc -l
```

number of files in the current directory

```
$ wc -c fred
```

size of fred in bytes

Finding things

fgrep *string file1 file2*

– print all lines in `file1` and `file2` which contain the characters `string`

(some) options:

`-l` list matching files only

`-v` find lines not containing the string

N.B. two other variants: `grep` and `egrep`

find *dir* **-name** *fname* **-print**

– list all files named `fname` in the directory `dir` and any subdirectory of `dir`

options: innumerable!

Examples

```
$ fgrep -l UNIX *.doc
```

list all '`.doc`' files containing the word '`UNIX`'

```
$ find /usr/home2/alan -name '*urgent*' -print
```

find all files within '`/usr/home2/alan`' whose file name includes '`urgent`' and print the names

Hands on

- create (using `cat >`) five files `ax, by, cy, dx, ex` make their content and lengths different

- set an environment variable `weekly`

```
whichfile="by ex"
```

- use it to `cat` the files

```
cat $whichfile
```

- what will happen if you quote it?

```
cat "$whichfile"
```

- try it!

- use the variable `whichfile` to an echo command which produces the following:

```
$whichfile="by ex"
```

☞ ☞ ☞ ☞ Hands on (ctd.) ✍ ✍ ✍

❍ imagine you are automating a backup procedure

☞ create two files `weekly` and `monthly`

```
$ cat >weekly
ax
by
cy
^D
$ echo ?x >monthly
$
```

☞ use them to list and word count files

```
ls `cat weekly`
wc `cat monthly`
```

☞ create an environment variable `whichfiles`

```
$ whichfiles="weekly"
```

☞ now create a command line which sorts the list referred to by `whichfiles` but make it generic. That is `sort `cat weekly`` will not do!

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Shell 5
Scripts and Arguments

Shell 5

Scripts and Arguments

- simple scripts

  #!/bin/sh

  chmod

- grouping commands

  (… ; …)

  { … ; … ; }

- exit codes

  exit *n*

  $?

- command line arguments

  $1, $2, …

  $#, $*, ${@:-"$@"}

- HERE files

  cat <<HERE

Shell Scripts

- if you do the same thing over and over

  . . . put it in a shell script

- shell scripts are files:

  ① starting with:

  #!/bin/sh

  ② containing shell commands

  ③ made executable by

  chmod a+x

- executed using a copy of the shell

```
$ cat >my-first-script
#!/bin/sh
echo hello world
$ chmod a+x my-first-script
$ my-first-script
hello world
$
```

Exit Codes

- as well as output and errors . . .
  . . . also an exit code

- an integer: 0 – success
  anything else – failure

- examine using $?
  – the exit code of the last command

```
$ cat fred
this is the contents
of the file called fred
$ echo $?
0
$ cat freda
cat: freda: No such file or directory
$ echo $?
1
$
```

Setting Exit Codes

- set the exit code of a script with:

  exit *n*

- exits (sub)shell immediately

- logs out if executed at the top-level !

```
$ cat >script-2
#!/bin/sh
echo exiting now
exit 17
echo this line never gets typed
$ chmod a+x script-2
$ script-2
exiting now
$ echo $?
17
$
```

Grouping

- brackets used to group commands

```
$ (echo hello; echo bye bye) >fred
$ cat fred
hello
bye bye
$
```

  - ❍ commands run in a sub-shell

- curly brackets similar

```
$ { echo hello; echo bye bye; } >fred
$ cat fred
hello
bye bye
$
```

  - ① no sub-shell
  - ② { and } treated as commands
    ⇒ start of line or after ;

Scope within Groups

- round brackets give a sub-shell

  ⇒ commands have local effect:
  i.e. `exit`, variables and `cd`

- curly brackets in outer-shell

  ⇒ all commands affect it

```
$ myvar="bye bye"
$ (myvar=fred; exit 53)
$ echo $? $myvar
53 bye bye
$ { myvar=fred; }
$ echo $myvar
fred
$ { exit 53; }
```

– system logged out!

Arguments

- general form of command

  command arg1 arg2 … argn

- each argument may be:
  - an option: e.g. -x
  - a file name: e.g. fred
  - anything else: e.g. "a message"

- within a script arguments are:

  $1, $2, $3, ...

- count of arguments in **$#**

  (N.B. C programmers $# ≠ argc)

```
$ cat >show-args
#!/bin/sh
echo nos=$# 1=$1 2=$2 3=$3 4=$4
$ chmod a+x show-args
$ show-args a b c
nos=3 1=a 2=b 3=c 4=
$
```

Quoting Arguments

- spaces separate arguments

```
$ show-args a bcd e23 -x
nos=4 1=a 2=bcd 3=e23 4=-x
```

- wildcards expand to separate names

```
$ show-args *5.doc
nos=2 1=fred5.doc 2=harry5.doc 3= 4=
```

- spaces in variables make several args

```
$ longvar="a b c d"
$ show-args $longvar
nos=4 1=a 2=b 3=c 4=d
```

- quotes create a single argument

```
$ show-args a "b c" d
nos=3 1=a 2=b c 3=d 4=
```

- but back quotes don't !

```
$ show-args `echo a b`
nos=2 1=a 2=b 3= 4=
```

Passing Them On

- sometimes want to pass arguments on to another command

```
$ cat >list
#!/bin/sh
echo listing of file
cat $1 $2 $3
```

  but how many arguments to pass?

- can get whole list:

$*	–	the whole list (unquoted)
$@	–	similar except . . .
"$*"	–	the whole list as one argument
"$@"	–	the whole list properly quoted
${@+"$@"}	–	safest for older shells

- use `shift` to remove first argument

```
$ cat >mess-wc
#!/bin/sh
echo message is $1
shift
wc $*    # doesn't count first argument
```

HERE files

- to give constant text to a command
  - single line – use echo and pipe

```
$ echo hello | cat >fred
```

  - lots of lines – use HERE file

```
$ wc <<HERE
> this is two lines
> of text
> HERE
       2       6      26
$
```

N.B. secondary prompt "> "

- you can use any terminator,
  not just HERE!

HERE file substitution

- variables are expanded in HERE files

```
$ myvar=fred
$ cat <<HERE
> Dear $myvar how are you?
> HERE
Dear fred how are you?
$
```

- prevent expansion with quotes

```
$ cat <<"HERE"
> Dear $myvar how are you?
> HERE
Dear $myvar how are you?
$
```

. . . or backslash

```
$ cat <<HERE
> Dear $myvar try typing \$myvar with a \\
> HERE
Dear fred try typing $myvar with a \
$
```

- wildcards never expanded

UNIX Power Tools

UNIX Power Tools

UNIX Power Tools

UNIX Power Tools

Commands 5

Numbers and Values

UNIX Power Tools

UNIX Power Tools

Commands 5
Numbers and Values

- evaluating expressions
  expr
- testing conditions
  test
  […]
- running commands
  eval
- doing something else
  . *command*
  exec

Evaluating Expressions

- we can set and use variables . . .
  . . . how do we calculate with them?

  expr expression

- **expr** is a program, it
  - evaluates the expression
  - sends the result to standard output

```
$ expr 1 + 2
3
```

- each item must be separate

```
$ expr 1+ 2
expr: syntax error
```

- beware of null values!

```
$ expr $undefinedVar + 2      # expands to expr + 2
expr: syntax error
```

- use with backquotes in scripts

```
mycount=`expr $mycount + 1`
```

Testing Conditions

- **expr** can evaluates logical conditions

```
$ expr 2 ">" 1 \& 3 \> 2      # & means and
1                             # 1 means true
```

  - operators need to be quoted or escaped
  - result to standard output

- **test** also evaluates logical conditons

```
$ test 2 -gt 1 -a 3 -gt 2     # -a means and
$ echo $?
0                             # 0 means true !!!
```

  - different syntax
  - returns result in exit code
  - zero exit code is success – true!

- shorthand using **[…]**

```
$ [ 1 = 2 ]
$ echo $?
1                             # false
```

  - needs spaces like curly brackets

Running Commands

- you may want to execute a command which is in a variable:

```
$ mycommand="echo hello"
$ $mycommand
hello
```

- but doesn't work for setting variables

```
$ setcommand="var=newval"
$ $setcommand
var=newval: not found
$
```

... or pipes, etc.

```
$ pipecommand="echo hello | cat"
$ $pipecommand
hello | cat
$
```

- **eval** does it right!

```
$ eval $setcommand
$ echo $var
newval
$ eval $pipecommand
hello
```

Doing Something Else

- normally commands run in a sub-shell
- you can control this:
- dot makes scripts run in the main shell

```
$ . .profile
```

  - used for setting variables in login scripts

- exec makes command replace the shell

```
$ ( echo first line
> exec echo hello
> echo last line never executed )
first line
hello
$
```

  - note use of brackets to force sub-shell

Hands on

- experiment with `(exit n)` followed by `echo $?`
- enter the script `show-args` on the slide "Arguments"
- experiment with different quoted arguments to it
- create a file `pass-on`

```
show-args $*
echo now quoted \*
show-args "$*"
echo now @
show-args $@
echo quoted @
show-args "$@"
echo now magic
show-args ${@+"$@"}
```

- try the following

```
$ pass-on a "b c" d
$ pass-on a "" d
$ pass-on ""
$ pass-on
```

Hands on (ctd.)

- the first script I create on any platform is:

```
$ cat >chx
chmod a+x $*
$ chmod a+x chx
```

- type it in and check you understand what it does

- you may find UNIX cannot find your new script
  if this happens try changing your PATH environment variable (which tells UNIX where to look for commands) to include the current directory:

```
$ PATH=.:$PATH
```

- write a script called `lines` which behaves as follows:

```
$ lines 150 180 long-file
```

  this should list lines 150 to 180 (inclusive) of the file called `long-file` with line numbers (150, 151 etc.)

- use chx to make `lines` executable

- not simply an exercise – this is exactly the script I wrote recently to help me with C compiler error messages.

  useful scripts don't have to be long ones!

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Shell 6
Control Structures

Shell 6
Control Structures

- testing conditions
  `if` ... `then` ... `else` ... `fi`
  `case` ... `in` ... `esac`

- looping
  `for` ... `do` ... `done`
  `while` ... `do` ... `done`
  `until` ... `do` ... `done`

- catching signals
  `trap`

- functions
  name`() { ... }`

Testing Conditions

- shell has an '**if**' statement

```
if [ $# -lt 2 ]
then
  echo not enough arguments
else
  cp $1 $2
fi
```

  - the 'condition' part is a command
  - typically test, but not always

- also '**case**' statement

```
case $1 in
  -e)  echo $2 ;;
  -c)  cat $2 ;;
   *)  echo bad first argument $1 ;;
esac
```

  - the patterns (before bracket) like wildcards
    ⇒ *) acts as default
  - fallthrough without double semicolon ;;

Looping – for

- **`for`** does the same thing to several values

```
$ for i in 1 2 3
> do
> echo -n " abc$i"
> done
 abc1 abc2 abc3
$
```

- useful for iterating over files

```
for i in *.doc
```

- can be used for iterating through arguments

```
for i in $*
```

special form handles quoting correctly

```
for i
do
    echo "listing of $i"
    cat $i
done
```

Looping – while and until

- **`while`** and **`until`** loops based on a condition

```
myct=1
while [ $myct -lt 10000 ]
do
    echo this is line $myct
    myct=`expr $myct + 1`
done
```

- some special commands for tests

`:`	–	always returns exit code 0
`true`	–	always returns exit code 0
`false`	–	always returns exit code 1

```
echo for ever
while :
do
    echo and ever
done
```

- `:`, `true` and `false` all ignore their arguments

  ⇒ `:` can be used for comments . . .

  . . . but `#` is better

Catching Signals

- UNIX signals caused by:
  - errors e.g. arithmetic overflow
  - user interrupts (ctrl-C)

  (see man signal for full list)

- signals force the script to exit

- may be untidy (e.g. temporary files)

- trap allows tidy up

```
tmpfile=/tmp/myprog$$
trap "rm $tmpfile" 0 1 2 3 15
  ... more commands which use tmpfile ...
```

- note:
  - signal **0** means normal shell exit
    (see man signal for full list)
  - use of **$$** - the process id to invent unique temporary file names

Functions

- also can define 'functions' in scripts

```
f() {
  myvar=$1
}
```

- used like normal command

```
f abc
```

- share variables with main script
- but have their own argument list

```
$ cat >fun-script
#!/bin/sh
myvar=xyz
f() {
  myvar=$1
}
f $2
echo $myvar
$ chx fun-script
$ fun-script a b
b
```

UNIX **Power Tools**

UNIX **Power Tools**

UNIX **Power Tools**

Commands 6

Scriptable Programs

UNIX **Power Tools**

UNIX **Power Tools**

UNIX Power Tools

Commands 6
Scriptable Programs

- shell works on whole files
- scriptable programs work within files

sed – 'stream editor'
rather archane - OK if you like ed

awk – C like programming language
process file line by line

perl – cross between awk and shell
used heavily in web CGI scripts

- focus on awk
- use shell + awk for maximum effect

awk – structure

awk -f *command-file data-file*

- ❍ processes *data-file* line by line
- ❍ uses awk script in *command-file*

- general format of awk script:

```
BEGIN { initial statements }
pattern{ statements }
pattern{ statements }
END { final statements }
```

- ❍ statements within { ... } may be many lines

- execution of script:
  - ① BEGIN statements executed before file read
  - ② patterns matched against each line of data
    relevant statements executed on match
  - ③ END statements executed at end of file

- patterns:
  - ❍ may be regular expressions e.g. /^[a-z]*$/
  - ❍ or general condition e.g. $1=="abc"
  - ❍ empty means every line

awk – variables

- two kinds of variable
  - simple names

```
BEGIN { count=0}
{ count=count+1}
END { print count}
```

(counts lines in data file)

  - field numbers
    - **$3** – third field
    - **$0** – the whole line

```
BEGIN { count=0; sum=0 }
$1=="add"{ count=count+1;
  sum=sum+$2
}
END { print sum/count}
```

(average of second field where first is "add")

- any variable may contain a number or string

- fields are separated by white space
  but default can be changed using variable FS

awk – statements

- assignment

  variable **=** *expression*

- expressions
  - C-like, including == for equality!
  - juxtaposition for string concatenation

```
var="hello " $2
```

- printing
  - the default action when no statements given is to print the line (that is not even {})
  - print the current line:

```
print
```

  - print selected fields or values

```
print "second two fields", $2, $3
```

  - formatted print (like C printf)

```
printf "1=%s 2=%d", $1, $2
```

awk – control structures

- standard set of conditionals and loops
  - for example:

```
if ( $2<0 ) {
    print $1 " is overdrawn"
}
```

```
for ( i=1; i<=10; i++ ) {
    print "This line is", i
}
```

  - all closely follow C syntax

- also special control over data file
  - normally all matching patterns are executed
  - can skip further matches with **next**

```
($1=="x"){ next }
{ print $2 }
```

  - prints the second field of each line except those with 'x' as the first field

awk – and more ...

- arrays / dictionaries

```
print a[3]
b["fred"] = 7
```

- output to named files

```
print "hello" >"fred"
```

- execution of shell commands

```
system("wc " $1)
```

- various built in functions:
  - numerical (e.g. exp, sqrt, log)
  - string manipulation including regular expression substitution

- 'new awk' has user functions too

  (called nawk on some systems)

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ experiment with awk, using some of the examples from the previous pages

☞ go over any examples from the slides

☞ the next page is an awk script and associated shell script for getting the bounding box information out of an encapsulated postscript picture

☞ notice how the shell script checks arguments and the awk script scans the file

☞ this is part of a suite of scripts I wrote to manipulate and edit 120 epsf pictures for a textbook on HCI which I co-authored

☞ it is the simplest . . . but quite typical

☞ general lesson – use each tool where it is best fitted

epsf-getbbx

```
#!/bin/sh
case $# in
    0)  awk -f epsf-getbbx.awk;;  # standard input
    1)  awk -f epsf-getbbx.awk $1;;
    *)  echo "usage:" $0 "{epsf-file}"
        exit 1;;
esac
exit 0
```

epsf-getbbx.awk

```
#  epsf-getbbx.awk
#  gets bounding box
#  looks for lines of the form:
#   `%%BoundingBox: 132 220 473 457'
#                   x0  y0  x1  y1

BEGIN {
}

$1 == "%%BoundingBox:"  {
  print $2, $3, $4, $5
  exit 0
}


{
  next
}

END {
  exit 1
}
```