Course Outline

Syllabus Overview

1. Qualities of software systems
    a. Correctness, clarity, reliability, efficiency, portability
2. Techniques for software construction
    a. Analysis, design, coding, testing, debugging, tuning
    b. Interface design, documentation, configuration
3. Tools for software construction
    a. Filters (grep, sed, cut, sort, uniq, tr, …)
    b. Scripting languages (shell , Perl, Python)
    c. Intro to programming for the web
    d. Analysis/configuration/documentation tools (git, gprof, make, …)

Topic 1: Filters

Filters

- A program that transforms a data stream
- They 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
- They follow common conventions:
  - Input can be specified by a list of file names
  - If no files are mentioned, the filter reads from standard input
  - The filename "-" corresponds to standard input

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

- Filters can have "variations" on a task, with either short or long form:
  - Short: -v
  - Long --verbose
  - --help or (-?) gives a list of all command-line options

Manual

- Unix **man**ual entries describe how each command option works

$ man *command*

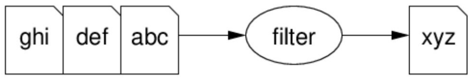
Shell Tools

I/O redirection

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

$ filter abc def ghi > xyz

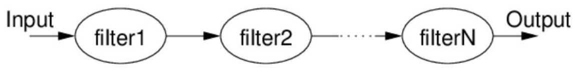
```
filter abc def ghi > xyz
```



Piping

Piping to use a combination of filters:

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



Common Commands

Cat (concatenate)

- Reads the text of the program “myProg.c” and writes it to standard output:

```
$ cat myProg.c
```

- Often used to ‘limit’ input based on the delimiter or field separator in formatted text data (e.g. columns broken by tabs, vertical bar, colons, semicolon etc.)
- Options:
  - -n (number output lines starting from 1)
  - -s (squeeze consecutive blank lines into single blank line)
  - -v (display control-characters in visible form)

wc (word counter)

- Summarizing filter used to count things
- Options:
  - -c (used to count the number of characters)
  - -w (used to count the number of words)
  - -l (used to count the number of lines)

```
$ wc file.txt
```

tr (transliterate characters)

- 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
  - 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
- Shofthand is available for specifying character lists:
  - ‘a-f’ is equivalent to ‘abcdef’
- E.g. a → 1, b → 2, c → 3

```
$ tr ‘abc’ ‘123’ < someText
```

- Options:
  - -c (complement - map all characters not occuring in sourceChars)
  - -s (squeeze adjacent repeated characters out - only copy the first)
  - -d (delete all characters in sourceChars - no destChars)
- 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

- Prints the first / last n (default 10) lines of input
- Options:
    - -n (set <u>n</u>umber of lines to be printed)
- Example:
    - Combine head and tail to select a range of lines (81-100) to output

```
head -n 100 | tail -n 20
```

egrep

- Select lines matching a pattern (Globally search with Regular Expressions and Print)
- Option:
    - -i (ignore upper/lower-case difference in matching)
    - -v (only display lines that do not match a pattern)
    - -w (only match pattern if it makes a complete word)

Regular Expressions (regex)

- A regular expression defines a set of strings
- Can be succinct and powerful
- Regex libraries are available for most languages
- Specify complex patterns concisely and precisely:

<u>Patterns</u>

- Default
    - cat matches strings with cat

- Alternation:
    - pattern1 | pattern 2 denotes the union of pattern1 and pattern2
    - E.g. perl|python|ruby matches any of the three strings

- Parentheses:
    - a(,a)* denotes a comma separated group of a's

- Escape character:
    - \ removes the special meaning of characters (e.g. *)

- Special characters:
    - . (dot) matches any single character
    - [abc] match one of any of the characters within the square brackets
    - [a-e] matches any characters in a-e
    - [^a-e] matches any character except a-e

- Anchoring matches:
    - Insisting that a pattern appears at the start or end of a string
        - ^[abc] matches either a b or c at the start of a string
        - cat$ matches cat at the end of a string

- Repetition of patterns:
    - p* denotes zero or more repetitions of p - $ [^X]*X matches any chars up to and including the first X
    - p+ denotes one or more repetitions of p - $ [0-9]+ matches any sequence of digits
    - p? denotes zero or one occurrence of p

cut (vertical slice)

- The cut command prints selected parts of input lines
    - Can select fields (assumed tab-separated columnated input)
    - Can select a range of character positions
- Lists are specified as ranges (e.g. 1-5) or comma-separated (e.g. 1,3,5)
- Options:
    - -f *listOfCols* (print only the specified fields)
    - -c *listOfPos* (print only chars in the specified position)
    - -d ‘c’ (use character c as the field separator)
- 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
```

paste (combine files)

- Displays several text files in “parallel” on output
- Lines from each file are separated by a tab character or specified delimiter(s)
- If files are different lengths, output uses empty strings for the shorter length file
- Options:
    - -s (interleaves lines)
- Example:

```
# 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)

- 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
- Options:
    - -r (reverse sort - descending order)
    - -n (numerical sort)
    - -d (dictionary order - ignore non-letters and non-digits)
    - -t ’c’ (use character c to separate columns)
    - -kn’ (sort on column n)

```
# 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)

- Removes all but one copy of adjacent identical lines
- Options:
    - -c (also print number of times each lines is duplicated)
    - -d (only print (one copy of) duplicated lines)
    - -u (only print lines that occur uniquely (once only))

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

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
- 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)
    - -i (ignore case)
    - -t c (tab character is c)

```
# 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)

- Provides the power of interactive-style editing in “filter mode”

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

- sed process:
    - 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 is very powerful and can also:
    - Partition lines based on patterns rather than columns
    - Extract ranges of lines based on patterns or line numbers
- Options
    - -n (no printing - displays no output but applies edit commands as normal)

Editing commands:

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

Selection command

- LineNo (selects the specified line)
- StartLineNo, EndLineNo (selects all lines between specified line numbers)
- /RegExp/ (selects all lines that match RegExp)
- /RegExp/,/RegExp2/ (selects all lines between lines matching reg exps)

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
```

```
# 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)

- 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
- Invocation:

find StartDirectory Tests Actions

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

```
 # 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  \(  -name '*07*'  -o  -mtime -1  \)  -

# 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 {} \;
```

Filters Summary

Function	Command	Description
Horizontal Slicing - select subset of lines	cat	Reads files sequentially and writes them to stdout
	head	Prints the first n lines of input
	tail	Prints the last n lines of input
	egrep	Selects lines matching a pattern
	sed	Selects lines matching a pattern and edits
	uniq	Removes all but one adjacent duplicate line
Vertical Slicing - select subset of columns	cut	Selects parts of input lines
	sed	Selects lines matching a pattern and edits
Substitution	tr	Converts text char-by-char based on a mapping
	sed	Selects lines matching a pattern and edits
Aggregation	wc	Determines word/char/line count
	uniq	Removes all but one adjacent duplicate line
Assembly - combining data sources	paste	Displays several text files in “parallel” on output
	join	Merges two files using the values in a field in each file as a common key
Reordering	sort	Copies input to output but ensures that the output is arranged in some particular order of lines
Viewing - always at end of pipeline	more	If text is too large to fit on screen, make it into pages
	less	Like more, but allows backward movement
File System Filter	find	Search for files based on specified properties
Programmable Filters	sed	Selects lines matching a pattern and edits
	perl	

Topic 2: Shell

Shell is a command interpreter that executes other programs. They can be either:

1. Graphical
    a. Easy for naive users to start using the system
2. Command-line
    a. Programmable and powerful tool

On Unix/Linux, bash has become the defacto standard shell

What Shells Do

All Unix shells have the same basic mode of operation:

```
loop
   if (interactive) print a prompt
   read a line of user input
   apply transformations to line
   split line into words (/\s+/)
   use first word in line as command name
   execute that command,
      using other words as arguments
end loop
```

The transformations applied to input lines include:

- Variable expansion
- File name expansion

Executing a shell script involves:

1. Finding the file containing the named program (PATH)
2. Starting the new process for execution of the program

```
$ sh bling    # file need not be executable
$ ./bling     # file must be executable
$ bling       # file must be executable and . in PATH
```

Permissions

- To make a file executable use:

$ chmod 755 *fileName*
$ chmod +x *fileName*

Shell as Interpreter

The shell can be viewed as a programming language interpreter. The shell has:

1. A state (collection of variables and their values)
2. Control (current location, execution flow)

Unlike other interpreters, the shell:

1. Modifies the program code before finally executing it
2. Has an infinitely extendible set of basic operations

Basic operations in shell scripts are a sequence of words

- A word is defined by any sequence of:
    - Non-whitespace characters
    - Characters enclosed in double-quotes
    - Characters enclosed in single-quotes

Shell Scripts

#!/bin/sh : tells the parent shell which interpreter should be used to execute the script (in this case, sh means shell)

Shell Variables and Pathname Expansion

General rules:

- No need to declare shell variables, simply use them
- Are local to the current execution of the shell
- All variables have type string
- Initial value of variable = empty string
- x=1 is equivalent to x="1"

Built-in variables with pre-assigned values:

1. $0 (the name of the command)
2. $1 (the first command-line argument)
3. $2 (the second command-line argument)
4. $3 (the third command-line argument)
5. $# (count of command-line arguments)
6. $* (all of the command-line arguments - together)
7. $@ (all of the command-line arguments - separately)
8. $? (exit status of the most recent command)
9. $$ (process ID of this shell)

If a string contains any of * ? [] it is matched against pathnames

- * matches zero or more of any character
- ? matches any one character
- [] matches any one character between the []

The string is replaced with all matching pathnames

Shell Debugging

Tips:

- The shell transforms commands before executing
- Can be useful to know what commands are executed
- Can be useful to know what transformations are produced
- "set -x" shows each command after transformation

Quoting

Quoting can be used for three purposes:

1. To group a sequence of words into a single word
2. To control the kinds of transformations that are performed
3. To capture the output of commands (back-quotes)

Three kinds of quotes:

1. Single-quote (‘) - grouping, turns off all transformations
    a. Useful to pass shell metacharacters in args
2. Double-quote (“) - grouping, no transformations except $ and `
    a. Construct strings using the values of shell variables (e.g. “x=$x, y=$y”)
    b. Prevent empty variables from vanishing (e.g. using test “$x” rather than test $x)
    c. Used for values obtained from the command line or a user (e.g. using test “ $1” rather than test $1)
3. Back-quote (`) - no grouping, capture command results
    a. E.g. using `command`, the shell will:
        i. Perform variable substitution, execute the resulting command and arguments, capture the standard output from the command, convert it to a single string, and use this string as the value of the expression

b. E.g. of back-quotes for a script that converts GIF files to PNG format

```
#!/bin/sh
# ungif - convert gifs to PNG format

for f in "$@"
do
    dir=`dirname "$f"`
    prefix=`basename "$f" .gif`
    outfile="$dir/$prefix.png"
    giftopnm "$f" | pnmtopng > "$outfile"
done
```

Connecting Commands

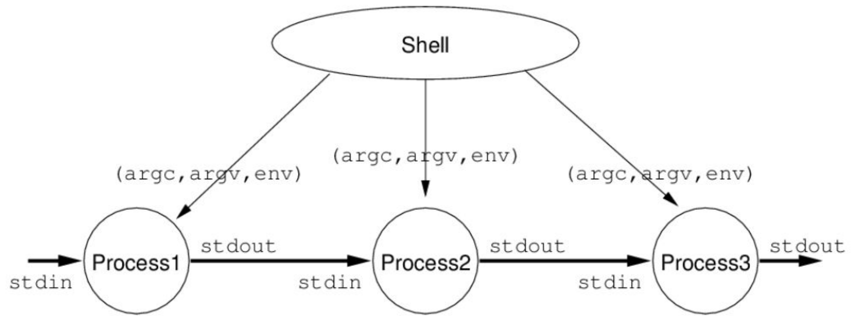
The shell provides I/O redirection to allow us to change where processes read from and write to. Beware: > truncates files before executing command - so always have backups

1. < infile (connect stdin to the file infile)
2. > outfile (connect stdout to the file outfile)
3. >> outfile (append stdout to the file outfile)
4. 2> outfile (connect stderr to the file outfile)
5. 2>&1 > outfile (connect stderr + stdout to outfile)

Many commands accept a list of file inputs (e.g. cat file1 file2 file3). These commands typically follow the conventions:

- Read contents of stdin if no filename arguments
- Treat the filename as meaning stdin

The shell sets up the environment for each command in a pipeline and connects them together:



Exit Status and Testing

Process exit status is used for control in shell scripts:

- Zero exit status means command successful
- Non-zero exit status means error occurred
- Mostly ignored in reality, example application to the right:

AND lists	cmd_1 && cmd_2 && ... && cmd_n
(cmd_{i+1} is executed only if cmd_i succeeds (zero exit status))	
OR lists	cmd_1 || cmd_2 || ... || cmd_n
(cmd_{i+1} is executed only if cmd_i fails (non-zero exit status))	

The test command performs a test or combination of tests and returns:

- A zero exit status if the test succeeds
- A non-zero exit status if the test fails

Useful testing features:

1. String comparison (= !=)
2. Numeric comparison (-eq -ne -lt)
3. Checks on files (-f -x -r)
4. Boolean operators (-a -o !)

Sequential Execution and Grouping

Combine commands in pipelines and AND/OR lists using semicolon separation or newline separation

```
cmd(1) ; cmd(2) ; … ; cmd(n)
or
cmd(1)
cmd(2)
…
cmd(n)
```

Commands can be grouped using (...) for execution in new sub-shell or {...} for execution in current shell

If Command

The if-then-else construct allows conditional execution:

```
if testList{1}
then
   commandList{1}
elif testList{2}
then
   commandList{2}
...
else
   commandList{n}
fi
```

Case Command

Case provides multi-way choice based on patterns:

```
case word in
pattern{1})  commandList{1} ;;
pattern{2}-2)  commandList{2}-2 ;;
...
pattern{n})  commandList{n} ;;
esac
```

- The *word* is compared to each *pattern* in turn, and once matched, the corresponding *commandList* is executed and the statement finishes

Loop Commands

While loops iterate based on a test command list:

```
while testList
do
   commandList
done
```

For loops set a variable to successive words from a list:

```
for var in wordList
do
   commandList  # ... generally involving  var
done
```

String Comparison	Description
Str1 = Str2	Returns true if the strings are equal
Str1 != Str2	Returns true if the strings are not equal
-n Str1	Returns true if the string is not null
-z Str1	Returns true if the string is null
Numeric Comparison	**Description**
expr1 -eq expr2	Returns true if the expressions are equal
expr1 -ne expr2	Returns true if the expressions are not equal
expr1 -gt expr2	Returns true if expr1 is greater than expr2
expr1 -ge expr2	Returns true if expr1 is greater than or equal to expr2
expr1 -lt expr2	Returns true if expr1 is less than expr2
expr1 -le expr2	Returns true if expr1 is less than or equal to expr2
! expr1	Negates the result of the expression
File Conditionals	**Description**
-d file	True if the file is a directory
-e file	True if the file exists (note that this is not particularly portable, thus -f is generally used)
-f file	True if the provided string is a file
-g file	True if the group id is set on a file
-r file	True if the file is readable
-s file	True if the file has a non-zero size
-u	True if the user id is set on a file
-w	True if the file is writable
-x	True if the file is an executable

Topic 3: Perl

Perl Intro

Perl = Practical Extraction and Report Language

- Can be used as a replacement for awk, sed, grep, other filters, shell scripts and C programs
- Makes it easy to build useful systems
- Readability can be a problem

Design Principles

1. Easy and concise to express common idioms
2. Multiple ways to do one thing
3. Use defaults wherever possible
4. Uses powerful, concise syntax but can accept ambiguity in some cases
5. A large language that users will learn subsets of

Running Perl

Perl programs can be invoked by:

1. Perl -w file.pl
2. perl -w -e ‘print “Hello, world\n”;’
3. Using #!/usr/bin/perl -w and making the program file executable

Syntax

Char	**Kind**	**Example**	**Description**
`#`	Comment	`# comment`	rest of line is a comment
`$`	Scalar	`$count`	variable containing simple value
`@`	Array	`@counts`	list of values, indexed by integers
`%`	Hash	`%marks`	set of values, indexed by *strings*
`&`	Subroutine	`&doIt`	callable Perl code (& optional)

Any unadorned identifiers are either:

1. Names of built in (or other) functions
    a. Chomp, split
2. Control-structures
    a. If, for, foreach
3. Literal strings

Variables

Perl variables:

- Perl has three basic kinds of variables:
    a. Scalars - a single atomic value (number or string)
    b. Arrays - a list of values, indexed by number
    c. Hashes - a group of values, indexed by string
- The variables do not need to be declared or initialised
- Many scalar operations have a ‘default source/target’
    a. If an argument is not specified, $_ is assumed

Arithmetic & Logical Operators

Arithmetic Operators

- Numeric: == != < <= > >= <=>
- String: eq ne lt le gt ge cmp

Concatenation “.”
String Compare “cmp”

Logical Operators

Operation	**Example**	**Meaning**
And	`x && y`	false if x is false, otherwise y
Or	`x || y`	true if x is true, otherwise y
Not	`! x`	true if x is not true, false otherwise
And	`x and y`	false if x is false, otherwise y
Or	`x or y`	true if x is true, otherwise y
Not	`not x`	true if x is not true, false otherwise

Control Structures

If Blocks:

1. If ()
2. Elsif ()
3. Else

Loops

1. While ()
2. Until ()
3. For (init; expr; step)
4. Foreach var (list)

Special Variables

`$_`	default input and pattern match
`@ARGV`	list (array) of command line arguments
`$0`	name of file containing executing Perl script (cf. shell)
$\$i$	matching string for i^{th} regexp in pattern
`$!`	last error from system call such as open
`$.`	line number for input file stream
`$/`	line separator, none if undefined
`$$`	process number of executing Perl script (cf. shell)
`%ENV`	lookup table of environment variables

Perl Arrays

An array is a sequence of scalars, indexed by position (0,1,2,...)

- Denoted by @array
- Array elements denoted by $array[index]
- $# array gives the index of the last element
-