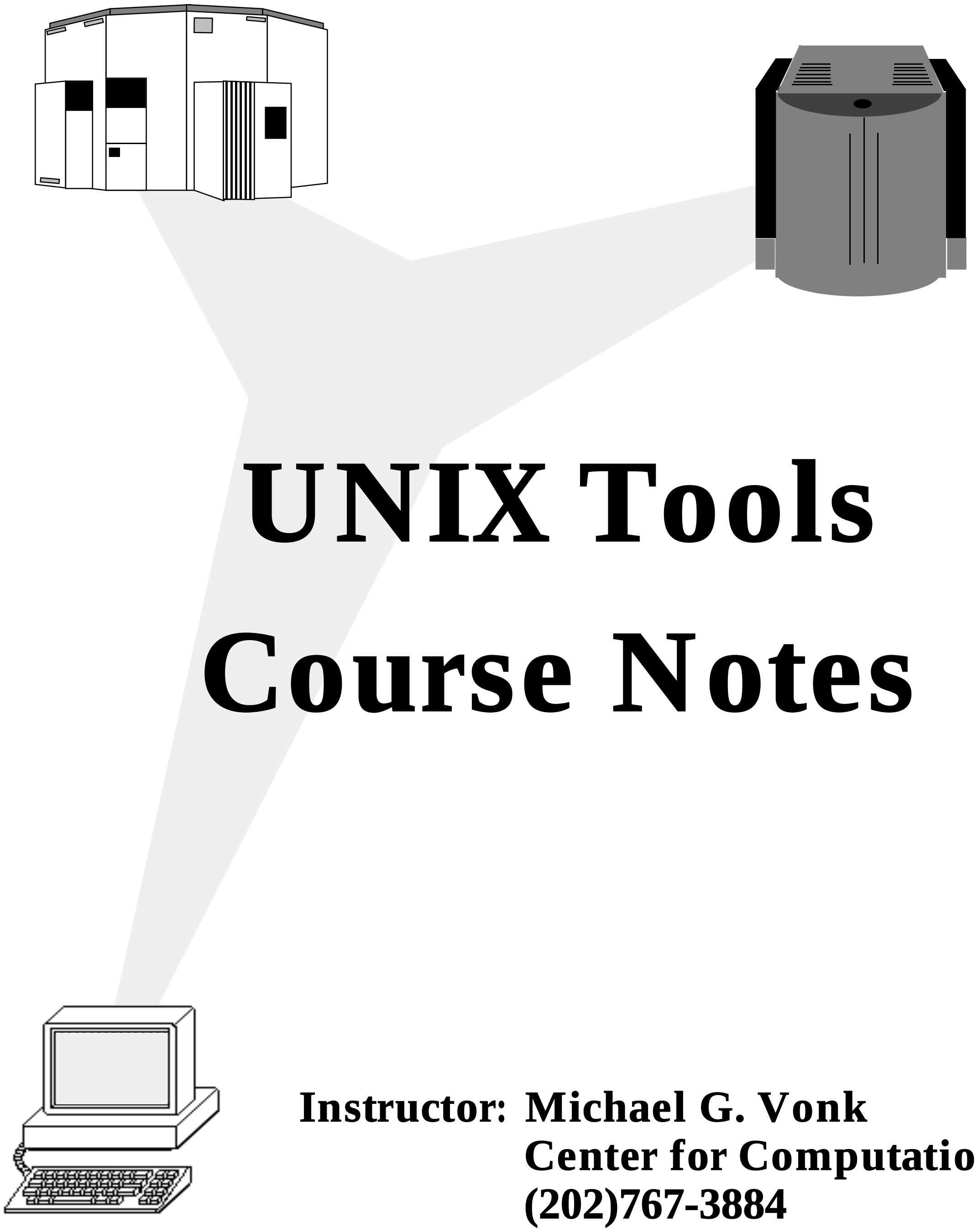
Naval Research Laboratory

Washington, DC 20375-5320



UNIX Tools Course Notes

Instructor: Michael G. Vonk
Center for Computational Science
(202)767-3884
michael.vonk@nrl.navy.mil

UNIX Tools

July 31, 1998

1. Introduction

There are many UNIX tools that enable users to efficiently solve a wide range of problems. This class provides an introduction to several of these commonly used tools. Further information can be found in the references sited at the end of the notes and in the appropriate `man` pages.

Topics to be covered include:

- writing shell scripts
- using regular expressions to match patterns of characters
- creating and modifying files using `sed` and `awk`
- using miscellaneous commands to manipulate data
- archival and compression utilities
- program development
- managing projects using make

The goal of this class is to be an extension of the topics covered in "Introduction to UNIX." A companion page containing examples used in this class is located at:

```
http://amp.nrl.navy.mil/code5595/
ccs-training/unix-tools/companion-page.html
```

2. Bourne Shell Scripts

While the C shell is typically used for interactive processing as it has more interactive functionality, the Bourne shell is better for writing shell scripts as it has a simpler syntax and more programming features.

Shell scripts are useful for:

- automating frequently performed and complex tasks
- performing tasks at specific times or regular time intervals

Further information can be found in the `sh` `man` page.

2.1. Shell Script Format and Execution

The basic format of a shell script is as follows:

```
#!/bin/sh
#
# Example script for "UNIX Tools" class.
# Compiles and executes a C program.
#
# Usage: % compile-and-run

gcc -o myprog main.c fun.c prod.c -lm
myprog
```

Example 1 compile-and-run

Scripts are executed by default under the Bourne shell, except when the first characters of the first line of the script are:

`#`	script will be run under C shell
`#!`*`command`*	script will be run under *`command`*

Shell scripts can be executed by:

1. Making the script executable and naming it on the command line:

   ```
   % chmod u+x script-name
   % script-name
   ```

   This assumes the current working directory is in your path, otherwise the following can be used:

   ```
   % ./script-name
   ```

   A common place for storing script files is in a `bin` directory under your home directory, and then to include `~/bin` in your path.

2. Running the script in a subshell

   ```
   % sh script-name
   ```

3. Running the script in the current shell

   ```
   % source script-name
   ```
   or
   ```
   $ . script-name
   ```
   (from Bourne shell)

4. Running the script as a batch job using the `batch`, `at`, or `cron` commands

2.2. Variable Assignment and Substitution

Variable assignments use the following format:

```
variable=value
```

where value is one of the following:

`ddd`	decimal number
`'string'`	string (all enclosed characters protected)
`"string"`	string ($, ‘ substitution performed within string, special characters can be protected using \)
`‘command‘`	command output[*]
`$var`	value of `var` (or nothing, if undefined)
`${var}`	same as above

2.3. Shell Metacharacters

The following metacharacters can be used as file name expansion characters:

`*`	Matches zero or more occurrences of any character (does not match "`.`" as first character of a filename)
`?`	Matches any single character
`[ccc]`	Matches one of the enclosed characters—a range of characters may be specified as `[a-z]`
`[!ccc]`	Matches any character not enclosed in `[]`

* In NCSA Telnet (Macintosh), the backquote may have been remapped to escape, check under Preferences (Global) in the Edit Menu

2.4. Passing Arguments to a Script

Arguments can be passed from the command line to a shell script and are assigned to the following special variables:

`$0`	name of command
`$1 - $9`	positional parameters
`$#`	number of positional parameters
`$*`	list of all positional parameters

Only nine positional parameters are available (although on some systems, this limit is removed). The `shift` command allows access to additional positional parameters—`$2` becomes `$1`, `$3` becomes `$2`, etc. The original first positional parameter is no longer available. (There is no un`shift` command.)

```
echo "Command: $0"
echo "Number of arguments: $#"
echo "Argument list: $*"
echo ""
echo "Argument #1 =  $1"
echo "Argument #2 =  $2"
echo "Argument #3 =  $3"
echo "Argument #4 =  $4"
```

Example 2 show-arguments

2.5. Script I/O

2.5.1. Writing Output

The `echo` command writes a string to standard output:

```
echo string
```

Special characters can be protected using `\`. The following special characters are frequently used:

`\c`	does not create a new line after writing *`string`* (must be last character in *`string`*)
`\n`	create a new line

2.5.2. Reading Input

The `read` command reads a line from standard input:

```
read variable-list
```

Input words are assigned to successive variables. Any remaining words are assigned to the last named variable.

```
echo "Enter name (First MI Last):  \c"
read first mi last

echo " "
echo "First Name     : $first"
echo "Middle Initial : $mi"
echo "Last Name      : $last"
```

Example 3 read-name

2.5.3. I/O Redirection

The following I/O redirection operators can be used:

`>` *`file`*	Redirects output to *`file`*
`>>` *`file`*	Appends redirected output to *`file`*
`<` *`file`*	Redirects input from *`file`*
`<<` *`marker`*	"Here" document—accepts shell input up to first occurrence of *`marker`* on a line by itself
`command1` `|` *`command2`*	Pipes *`command1`*'s standard output into *`command2`*'s standard input

The following illustrates the use of a "here" document:

```
name="Michael"

cat > hello.c <<EOF

#include <stdio.h>
main() {
    printf("Hello, $name...\n");
}

EOF

gcc -o hello hello.c
hello
```

Example 4 here-document

2.6. Control Structures

A list of commands can be conditionally executed based on a specified condition, which can take one of the following two forms:

1. Result of the `test` command:

   `test` *`condition`*

   or

   `[` *`condition`* `]` (the spaces are required)

2. Exit status of a command (or of the last command in a list)

The `test` result or exit status which is stored in `$?` is zero for true (success) and non-zero for false (failure).

Logical operators include:

`-a`	and
`-o`	or
`!`	not

Comparison operators include:

String	**Numeric**
`=`	`-eq`
`!=`	`-ne`
`string` (true if not null)	`-gt`
	`-ge`
	`-lt`
	`-le`

Tests can be performed on files and strings, including:

`-f` *`file`*	true if *`file`* exists and is an ordinary file
`-d` *`file`*	true if *`file`* exists and is a directory
`-z string`	true if string length is 0
`-n string`	true if string length is not 0

See the `test(1)` man page for further details.

2.6.1. Conditional Statements

The `if` statement uses the following syntax:

```
if condition
then
   command-list
else if condition-2
   command-list
else
   command-list
fi
```

The `else if` and `else` constructs are optional. `elif` can be used as a shorthand for `else if`.

```
if who | grep -s $1 > /dev/null
then
  echo $1 is logged in
else
  echo $1 not available
fi
```

Example 5 check-user

The `case` statement provides multi-way branching:

```
case variable in
  pat1)         command-list
                ;;
  pat2 | pat3)  command-list
                ;;
  *)            command-list
                ;;
esac
```

2.6.2. Looping Statements

The `while` and `until` contructs have the following syntax:

```
while condition
do
  command-list
done

until condition
do
  command-list
done
```

The `for` loop has the general form:

```
for var in list
do
  command-list
done
```

If ’`in` *`list`*’ is omitted, the loop is executed once for each positional parameter (i.e. ’`in $*`’ is assumed).

There are two constructs for jumping out of loops:

`break`	terminates execution of innermost enclosing loop, causing execution to resume after the nearest `done` statement
`continue`	causes execution to resume at the `while`, `until` or `for` statement which begins the loop containing the `continue` command

2.6.3. Using the `exit` Command to Exit Shell Scripts

The `exit` statement will exit the current shell script. It can be given a numeric argument which is the script's exit status. If omitted the exit status of the last command executed is used. For example:

```
exit 1
```

2.6.4. The && and || Operators

The && operator can be used to execute a command if the previous command is successful:

```
command1 && command2
```

The || operator can be used to execute a command if the previous command fails:

```
command1 || command2
```

For example:

```
   # Execute myprog--print a message if
   # execution is successful

   myprog && echo "Execution successful"

   # Execute myprog--print a message if
   # execution fails

   myprog || echo "Execution failed"
```

Example 6 test-execution

2.7. Arithmetic Operators

The shell does not have any arithmetic features built in, so you must use the `expr` command, as shown below:

> *`variable`* `= ‘expr` *`expression`*`‘`

For example:

```
   index = ‘expr $index + 1‘
```

Further details can be found in the `expr(1)` man page.

2.8. Signal Trapping

The `trap` command can be used to catch or ignore operating system signals:

```
trap 'command-list' signal-list
```

The commands in *`command-list`* are executed when the signal is trapped and then control is returned to the place at which it was interrupted.

If *`command-list`* is not specified, then the action taken on receipt of any signal in *`signal-list`* is reset to the default action. If *`command-list`* is an explicitly quoted null command (`' '` or `" "`), then the signals in *`signal-list`* are ignored.

The following signals are commonly trapped:

0	shell exit (for any reason, including end of file)
1	hangup
2	interrupt (`^C`)
3	quit (causes program to produce a core dump)
9	kill (cannot be caught or ignored)
15	terminate

The `kill` command can be used to send a signal to a script:

```
% kill [-signo] pid
```

Using `kill` without a specified signal number results in the signal number `15` being sent. The full list of signal numbers can be shown as follows:

```
% kill -l
```

To check what traps are currently set, use `trap` by itself:

```
% trap
```

```
# Remove a temporary file if script is
# interrupted

temp=/tmp/file.$$
trap 'rm $temp; exit' 0 1 2 3 15
  .
  .
  .
```

Example 7 signal-trap

2.9. Debugging Shell Scripts

The following debugging options can be used either on the command line or with the `set` command:

`-e`	in non-interactive mode, exit immediately if a command fails
`-x`	print commands and their arguments as they are executed
`-v`	print shell input lines as they are read
`-n`	read commands but do not execute them

For example:

```
$ sh -x script-name argument-list
```

3. Regular Expressions

Regular expressions are used to match a pattern of characters. Metacharacters (special characters not interpreted literally) are used to match a sequence of actual characters.

Not all utilities use the same set of metacharacters:

• basic set	used in `grep` and `sed`
• extended set	used in `egrep` (extended `grep`) and `awk`

Metacharacters used in regular expressions may be the same as those used as shell metacharacters, but have different meanings.

Table 1 Basic Set of Metacharacters (`grep` and `sed`)

`.`	Any single character (except newline)
`*`	Zero or more occurences of previous regexp
`[`*`ccc`*`]`	Any one of a class of characters: "`^`" when first, reverses the match "`-`" when not first or last, indicates a range "`]`" when first, is a member of the class Other characters (except "`\`") lose their meaning
`^`	As the first character in a regular expression, matches the beginning of the line
`$`	As the last character in a regular expression, matches the end of the line
`\{`*`n`*`,`*`m`*`\}`	Range of occurrences of preceding regexp (`sed` and `grep` only)
`\`	Escapes the following metacharacter

Basic Examples

The `grep` command prints lines from a file that contain a specified regular expression:

```
% grep 'regular-expression' filename
```

Since *`regular-expression`* can contain metacharacters also used by the shell, *`regular-expression`* is enclosed in single quotes.

For example, the following regular expressions could be used:

`80.86`	"80236", "80386", etc.
`❑.*❑`	longest possible* string between spaces
`❑[^❑]*❑`	shortest possible string between spaces
`[a-zA-Z]`	any alphabetic character
`^$`	blank lines
`^❑*$`	blank lines possibly containing spaces
`❑❑*$`	lines ending in one or more spaces
`\.$`	lines ending in a period
`❑\{2,4\}`	two to four spaces

* A regular expression tries to match the longest string possible.

Table 2 Extended Set of Metacharacters (`egrep` and `awk`)

`+`	One or more occurrences of the previous regular expression
`?`	Zero or one occurrences of the previous regular expression
`|`	Specifies that either the preceding or following regular expression can be matched
`( )`	Groups regular expressions

Extended Examples

`❏+`	one or more spaces
`80[23]?86`	"`8086`", "`80236`" or "`80386`"
`(UN|A)IX`	"`UNIX`" or "`AIX`"

Seemingly Incomprehensible Example

```
(^|❏)["[{(]*book[]})"?!.,;:'s]*(❏|$)
```

4. The `sed` Editor

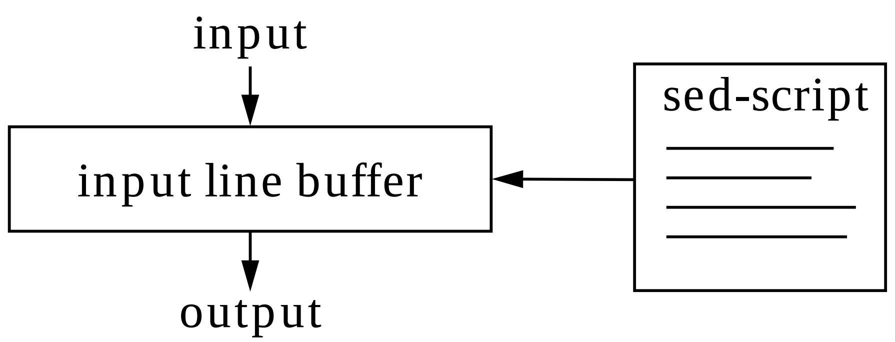
The `sed` editor is a "non-interactive," stream-oriented editor commonly used to:

- make a series of changes (such as search and replace) across a number of files
- filter data for use with other applications
- edit very large files that would be too slow to edit interactively

It is "non-interactive" in that `sed` commands are read from a script rather than being entered interactively, and it is stream oriented in that input flows through the program and is directed to standard output—the input file itself is not changed.

`sed` loops through each input file as follows:

- reads one line of input into a buffer
- executes all applicable commands on the buffer
- writes buffer contents to standard output



All editing commands (unless line addressing is used to restrict the lines affected) are applied in order to every line of input.

4.1. Invoking the `sed` Editor

The `sed` utility can be invoked by specifying `sed` instructions on the command line or by placing them in a `sed` script:

```
% sed [-e] 'instruction-list' input-file
% sed -f script-file input-file
```

Each line of input is written, perhaps modified, to standard output (which can be redirected, using ">"). To suppress this, the `-n` option can be used. Selected lines can then be output using the `p` command.

4.2. Command Format and Line Addressing

Command format:

```
[address1 [,address2]][!] command
```

An *address* can be:

- a regular expression enclosed in slashes
- a line number (not reset between input files)
- a `$` which denotes the last line

There can be 0, 1, or 2 addresses specified:

- 0 *command* applied to all lines
- 1 *command* applied to specified lines
- 2 *command* applied to specified range of lines

An exclamation point following an address can be used to execute a command on all lines except those specified.

For example:

`/lunch/`*`command`*	execute *`command`* on all lines containing the string "lunch"
`1,10`*`command`*	execute *`command`* on lines 1-10
`12,$`*`command`*	execute *`command`* on all lines starting at line 12
`1,/^$/`*`command`*	execute *`command`* on all lines up to the first blank line
`/dinner/!`*`command`*	execute *`command`* on all lines not containing the string "dinner"

Comments can be included in `sed` scripts by prefixing them with '#'. (Some systems allow comments only on the first line.)

Commands may be grouped using `{}`, allowing the nesting of ranges and multiple commands to be executed on a single range.

```
address {
    command1
        .
        .
        .
    commandn
}
```

A common `sed` error is extraneous blanks at the end of a command—extra spaces at the end are not permitted. sed prints "`Command garbled`" when it doesn't understand a command.

4.3. Basic sed Commands

Table 3 Basic sed Commands

a	Append *text* following each line specified*: [*address*]a\\ *text*
c	Replace specified lines with *text*. If a range of lines is specified, replace the entire range with a single copy of *text**: [*address1*][,*address2*]c\\ *text*
d	Delete line (do not write to standard output): [*address1*][,*address2*]d
i	Insert *text* before each line specified*: [*address*]i\\ *text*
p	Print specified lines (causes duplicate printing unless -n specified on command line): [*address1*][,*address2*]p
s	Substitute *pat2* for *pat1* on specified lines: [*add1*][,*add2*]s/*pat1*/*pat2*/*flags* The "g" flag changes all occurrences on each line
y	Translate each character in *string1* to corresponding character in *string2*: [*add1*][,*add2*]y/*string1*/*string2*/

* *text* may be continued over new lines using "\\".

4.4. `sed` **Examples**

To delete all completely blank lines from a file:

```
% sed -e ’/^$/d’ blanks.dat > noblanks.dat
```

To trim trailing spaces at the end lines from a file:

```
% sed -e ’s/ *$//’ trailing-blanks.dat
```

To substitute one string for another everywhere it is found:

```
% sed 's/Unix/UNIX/g' chapter1.txt > new.txt
```

To print only the lines containing a specified pattern:

```
% sed -n -e '/lunch/p' todo.dat
```

To make a series of substitutions across several files:

```
#!/bin/sh

for file in *.c
do
  sed -e ’s/^#include "a.h"/#include "b.h"/
          s/start=10/start=20/
         ’ $file > $file.new
  mv $file.new $file
done
```

Example 8 multiple-changes

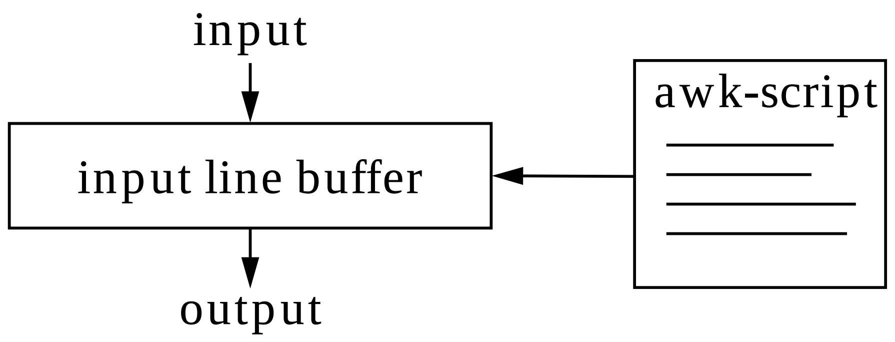
5. The awk Utility

The awk utility is designed to make information retrieval and text manipulation easy. awk can be used to perform a variety of data processing tasks, including:

- generating reports
- filtering data for use with other applications
- reformatting data

awk is a pattern matching programming language which is best used on files with some kind of structure. It allows you to use the structure of the file in writing procedures for inserting and extracting data.

awk takes a line of input, executes each of the instructions from the script file, and writes the processed line to standard output.



A newer version of awk, called nawk (for new awk), is available on some systems. Also available from the FSF is GNU awk, known as gawk, which implements all of the features of nawk, with many new features.

5.1. Invoking the `awk` Utility

The `awk` utility can be invoked by specifying `awk` instructions on the command line or by placing them in an `awk` script:

```
% awk 'instructions' file-list
% awk -f script file-list
```

Output can be redirected as follows:

```
% awk -f script file-list > output-file
```

5.2. Command and Script Format

Commands have two parts—a pattern and a procedure:

```
pattern{procedure}
```

If the pattern matches the current line, the procedure is executed. Patterns can be:

- regular expressions enclosed in slashes
- relational expressions (such as `$2=100`)

If no pattern is specified, the procedure is applied to all lines. If no procedure is specified, all lines matching the pattern are printed.

Scripts can be made up of three sections:

BEGIN	executed before any data is read; used to set variables and to define record and field separators
main section	made up of patterns to be searched for in the input data and actions to be taken
END	executed after last input line has been processed; useful for printing report totals

A typical `awk` script looks like the following:

```
BEGIN {procedure}

pattern{procedure}
pattern{procedure}
pattern{procedure}

END {procedure}
```

Multiple patterns can be specified as follows:

pat1 && *pat2*	and
pat1 || *pat2*	or
pat1, *pat2*	range of lines

5.3. Variables and Separators

By default, each input line is a record and each word within a record is a field. The following variables are used with records and fields:

`RS`	record separator (default is newline)
`NR`	current record number
`FS`	field separator (default is spaces or tabs, can also be reset using the `-F` option)
`NF`	number of fields in current record
`$0`	represents the entire line
`$1`, `$2`, ...	represent each word in the line

5.4. Basic `awk` Commands

Variable Assignment

Two types of variables are commonly used—integers and strings. Variable assignment occurs as follows:

```
total  = 0
total  = total + $1
string = "Hello, world"
```

Printing

Printing commands include `print` and `printf`.

For example:

```
print $1, $2
printf("%10d\n", total)
```

Two variables control how printing occurs:

`ORS`	Output Record Separator (default is a new-line)
`OFS`	Output Field Separator (default is a space)

Flow Control

Flow control statements include `if`, `while`, and `for`.

5.5. `awk` Examples

Print the number of lines in a file:

```
% awk 'END {print NR}' filename
```

Print the last field of every input line:

```
% awk '{print $NF}' filename
```

Print every line with more than four input fields:

```
% awk 'NF > 4' filename
```

Print the second field divided by 100 and erase the fourth field:

```
% awk '{$2/= 100; $4= ""; print}' filename
```

Print all the lines between two patterns:

```
% awk '/pattern1/,/pattern2/' filename
```

6. Miscellaneous Commands

The commands in this section are useful for manipulating files.

Cutting Columns or Fields from a File

```
% cut -ccolumns file
```

Options:

`-f`*`fields`*	cut tab delimited fields
`-d`*`char`*	specify delimiter

```
% ls -l | cut -c1-10
% cut -f1 table-data
% cut -d: -f1 /etc/passwd
```

Pasting Files Together

```
% paste file1 ... filen
```

Options:

`-d`*`char`*	specify delimiter (default is a tab)

```
% paste names positions > roster
```

Translating Characters

```
% tr string1 string2
```

This will translate characters found in *string1* to corresponding character is *string2*.

Options:

`-d`	delete characters in *string1*
`-s`	squeeze repeated characters in *string2* to single characters

```
% tr '[a-z]' '[A-Z]' < roster
% tr '\11' ' ' < roster          ('\11' is a tab)
% tr -d '\11' < roster
% tr -s ' ' ' '
```

Sorting Files

```
% sort [options] file
```

Options:

`-u`	remove duplicates
`-r`	sort in descending order
`-n`	sort numerically
`+`*n*	sort on field *n*
`-t`	specify field delimiter (default is a space or tab)

```
% sort roster
% sort +1n roster
% sort +2n -t: /etc/passwd
```

Removing Consecutive Duplicate Lines from a File

```
% uniq filename
```

Options:

`-d`	list duplicates, but don't remove them

```
% uniq -d duplicates
```

7. Archival and Compression Utilities

The utilities described in this section can be used to manage groups of files and compress them.

7.1. The `tar` Utility

The `tar` (short for tape archiver) utility can be used to group sets of related files into a single large file for easy storage and transportability. (`tar` is typically used to create archive datasets on disk rather than archive tapes.)

7.1.1. Archiving Files

`tar` can be used to archive entire directory structures or individual files. The following archives a directory and all its subdirectories:

```
% tar -cvf misc.tar misc
```

To archive a set of individual files, the following is used:

```
% tar -cvf filename.tar file-specifier
```

Options used in archiving files:

`-c`	creates a new tar file
`-v`	verbose mode
`-f` *`filename`*	specifies a tar file name

Note Issue the `tar` command from a directory other than the one being archived—otherwise the `tar` dataset may contain itself.

7.1.2. Listing Archive Contents

The table of contents for a `tar` dataset can be displayed using the `-t` option as follows:

```
% tar -tf misc.tar
```

7.1.3. Extracting Files From `tar` Archives

The `tar` command can also be used to extract files from `tar` datasets. The basic format is as follows:

```
% tar -xvf misc.tar
```

The extracted files will have their original subdirectories, filena-mes, protections, and modification times.

To extract a specific directory, use:

```
% tar -xvf filename.tar directory-name
```

This will extract the entire directory specified. If the directory does not exist, it is created, otherwise, the files are replaced.

7.2. The `compress` and `uncompress` Commands

The `compress` command can be used to reduce file storage. Source code and English text files can typically be compressed 50-60%. Compressed files can be restored to their original form using the `uncompress` utility.

The following will compress all the text files in the current directory:

```
% compress *.txt
```

The specified files will be compressed in place and given the extension "`.Z`". The set of files can then be uncompressed as follows:

```
% uncompress *.txt.Z
```

Groups of files are quite often tarred and then compressed resulting in a file with the extension "`.tar.Z`". These files can be uncompressed and detarred as follows:

```
% tar -cvf misc.tar misc
% compress misc.tar

% uncompress misc.tar.Z
% tar -xvf misc.tar
```

8. Program Development

8.1. Program Development Cycle

The basic program development cycle is as follows:

```
% vi hello.c
% gcc hello.c
% a.out
```

C compilers (`cc`) are available on most UNIX systems and are located in the standard path. (The examples in this section, however, use Gnu's `gcc` compiler.)

Compiling and linking are performed in one step by default. Executable files are named `a.out` by default. To rename executables, use the `-o` option:

```
% gcc -o hello hello.c
```

Compiling several programs and then linking (including the math library) separately:

```
% gcc -c main.c prod.c fun.c
% gcc -o myprog main.o prod.o fun.o -lm
```

Programs can be executed by simply entering their filename:

```
% myprog
```

If, however, the current directory is not in your path variable, you can execute it as follows:

```
% ./myprog
```

8.2. Common Compiler and Linker Options

The following option are often used with the `cc` and `gcc` compilers:

`-c`	Compiles but skips link step. Generates an object file named "`*filename*.o`". This option is useful for compiling several modules and linking them later.
`-o *filename*`	Names output file (executable files are named "`a.out`" by default).
`-g`	Generates symbol table for use with the dbx debugger (overrides `-O`). For separate compilation and linkage, make sure to use `-g` on both steps.
`-p`	Prepares object files for profiling with prof.
`-I*pathname*`	Adds pathname to list of directories in which the C preprocessor searches for `#include` files with relative pathnames.
`-l*x*`	Links in library `lib*x*.a` found in `/lib`, `/usr/lib`, or `/usr/local/lib`. To specify some other library, you must specify its full path, e.g. `-l/usr/people/smith/mylib.a`
`-L*directory*`	Adds *`directory`* to beginning of list of directories in which to search for libraries.

8.3. Execution Timing and Profiling

8.3.1. The `time` Command

The `time` command can be used to quickly analyze program performance. `time` requires no special compilation or linkage options and can be used as follows on any executable program, script, or command:

```
% time myprog
1702.4u 14.6s 48:19 59% 0+10496k 1+2io 9pf+0w
%
```

Output from the `time` command is comprised of the following:

- Time spent on user code (seconds)
- Time spent executing system code on behalf of user (seconds)
- Time to completion (minutes:seconds)
- Percentage of machine resources used
- Average shared (program) memory + private memory (kilobytes)
- Number of reads + writes
- Number of page faults + swap-outs

8.3.2. The `prof` Utility

The `prof` utility can be used to produce an execution profile of a program. The `-p` option must be specified on the `cc` or `gcc` command to link in the `prof` libraries:

```
% gcc -o othello othello.c -p
```

`prof` correlates the symbol table from the executable program with the profile file (`mon.out`, by default) produced when the program is executed and displays the percentage of time spent in each routine, number of times the routine was called, and number of milliseconds per call:

```
% gcc -o othello othello.c -p
% othello
% prof othello

  %time  cumsecs    #call  ms/call  name
   42.9   269.29   227259     1.18  _createResponseList
   13.6   354.66                    mcount
   11.7   427.77 55725499     0.00  .mul
    8.0   477.65 22284595     0.00  _strcmp
             .
             .
             .
    0.0   627.35      682     0.00  _strlen
%
```

8.4. Syntax Checkers (`lint`)

The `lint` utility detects features of C programs that are likely to be bugs, non-portable, or wasteful. It also performs stricter type checking than does the C compiler:

```
    % lint othello.c
```

Problems noted include:

- unreachable statements
- loops not entered at the top
- automatic variables declared and not used
- logical expressions with constant values

Function calls are checked for inconsistencies such as:

- functions that return values in some places and not in others
- functions called with varying numbers of arguments
- functions that pass arguments of a type other than the type the function expects to receive
- functions whose values are not used
- calls to functions not returning values that use the non-existent return values of the function

`lint` is typically available on all UNIX systems.

9. Managing Projects Using `make`

The `make` utility uses instructions provided in a user written description file (makefile) to:

- Automate the program development process
- Eliminate unnecessary recompiling of unchanged code

The makefile is used to record dependency relationships. `make` compares the modification time of a dependency with that of the target and, if newer, rebuilds the target.

Once the makefile is written, the compilation and linkage process is as simple as:

```
% make
```

`make` searches for a description file named `makefile` or `Makefile` in the current directory (`Makefile` is the convention since capital letters appear before lowercase in ls output).

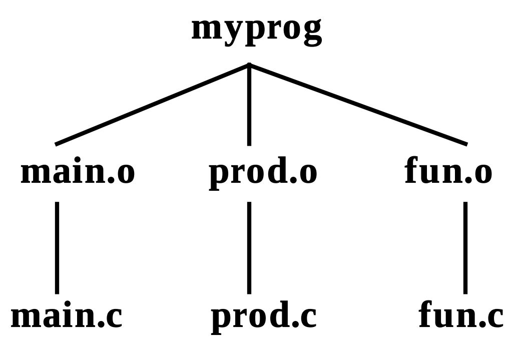
Note Source programs should first be divided into individual files for each routine (the `csplit` utility can be used to do this)

9.1. Simple `make` Example

The C program which will be used throughout this section consists of the following routines:

```
main
prod
fun
```

The following diagram shows the dependencies between the executable program, object files, and source files.



The executable `myprog` depends on `main.o`, `prod.o`, and `fun.o`, each of which depend on their respective source files.

These dependency relationships are illustrated in the following makefile.

```
myprog: main.o prod.o fun.o
   gcc -o myprog main.o prod.o fun.o -lm

main.o: main.c
   gcc -c main.c

prod.o: prod.c
   gcc -c prod.c

fun.o: fun.c
   gcc -c fun.c
```

Example 9 makefile #1

This example shows that the executable `myprog` depends on the three object files. Each object file depends on its source file. The `gcc` commands shown are used to build each target.

This makefile can then be used and the executable run as follows:

```
% make
% myprog
```

9.2. Makefile Description

A makefile specifies the sequence of operations to be performed and dependency relationships.

Makefiles consist of 3 parts:

1. Rules	Define targets to be built, their dependencies, and a set of commands used to build the target
2. Macros	Define variables for use within make
3. Suffix Rules	Specify a set of commands for building a file with one suffix from another file with the same basename but a different suffix

Comments can be including by prefixing them with #

`make` processes targets as it encounters them using its depth-first dependency scan. Target entries not encountered during dependency scan are not processed.

9.3. Rules

makefile rules have the following syntax:

```
target: dependency [dependency ...]
<TAB>   command
```

where:

• target	Defines the object of the operation
• dependency	Files on which the target depends
• command	Specifies how to build the target (multiple commands can be specified on separate lines or by separating them with semicolons)

If a dependency has been updated more recently than the target, `make` updates the target by running the command.

If you don't specify a list of commands to build a target, `make` attempts to use a user supplied or default suffix rule.

9.4. Macros

Macros are simple variables used to simplify makefiles and are defined as follows:

```
macro=value
```

Macros can be referenced in any of the following ways:

```
$(macro)
${macro}
$x                 (if macro name is only one character long)
```

For example, the executable name and the list of object files in Example 1 could be replaced by macros as follows:

```
EXE = myprog
OBJ = main.o prod.o fun.o

$(EXE): $(OBJ)
    gcc -o $(EXE) $(OBJ) -lm
```

Undefined macro references are replaced by empty strings.

Macro values can be overridden using command line options. For example:

```
% make "EXE=testprog"
```

In addition to user defined macros, there are several internally defined "dynamic" macros, three of which are shown below:

Symbol	Value
`$?`	List of dependencies newer than target
`$@`	Name of the current target
`$<`	Name of the dependency, as if selected by `make` for use with an implicit rule

```
EXE     = myprog
OBJ     = main.o prod.o fun.o
CC      = gcc
CFLAGS  = -c
LDLIBS  = -lm

$(EXE): $(OBJ)
   $(CC) -o $@ $(OBJ) $(LDLIBS)

main.o: main.c
   $(CC) $(CFLAGS) $?

prod.o: prod.c
   $(CC) $(CFLAGS) $?

fun.o: fun.c
   $(CC) $(CFLAGS) $?
```

Example 10 makefile #2

9.5. Suffix Rules

Suffix rules are used to specify how to build files with one suffix from files with the same basename but with a different suffix. Suffix rules are used:

- when there is no rule for a specified target
- when you don't specify a command to build a target

Adding suffix rules:

1. Add the suffixes of both the target and dependency files to the suffixes list, if necessary, by providing them as dependencies to the `.SUFFIXES` special target
2. Add a target entry for the suffix rule

```
EXE     = myprog
OBJ     = main.o prod.o fun.o
CC      = gcc
CFLAGS  = -c
LDLIBS  = -lm

SUFFIXES: .o .c

$(EXE): $(OBJ)
   $(CC) -o $(EXE) $(OBJ) $(LDLIBS)

.c.o:
   $(CC) $(CFLAGS) $<
```

Example 11 makefile #3

Default suffix rules are listed in the file:

```
/usr/include/make/default.mk
```

Using default suffix rules, Example 3 can be simplified to:

```
EXE     = myprog
OBJ     = main.o prod.o fun.o
CC      = gcc
LDLIBS  = -lm

$(EXE): $(OBJ)
   $(CC) -o $(EXE) $(OBJ) $(LDLIBS)
```

Example 12 makefile #4

9.6. Include File Dependencies

The following example shows dependency relationships for "include" files:

```
EXE     = myprog
OBJ     = main.o prod.o fun.o
CC      = gcc
LDLIBS  = -lm

$(EXE): $(OBJ)
   $(CC) -o $(EXE) $(OBJ) $(LDLIBS)

$(OBJ): header.h
```

Example 13 Include File Dependencies

9.7. Invoking `make`

Make is invoked as follows:

```
% make [options] [target]
```

Commonly used options include:

`-f` *`filename`*	uses named file for make description file instead of `makefile` or `Makefile`
`-n`	displays commands `make` is to perform without executing themuseful for debugging makefiles

By default, `make` attempts to build the first target it encounters. By specifying a target, it is possible to update an individual target.

In some cases, relying on the use of default suffix rules and macros can completely eliminate the need for a makefile.

9.8. Common `make` Mistakes

Problems frequently encountered when using `make`:

- Forgetting to begin command lines with a tab (spaces cannot be used)
- Forgetting to continue non-comment lines with a backslash (\)
- Confusing dependencies—using a source file name where an object file name should be used, etc.

Suggestion:

- Develop a template makefile using default rules, macros, etc. and modify this as necessary for new projects

10. References

Many online and hardcopy references were used in creating this class. Among the best of these are the following:

- "sed and awk"

  by Dale Dougherty
  O'Reilly and Associates, Inc.
  ISBN: 0-937175-59-5

- "Managing Projects with make"

  by Andrew Oram & Steve Talbott
  O'Reilly and Associates, Inc.
  ISBN: 0-937175-90-0

- the various `man` pages

11. Summary

It is important to know what tools you have available to you, and how to use those tools. There are several hurdles to using these tools:

1. learn how the tools work
2. apply UNIX regular expression syntax
3. get the knack of script writing

Hopefully these notes will help you efficiently overcome these hurdles.