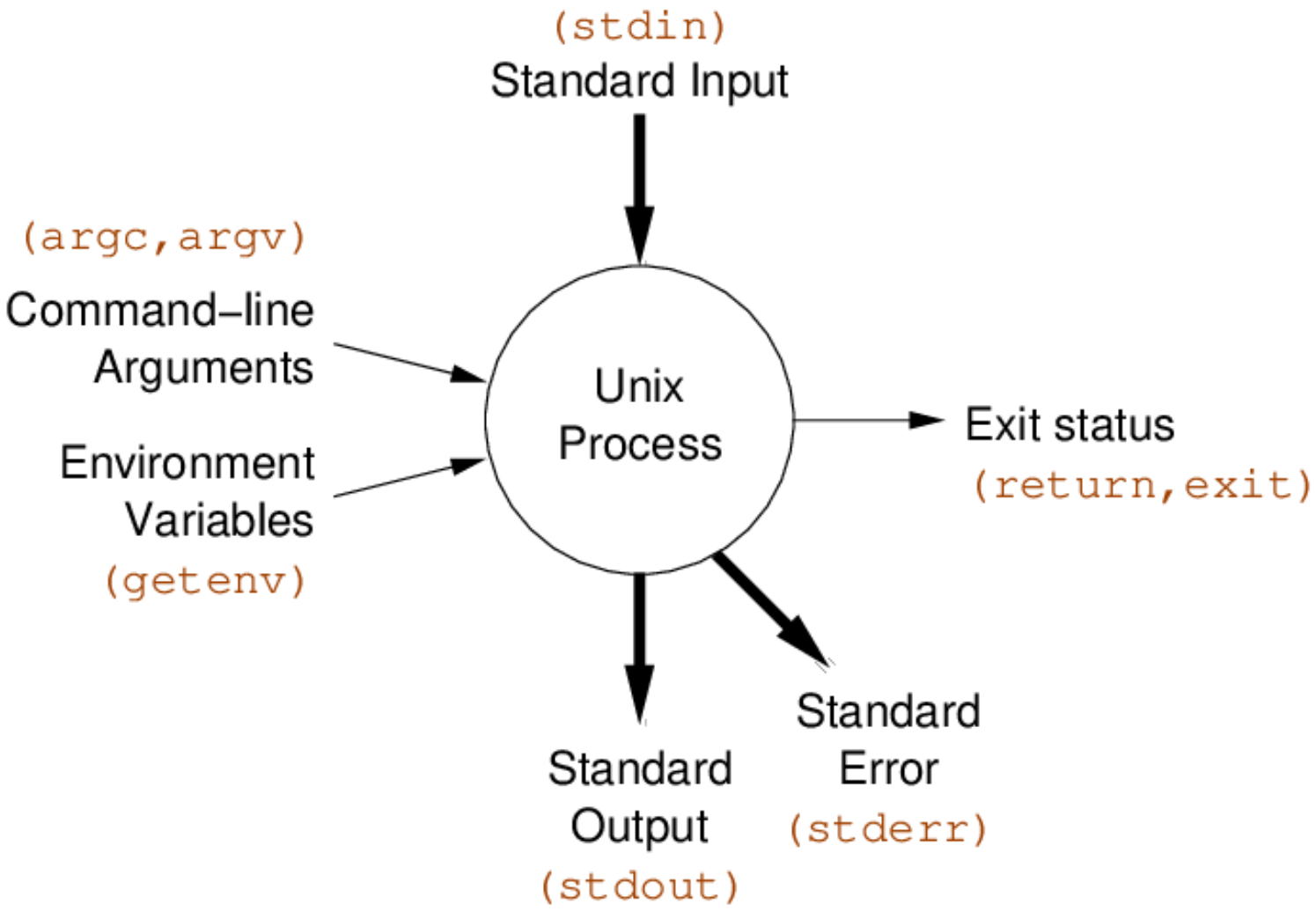
Unix Processes

A Unix process executes in this environment



Unix Processes: C programmer's view

Components of process environment (C programmer's view):

- `char *argv[]` - command line arguments
- `int argc` - size of `argv[]`
- `char *env[]` - `name=value` pairs from parent proces
- `FILE *stdin` - input byte-stream, e.g. `getchar)`
- `FILE *stdout` - input byte-stream, e.g. (`putchar()`
- `FILE *stderr` - output byte-stream, e.g. `fputc(c, stderr)`
- `exit(int)` - terminate program, set exit status
- `return int` - terminate `main()`, set exit status

Output a file: C Code

```c
// write bytes of stream to stdout
void process_stream(FILE *in) {
    while (1) {
        int ch = fgetc(in);
        if (ch == EOF)
            break;
        if (fputc(ch, stdout) == EOF) {
            fprintf(stderr, "cat:");
            perror("");
            exit(1);
        }
    }
}
```

Process files/stdin: C Code

```c
// process files given as arguments
// if no arguments process stdin
int main(int argc, char *argv[]) {
    if (argc == 1)
        process_stream(stdin);
    else
        for (int i = 1; i < argc; i++) {
            FILE *in = fopen(argv[i], "r");
            if (in == NULL) {
                fprintf(stderr, "cat: %s: ", argv[i]);
                perror("");
                return 1;
            }
            process_stream(in);
            fclose(in);
        }
    return 0;
}
```

Count Lines, Words, Chars: C Code

```c
// count lines, words, chars in stream
void count_file(FILE *in) {
    int n_lines = 0, n_words = 0, n_chars = 0;
    int in_word = 0, c;
    while ((c = fgetc(in)) != EOF) {
        n_chars++;
        if (c == '\n')
            n_lines++;
        if (isspace(c))
            in_word = 0;
        else if (!in_word) {
            in_word = 1;
            n_words++;
        }
    }
    printf("%6d %6d %6d", n_lines, n_words, n_chars);
}
```

What is a filter?

Filter: a program that transforms a data stream.
On Unix, filters are commands that:

- read text from their standard input or specified files
- perform useful transformations on the text stream
- write the transformed text to their standard output

What is a filter?

Example: `cat MyProg.c`

- reads the text of the program in the file `MyProg.c`
- writes the (untransformed) text to standard output (i.e. the screen)

Example: `cat <MyProg.c`

- the shell (command interpreter) connects the file `MyProg.c` to standard input of `cat`
- `cat` reads its standard input
- writes the (untransformed) text to standard output (i.e. the screen)

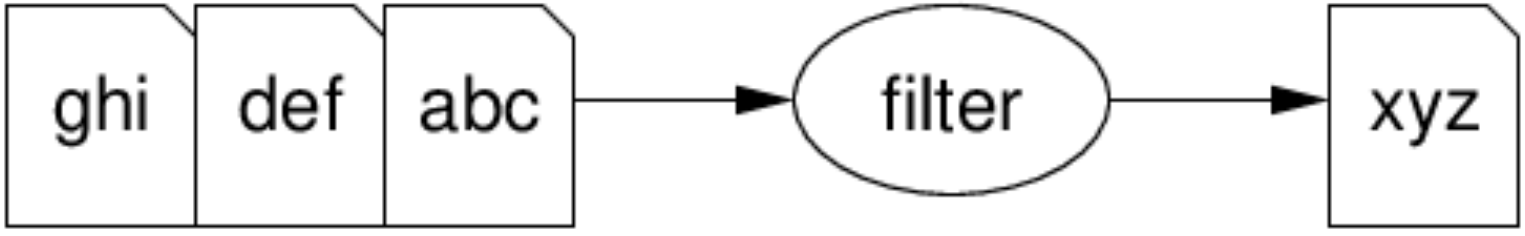
Using Filters

Shell I/O redirection can be used to specify filter source and destination:

```
filter < abc > xyz
```

Alternatively, most filters allow multiple sources to be specified:

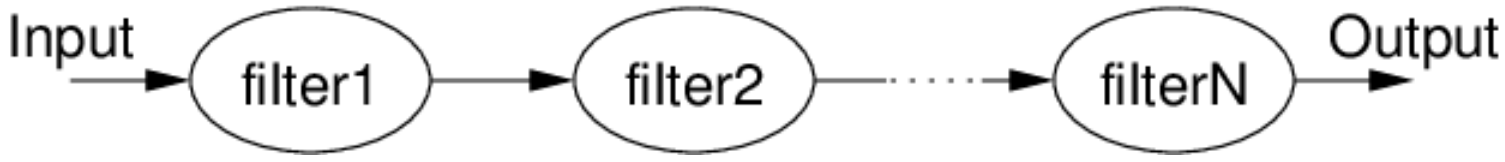
```
filter abc def ghi > xyz
```



Using Filters

In isolation, filters are reasonably useful
In combination, they provide a very powerful problem-solving toolkit.
Filters are normally used in combination via a pipeline:

```
filter1 | filter2 | ... | filterN
```



Note: similar style of problem-solving to function composition.

Using Filters

Unix filters use common conventions for command line arguments:

- input can be specified by a list of file names
- if no files are mentioned, the filter reads from standard input (which may have been connected to a file)
- the filename "–" corresponds to standard input

Examples:

```
# read from the file data1
filter data1
# or
filter < data1

# read from the files data1 data2 data3
filter data1 data2 data3

# read from data1, then stdin, then data2
filter data1 - data2
```

If filter doesn't cope with named sources, use `cat` at the start of the pipeline

Using Filters

Filters normally perform multiple variations on a task.
Selection of the variation is accomplished via command-line options:

- options are introduced by a - ("minus" or "dash")
- options have a "short" form, - followed by a single letter (e.g. `-v`)
- options have a "long" form, -- followed by a word (e.g. `--verbose`)
- short form options can usually be combined (e.g. `-av` vs `-a -v`)
- `--help` (or `-?`) often gives a list of all command-line options

Filters: Option

Most filters have *many* options for controlling their behaviour.
Unix **man**ual entries describe how each option works.
To find what filters are available: `man -k` *keyword*
The solution to all your problems: **RTFM**

Delimited Input

Many filters are able to work with text data formatted as *fields* (columns in spreadsheet terms).
Such filters typically have an option for specifying the delimiter or field separator.
(Unfortunately, they often make different assumptions about the default column separator)
Example (tab-separated columns):

```
John    99
Anne    75
Andrew  50
Tim     95
Arun    33
Sowmya  76
```

Delimited Input

Example (verticalbar-separated columns, enrolment file):

```
COMP1011|2252424|Abbot, Andrew John    |3727|1|M
COMP2011|2211222|Abdurjh, Saeed        |3640|2|M
COMP1011|2250631|Accent, Aac-Ek-Murhg  |3640|1|M
COMP1021|2250127|Addison, Blair        |3971|1|F
COMP4012|2190705|Allen, David Peter    |3645|4|M
COMP4910|2190705|Allen, David Pater    |3645|4|M
```

Example (colon-separated columns, old Unix password file):

```
root:ZHolHAHZw8As2:0:0:root:/root:/bin/bash
jas:nJz3ru5a/44Ko:100:100:John Shepherd:/home/jas:/bin/bash
cs1021:iZ3sO90O5eZY6:101:101:COMP1021:/home/cs1021:/bin/bash
cs2041:rX9KwSSPqkLyA:102:102:COMP2041:/home/cs2041:/bin/bash
cs3311:mLRiCIvmtI9O2:103:103:COMP3311:/home/cs3311:/bin/bash
```

`cat`: the simplest filter

The `cat` command copies its input to output unchanged (identity filter).
When supplied a list of file names, it con**cat**enates them onto stdout.
Some options:

`-n`	**n**umber output lines (starting from 1)
`-s`	**s**queeze consecutive blank lines into single blank line
`-v`	display control-characters in **v**isible form (e.g. `^C`)

The `tac` command copies files, but reverses the order of lines.

`wc`: word counter

The `wc` command is a summarizing filter.
Useful with other filters to count things.

`-c`	counts the number of **c**haracters (incl. `\n`)
`-w`	counts the number of **w**ords (non-white space)
`-l`	counts the number of **l**ines

Some filters find counting so useful that they define their own options for it (e.g. `grep -c`)

`tr`: transliterate characters

The `tr` command converts text char-by-char according to a mapping.

```
tr 'sourceChars' 'destChars' < dataFile
```

Each input character from *sourceChars* is mapped to the corresponding character in *destChars*.
Example:

```
tr 'abc' '123' < someText
```

Has *sourceChars*=`'abc'`, *destChars*=`'123'`, so:
`a` $\rightarrow$ `1`, `b` $\rightarrow$ `2`, `c` $\rightarrow$ `3`
Note: `tr` doesn't accept file name on command line.

`tr`: transliterate characters

Characters that are not in *sourceChars* are copied unchanged to output.
If there is no corresponding character (i.e. *destChars* is shorter than *sourceChars*), then the last char in *destChars* is used.
Shorthands are available for specifying character lists:
E.g. `'a-z'` is equivalent to `'abcdefghijklmnopqrstuvwxyz'`
Note: newlines will be modified if the mapping specification requires it.

`tr`: transliterate characters

Some options:

`-c`	map all characters *not* occurring in *sourceChars* (**c**omplement)
`-s`	**s**queeze adjacent repeated characters out (only copy the first)
`-d`	**d**elete all characters in *sourceChars* (no *destChars*)

tr: transliterate characters

Examples:

```
# map all upper-case letters to lower-case equivalents
tr 'A-Z' 'a-z' < text

# simple encryption (a->b, b->c, ... z->a)
tr 'a-zA-Z' 'b-zaB-ZA' < text

# remove all digits from input
tr -d '0-9' < text

# break text file into individual words, one per line
tr -cs 'a-zA-Z0-9' '\n' < text
```

`head`/`tail`: select lines

- `head` prints the first *n* (default 10) lines of input.
  E.g. `head file` prints first 10 lines of `file`.
  `-n` option changes numbe rof lines head/tail prints.
- The `tail` prints the last *n* lines of input.
  E.g. `tail-n 30 file` prints last 30 lines of `file`.
- Combine `head` and `tail` to select a range of lines.
  E.g. `head -n 100 | tail -n 20` copies lines 81..100 to output.

With more than one file prefixes with name (see labs).

`egrep`: select lines matching a pattern

The `egrep` command only copies to output those lines in the input that match a specified pattern.
The pattern is supplied as a regular expression on the command line (and should be quoted using single-quotes).
Some options:

`-i`	ignore upper/lower-case difference in matching
`-v`	only display lines that *do not* match the pattern
`-w`	only match pattern if it makes a complete word

The grep family

`egrep` is one of a group of related filters using different kinds of pattern match:

- `grep` uses a limited form of POSIX regular expressions (no `+ ? |` or parentheses)
- `egrep` (extended grep) implements the full regex syntax
- `fgrep` finds any of several (maybe even thousands of) fixed strings using an optimised algorithm.

(The name `grep` is an acronym for **G**lobally search with **R**egular **E**xpressions and **P**rint)

Regular Expressions

A *regular expression* (regex) defines a a set of strings.
A *regular expression* usually thought of as a pattern
Specifies a possibly infinite set of strings.
They can be succinct and powerful.
Regular expressions libraries are available for most languages.
In the Unix environment:

- a lot of data is available in plain text format
- many tools make use of regular expressions for searching
- effective use of regular expressions makes you more productive

A POSIX standard for regular expressions defines the "pattern language" used by many Unix tools.

Regular expressions Basics

Regular expressions specify complex patterns concisely & precisely.

- Default: a character matches itself. E.g. **a** has no special meaning so it matches the character **a**.
- Repetition: *p** denotes zero or more repetitions of *p*.
- Alternation: $pattern_1$ | $pattern_2$ denotes the union of $pattern_1$ and | $pattern_2$.

  E.g. `perl|python|ruby` matches any of the strings `perl`, `python` or `ruby`

- Parentheses are used for grouping e.g. *a*(,*a*)* denotes a comma separated list of *a*'s.
- The special meanings of characters can be removed by escaping them with \ e.g. * matches the * character anywhere in the input.

The above 5 special characters ()*—\are sufficient to express any regular expression but many more features are present for convenience & clarity.

Patterns for matching Single Characters

- The special pattern `.` (dot) matches any single character.
- Square brackets provide convenient matching of any **one** of a set of characters.
  [*listOfCharacters*] matches any single character from the list of characters. E.g. `[aeiou]` matches any vowel.
- A shorthand is available for ranges of characters [*first* – *last*]
  Examples: `[a-e]` `[a-z]` `[0-9]` `[a-zA-Z]` `[A-Za-z]` `[a-zA-Z0-9]`
- The matching can be inverted [`^`*listOfCharacters*]
  E.g. `[^a-e]` matches any character *except* one of the first five letters
- Other characters lose their special meaning inside bracket expressions.

Anchoring Matches

We can insist that a pattern appears at the start or end of a string

- the start of the line is denoted by `^` (uparrow) E.g. `^[abc]` matches either `a` or `b` or `c` at the start of a string.
- the end of the line is denoted by `$` (dollar) E.g. `cat$` matches `cat` at the end of a string.

Repetition

We can specify repetitions of patterns

- *p*`*` denotes zero or more repetitions of *p*
- *p*`+` denotes one or more repetitions of *p*
- *p*`?` denotes zero or one occurence of *p*

E.g. `[0-9]+` matches any sequence of digits (i.e. matches integers)
E.g. `[-'a-zA-Z]+` matches any sequence of letters/hyphens/apostrophes
(this pattern could be used to match words in a piece of English text, e.g. `it's`, `John`, ...)
E.g. `[^X]*X` matches any characters up to and including the first `X`

Repetition

If a pattern can match several parts of the input, the first match is chosen.
Examples:

Pattern	**Text** (with match underlined)
`[0-9]+`	i=*1234* j=56789
`[ab]+`	*aabbababababaaa*cabba
`[+]+`	C*++* is a hack

Regular Expression Examples

Regex	**Matches**
`abc`	the string of letters `"abc"` E.g. `abc`
`a.c`	strings of letters containing `'a'` followed by `'c'` with any single character in between E.g. `abc, aac, acc, aXc, a2c, ...`
`ab*c`	strings of letters containing `'a'` followed by `'c'` with any number of `'b'` letters in between E.g. `ac, abc, abbc, abbbc, ...`
`a|the`	either the string `"a"` or the string `"the"` E.g. `a, the`
`[a-z]`	any single lower-case letter E.g. `a, b, c, ... z`

`cut`: vertical slice

The `cut` command prints selected parts of input lines.

- can select fields (assumes tab-separated columnated input)
- can select a range of character positions

Some options:

`-f`*listOfCols*	print only the specified fields (tab-sepa
`-c`*listOfPos*	print only chars in the specified positio
`-d'`*c*`'`	use character *c* as the field separator

Lists are specified as ranges (e.g. `1-5`) or comma-separated (e.g. `2,4,5`).

cut: vertical slice

Examples:

```
# print the first column
cut -f1 data

# print the first three columns
cut -f1-3 data

# print the first and fourth columns
cut -f1,4 data

# print all columns after the third
cut -f4- data

# print the first three columns, if '|'-separated
cut -d'|' -f1-3 data

# print the first five chars on each line
cut -c1-5 data
```

Unfortunately, there's no way to refer to "last column" without counting the columns.

paste: combine files

The paste command displays several text files "in parallel" on output.
If the inputs are files a, b, c

- the first line of output is composed of the first lines of a, b, c
- the second line of output is composed of the second lines of a, b, c

Lines from each file are separated by a tab character or specified delimiter(s).
If files are different lengths, output has all lines from longest file, with empty strings for missing lines.
Interleaves lines instead with -s (serial) option.

paste: combine files

Example: using paste to rebuild a file broken up by cut.

```
# assume "data" is a file with 3 tab-separated columns
cut -f1 data > data1
cut -f2 data > data2
cut -f3 data > data3
paste data1 data2 data3 > newdata
# "newdata" should look the same as "data"
```

`sort`: sort lines

The `sort` command copies input to output but ensures that the output is arranged in some particular order of lines.
By default, sorting is based on the first characters in the line.
Other features of `sort`:

- understands that text data sometimes occurs in delimited fields.
  (so, can also sort fields (columns) other than the first (which is the default))
- can distinguish numbers and sort appropriately
- can ignore punctuation or case differences
- can sort files "in place" as well as behaving like a filter
- capable of sorting *very large* files

`sort`: sort lines

Some options:

`-r`	sort in descending order (**r**everse sort)
`-n`	sort numerically rather than lexicographically
`-d`	dictionary order: ignore non-letters and non-digits
`-t'`*c*`'`	use character *c* to separate columns (default: space)
`-k`*n*`'`	sort on column *n*

Note: the `' '` around the separator char are usually not necessary, but are useful to prevent the shell from mis-interpreting shell meta-characters such as `'|'`.
Hint: to specify TAB as the field delimiter with an interactive shell like bash, type CTRL-v before pressing the TAB key.

sort: sort lines

Examples:

```
# sort numbers in 3rd column in descending order
sort -nr -k3 data

# sort the password file based on user name
sort -t: -k5 /etc/passwd
```

`uniq`: remove or count duplicates

The `uniq` command by default removes all but one copy of *adjacent* identical lines.
Some options:

`-c`	also print number of times each line is duplicated
`-d`	only print (one copy of) duplicated lines
`-u`	only print lines that occur uniquely (once only)

Surprisingly useful tool for summarising data, typically after extraction by cut. Always preceded by sort (why?).

```
# extract first field, sort, and tally
cut -f1 data  |  sort  |  uniq -c
```

`join`: database operator

`join` merges two files using the values in a field in each file as a common key.
The key field can be in a different position in each file, but the files must be ordered on that field. The default key field is 1.
Some options:

`-1` *k*	key field in first file is *k*
`-2` *k*	key field in second file is *k*
`-a` *N*	print a line for each unpairable line in file *N* (1 or 2)
`-i`	ignore case
`-t` *c*	tab character is *c*

join: database operator

Given these two data files (tab-separated fields)

```
# data1:                          # data2:
Bugs Bunny      1953              Warners Bugs Bunny
Daffy Duck      1948              Warners Daffy Duck
Donald Duck     1939              Disney  Goofy
Goofy   1952                      Disney  Mickey Mouse
Mickey Mouse    1937              Pixar   Nemo
Nemo    2003
Road Runner     1949
```

the command `join -t' ' -2 2 -a 1 data1 data2` gives

```
Bugs Bunny      1953    Warners
Daffy Duck      1948    Warners
Donald Duck     1939
Goofy   1952    Disney
Mickey Mouse    1937    Disney
Nemo    2003    Pixar
Road Runner     1949
```

sed: stream editor

The sed command provides the power of interactive-style editing in "filter-mode".
Invocation:

```
sed -e 'EditCommands' DataFile
sed -f EditCommandFile DataFile
```

How sed works:

- read each line of input
- check if it matches any patterns or line-ranges
- apply related editing commands to the line
- write the transformed line to output

sed: stream editor

The editing commands are very powerful and subsume the actions of many of the filters looked at so far.
In addition, sed can:

- partition lines based on patterns rather than columns
- extract ranges of lines based on patterns or line numbers

Option -n (**n**o printing):

- applies all editing commands as normal
- displays no output, unless p appended to edit command

sed: stream editor

Editing commands:

`p`	print the current line
`d`	delete (don't print) the current line
`s/`*RegExp*`/`*Replace*`/`	substitute first occurrence of string matching *RegExp* by *Replace* string
`s/`*RegExp*`/`*Replace*`/g`	substitute all occurrences of string matching *RegExp* by *Replace* string
`q`	terminate execution of `sed`

sed: stream editor

All editing commands can be qualified by line addresses or line selector patterns to limit lines where command is applied:

LineNo selects the specified line

StartLineNo,*EndLineNo*

selects all lines between specified line numbers

/*RegExp*/ selects all lines that match *RegExp*

/*RegExp*1/,/*RegExp*2/

selects all lines between lines matching reg exps

sed: stream editor

Examples:

```
# print all lines
sed -n -e 'p' < file

# print the first 10 lines
sed -e '10q' < file
sed -n -e '1,10p' < file

# print lines 81 to 100
sed -n -e '81,100p' < file

# print the last 10 lines of the file?
sed -n -e '$-10,$p' < file  # does NOT work
```

sed: stream editor

More Examples:

```
# print only lines containing 'xyz'
sed -n -e '/xyz/p' < file

# print only lines {\em{not}} containing 'xyz'
sed -e '/xyz/d' < file

# show the passwd file, displaying only the
# lines from "root" up to "nobody" (i.e. system accounts)
sed -n -e '/^root/,/^nobody/p' /etc/passwd

# remove first column from ':'-separated file
sed -e 's/[^:]*://' datafile

# reverse the order of the first two columns
sed -e 's/\([^:]*\):\([^:]*\):\(.*\)$/\2:\1:\3/'
```

`find`: search for files

The `find` commands allows you to search for files based on specified properties (a filter for the file system)

- searches an entire directory tree, testing each file for the required property.
- takes some action for all "matching" files (usually just print the file name)

Invocation:

```
find StartDirectory Tests Actions
```

where

- the *Tests* examine file properties like name, type, modification date
- the *Actions* can be simply to print the name or execute an arbitrary command on the matched file

find: search for files

Examples:

```
# find all the HTML files below /home/jas/web
find  /home/jas/web  -name '*.html'  -print

# find all your files/dirs changed in the last 2 days
find  ~  -mtime -2  -print

# show info on files changed in the last 2 days
find  ~  -mtime -2  -type f  -exec ls -l {} \;

# show info on directories changed in the last week
find  ~  -mtime -7  -type d  -exec ls -ld {} \;

# find directories either new or with '07' in their name
find  ~  -type d  \(  -name '*07*'  -o  -mtime -1  \)  -pri
```

find: search for files

More Examples:

```
# find all {\it{new}} HTML files below /home/jas/web
find  /home/jas/web  -name '*.html'  -mtime -1  -print

# find background colours in my HTML files
find  ~/web  -name '*.html'  -exec grep -H 'bgcolor' {} \;

# above could also be accomplished via ...
grep  -r  'bgcolor'  ~/web

# make sure that all HTML files are accessible
find  ~/web  -name '*.html'  -exec chmod 644 {} \;

# remove any really old files ... Danger!
find  /hot/new/stuff  -type f  -mtime +364  -exec rm {} \;
find  /hot/new/stuff  -type f  -mtime +364  -ok rm {} \;
```

Filter summary by type

- *Horizontal slicing* - select subset of lines:
  `cat`, `head`, `tail`, `*grep`, `sed`, `uniq`
- *Vertical slicing* - select subset of columns: `cut`, `sed`
- *Substitution*: `tr`, `sed`
- *Aggregation, simple statistics*: `wc`, `uniq`
- *Assembly* - combining data sources: `paste`, `join`
- *Reordering*: `sort`
- *Viewing* (always end of pipeline): `more`, `less`
- *File system filter*: `find`
- *Programmable filters*: `sed`, (and `perl`)