COP 4342, Fall 2006: Introduction

- ☞ History of Unix
- ☞ Shells: what they are and how they work
- ☞ Commands: built-in, aliases, and program invocations
- ☞ Tree-structured resources: processes and files
- ☞ Finding more information: `man`, `info`, and `Google`.

History of Unix

- ☞ Unix is now more than 30 years old, began in 1969 (*The Evolution of the Unix Time-sharing System*, Ritchie at http://cm.bell-labs.com/cm/cs/who/dmr/hist.htr

Introduction to Unix

☞ Started at AT&T's Bell Labs, originally derived from MULTICS. Original hardware was a DEC PDP-7, and the filesystem was hierarchical but did not have path names (i.e., there was no equivalent to name such as `/etc/hosts`, it would just be `hosts`; directory information was kept in a special file called dd)

☞ Rather than a *product* from a manufacturer, Unix began as collaboration with these goals:

- Simplicity
- Multi-user support
- Portability
- Universities could get source code easily
- Users shared ideas, programs, bug fixes

Introduction to Unix

- The development of early Unix was user-driven rather than corporate-driven
- Note that Liñux and the BSDs (FreeB̃SD, OpenB̃SD, NetB̃SD) now flouri̇̃sh in simĩlar “open source” enṽironm̃ents (`http:/̃/www.f̃reebsd~org, http:` `http:/̃/www.ñetbsd.õrg`)
- The first meeting of the **Unix User Group** was in May, 1974; this group would late become the **Usenix Association**

Old Unix

- ☞ Processes were very different

Old Unix

- Originally
  - Parent first closed all of its open files
  - Then it linked to the executable and opened it
  - Then the parent copied a bootstrap to the top of memory and jumped into the bootstrap

Old Unix

- The bootstrap copied the code for the new process over the parent’s code and then jumped into it
- When the child did an exit, it first copied in the parent process code into its code area, and then jumped back into the parent code at the beginning

Old Unix

- Today the parent does:
  - fork(2) (to create a new child process)
  - exec*(2) (to have the child process sta
    executing a new program)
  - wait*(2) (to wait on the child (or at l
    check on its status if non-blocking))

Unix Today

- ☞ Linux: a complete Unix-compatible operating system
  - ➾ Runs on huge array of hardware, from IBM's biggest machines down to commodity routers such as the Linksys WRT54G (which you can even hack, see *Linux on Linksys Wi-Fi Routers* at Linux Journal (http://www.linuxjournal.com/article/7322)).
  - ➾ Based on Linux Torvalds' kernel (he is still in charge of kernel development, though now many

people work on the kernel)

- The Linux distribution on the linprog machines is Scientific Linux; it includes a full development environment, X-Windows, NFS, office environment products (word processors, spreadsheets, etc), C, C++, Fortran, several mail systems (exim, postfix, and sendmail) and whole lot more (a full install is 5 gigabytes)
- Linux is mostly POSIX.1 compliant (for a good FAQ on POSIX, see http://www.opengroup.org/austin/p

Command Line versus Graphical Interface

- ☞ Typing is faster than mousing
- ☞ Graphics are computationally expensive, terminal handling is computationally inexpensive
- ☞ Easy to automate command lines, especially by utilizing histories
- ☞ Unix tools are designed to act as filters

The Layers of Unix

- ☞ Kernel → Provides access to system resources, both virtual and physical
- ☞ Shell → Provides a means to start other processes via keyboard input and screen output
- ☞ Tools → The vast array of programs that you can run to accomplish tasks

Some definitions

☞ "executable" $\longrightarrow$ A file that can be "executed" by creating a new process. There are two basic types of executables: binary executables, which natively run on hardware, and "script" executables which first invoke an interpreter. Script executables generally are human-readable (though, for instance, Zend PHP scripts can be pre-compiled into a crude intermediate representation.)

- ☞ process → An activation of a program. A process creates an entry in the process table (however, in Linux, a thread, which is retains the execution context of the caller, also goes into the process table.)

- ☞ daemon → Generally a persistent process (or at least the child of a persistent process) that is usually intended to provide some sort of service.

Some definitions

- ☞ user shell → Provides an environment that accepts keyboard input and provides screen output in order to allow a user to execute programs.
- ☞ “built-in” command → A “built-in” command does not cause the execution of a new process; often, it is used to change the state of a shell itself.
- ☞ alias → An alias expands to another command

☞ variable → A way to reference state in a shell

☞ flag → A way to specify options on the command line, generally via either a single dash or a double dash

Characteristics of Filters

- ☞ Should read from stdin and write to stdout by default (though some older utilities require explicit flags).
- ☞ Generally, filters should not read configuration files but should instead take their input from stdin and look at the command line for options via command line "flags".
- ☞ The output from one filter ideally should be easily readable by another filter.

Well-known shells

- ☞ bash
- ☞ sh
- ☞ csh
- ☞ ksh
- ☞ tcsh
- ☞ zsh

Unix Files

- ☞ Unix files normally follow the paradigm of a “byte-stream”
- ☞ Filenames may consist of most characters except the NUL byte and “/”
- ☞ They are case sensitive
- ☞ Periods are generally used for any filename extensions

- ☞ Filenames that start with a period are treated somewhat differently

- ☞ Unix does not generally make automatic backups of files

Some popular extensions

- `.c .h` → C files
- `.pl .pm` → Perl files
- `.py` → Python files
- `.cpp .c++ .CC` → C++ files
- `.s` → assembly files
- `.o` → object file

- `.gz` → gzipped file
- `.rpm` → rpm file

Wildcards and globbing

- "*" matches any string
- "?" matches any one character
- "[]" lets you specify a character class
- Note: you can use "[][]" to specify match "]" or "["

Filesytems

- ☞ Directories which are tree-structured
- ☞ Directories are just special files that contain pointers to other files (including other directories)
- ☞ / is the root of a filesystem
- ☞ CWD or “Current Working Directory” is the default directory for a process

Filesytem paths

- ☞ In Unix, we use / to distinguish elements in a path
- ☞ Absolute paths start with / and start at the root
- ☞ Relative paths start with any other character and are interpreted as relative to the current working directory

More on paths

- ☞ “.” is a special path (actually in the filesystem) that points at the current directory
- ☞ “..” is a special path (actually in the filesystem) that points at the parent directory
- ☞ “ /” is often understood by a shell as the home directory of the current user
- ☞ “ username/” is often understood by a shells as the

home directory of “username”

Listing files

☞ `ls` → show all of the non-dot files as a simple multicolumn listing

☞ `ls -l` → show a detailed listing, one line per file

☞ `ls -a` → include the dot files

☞ `ls -d DIRNAME` → just show the information about the directory and not its contents

☞ `ls NAME NAME ...` → show the named files (if they exist)

File permissions, user classes

- ☞ owner → Each file in the filesystem has an uid associated with it called the owner
- ☞ group → Each file in the filesystem also a gid associated with it called the group
- ☞ others → Refers to all others users

File permissions, rwx

- ☞ `r` → permission to read a file
- ☞ `w` → permission to write to a file
- ☞ `x` → permission to execute a file

Changing permissions with `chmod`

- ☞ Octal notation : `chmod 4755 /bin/ls`
- ☞ Symbolic notation : `chmod og+w /etc/hosts`

Removing files

- ☞ `rm FILENAME` removes the named files
- ☞ `rm -r DIRNAME` removes a directory, even if it has some contents
- ☞ `rm -f NAME` removes a file (if possible) without complaining or query
- ☞ `rm -i NAME` queries any and all removals before they are committed

- ☞ `rmdir DIRNAME` removes directory iff it is empty

- ☞ Recovering files after deletion is generally very hard (if not impossible) and if the filesystem is not quiescent, it becomes increasingly difficult to do

Manipulating files with `cp` and `mv`

- ☞ `cp FILE1 FILE2` copies a file
- ☞ `cp -r DIR1 DIR2` copies a directory; creates DIR2 if it doesn't exist otherwise puts the new copy inside of DIR2
- ☞ `cp -a DIR1 DIR2` like -r, but also does a very good job of preserving ownership, permissions, soft links and so forth

☞ `mv NAME1 NAME2` moves a file directory

Standard i/o

- ☞ Each process that starts on the system starts with three active file descriptors: 0, 1, and 2
- ☞ 0 $\longrightarrow$ is standard input, and is where a process by default expects to read input
- ☞ 1 $\longrightarrow$ is standard output, and is where a process by default will write output
- ☞ 2 $\longrightarrow$ is standard error, and is where a process by

default sends error messages

Redirection

- You can use and $<$ to provide simple redirection
- You can be explicit in bash and provide the actual file descriptor number
- For instance, in bash you can do “ls whatever 2 /dev/null” will make any error message disappear like the -f option in rm.
- You can use to append to a file

Displaying files

- ☞ `cat` $\longrightarrow$ Lets you see the contents with no paging
- ☞ `more` $\longrightarrow$ Pages output
- ☞ `less` $\longrightarrow$ Also pages output, will let you go backwards even with piped input
- ☞ `head` $\longrightarrow$ Just show the first lines of a file
- ☞ `tail` $\longrightarrow$ Just show the end lines of a file

Piping

- A pipe "|" simply lets you join the output of one program to the input of another
- The "tee" program lets you split the output of one program to go to the input of a program and to stdout

Finding more information

- ☞ The `man` program is a great place to start.
- ☞ The `info` program puts you in an emacs session.
- ☞ `Google` is your friend.

Processes

- ☞ Executables can be executed as processes
- ☞ Keyboard control of jobs
- ☞ `ps`, `top`, `pstree`
- ☞ `kill` doesn't kill, it sends signals
- ☞ `cron`, `anacron`

Executables can be executed as processes

- ☞ A process has an entry in the process table, and is initially loaded from a file in the filesystem
- ☞ An executable is a file in the filesystem which
  - ➾ Has the appropriate “x” flag(s) set
  - ➾ Either begins with a line of the form `#!/SOME/OTHER` or is in a binary format such as ELF or COFF

“Foreground” versus “Background”

- ☞ A process that is in the “foreground” of a shell means that the shell is waiting for the process to finish before accepting more input.
- ☞ A process that is in the “background” of a shell means that the shell will accept other commands while the process is executing. Generally, a “background” process can be brought to the “foreground”.

Shell communication with processes

☞ If a process is in the foreground, then by default when a ctrl-c is pressed and then mapped by stty to send a signal SIGINT, that SIGINT will be propagated to the foreground process. By default when a ctrl-z is pressed and then mapped by stty to send a signal SIGSTOP to the foreground process suspending the process. From there, you can either terminate it, put it in the background, or unsuspend it back to the foreground.

☞ If a process is in the background, you can use `kill` to explicitly send signals.

Shell job control

- ☞ You can place many processes simultaneously in the background; most shells will keep track of these and allow you to also access them via logical pids.
- ☞ You can either use ctrl-z / bg for a process that is in the foreground, or use a terminal “&” when you start the process.

Shell job control continued

- ☞ You can use `jobs` to keep up with which jobs you have running.

- ☞ You can use `fg %N` to bring job N back to the foreground.

`ps`

- ☞ You can also use `ps` to look at various portions of the process table.
- ☞ My favorites are `ps alxwww` and `ps -elf`.
- ☞ You pick and choose whatever format you like for output with the `ps -o --sort` option. For example, `ps -e -opid,uid,cmd --sort=uid`
- ☞ You can also show threads with the `ps -m` option.

`kill`

☞ Sending signals:

- `kill -KILL pid` → "unstoppable" kill (aka `kill -9 pid`)
- `kill -TERM pid` → terminate, usually much cleaner
- `kill -HUP pid` → either reload or terminate, usually clean if termination
- `kill -STOP pid` → suspend a process
- `kill -CONT pid` → restart a suspended process

☞ `kill` is generally a built-in, but there is also usually a kill program. The program version will not usually work with logical pids (unless your shell happens to translate logical pids to real pids before invoking kill, or the kill program is written such that it reparses the command line. For example, try `/usr/bin/kill -STOP %1`).

`top`

☞ The program `top` gives you a dynamic view of the process table.

☞ You can make it run faster with the “s” command.

☞ You can do “snapshots” with the `-b` (batch) option and the `-i iterations` option.

`pstree`

☞ Shows processes as a tree. Some options are:

- `-c` → Disable compaction.
- `-G` → Try to make graphical line drawing rather than just character
- `-Hpid` → Try to highlight a particular process and its ancestors
- `-p` → Show pids

☞ You can limit output to a user (specified by a user name) or to pid (specified by pid number)

`cron`

- You can run programs at arbitrary times with `cron`
  - Use `crontab -e` to edit your crontab (you can set EDITOR to specify an editor)
  - The five time fields are `minute, hour, dayOfMon` `month, dayOfWeek` where Sunday=0 for dayOfWeek

Shell Programming Topics

- ☞ Creating Shell Scripts
- ☞ Globbing
- ☞ Aliases, Variables/Arguments, and Expressions

Shell Programming Topics

- ☞ Shells, data, and debugging
- ☞ Structuring control flow
- ☞ Exit status

Shell Programming Topics

- Not (just) globbing: regular expressions
  - `grep, awk, perl` all use regular expressions

Advantages of shell scripts

- ☞ Can very easily automate a group of tasks, especially those with i/o that are related
- ☞ Can very easily leverage powerful Unix tools

Disadvantages of shell scripts

- Shell scripts execute slowly.
- Advanced programming techniques aren’t a feature of shell programming. Abstraction and encapsulation are poorly supported.

What shell to use

- ☞ For programming, most people have preferred `sh` and its derivatives such as `bash`.
- ☞ We will use `bash` for programming, although we will also talk about `csh` when appropriate in command shells.

What shell to use

☞ In the past, many people have preferred `csh` and `tcsh` as command line shells; however, it appears that `bash` is now preferred since its support for command line editing is quite strong and it also is quite useful for shell programming.

What shell to use

☞ There is also program `busybox` which is also worth knowing about. It is a shell — and a lot more. The binary itself includes many other programs such as `head`, `tail`, `ps`, `top`, `find`, `crontab`, and tar as built-ins.

Finding more information

- ☞ `man bash`
- ☞ `man {alias, bg, bind, break, builtin, cd command, compgen, ...}`
- ☞ `info bash`
- ☞ Google `bash`

Creating a script

☞ By convention, we use an extension of `.sh` for shell scripts.

☞ The first line needs to be

```
#!/bin/bash
#!/bin/sh
#!/bin/csh
#!/sbin/bash
```

Creating a script

☞ Now you should put some comments:

```
# 2006 09 06 -- original version by rdl
# 2006 09 07 -- updated ‘‘text’’ by rdl
#
# this shell program is used to confabula
#
```

Using echo

☞ The program (and builtin) echo is useful for sending a given string or strings to stdout.

```
[langley@sophie 2006-Fall]$ echo a b c
a b c
[langley@sophie 2006-Fall]$ echo "a b c"
a b c
[langley@sophie 2006-Fall]$ echo "$SHELL a b c"
/bin/bash a b c
[langley@sophie 2006-Fall]$ echo $SHELL a b c
/bin/bash a b c
[langley@sophie 2006-Fall]$ echo '$SHELL a b c'
$SHELL a b c
```

Shell variables

- ☞ Do not have to be declared: just use them. (If you want to, you can declare them with `declare`; generally only useful to make variables read-only.)
- ☞ Can be assigned a value, or can just have a blank value
- ☞ Can dereferenced with a “$”

Shell variables

Examples:

```
[langley@sophie 2006-Fall]$ a=b
[langley@sophie 2006-Fall]$ b=$a
[langley@sophie 2006-Fall]$ echo "a = $a , b = $b"
a = b , b = b
```

`read`ing values from the command line

From the man page for `bash`:

```
‘‘One  line  is  read  from  the  standard input, . . .  and  the
first word is assigned to the first name, the second word to the
second name, and so on, with leftover words and their  interven-
ing  separators  assigned  to the last name.  If there are fewer
words read from the input stream than names, the remaining names
are  assigned  empty  values.  The characters in IFS are used to
split the line into words.’’
```

`read` example

```
[langley@sophie 2006-Fall]$ read a b c d e f
apple beta cherry delta eta figs and more
[langley@sophie 2006-Fall]$ echo "$a -- $b -- $c -- $d -- $e -- $f"
apple -- beta -- cherry -- delta -- eta -- figs and more
```

`read` example

It is also good to note that you can also specify that items are to go into an array rather than just individually named variables with the `-a ARRAYNAME` option.

For example:

```
[langley@sophie 2006-Fall]$ read -a arr
a b c d e f g h
[langley@sophie 2006-Fall]$ for i in 0 1 2 3 4 5 6 7
> do
> echo ${arr[$i]}   # note the odd syntax to deref!
> done
a
```

b
c
d
e
`f`
g
h

Command line parameters

☞ When you call a shell script, command line parameters are automatically setup with $1, $2, etc...

```
[langley@sophie 2006-Fall]$ ./Script1.sh abc def ghi
first 3 args: 'abc' 'def' 'ghi'
```

☞ $0 refers to the name of the command (the first item)

More on command line arguments

- ☞ $# refers to the number of command line arguments.
- ☞ $@ refers to the all of the command lines arguments in one string.

Example:

```
[langley@sophie 2006-Fall]$ ./Script2.sh abc def ghi jkl
There are 4 arguments: abc def ghi jkl
```

Debugging tips

- ☞ The options `-x` and `-v` are very helpful. You can either add them to the initial `#!` line, or you can call the shell at the command line:
- ☞ bash -xv Script1.sh abc def

Example:

```
[langley@sophie 2006-Fall]$ bash -xv Script1.sh ls asd asdf asdf
#!/bin/bash
```

```
# 2006 09 06 -- Small test script

echo "first 3 args: ’$1’ ’$2’ ’$3’"
+ echo ’first 3 args: ’\’’ls’\’’ ’\’’asd’\’’ ’\’’asdf’\’’’
first 3 args: ’ls’ ’asd’ ’asdf’
echo "cmd: ’$0’"
+ echo ’cmd: ’\’’Script1.sh’\’’’
cmd: ’Script1.sh’
[langley@sophie 2006-Fall]$ bash -x Script1.sh ls asd asdf asdf
+ echo ’first 3 args: ’\’’ls’\’’ ’\’’asd’\’’ ’\’’asdf’\’’’
first 3 args: ’ls’ ’asd’ ’asdf’
+ echo ’cmd: ’\’’Script1.sh’\’’’
cmd: ’Script1.sh’
```

Testing

☞ You can test with square brackets:

```
$ [ $ -e /etc/hosts $ ] $
```

☞ You can also test with `test`:

```
test -e /etc/hosts
```

Testing

Example:

```
[langley@sophie 2006-Fall]$ if test -e /etc/hosts
> then
> echo exists
> fi
exists
[langley@sophie 2006-Fall]$ if [ -e /etc/hosts ]
> then
> echo exists
> fi
exists
```

File testing conditions

You can readily check various file status items:

```
[ -d DIR ]          #  True if directory DIR exists.
[ -e SOMETHING ]    #  True if file or directory SOMETHING exists.
[ -f FILE ]         #  True if regular file FILE exists.
[ -r SOMETHING ]    #  True if file or directory SOMETHING exists and is r
[ -s SOMETHING ]    #  True if file or directory SOMETHING exists and
                    #  has a size greater than zero.
[ -x SOMETHING ]    #  True if file or directory SOMETHING exists and
                    #  is ‘‘executable’’ by the current userid.
```

Numeric testing conditions

You can readily check various numeric values:

```
[ 0 -eq 1 ]   # equality
[ 1 -ne 1 ]   # inequality
[ 1 -lt 1 ]   # less than
[ 1 -gt 1 ]   # greater than
[ 1 -le 1 ]   # less than or equal
[ 1 -ge 0 ]   # great than or equal
```

String testing conditions

You can readily check various numeric values:

```
[ -z STRING ]      # is the string STRING zero length?
[ -n STRING ]      # is the string STRING non-zero length?
[ STR1 == STR2 ]   # ‘‘bash’’ equality; POSIX prefers ‘‘=’’
[ STR1 != STR2 ]   # inequality
[ STR1 < STR2 ]    # less than
[ STR1 > STR2 ]    # greater than
```

Note that it is a very good idea to “” quote any string variables; otherwise, the corresponding blank in `if [ $var1 != ‘‘today’’ ]` becomes `if [`

```
!= ''today'' ]!
```

exit

☞ You can explicitly exit a shell with `exit`, which can take an argument which will give the exit status of the process. (If you don't specify the optional value, the exit status for the whole shell will take the value of the last command to execute.)

```
[langley@sophie 2006-Fall]$ bash
[langley@sophie 2006-Fall]$ exit 3
exit
[langley@sophie 2006-Fall]$ echo $?
3
```

if / then

☞ We can write `if / then` statements like:

```
if condition
then
   [ ... statements ... ]
fi
```

Quoting

- ☞ Single quotes stop any globbing or variable expansion within them, and create a single token (i.e., whitespace within the quotes is not treated as a separator character.)
- ☞ Double quotes allow globbing and variable expansion within them, and create a single token (i.e., whitespace within the quotes is not treated as a separator character.)

☞ You can use the backslash to quote any single character.

Quoting examples

```
animal=''horse''
echo $animal     #prints: horse
echo '$animal'   #prints: $animal
echo ''$animal'' #prints: horse
cost=2000
echo 'cost: $cost'      #prints: cost: $cost
echo ''chost: $cost''   #prints: cost: 2000
echo ''cost: \$cost''   #prints: cost: $cost
echo ''cost: \$$cost'' #prints: cost: $2000
```

Multiple conditions

```
[ $1 -eq $2 ] && [ -e /etc/hosts ]
[ $1 -eq $2 ] || [ -d /etc ]
```

General if/then/else

```
if condition
then
  [ ... statements ... ]
elif condition
then
  [ ... statements ... ]
[ ... more elifs ... ]
else
  [ ... statements ... ]
fi
```

If example

```
#!/bin/bash
# 2006 09 08 - demonstrate if / then / else
if [ "x$1" != "x" ] && [ -f "$1" ]
then
   echo -n "Remove $1 (n)? "
   read answer
   if [ $answer == "y" ] || [ $answer == "Y" ] || [ $answer == "yes" ]
   then
      echo "Would remove"
   else
      echo "Would NOT remove"
   fi
else
   echo "Please specify a regular file"
fi
```

If example

```
#!/bin/bash
# 2006 09 08 - demonstrate if / then / else
if [ "x$1" == "x" ]
then
  echo "Please specify a regular filename!"
  exit 1
elif [ ! -f "$1" ]
then
  echo "$1 is not a regular file!"
  exit 1
else
   echo -n "Remove $1 (n)? "
   read answer
   if [ $answer == "y" ] || [ $answer == "Y" ] || [ $answer == "yes" ]
   then
```

```
        echo "Would remove"
    else
        echo "Would NOT remove"
    fi
fi
```

The case statement

`case WORD in PATTERN1 ) COMMANDS ;; PATTERN2 ) COMMANDS ;; ...  esac`

The idea here is that WORD is tested against the various PATTERNs listed, in order. The first match then executes the associated COMMANDs.

Case example

```
#!/bin/bash
# 2006 09 08 - case example
case $1 in
  "yes")
     echo "Thanks!"
     exit 0
     ;;
  "no")
     echo "Okay!"
     exit 1
     ;;
  *)
     echo "Please use either ’yes’ or ’no’ (case-sensitive)"
     ;;
esac;
```

While/until loops

```
while list; do list; done;
```

```
until list; do list; done;
```

`while` executes the do list as long as the **last** command in the `list` returns 0. `until` executes until the last command in the `list` returns 0.

`while` example

```
#!/bin/bash
# 2006 06 08 -- rdl
echo -n "Now 'finish' ? "
read cmd
while test $cmd != "finish"
do
   rm NONEXIST
   echo "Status of \$? == $?"
   echo -n "Now 'finish' ? "
   read cmd
done
```

`until` example

```
#!/bin/bash
# 2006 06 08 -- rdl
echo -n "Now ’finish’ ? "
read cmd
until test $cmd == "finish"
do
   rm NONEXIST
   echo "Status of \$? == $?"
   echo -n "Now ’finish’ ? "
   read cmd
done
```

Shifting the arguments

You can "shift" the argument list, eliminating the current $1 and replacing it with the current $2, and so forth:

Shifting the arguments

```
#!/bin/bash
while [ $# -gt 0 ]
do
  echo "$# --> arguments == '$@'"
  shift;
done
```

Shifting the arguments

```
[langley@sophie 2006-Fall]$ ./Script8.sh a b c d e f g h
8 --> arguments == 'a b c d e f g h'
7 --> arguments == 'b c d e f g h'
6 --> arguments == 'c d e f g h'
5 --> arguments == 'd e f g h'
4 --> arguments == 'e f g h'
3 --> arguments == 'f g h'
2 --> arguments == 'g h'
1 --> arguments == 'h'
[langley@sophie 2006-Fall]$
```

`exit`

We have already talked about `exit`, but to reiterate some points about exit:

☞ An `exit` status of zero should indicate success. It is a good idea to use an explicit `exit NUM` in scripts.

☞ An `exit` status that is non-zero should indicate failure.

☞ C programs use `exit(NUM)` to return a status.

exit example

```
#/bin/bash
# 2006 09 08 -- rdl Script9.sh
if ./Script10.sh
then
  echo -n "Enter filename: "
  read filename
  echo "You entered '$filename'"
else
  echo "Okay, no filename needed."
fi
```

exit example

```
#/bin/bash
# 2006 09 08 -- rdl Script9.sh
while /bin/true
do
  echo -n "Should I ask for a filename? "
  read answer
  case $answer in
     "no")
          exit 1
          ;;
     "yes")
          exit 0
          ;;
     *)
          ;;
```

```
  esac
done
```

Regular expressions

Regular expressions are a convenient way to describe a sequence of characters, and regular expressions are part of such programs as `emacs`, `awk`, and `perl`.

Regular expressions: operations

Concatenation: just place items adjacent, such `ab`, `xyz`, or `somechars`

Regular expressions: operations

Repetition: we use “*” to indicate repetition zero or more times:

`a*b == b, ab, aab, aaab, ...`

Regular expressions: operations

Special case of repetition: we can specify one or more times with +:

```
a+b == ab, aab, aaab, ...
```

Regular expressions: characters and classes

The dot “.” can indicate any character, such as

```
a.b == a1b, a2b, a3b, ...
```

Regular expressions: characters and classes

To specify a class of characters, you can use the [ ] syntax:

```
[abc] == a, b, c
[a-d] == a, b, c, d
[â-z] == NOT a lower case character
[0-9] == 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Anchoring

You can “anchor” an expression to either the beginning of a string or its end, or both. Use ˆto indicate the beginning of a line, and $ to indicate the end:

âbc$ matches a line that consists exactly of abc

abc$ matches a line that ends in abc

âbc maches a lines that begins with abc

Alternation and grouping

You can specify a group with round brackets “(“ and “)”.

You can specify alternatives with a vertical “|”

`(abc)|(def)` matches either `abc` or `def`

Note on grouping

It also possible in many instances possible to make a reference to whatever matched a group in round brackets.

Check chapter 32 for more on regular expressions

32.20 has a good summary of metacharacters for different programs.

32.21 has a reference with many useful examples

Using `grep/egrep`

You can use the `grep` program to find strings in files. The “-i” option makes the search case-insensitive. If no file or files are specified, then `grep` looks to stdin for input. `grep` also adds “?” as a special character that matches 0 or 1 instance of any character.

Examples with `grep/egrep`

```
egrep  [Ll]angley *   # finds instances of ‘‘langley’’ or
                      # ‘‘Langley’’ in all files in the
                      # current working directory
egrep -i she?p *      # finds case-insensitive instances of
                      # shep and she.p
egrep -c /bin/bash *  # shows filename and
                         # number of matches
```

Popular options with `grep/egrep`

- ☞ -i → case-insensitive
- ☞ -c → display count of matching lines rather all matching lines
- ☞ -v → invert the matching
- ☞ -H → always show filenames
- ☞ -h → always suppress filenames

☞ -l → just show the filenames that have one or more matches

wc

You can use the wc program to count characters, words, and lines:

```
wc -l *     # count the number of lines in all files
wc -w *     # count the number of words in all files
wc -c *     # count the number of characters in all files
wc -lw *    # count the number of words and lines in all files
wc *        # count words, characters, and lines in all files
```

`touch`

Touch is usually a program, but it can be a shell built-in such as with `busybox`.

The `touch` program by default changes the access and modification times for the files listed as arguments. If the file does not exist, it is created as a zero length file (unless you use the `-c` option.)

You can also set either or both of the times to arbitrary values, such as with the `-t`, `-d`, `-B`, and `-F` options.

Backquotes and textual substitution

If you surround a command with backquotes, the standard output of the command is substituted for the quoted material.

For instance,

```
[langley@sophie 2006-Fall]$ echo ‘ls 0*tex‘
01-introduction.tex 02-processes.tex 03-shells1.tex 03-shells2.tex 04-she
[langley@sophie 2006-Fall]$ echo ‘egrep -l Langley *‘
03-shells2.tex Syllabus-Fall.html Syllabus-Fall.html.1 Syllabus Summer.ht
[langley@sophie 2006-Fall]$ now=‘date‘
[langley@sophie 2006-Fall]$ echo $now
Mon Sep 11 09:55:09 EDT 2006
```

Backquotes and textual substitution

```
if [ ‘wc -l < /etc/hosts‘ -lt 10 ]; then echo "lt"; fi
 # use ‘‘<’’ to prevent filename from
```

xargs

```
xargs COMMAND -n N [INITIAL-ARGUMENTS]
```

`xargs` reads from stdin to obtain arguments for the COMMAND. You may specify initial arguments with the COMMAND. If you specify `-n N`, then only up to N arguments are given to any invocation of COMMAND. For instance,

```
[langley@sophie 2006-Fall]$ cat /etc/hosts | xargs -n 1 ping -c 1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.075 ms
```

```
--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.075/0.075/0.075/0.000 ms, pipe 2
PING localhost.localdomain (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=64 time=0

--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.060/0.060/0.060/0.000 ms, pipe 2
PING localhost.localdomain (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=64 time=0
```

The `for` statement

```
for name in LIST0 ; do LIST1 ; done
for name ; do LIST1 ; done   # useful for scripts
for (( EXPR1 ; EXPR2 ; EXPR3 )) ; do LIST1 ; done
```

In the last form, EXPR? are evaluated as arithmetic expressions.

The `for` statement

```
[langley@sophie 2006-Fall]$ for (( ip = 0 ; ip < 5 ; ip = ip+1)) do echo
0
1
2
3
4
```

The `for` statement

```
for i in ‘cat /etc/hosts‘
do
  ping -c 1 $i
done
```

`break` and `continue` statements

`break` terminates the current loop immediately and goes on to the next statement after the loop. `continue` starts the next iteration of a loop.

`break` and `continue` statements

For example,

```
for name in *
do
  if [ -f ‘‘$name’’ ]
  then
     echo ‘‘skipping $name’’
     continue
  else
     echo ‘‘process $name’’
  fi
done
```

`expr`

You can use `expr` to evaluate arithmetic statements, some regular expression matching, and some string manipulation. (You can also use either `bc` or `dc` for more complex arithmetic expressions.)

expr

```
files=10
dirs=‘expr $files + 5‘
limit=15
if [ ‘expr $files + $dirs‘ < $limit’’ ]
then
 echo ‘‘okay’’
else
 echo ‘‘too many!’’
fi
```

awk

One of the more powerful programs found on Unix machines is `awk`, and its updated versions, `nawk` and `gawk`.

It is most useful for handling text information that is separated into a series of uniform records. The most common one that it handles is records of one line, divided by either column numbers or by a field separator. For instance, handling the password file is a snap with `awk`.

awk

The password file on a Unix machine looks something like:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
```

awk

You can quickly get a list of usernames into a single string variable with:

```
[langley@sophie 2006-Fall]$ usernames=‘awk -F: ’{print $1}’ /etc/passwd‘
[langley@sophie 2006-Fall]$ echo $usernames
root bin daemon adm lp sync shutdown halt mail
[langley@sophie 2006-Fall]$ usernames=‘awk ’{print $1}’ FS=: /etc/passwd‘
[langley@sophie 2006-Fall]$ echo $usernames
root bin daemon adm lp sync shutdown halt mail
```

awk

Fundamentally, awk scripts consist of a series of pairs:

```
PATTERN { ACTION }
```

awk

where the PATTERN can be a

- *☞ /regular expression/*
- *☞ relational expression*
- *☞ pattern-matching expression*
- ☞ *BEGIN* or *END*

awk

By default, the record separator is a newline so awk works on a line-by-line basis by default.

If no PATTERN is specified, then the ACTION is always taken for each record.

If no ACTION specified, then the each records that matches a pattern is written to stdout.

awk

You can specify that an ACTION can take place before any records are read with the keyword `BEGIN` for the PATTERN.

You can specify that an ACTION can take place after all records are read with the keyword `END` for the PATTERN.

With PATTERNs, you can also negate (with `!`) them, logically “and” two PATTERNs (with &&), and logically “or” two PATTERNs (with ||).

awk

Some examples of regular expressions in awk:

```
[langley@sophie 2006-Fall]$ awk '/[Ll]angley/ {print $0}' /etc/passwd
langley:x:500:500:Randolph Langley:/home/langley:/bin/bash
[langley@sophie 2006-Fall]$ awk '/^#/' /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
```

awk

- ☞ $0 refers to the whole record, $N refers to the Nth field in a record
- ☞ NF refers to the number of fields in a record (example, `awk -F: 'END print NF' /etc/passw`
  tells you that there are seven fields used in the password file.)
- ☞ NR refers to which record (by default, line) you are currently at.

awk

Some examples of relational expressions:

```
$1 == ‘‘lane’’  # does the first field equal the string ‘‘lane’’?
$1 == $7        # are fields one and seven equal?
NR > 1000       # have we processed more than 1000 records?
NF > 10         # does this record have more than 10 fields?
NF > 5 && $1 = ‘‘me’’ # compound test
/if/&&/up/      # does the record contain both strings if and up?
```

awk

You can also check a given field against a regular expression:

```
$1 ~ /D[Rr]\./   # does the first field contain a Dr. or DR.?
$1 !~ /#/       # does the first field have a # in it?
```

awk

ACTIONs are specified with { }. You can use semicolons to separate statements with the braces (although newlines work also). Popular statements are `print`, `if {} else {}`, and `system`.

`awk` is very powerful! Henry Spencer wrote an assembler in `awk`.

awk example scripts

```
{ print $1, $2 }   # print the first two fields of each record

$0 !~ /^$/         # print all non-empty lines

$2 > 0 && $2 < 10 { print $2 }  # print field 2 if it is 0 < $2 < 10

BEGIN {FS='':''
sum = 0}    # sum field 3 and print the sum
{sum += $3}
END {print sum}
```

The `tr` utility

Allows you to delete, replace, or “squeeze” characters from standard input. The `-d` option deletes the characters specified in the first argument; `-s` squeeze removes all repetitions of characters in the first argument with a single instance of the character. The normal mode is to substitute characters from the first argument with characters from the second argument.

The `tr` utility

```
[langley@sophie 2006-Fall]$ cat /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1                localhost.localdomain localhost
128.186.120.8                sophie.cs.fsu.edu
127.0.0.1                a.as-us.falkag.net
127.0.0.1                clk.atdmt.com
[langley@sophie 2006-Fall]$ cat /etc/hosts | tr 'a-z' 'A-Z'
# DO NOT REMOVE THE FOLLOWING LINE, OR VARIOUS PROGRAMS
# THAT REQUIRE NETWORK FUNCTIONALITY WILL FAIL.
127.0.0.1                LOCALHOST.LOCALDOMAIN LOCALHOST
128.186.120.8                SOPHIE.CS.FSU.EDU
127.0.0.1                A.AS-US.FALKAG.NET
127.0.0.1                CLK.ATDMT.COM
```

More `tr` examples

```
tr '&' '#'     translate ampersands to hash

tr -s '\t'     squeeze consecutive tabs to one tab
```

More `tr` examples

```
[langley@sophie 2006-Fall]$ cat /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1                localhost.localdomain localhost
128.186.120.8                sophie.cs.fsu.edu
127.0.0.1                a.as-us.falkag.net
127.0.0.1                clk.atdmt.com
[langley@sophie 2006-Fall]$ tr -s ’\t’ < /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1        localhost.localdomain localhost
128.186.120.8        sophie.cs.fsu.edu
127.0.0.1        a.as-us.falkag.net
127.0.0.1        clk.atdmt.com
```

More `tr` examples

`tr -d '\015'` delete carriage returns from a DOS file

basename

`basename` lets you remove leading directory strings. It can also remove suffixes simply by specifying the suffix as a second argument.

```
[langley@sophie 2006-Fall]$ basename ‘pwd‘
2006-Fall
[langley@sophie 2006-Fall]$ var1=/etc/inetd.conf
[langley@sophie 2006-Fall]$ basename $var1 .conf
inetd
```

`dirname`

`dirname` does the opposite function of `basename`: it returns the leading path components from a directory name.

```
[langley@sophie 2006-Fall]$ echo ‘pwd‘
/mnt-tmp/Lexar/fsucs/cop-4342/2006-Fall
[langley@sophie 2006-Fall]$ dirname ‘pwd‘
/mnt-tmp/Lexar/fsucs/cop-4342
[langley@sophie 2006-Fall]$ dirname 05-shells4.tex
.
[langley@sophie 2006-Fall]$ dirname ‘pwd‘/xyz
/mnt-tmp/Lexar/fsucs/cop-4342/2006-Fall
```

`sort`

For all of the files listed, `sort` will sort the concatenated lines of those files to stdout. The most useful options are `-f`, which means to fold case, `-n` to sort numerically rather alphabetically, `-u` to remove duplicates (“u” is short for “unique”), and `-r` to reverse the order of the sort.

You can specify particular fields to sort by specifying a field separator (whitespace is the default) with the `-t` option, and then using `-k` to specify particular fields.

`sort` examples

```
[langley@sophie 2006-Fall]$ sort /etc/passwd
adm:x:3:4:adm:/var/adm:/sbin/nologin
amanda:x:33:6:Amanda user:/var/lib/amanda:/bin/bash
apache:x:48:48:Apache:/var/www:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
canna:x:39:39:Canna Service User:/var/lib/canna:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
desktop:x:80:80:desktop:/var/lib/menu/kde:/sbin/nologin
```

`sort` examples

```
[langley@sophie 2006-Fall]$ sort -r /etc/passwd
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
wnn:x:49:49:Wnn Input Server:/var/lib/wnn:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
vmail:x:502:502::/home/vmail:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
user1:x:505:505::/home/user1:/bin/bash
test:x:503:503::/home/test:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
```

`sort` examples

```
[langley@sophie 2006-Fall]$ sort -k3,3n -t: /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
```

`sort` examples

```
[langley@sophie 2006-Fall]$ sort -k4,4n -k3,3n -t: /etc/passwd
root:x:0:0:root:/root:/bin/bash
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
operator:x:11:0:operator:/root:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
```

`groff` and `gtbl`

☞ There are a lot of great packages out there, such as `graphviz`. A handy one is `groff`, a derivative of the ancient `troff` and `nroff` families. ("roff" comes from "runoff"; `man` pages are traditionally written in `nroff` format.)

☞ You can use `gtbl` with `groff` to quickly make nice PostScript tables.

```
gtbl some.tr | groff > /tmp/some.ps
```

groff and gtbl

```
.sp 10     # skip 10 lines
.ps 14     # point size 14 pt
.TS        # table start
center box tab(/);   # center the table in the page, put it in a box, and
c c c c    # center the first line
r r r r .  # right justify the rest
.sp .2v    # skip down 2/10s of a line
Last / First / Age / Zipcode    # column headers
.sp .1v    # skip down 1/10 of a line
_          # horizontal rule
_          # horizontal rule
.sp .3v    # skip down 3/10s of a line
Gordon/Flash/91/91191   # record one
.sp .2v                 # skip down
Jones/Carol/20/32399    # record two
```

```
.sp .2v                 # skip down
Miller/Bob/23/32499     # record three
.sp .2v                 # skip down
Yagi/Akihito/22/32111   # record four
.sp .1v                 # skip down
.TE                     # table end
```

`fmt`

Another great little utility is `fmt` which lets you quickly reformat a document.

You can use `-w` to control the width. `fmt` also prefers to see two spaces after a question mark, period, or exclamation point to indicate the end of a sentence.

`fmt` example

[langley@sophie 2006-Fall]$ cat lincoln.txt Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

`fmt` example

```
[langley@sophie 2006-Fall]$ fmt lincoln.txt
Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation,
or any nation so conceived and so dedicated, can long endure. We are met
on a great battle-field of that war. We have come to dedicate a portion
of that field, as a final resting place for those who here gave their
lives that that nation might live. It is altogether fitting and proper
that we should do this.
```

`fmt` example

```
[langley@sophie 2006-Fall]$ fmt -w 20 lincoln.txt
Four score and
seven years ago our
fathers brought
forth on this
continent, a new
nation, conceived
in Liberty, and
dedicated to the
proposition that
all men are created
equal.
```

`cut`

☞ `cut` allows you to extract columnar portions of a file. The columns can be specified either by a delimiter (the default delimiter is the tab character.)

☞ You can specify a delimiter with the `-d` option.

☞ You must specify either at least one field number with `-f`, a byte number with `-b`, or a character number with `-c`. With ordinary ASCII text, `-b` and `-c` mean the same thing, but if we ever get multi-byte characters handled correctly, it shouldn't.

`cut` examples

```
[langley@sophie 2006-Fall]$ cut -c 1 /etc/hosts
#
#
1
1
1
1
```

`cut` examples

```
[langley@sophie 2006-Fall]$ cut -b 1 /etc/hosts
#
#
1
1
1
1
```

`cut` examples

```
[langley@sophie 2006-Fall]$ cut -f1 /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1
128.186.120.8
127.0.0.1
127.0.0.1
```

`cut` examples

```
[langley@sophie 2006-Fall]$ cut -c1-10 /etc/hosts
# Do not r
# that req
127.0.0.1
128.186.12
127.0.0.1
127.0.0.1

[langley@sophie 2006-Fall]$ cut -d: -f1,5 /etc/passwd
netdump:Network Crash Dump user
sshd:Privilege-separated SSH
rpc:Portmapper RPC user
rpcuser:RPC Service User
nfsnobody:Anonymous NFS User
```

`paste`

`paste` lets you put two or more files together as columns. By default, the columns will be joined with a tab character, but you can use the `-d` option to specify a different delimiter.

`paste` example

```
prompt% cut -f1 /etc/hosts > /tmp/f1
prompt% cut -d: -f5 /etc/passwd /tmp/f2
prompt% paste -d: /tmp/f1 /tmp/f2
# Do not remove the following line, or various programs:root
# that require network functionality will fail.:bin
127.0.0.1:daemon
128.186.120.8:adm
127.0.0.1:lp
127.0.0.1:sync
```

`head` **and** `tail`

- ☞ These programs, as mentioned before, allow you to excerpt the initial or the final lines of a file.
- ☞ Used in combination, you can isolate an arbitrary range of lines.
- ☞ You can also use the `-f` option with `tail` to monitor a file for changes.
- ☞ By default, if you specify multiple files, you get a nice

little header to distinguish them.

`head` and `tail` examples

```
head /etc/passwd    # print the first 10 lines of passwd
tail -20 /etc/passwd   # print the last 20 lines of passwd
head -15 /etc/passwd | tail -5 # print lines 10 - 15 of passwd
tail -f /var/log/messages  # monitor the log ‘‘messages’’ file
```

sed

Chapter 34 of UPT has a good section on `sed`.

`sed` is a “stream editor.” It can edit files in place.

You can specify multiple `sed` scripts with `-e`.

`sed` examples

```
[langley@sophie 2006-Fall]$ sed "s/1/9/" < /etc/hosts
# Do not remove the following line, or various programs
# that require network functionality will fail.
927.0.0.1                    localhost.localdomain localhost
928.186.120.8                    sophie.cs.fsu.edu
927.0.0.1                    a.as-us.falkag.net
927.0.0.1                    clk.atdmt.com
```

`sed` **examples**

```
[langley@sophie 2006-Fall]$ sed -e "s/1/9/" -e "s/a/A/g" < /etc/hosts
# Do not remove the following line, or vArious progrAms
# thAt require network functionAlity will fAil.
927.0.0.1                locAlhost.locAldomAin locAlhost
928.186.120.8                sophie.cs.fsu.edu
927.0.0.1                A.As-us.fAlkAg.net
927.0.0.1                clk.Atdmt.com
```

Setting up your environment

- ☞ Environment variables – these variables are passed to child processes
- ☞ Aliases – modify the meaning of “commands”
- ☞ History – a record of your shell commands
- ☞ Command completion – lets you save keystrokes

Environmental variables

☞ Environmental variables are passed to child processes at invocation. (The child process can of course ignore them if it likes.)

☞ Children cannot modify parent's environmental variables – any modification by a child process are local to the child and any children it might create.

Environmental variables

☞ The traditional C "main" is usually defined something like:

```
int main(int argc, char *argv[], char *en
```

Setting environmental variables

CSH/TCSH: `setenv VARIABLE VALUE`

BASH: `export VARIABLE=VALUE`

old SH: `VARIABLE=VALUE ; export VARIABLE`

Note: there are a few special variables such as `path` and `home` that CSH/TCSH autosynchronizes between the two values.

Setting environmental variables

```
[langley@sophie 2006-Fall]$export VAR1=value
[langley@sophie 2006-Fall]$ bash
[langley@sophie 2006-Fall]$ echo $VAR1
value
[langley@sophie 2006-Fall]$ exit
exit
[langley@sophie 2006-Fall]$ csh
[langley@sophie 2006-Fall]$ echo $VAR1
value
```

Setting environmental variables

```
[langley@sophie 2006-Fall]$ csh
[langley@sophie 2006-Fall]$ setenv VAR2 bigvalue
[langley@sophie 2006-Fall]$ csh
[langley@sophie 2006-Fall]$ echo $VAR2
bigvalue
[langley@sophie 2006-Fall]$ exit
[langley@sophie 2006-Fall]$ exit
[langley@sophie 2006-Fall]$ bash
[langley@sophie 2006-Fall]$ echo $VAR2
bigvalue
```

Unsetting environmental variables

CSH/TCSH: `unsetenv VAR`

SH/BASH: `unset VAR`

You can also leave it as local variable in bask with `export -n VAR`.

Unsetting environmental variables

```
[langley@sophie 2006-Fall]$ csh
[langley@sophie 2006-Fall]$ setenv VAR99 testvar
[langley@sophie 2006-Fall]$ csh
[langley@sophie 2006-Fall]$ echo $VAR99
testvar
[langley@sophie 2006-Fall]$ unsetenv VAR99
[langley@sophie 2006-Fall]$ echo $VAR99
VAR99: Undefined variable.
[langley@sophie 2006-Fall]$ exit
[langley@sophie 2006-Fall]$ exit
[langley@sophie 2006-Fall]$ echo $VAR99
testvar
```

Unsetting environmental variables

```
[langley@sophie 2006-Fall]$ export VAR50=test
[langley@sophie 2006-Fall]$ bash
[langley@sophie 2006-Fall]$ echo $VAR50
test
[langley@sophie 2006-Fall]$ unset VAR50
[langley@sophie 2006-Fall]$ echo $VAR50

[langley@sophie 2006-Fall]$ exit
exit
[langley@sophie 2006-Fall]$ echo $VAR50
test
[langley@sophie 2006-Fall]$ export -n VAR50
[langley@sophie 2006-Fall]$ echo $VAR50
test
[langley@sophie 2006-Fall]$ bash
```

```
[langley@sophie 2006-Fall]$ echo $VAR50
```

Displaying your environment

BASH: `env, printenv, set, declare -x, typeset -x`

CSH: `env, printenv, setenv`

Predefined environmental variables

What is "predefined" is not so much the value of the variable as its name and its normal use.

- ☞ `PATH` : a list of directories to visit. They are delimited with ":". Note that csh/tcsh "autosynchronize" this variable.
- ☞ `EDITOR` : the default editor to start when you run a program that involves editing a file, such as `crontab -e`.

☞ `PRINTER` : the default printer to send to.

☞ `PWD` : your present working directory.

☞ `HOME` : your home directory.

☞ `SHELL` : the path to your current shell. (Be cautious with this one: in some shells, it is instead `shell`).

☞ `USER` : your username.

☞ `TERM` : your terminal type.

☞ `DISPLAY` : used by programs to find the X server to display their windows.

Aliases

An `alias` allows you to abbreviate a command. For instance, instead of using `/bin/ls -al`, you might abbreviate it to `ll` with:

SH/BASH: `alias ll=‘‘/bin/ls -al’’`

CSH/TCSH: `alias ll ‘‘/bin/ls -al’’`

Removing aliases

You can remove an alias with `unalias`.

Example:

```
unalias ll
```

`which`, `whatis`, `whereis`, `locate`

The program (or built-in) `which` simply gives you the path to the named executable as it would be interpreted by your shell invoking that executable, and is created by examining your path.

The program `locate` looks in a database for all accessible files in the filesystem that contain the substring you specify. You can also specify a regular expression, such as

```
locate -r 'ab.*ls'
```

The program `whatis` will give you the description line from the `man` page for the command you specify. (N.B.: You can also search the `man` page descriptions with `man -k keyword`.)

The program `whereis` will give you both the path to the executable named and the page to its manpage.

Setting your prompt

SH/BASH: `PS1='% '`

CSH/TCSH: `set prompt='% '`

“Sourcing” commands

Because ordinarily running a shell script means first forking a child process and then exec-ing the script in that child shell, it is not possible to modify the current shell’s environmental variables from just running a script.

Instead, we do what is called “sourcing” the script, which means simply executing its commands (such as setting environmental variables) inside the current shell process.

CSH/TCSH: `source FILE`

SH/BASH: `.   FILE`

N.B.: modern versions of bash also support the `source` built-in.

`.login , .profile`

When you login, your user shell is started with `-l`. For sh/bash, this means that shell will source your `.profile` file; for csh/tcsh, this means sourcing your `.login` file.

Typically, you would want your environmental variables in that file, and any other one-time commands that you want to do when logging in, such as checking for new email.

Shell `.*rc` files

For each shell that you start, generally a series of "run command" files, abbreviated as "rc" will be sourced. In these you can set up aliases and variables that you want for every shell (including those that are not interactive, such as those running under a crontab.)

BASH: `.bashrc`

CSH: `.cshrc`

There is also a `.tschrc` for `tcsh`. History, `sh` did not look for configuration files except when invoked as a login shell.

`.*rc` files in general

In general, many program use .*rc files. Some will ask you to setup the file; some will create it for you. Some want a whole directory.

- `.gvimrc`
- `.procmailrc`
- `.gtkrc`
- `.xfigrc`

☞ `.acrorc`

`.gvimrc`

- ☞ Set the background
- ☞ Set the size and type of the font
- ☞ Set the size of the window in characters
- ☞ Turn on or off syntax highlighting

`.procmailrc`

The syntax is quite obscure, but you can apply arbitrary rules to your incoming email via your `.procmailrc` file.

`.procmailrc` example

```
DOMAIN="<$1>"
RECIPIENT="<$2>"
WHATSIT="<$3>"
VERBOSE=on
LOGFILE=/tmp/procmail2.log
LOGABSTRACT=all
ROOTHOMEDIR=/home/vmail-users
ROOTINBOXDIR=/var/spool/vbox

:0
* RECIPIENT ?? ()\/[^<]*@
* MATCH ?? ()\/.*[^@]
{
   USER=$MATCH
}
```

```
:0 a
* DOMAIN ?? ()\/[^<].*[^>]
{
   DOMAINNOBRACKET=$MATCH
}

:0 a
${ROOTINBOXDIR}/${DOMAINNOBRACKET}/${USER}
```

Shell history

You can modify the number of lines kept in your history:

bash: `HISTSIZE=SOMENUMBER`

csh/tcsh: `set history=SOMENUMBER`

Your shell history lets you do many things: search commands that you ran in the past, re-execute commands, modify them, or save them off (bash lets you do the latter automatically in your

`.bash_history` file.)

Command history substitution

- `!!` → repeat last command
- âb̂ → repeat last command, but change a to b
- `!-N` → repeat the command N back in your history
- `history` → display the history
- `history N` → display the last N lines of history
- `!N` → repeat command N

☞ `!STRING` → repeat the last command that started with STRING.

Using previous command arguments

- ☞ `!$` → refers to the last argument of the previous command
- ☞ `!caret` → refers to the first argument of the previous command
- ☞ `!*` → refers to the all of the arguments of the previous command
- ☞ `!:n*` → refers to the arguments N through the last

argument of the previous command

Command line manipulation

You can use the arrow keys to move through your history, and back and forth on command lines.

With bash, you can use the default emacs key-bindings for thing such as end-of-line (ctrl-e) and beginning-of-line (ctrl-b).

Complete word function

If you are in the first word of a command, you can find all the matching commands up to that point with a TAB character.

If you are else in the line, you can use the TAB character to show all matching filenames in the current working directory, or if you have started an absolute path, then matching items down the path.

Perl

- ☞ Introduction
- ☞ Scalars
- ☞ Lists and arrays
- ☞ Control structures
- ☞ I/O
- ☞ Associative arrays/hashes

- ☞ Regular expressions
- ☞ Subroutines and objects
- ☞ Dealing with files
- ☞ Directory and file manipulation

Perl history

PERL stands for “Practical Extraction and Report Language” (although there is the alternative “Pathologically Eclectic Rubbish Lister”.)

It was created by Larry Wall and became known in the 1990s.

It was available both from ucbvax and via Usenet.

Perl is released under the Artistic License and under the GNU General Public and License.

Perl’s Artistic License

6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package. If such scripts or library files are aggregated with this Package via the so-called “undump” or “unexec” methods of producing a binary executable image, then distribution of such an

image shall neither be construed as a distribution of this Package nor shall it fall under the restrictions of Paragraphs 3 and 4, provided that you do not represent such an executable image as a Standard Version of this Package.

7. C subroutines (or comparably compiled subroutines in other languages) supplied by you and linked into this Package in order to emulate subroutines and variables of the language defined by this Package shall not be considered part of this Package, but are the equivalent of input as in Paragraph 6, provided

these subroutines do not change the language in any way that would cause it to fail the regression tests for the language.

Advantages of Perl

- ☞ Perl 5 is a pleasant language to program in.
- ☞ It fills a niche between shell scripts and conventional languages.
- ☞ It is very appropriate for system administration scripts.
- ☞ It is very useful for text processing.

☞ It is a high level language with nice support for objects. A Perl program often will take far less space than the equivalent C or C++ program.

Perl is Interpreted

- ☞ Perl is first "compiled" into bytecodes; those bytecodes are then interpreted. Ruby, Python, and Java all do essentially the same thing.
- ☞ This is faster than shell interpretation, particularly when you get into some sort of loop. It is still slower than standard compilation.
- ☞ On the machine I tested, an empty loop in bash for 1 million iterations takes 34 seconds; 1 million

iterations of an empty loop in Perl takes 0.47 seconds. 1 million iterations of empty loop in C run in 0.001 to 0.003 seconds.

A Perl Program

```
#!/usr/bin/perl -w
# 2006 09 18 - rdl
use strict;
print ‘‘Hello, World!\n’’;
exit 0;
```

The first line indicates that we are to actually execute “/usr/bin/perl”. (The “-w” indicates “please whine”.) The second line is a comment. The third line makes it mandatory to declare variables. (Notice that statements are terminated with semicolons.) The 4th line does our Hello World, and 5th line terminates the program.

Basic concepts

- ☞ There is no explicit "main", but you can have subroutines.
- ☞ Features are taken from a large variety of languages, but especially shells and C.
- ☞ It is very easy to write short programs that pack a lot of punch.

Similarities to C

- ☞ Many operators
- ☞ Many control structures
- ☞ Supports formatted i/o
- ☞ Can access command line arguments
- ☞ Supports access to i/o streams, including stdin, stdout, and stderr.

Similarities to shell programming

- ☞ Comment syntax of #
- ☞ $variables
- ☞ Interpolation of variables inside of quoting.
- ☞ Support command line arguments.
- ☞ Implicit conversion between strings and numbers.

- ☞ Support for regular expressions.
- ☞ Some control structures.
- ☞ Many specific operators similar to shell commands and Unix command syntax.

Scalars

Scalars represent a single value:

```
my $var1 = ''some string'';

my $var2 = 23;
```

Scalars are strings, integers, or floating point numbers.

There are also "magic" scalars which appear in Perl code. The most common one is $_, which means the “default” variable, such as when you just do a `print` with no argument, or are looping over the contents of a list. The “current” item would be referred to by `$_`.

Numbers

Both integers and floating point numbers are actually stored as double precision values —unless you invoke the “use integer” pragma:

```
#!/usr/bin/perl -w
# Script19.pl
# 2006-09-18 - rdl. Illustrate use of "use integer"
use strict;
use integer;
my $w = 100;
my $x = 3;
print "w / x = " . $w/$x . "\n";
[langley@sophie 2006-Fall]$ ./Script19.pl
w / x = 33
```

Floating point literals

- ☞ Floating point literals are similar to those of C.
- ☞ All three of these literals represent the same value:

```
12345.6789
123456789e-4
123.456789E2
```

Integer decimal literals

☞ Similar to C:

```
0   -99   1001
```

☞ Can use underscore as visual separator:

```
2_333_444_555_666
```

Other integral literals

- Hexadecimal:

```
0xff12    0x991b
```

- Octal:

```
0125    07611
```

- Binary:

```
0b101011
```

C-like operators

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>=</td><td>Assignment</td></tr>
<tr><td>+ - * / %</td><td>Arithmetic</td></tr>
<tr><td>& | << >></td><td>Bitwise</td></tr>
<tr><td>< > ≤ ≥</td><td>Relational</td></tr>
<tr><td>&& || !</td><td>Logical</td></tr>
<tr><td>+= -= *=</td><td>Binary assignment</td></tr>
<tr><td>++ –</td><td>Increment/Decrement</td></tr>
<tr><td>? :</td><td>Ternary</td></tr>
<tr><td>,</td><td></td></tr>
</table>

Operators not similar to C operators

Operator	Meaning
*	Exponentiation
¡=¿	Numeric comparison
x	String repetition
.	String concatenation
eq ne lt gt ge le	String relations
cmp	String comparison
=¿	Like comma but forces first left word to be string

Strings

Strings are a base type in Perl.

Strings can be either quoted to allow interpolation (both metacharacters and variables), or quoted so as not to be. Double quotes will allow this, single quotes prevent interpolation.

Single quoted strings using ’

Single quoted strings are not subject to most interpolation.

However, there are two to be aware of: (1) Use \’ to indicate a literal single quote inside of a single quoted string that was defined with ’. (You can avoid this by using the q// syntax.) (2) Use \\ to insert a backslash; other \SOMECHAR are not interpolated inside of single quoted strings.

Double quoted strings

You can specify special characters in double quoted strings easily:

```
print "this is an end of line\n";
print "there are \t tabs \t embedded \t here \n";

print "embedding double quotes \" are easy \n";

print "that costs \$1000 \n";
```

```
print "the variable \$variable ";
```

String operators

☞ The period "." is used to indicate string concatenation.

☞ The "x" operator is used to indicate string repetition:

``abc '' x 4 → ``abc abc abc abc ''

Implicit conversions atwixt numbers and strings

Perl will silently convert numbers and strings where appropriate.

For instance:

```
"5" x "10" → "5555555555"

"2" + "2" → 4

"2 + 2" .  4 → "2 + 24"
```

Scalars

- Ordinary scalar variables begin with $
- They correspond to the regular expression $[a-zA-Z][a-zA-Z0-9_]*
- Scalar can hold integers, strings, or floating point numbers.

Declaring scalars

☞ I recommend you use the pragma `use strict;` – and if you do so, then you will have to explicitly declare all of your variables before using them. Use `my` to declare your variables.

☞ You can declare and initialize one or more variables with `my`:

```
my $a;
my ($a,$b);
my $a = ‘‘value’’;
my ($a,$b) = (‘‘a’’, ‘‘b’’);
```

- ☞ Variable declarations can occur almost anywhere

Variable interpolation

You can use the special form `${variablename}` when you are trying to have a variable name interpreted when it is surrounded by non-whitespace:

```
[langley@sophie 2006-Fall]$ perl
$a = 12;
print "abc${a}abc\n";
abc12abc
```

Undef value

A variable has the special value `undef` when it is first created (it can also be set with the special function `under()` and can be tested with the special function `defined()`).

An `undef` variable is treated as zero if it is used numerically.

An `undef` variable is treated as an empty string if it is used as a string value.

The `print` operator

- ☞ The `print` operator can print a list of expressions, such as strings, variables, or a combination of operands and operators.
- ☞ By default, it prints to stdout.
- ☞ The general form is `print [expression [, expression]*];`

The line input operator <STDIN>

☞ You can use <STDIN>to read a single of input:

```
$a = <STDIN>
```

☞ You can test for end of input with `defined($a)`.

The `chomp` function

You can remove the newline from a string with `chomp`:

```
$line = <STDIN>;
chomp($line);

chomp($line = <STDIN>);
```

The `chomp` function

```
[langley@sophie 2006-Fall]$ perl
chomp($line = <STDIN>);
print $line;
abcdefghijik
abcdefghijik[langley@sophie 2006-Fall]$
```

String relational operators

The string relational operators are `eq`, `ne`, `gt`, `lt`, `ge`, and `le`.

Examples:

```
100 lt 2
"x" le "y"
```

String length

You can use the `length` function to give the number of characters in a string.

Scalar values “typecast” to boolean values

Many of Perl’s control structures look for a boolean value. Perl doesn’t have an explicit “boolean” type, so instead we use the following “typecasting” rules for scalar values:

☞ If a scalar is a number, then 0 is treated as false, and any other value is treated as true.

☞ If a scalar is a string, then “0” and the empty string are treated as false, and any other value as true.

☞ If a scalar is not defined, it is treated as false.

If elsif else

Note that both `elsif` and `else` are optional, but curly brackets are never optional, even if the block contains one statement.

```
if(COND)
  {
  }
[elsif
  {
  }]*
[else
  {
  }]
```

`if-elsif-else` examples

`if` example:

```
if($answer == 12)
{
  print "Right -- one year has twelve months!\n";
}
```

`if-elsif-else` examples

`if/else` example:

```
if($answer == 12)
{
  print "Right -- one year has twelve months!\n";
}
else
{
  print "No, one year has twelve months!\n";
}
```

`if-elsif-else` examples

`if-elsif-else` example:

```
if($answer < 12)
{
  print "Need more months!\n";
}
elsif($answer > 12)
{
  print "Too many months!\n";
}
else
{
  print "Right -- one year has twelve months!\n";
}
```

`if-elsif-else` examples

`if-elsif-elsif` example:

```
if($a eq "struct")
{
}
elsif($a eq "const")
{
}
elsif($a ne "virtual")
{
}
```

`defined()` function

You can test to see if a variable has a defined value with `defined()`:

```
if(!defined($a))
{
  print "Use of undefined value is not wise!";
}
```

The `while` construction

```
while(<boolean>)
{
  <statement list>
}
```

As with `if-elsif-else`, the curly brackets are not optional.

`while` examples

```
while(<STDIN>)
{
  print;
}
```

[You might note that we are using the implicit variable $_ in this code fragment.]

`until` control structure

```
until(<boolean>)
{
  <statement list>
}
```

The `until` construction is the opposite of the `while` construction since it executes the `<statement list>` until the `<boolean>` test becomes true.

`until` example

```
#!/usr/bin/perl -w
# 2006 09 20 -- rdl script22.pl
use strict;
my $line;
until(! ($line=<STDIN>))
{
  print $line;
}
```

`for` control structure

```
for(<init>; <boolean test>; <increment>)
{
  <statement list>
}
```

Very similar to the C construction. The curly brackets again are not optional.

`for` example

```
for($i = 0; $i<10; $i++)
{
  print "\$i * \$i = " . $i*$i . "\n";
}
```

Lists and Arrays

- ☞ A list in Perl is an ordered collection of scalars.
- ☞ An array in Perl is a variable that contains an ordered collection of scalars.

List literals

- ☞ Can represent a list of scalar values
- ☞ General form:

```
( <scalar1>, <scalar2>, ...  )
```

List literals

☞ Examples:

```
(0, 1, 5)       # a list of three scalars that are numbers
('abc', 'def') # a list of two scalars that are strings
(1, 'abc', 3)  # can mix values
($a, $b)        # can have values determined at runtime
()              # empty list
```

Using qw syntax

You can also use the “quoted words” syntax to specify list literals:

```
('apples', 'oranges', 'bananas')
qw/ apples oranges bananas /
qw! apples oranges bananas !
qw( apples oranges bananas )
qw< apples oranges bananas >
```

List literals, cont'd

☞ You can use the range operator ".." to create list elements.

☞ Examples:

```
(0..5)         #
(0.1 .. 5.1) # same since truncated (not {\tt floor()}!)
(5..0)         # evals to empty list
(1,0..5,'x' x 10) # can use with other types...
($m..$n)       # can use runtime limits
```

Array variables

☞ Arrays are declared with the “@” character.

```
my @a;
my @a = ('a', 'b', 'c');
```

☞ Notice that you don’t have to declare an array’s size.

Arrays and scalars

- ☞ Arrays and scalars are in separate name spaces, so you can have two different variables $a and @a.
- ☞ Mnemonically, “$” does look like “S”, and “a” does resemble “@”.

Accessing array elements

- ☞ Accessing array elements in Perl is syntactically similar to C.
- ☞ Perhaps somewhat counterintuitively, you use `$a[<num>]` to specify a scalar element of an array named `@a`.
- ☞ The index `<num>` is evaluated as a numeric expression.

☞ By default, the first index in an array is 0.

Examples of arracy access

```
$a[0] = 1;          # assign numeric constant
$a[1] = "string";   # assign string constant
print $m[$a];       # access via variable
$a[$c] = $b[$d];    # copy elements
$a[$i] = $b[$i];    #
$a[$i+$j] = 0;      # expressions are okay
$a[$i]++;           # increment element
```

Assign list literals

You can assign a list literal to an array or to a list of scalars:

```
($a, $b, $c) = (1, 2, 3);          # $a = 1, $b = 2, $c = 3
($m, $n) = ($n, $m);               # works!
@nums = (1..10);                   # $nums[0]=1, $nums[1]=2, ...
($x,$y,$z) = (1,2)                 # $x=1, $y=2, $z is undef
@t = ();                           # t is defined with no elements
($a[1],$a[0])=($a[0],$a[1]);       # swap works!
@kudomono = ('apple','orange');    # list with 2 elements
@kudomono = qw/ apple orange /;    # ditto
```

Array-wide access

Sometimes you can do an operation on an entire array. Use the `@array` name:

```
@x = @y;            # copy array y to x
@y = 1..1000;       # parentheses are not requisite
@lines = <STDIN>    # very useful!
print @lines;       # works in Perl 5, not 4
```

Printing entire arrays

☞ If an array is simply printed, it comes out something like

```
@a = ('a','b','c','d');
print @a;
abcd
```

☞ If an array is interpolated in a string, you get spaces: @a = ('a','b','c','d'); print "@a"; a b c d

Arrays in a scalar context

Generally, if you specify an array in a scalar context, the value returned is the number of elements in the array.

```
@array1 = ('a', 3, 'b', 4, 'c', 5);  # assign array1 the values of list
@array2 = @array1;                   # assign array2 the values of array1
$m = @array2;                        # $m now has value 6
$n = $m + @array1                    # $n now has value 12
```

Using a scalar in an array context

If you assign an array a scalar value, that array will be just a one element array:

```
$m = 1;
@arr = $m;          # @arr == ( 1 );
@yup = "apple";     # @yup == ( "apple" );
@arr = ( undef );   # @arr == ( undef );
@arr = ();          # @arr is now empty, not an array with one undef value!
```

Size of arrays

Perl arrays can be any size up to the amount of memory available for the process. The number of elements can vary during execution.

```
my @fruit;                # has zero elements
$fruit[0] = "apple";      # now has one element
$fruit[1] = "orange";     # now has two elements
$fruit[99] = 'plum';      # now has 100 elements, most of which are undef
```

Last element index

Perl has a special scalar form `$#arrayname` that returns a scalar value that is equal to the index of the last element in the array.

```
for($i = 0; $i<=$#arr1; $i++)
{
  print "$arr1[$i]\n";
}
```

Last element index use

You can also use this special scalar form to truncate an array:

```
@arr = (1..100);    # arr has 100 elements...
$#arr = 9;          # now it has 10
print "@arr";
1 2 3 4 5 6 7 8 9 10
```

Using negative array indices

A negative array index is treated as being relative to the end of the array:

```
@arr = 1..100;
print $arr[-1];    # similar to using $arr[$#arr]
100
print $arr[-2];
99
```

Arrays as stacks

- ☞ Arrays can be used as stacks, and Perl has built-ins that are useful for manipulating arrays as stacks: `push`, `pop`, `shift`, and `unshift`.
- ☞ `push` takes two arguments: an array to push onto, and what is to pushed on. If the new element is an array, then the elements of that array are appended to the original array as scalars.
- ☞ A `push` puts the new element(s) at the end of the

original array.

☞ A pop removes the last element from the array specified.

Examples of push and pop

```
push @nums, $i;
push @ans, "yes";
push @a, 1..5;
push @a, @b;         # appends the elements of b to a
push @a, (1, 3, 5);
pop @a;
push(@a,pop(@b));    # moves the last element of b to end of a
@a = (); @b = (); push(@b,pop(@a))  # b now has one undef value
```

`shift` and `unshift`

- ☞ `shift` removes the first element from an array
- ☞ `unshift` inserts an element at the beginning of an array

Examples of `shift` and `unshift`

```
@a = 1..10;
unshift @a,99;       # now @a == (99,1,2,3,4,5,6,7,8,9)
unshift @a,('a','b') # now @a == ('a','b',99,1,2,3,4,5,6,7,8,9)
$x = shift @a;       # now $x == 'a'
```

`foreach` control structure

You can use `foreach` to process each element of an array or list.

It follows the form:

```
foreach $SCAlAR (@ARRAY or LIST)
{
  <statement list>
}
```

(You can also `map` for similar processing.)

`foreach` examples

```
foreach $a (@a)
{
  print "$a\n";
}
map {print "$_\n";} @a;

foreach $item (qw/ apple pear lemon /)
{
  push @fruits,$item;
}
map {push @fruits, $_} qw/ apple pear lemon/;
```

The default variable `$_`

`$_` is the default variable (and is used in the previous `map()` examples). It is used as a default when at various times, such as when reading input, writing output, and in the `foreach` and `map` constructions.

The default variable $_

```
while(<STDIN>)
{
  print;
}

$sum = 0;
foreach(@arr)
{
  $sum += $_;
}

map { $sum += $_} @arr;
```

Input from the “diamond” operator

Reading from $<>$ causes a program to read from the files specified on the command line or stdin if no files are specified.

Example of diamond operator

```
#!/usr/bin/perl -w
# 2006 09 22 - rdl script23.pl
while(<>)
{
    print;
}
```

You can either use `./Script23.pl < /etc/hosts` or `./Script23.pl /etc/hosts /etc/resolv.conf`

The `@ARGV` array

There is a builtin array called `@ARGV` which contains the command lines arguments passed in by the calling program.

Note that `$ARGV[0]` is the first argument, not the name of the Perl program being invoked

Using @ARGV

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script24.pl
# do the equivalent of a shell’s echo:
use strict;
my $a;
while($a = shift @ARGV)
{
  print "$a ";
}
print "\n";
```

Using @ARGV

```
#!/usr/bin/perl -w
# 2005 09 25 - rdl Script25.pl
# count the number of arguments
use strict;
my $count = 0;
map { $count++ } @ARGV;
print "$count\n";
```

Loop control operators

Perl has three interesting operators to affect looping: `next`, `last`, and `redo`.

☞ `next` → start the next iteration of a loop immediately

☞ `last` → terminate the loop immediately

☞ `redo` → restart this iteration (very rare in practice)

The `next` operator

The `next` operator starts the next iteration of a loop immediately, much as `continue` does in C.

The `next` operator

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script26.pl
# sum the positive elements of an array to demonstrate next
use strict;
my $sum = 0;
my @arr1 = -10..10;
foreach(@arr1)
{
    if($_ < 0)
    {
        next;
    }
    $sum += $_;
}
print $sum;
```

The `last` operator

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script27.pl
# read up to 100 items, print their sum
use strict;
my $sum = 0;
my $count = 0;
while(<STDIN>)
{
    $sum += $_;
    $count++;
    if($count == 100)
    {
        last;
    }
}
print "\$count == $count, \$sum == $sum \n";
```

The `redo` operator

The rarely used `redo` operator goes back to the beginning a loop block, but it does not do any retest of boolean conditions, it does not execute any increment-type code, and it does not change any positions within arrays or lists.

The `redo` operator

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script29.pl
# demonstrate the redo operator
use strict;
my @strings = qw/ apple plum pear peach strawberry /;
my $answer;
foreach(@strings)
{
    print "Do you wish to print '$_'? ";
    chomp($answer = uc(<>));
    if($answer eq "YES")
    {
        print "PRINTING $_ ...\n";
        next;
    }
```

```
    if($answer ne "NO")
    {
        print "I don’t understand your answer ’$answer’! Please use eithe
        redo;
    }
}
```

The `reverse` function

If used to return a list, then it reverses the input list.

If used to return a scalar, then it first concatenates the elements of the input list and then reverses all of the characters in that string.

Also, you can `reverse` a hash, by which the returned hash has the keys and values swapped from the original hash. (Duplicate `value` $\longrightarrow$ `key` in the original hash are chosen randomly for the new `key` $\longrightarrow$

`value`.)

Examples of `reverse`

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script30.pl
# demonstrate the reverse function
use strict;
my @strings = qw/ apple plum pear peach strawberry /;
print "\@strings = @strings\n";
my @reverse_list = reverse(@strings);
my $reverse_string = reverse(@strings);
print "\@reverse_list = @reverse_list\n";
print "\$reverse_string = $reverse_string\n";
```

Example of `reverse` for hash

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script31.pl
# demonstrate the reverse operator
use strict;
my %strings = ( ’a-key’ , ’a-value’, ’b-key’, ’b-value’, ’c-key’, ’c-valu
print "\%strings = ";
map {print " ( \$key = $_ , \$value = $strings{$_} ) "} (sort keys %strin
print " \n";
my %reverse_hash = reverse(%strings);
print "\%reverse_hash = ";
map {print " ( \$key = $_ , \$value = $reverse_hash{$_} ) "} (sort keys %
print " \n ";
```

Example of `reverse` for hash with duplicate

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script33.pl
# demonstrate the reverse operator for hash with duplicate values
use strict;
my %strings = ( ’a-key’ , ’x-value’, ’b-key’, ’x-value’, ’c-key’, ’x-valu
print "\%strings = ";
map {print " ( \$key = $_ , \$value = $strings{$_} ) "} (sort keys %strin
print " \n";
my %reverse_hash = reverse(%strings);
print "\%reverse_hash = ";
map {print " ( \$key = $_ , \$value = $reverse_hash{$_} ) "} (sort keys %
print " \n ";
```

Examples of `reverse`

```
#!/usr/bin/perl -w
# 2006 09 25 - rdl Script32.pl
# demonstrate the reverse operator
use strict;
my $test = reverse(qw/ 10 11 12 /);
print "\$test = $test\n";
```

The `sort` function

The `sort` function is only defined to work on lists, and will only return sensible items in a list context. **By default, `sort` sorts lexically.**

The `sort` function

```
  # Example of lexical sorting
@list = 1..100;
@list = sort @list;
print "@list ";
1 10 100 11 12 13 14 15 16 17 18 19 2 20
21 22 23 24 25 26 27 28 29 3 30 31 32 33
34 35 36 37 38 39 4 40 41 42 43 44 45 46
47 48 49 5 50 51 52 53 54 55 56 57 58 59
6 60 61 62 63 64 65 66 67 68 69 7 70 71 72
```

73 74 75 76 77 78 79 8 80 81 82 83 84 85 86 87 88 89 9 90 91 92 93 94 95 96 97 98 99

The `sort` function

You can define an arbitrary sort function. Our earlier mention of the <=> operator comes in handy now:

```
  # Example of numerical sorting
@list = 1..100;
@list = sort { $a <=> $b } @list;
print "@list ";
@list = 1..100;
@list = sort { $a <=> $b } @list;
print "@list";
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73
74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99 100

The `sort` function

The `$a` and `$b` in the function block are actually package global variables, and should not be declared by you as `my` variables.

The `sort` function

You can also use the `cmp` operator quite effectively in these type of anonymous sort functions:

```
@words = qw/ apples Pears bananas Strawberries cantaloupe grapes Blueberr
@words_alpha = sort @words;
@words_noncase = sort { uc($a) cmp uc($b) } @words;
print "\@words_alpha = @words_alpha\n";
print "\@words_noncase = @words_noncase\n";
# yields:
@words_alpha = Blueberries Pears Strawberries apples bananas cantaloupe g
@words_noncase = apples bananas Blueberries cantaloupe grapes Pears Straw
```

Hashes

We have already used a few examples of hashes. Let's go over exactly what is happening with them:

- ☞ A hash is similar to an array in that it has an index and in that it may take an arbitrary number of elements.

- ☞ An index for a hash is a string, not a number as in an array.

☞ Hashes are also known as “associative arrays.”

☞ The elements of a hash have no particular order.

☞ A hash contains key-value pairs; the keys will be unique, and the values are not necessarily so.

Hash declarations

- ☞ Hashes are identified by the % character.
- ☞ The name space for hashes is separate from that of scalar variables and arrays.

Hash element access

☞ One uses the syntax `$hash{$key}` to access the value associated with key `$key` in hash `%hash`.

☞ Perl expects to see a string as the key, and will silently convert scalars to a string, and will convert arrays silently.

Examples

```
$names[12101] = ’James’;
$names[12101] = ’Bob’;     # overwrites value ’James’
$name = $names[12101];     # retrieve value ’Bob’;
$name = $names[11111];     # undefined value returns undef

%hash = (’1’, ’1-value’, ’a’, ’a-value’, ’b’, ’b-value’);
@array = (’a’);
print $hash{@array};
# yields
1-value
```

Examples

```
%names = (1, 'Bob', 2, 'James');
foreach(sort(keys(%names)))
{
  print "$_ --> $names{$_}\n";
}
# yields
1 --> Bob
2 --> James

map { print "$_ --> $names{$_}\n"; } sort(keys(%names));
# yields
1 --> Bob
2 --> James
```

Referring to a hash as a whole

As might have been gleaned from before, you can use the % character to refer a hash as a whole.

```
%new_hash = %old_hash;
%fruit_colors = ( 'apple' , 'red' , 'banana' , 'yellow' );
%fruit_colors = ( 'apple' => 'red' , 'banana' => 'yellow' );

print "%fruit_colors\n";      # only prints '%fruit_colors', not keys or
@fruit_colors = %fruit_colors;
print "@fruit_colors\n";      # now you get output...
# yields
banana yellow apple red
```

The `keys` and `values` functions

You can extract just the hash keys into an array with the `keys` function.

You can extract just the hash values into an array with the `values` function.

```
%fruit_colors = ( ’apple’ => ’red’ , ’banana’ => ’yellow’ );
@keys = keys(%fruit_colors);
@values = values(%fruit_colors);
print "\@keys = ’@keys’ , \@values = ’@values’\n";
# yields
@keys = ’banana apple’ , @values = ’yellow red’
```

The each function

Perl has a “stateful” function each that allows you to iterate through the keys or the key-value pairs of a hash.

```
%fruit_colors = ( ’apple’ => ’red’ , ’banana’ => ’yellow’ );
while( ($key, $value) = each(%fruit_colors) )
{
  print "$key --> $value\n";
}
```

The each function

Note: if you need to reset the iterator referred to by each, you can just make a call to either `keys(%fruit_colors)` or `values(%fruit_colors)` – so don't do that accidentally!

```
%fruit_colors = ( 'apple' => 'red' , 'banana' => 'yellow' );
while( ($key, $value) = each(%fruit_colors) )
{
  print "$key --> $value\n";
  # ...
  @k = keys(%fruit_colors);   # resets iterator!!!
}
# yields loop!
```

```
banana --> yellow
banana --> yellow
banana --> yellow
banana --> yellow
banana --> yellow
 ....
```

The `exists` function

You can check if a key exists in hash with the `exists` function:

```
if(exists($hash{'SOMEVALUE'}))
{
}
```

The `delete` function

You can remove a key-value pair from a hash with `delete`:

```
delete($hash{'SOMEVALUE'});
```

`printf`

- ☞ `printf` in Perl is very similar to that of C.

- ☞ `printf` is most useful when when printing scalars. Its first (non-filehandle) argument is the format string, and any other arguments are treated as a list of scalars:

```
printf "%s %s %s %s", ("abc", "def") , ("ghi", "jkl");
# yields
abc def ghi jkl
```

printf

- Some of the common format attributes are
  - `%[-][N]s` $\longrightarrow$ format a string scalar, N indicates maximum characters expected for justification, - indicates to left-justify rather than default right-justify.
  - `%[-|0][N]d` $\longrightarrow$ format a numerical scalar as integer, N indicates maximum expected for justification “-” indicates to left-justify, “0” indicates zero-fill (using both “-” and “0” results in left-justify, no zero-fill.)
  - `%[-|0]N.Mf` $\longrightarrow$ format a numerical scalar as

floating point. “N” gives the total length of the output, and “M” give places after the decimal. After the decimal is usually zero-filled out (you can toggle this off by putting “0” before “M”.) “0” before N will zero-fill the left-hand side; “-” will left-justify the expression.

Examples of `printf()`

```
printf "%7d\n", 123;
# yields
    123

printf "%10s %-10s\n","abc","def";
# yields
       abc def
```

Examples of `printf()`

```
printf "%10.5f %010.5f %-10.5f\n",12.1,12.1,12.1;
# yields
  12.10000 0012.10000 12.10000

$a = 10;
printf "%0${a}d\n", $a;
# yields
0000000010
```

Perl regular expressions

- Much information can be found at `man perlre`.

- Perl builds support for regular expressions as a part of the language like awk but to a greater degree. Most languages instead simply give access to a library of regular expressions (C, PHP, Javascript, and C++, for instance, all go this route.)

- Perl regular expressions can be used in conditionals, where if you find a match then it evaluates to true, and if no match, false.

```
$_ = "howdy and hello are common";
if(/hello/)
{
  print "Hello was found!\n";
}
else
{
  print "Hello was NOT found\n";
}
# yields
Hello was found!
```

What do Perl patterns consist of?

- ☞ Literal characters to be matched directly
- ☞ "." (period, full stop) matches any one character (except newline unless coerced to do so)
- ☞ "*" (asterisk) matches the preceding item zero or more times
- ☞ "+" (plus) matches the preceding item one or more times

- ☞ “?” (question mark) matches the preceding item zero or one time

- ☞ “(” and “)” (parentheses) are used for grouping

- ☞ “|” (pipe) expresses alternation

- ☞ “[” and “]” (square brackets) express a range, match one character in that range

Examples of Perl patterns

`/abc/`	Matches “abc”
`/a.c/`	Matches “a” followed by any character (except newline) and then a
`/ab?c/`	Matches “ac” or “abc”
`/ab*c/`	Matches “a” followed by zero or more “b” and then a “c”
`/ab|cd/`	Matches “abd” or “acd”
`/a(b|c)+d`	Matches “a” followed by one or more “b” or “c”, and then a “d”
`/a[bcd]e/`	Matches “abe”, “ace”, or “ade”
`/a[a-zA-Z0-9]c/`	Matches “a” followed one alphanumeric followed by “c”
`/a[^a-zA-Z]/`	Matches “a” followed by anything other than alphabetic character

Character class shortcuts

You can use the following as shortcuts to represent character classes:

`\d`	A digit (i.e., `0-9`)
`\w`	A word character (i.e., `[0-9a-zA-Z_]`)
`\s`	A whitespace character (i.e., `[\f\t\n ]`)
`\D`	Not a digit (i.e., `[^0-9]`)
`\W`	Not a word (i.e., `[^0-9a-zA-Z_]`)
`\S`	Not whitespace

General quantification

You can specify numbers of repetitions using a curly bracket syntax:

```
a{1,3}          # ‘‘a’’, ‘‘aa’’, or ‘‘aaa’’
a{2}            # ‘‘aa’’
a{2,}           # two or more ‘‘a’’
```

Anchors

Perl regular expression syntax lets you work with context by defining a number of “anchors”: \A, ^, \Z, $, \b.

/\ba/	Matches if “a” appears at the beginning of a word
/\Aa$/	Matches if “a” appears at the end of a line
/\Aa\Z/	Matches if a line is exactly “a”
/^Aa$/	Matches if a line is exactly “a”

Remembering substring matches

- ☞ Parentheses are also used to remember substring matches.
- ☞ Backreferences can be used within the pattern to refer to already matched bits.
- ☞ Memory variables can be used after the pattern has been matched against.

Backreferences

- ☞ A backreference looks like \1, \2, etc.
- ☞ It refers to an already matched memory reference.
- ☞ Count the left parentheses to determine the back reference number.

Backreference examples

```
/(a|b)\1/          # match ‘‘aa’’ or ‘‘bb’’
/((a|b)c)\1/       # match ‘‘acac’’ or ‘‘bcbc’’
/((a|b)c)\2/       # match ‘‘aba’’ or ‘‘bcb’’
/(.)\1/            # match any doubled characters except newline
/\b(\w+)\s+\b\1\s/ # match any doubled words
/(['"])(.*)\1/     # match strings enclosed by single or double quotes
```

Remember, perl matching is by default greedy

For example, consider the last backreference example:

```
$_ = "asfasdf 'asdlfkjasdf ' werklwerj'";
if(/(['"])(.*)\1/)
{
  print "matches $2\n";
}
# yields
matches asdlfkjasdf ' werklwerj
```

Memory variables

- ☞ A memory variable has the form $1, $2, etc.
- ☞ It indicates a match from a grouping operator, just as back reference does, but after the regular expression has been executed.

```
$_ = "  the larder  ";
if(/\s+(\w+)\s+/)
{
   print "match = '$1'\n";
}
# yields
match = 'the'
```

Regular expression “binding” operators

Up to this point, we have considered only operations against $_.

Any scalar can be tested against with the =~ and !~ operators.

```
"STRING" =~ /PATTERN/;

"STRING" !~ /PATTERN/;
```

Examples

```
$line = "not an exit line";
if($line !~ /^exit$/)
{
   print "$line\n";
}
# yields
not an exit line

# skip over blank lines...
if($line =~ /$^/)
{
  next;
}\
```

Automatic match variables

You don’t have to necessarily use explicit backreferences and memory variables. Perl also gives you three default variables that can be used after the application of any regular expression; they refer to the portion of the string matched by the whole regular expression.

```
$‘      refers to the portion of the string before the match
$&      refers to the match itself
$’      refers to the portion of the string after the match
```

Example of automatic match variables

```
$_ = "this is a test";
/is/;
print "before: < $‘ >  \n";
print "after:  < $’ >  \n";
print "match:  < $& >  \n";
# yields
before: < th >
after:  <  is a test >
match:  < is >
```

Example of automatic match variables

```
#!/usr/bin/perl -w
# 2006 09 27 - rdl Script34.pl // change = to =:
use strict;
while(<>)
{
  /=/;
  print "$‘=:$’\n";
}
```

Other delimiters: Using the “m”

You can use other delimiters (some are paired items) rather than just a slash, but you must use the “m” to indicate this. (See `man perlop` for a good discussion.)

For instance:

```
m/.../  m{...}  m[...] m(...)
m!...!  m,...,  m^...^  m#...#
```

Example

```
# not so readable way to look for a URL reference
if ($s =˜ /http:\/\/\//)

# better
if ($s =˜ mˆhttp://ˆ )
```

Option modifiers

There are a number of modifiers that you can apply to your regular expression pattern:

Modifier	Description
i	case insensitive
s	treat string as a single line
g	find all occurrences

Regular expressions and case insensitivity

As previously mentioned, you can make matching case insensitive with the `i` flag:

```
/\b[Uu][Nn][Ii][Xx]\b/;        # explicitly giving case folding

/\bunix\b/i;                   # using ‘‘i’’ flag to fold code
```

Really matching any character with "."

As mentioned before, usually the "." (dot, period, full stop) matches any character except newline. You make it match newline with the s flag:

```
/"(.|\n)*"/;                    # match any quoted string, even with newline
/"(.*)"/s;                      # same meaning, using ‘‘s’’ flag
```

N.B. – I like to use the flags `///six;` as a personal default set of flags with Perl regular expressions.

Going global with the ''g'' flag

You can make your matching global with the g flag. For ordinary matches, this means making them stateful: Perl will remember where you left off with each reinvocation of the match unless you change the value of the variable, which will reset the match.

Going global with the ‘‘g’’ flag

```
#!/usr/bin/perl -w
# 2006 09 29 - rdl Script36.pl
# shows the //g as stateful...
while(<>)
{
    while(/[A-Z]{2,}/g)
    {
        print "$&\n" if (defined($&));
    }
}
```

Interpolating variables in patterns

You can even specify a variable inside of a pattern – but you want to make sure that it gives a legitimate regular expression.

Interpolating variables in patterns

```
my $var1 = "[A-Z]*";
if( "AB" =~ /$var1/ )
{
  print "$&";
}
else
{
  print "nopers";
}
# yields
AB
```

Regular expressions and substitution

- ☞ The s/.../.../ form can be used to make substitutions in the specified string.
- ☞ If paired delimiters are used, then you have to use two pairs of the delimiters.
- ☞ g after the last delimiter indicates to replace more than just the first occurrence.
- ☞ The substitution can be bound to a string using =~ .

Otherwise it makes the substitutions in $_.

☞ The operation returns the number of replacements performed, which can be more than one with the ’g’ option.

Examples

```
#!/usr/bin/perl -w
# 2006 09 29 - rdl Script37.pl
# shows s///g... by removing acronyms
use strict;
while(<>)
{

    s/([A-Z]{2,})//g;
    print;
}
```

Examples

```
s/\bfigure (\d+)/Figure $1/  # capitalize references to figures
s{//(.*)}{/\*$1\*/}          # use old style C comments
s!\bif(!if (!                # put a blank between  if and  (
s(!)(.)                      # tone down that message
s[!][.]g                     # replace all occurrences of '!' with '.'
```

Case shifting

You can use `\U` and `\L` to change follows them to upper and lower case:

Case shifting

```
$text = " the acm and the ieee are the best! ";
$text =~ s/acm|ieee/\U$&/g;
print "$text\n";
# yields
 the ACM and the IEEE are the best!
```

Case shifting

```
$text = "CDA 1001 and COP 3101

are good classes, but COP 4342 is better!";
$text =~ s/\b(COP|CDA) \d+/\L$&/g;
print "$text\n";
# yields
cda 1001 and cop 3101

are good classes, but cop 4342 is better!
```

Using `tr///` (also known as `y///`)

☞ In Perl you can also convert one set of characters to another using the `tr/.../.../` form. (Or if you like, you can use `y///`.)

☞ Much like the program `tr`, you specify two lists of characters, the first to be substituted, and the second what to substitute.

- ☞ `tr` returns the number of items substituted (or deleted.)

- ☞ The modifer d deletes characters not replaced.

- ☞ The modifer s “squashes” any repeated characters.

Examples (from the `perlop` man page)

```
$ARGV[1] =~ tr/A-Z/a-z/;    # canonicalize to lower case
$cnt = tr/*/*/;             # count the stars in $_
$cnt = $sky =~ tr/*/*/;     # count the stars in $sky
$cnt = tr/0-9//;            # count the digits in $_
```

More examples

```
# get rid of redundant blanks in $_
tr/ //s;

# replace [ and { with ( in $text
$text =~ tr/[{/(/;
```

Using `split`

The `split` function breaks up a string according to a specified separator pattern and generates a list of the substrings.

Using `substring`

For example:

```
$line = " This sentence contains five words. ";
@fields = split / /, $line;
map {  print "$count --> $fields[$count]\n"; $count++; } @fields;
# yields
 -->
1 --> This
2 --> sentence
3 --> contains
4 --> five
5 --> words.
```

Using the `join` function

The `join` function does the reverse of the `split` function: it takes a list and converts to a string.

However, it is different in that it doesn't take a pattern as its first argument, it just takes a string:

```
@fields = qw/ apples pears cantaloupes cherries /;
$line = join "<-->", @fields;
print "$line\n";
# yields
apples<-->pears<-->cantaloupes<-->cherries
```

Filehandles

[Also see `man perlfaq5` for more detail on this subject.]

A filehandle is an I/O connection between your process and some device or file. Perl output is buffered.

Perl has three predefined filehandles: `STDIN`, `STDOUT`, and `STDERR`.

Filehandles

Unlike other variables, you don’t declare filehandles. The convention is to use all uppercase letters for filehandle names. (Especially important if you deal with anonymous filehandles!)

The open operator takes two arguments, a filehandle name and a connection (e.g. filename). The connection can start with ”< , > , or ”>> to indicate read, write, and append access.

Examples

```
open IN,  in.dat ;     # open  in.dat for input
open IN2,  <$file ;    # open filename in $file for input
open OUT,  >out.dat ;  # open  out.dat for output
open LOG,  >>log.txt ; # open  log.txt to append output
```

Closing filehandles

The close operator closes a filehandle. This causes any remaining output data associated with this filehandle to be flushed to the file.

Perl automatically closes filehandles at the end of a process, or if you reopen it.

Examples

```
close IN;  # closes the IN filehandle
close OUT; # closes the OUT filehandle
close LOG; # closes the LOG filehandle
```

Testing open

You can check the status of opening a file by examining the result of the open operation. It returns a true value if it succeeded, and a false one if it failed.

```
if (!open OUT,  >out.dat ) {
      die  Could not open out.dat. ;
}
```

Using a filehandle

```
Open IN,  <in.dat ;
Open OUT,  >out.dat ;
$i = 1;
while ($line = <IN>) {
     printf OUT  %d: $line , $i;
}
```

Note that a comma is not used after the filehandle in a `print` or `printf` statement.

Reopening a filehandle

You can reopen a standard filename. This allows you to perform input or output in a normal fashion, but to redirect the I/O from/to a file within the Perl program.

Examples of reopening a filehandle

```
# redirect standard output to out.txt
open STDOUT, >out.txt ;
printf Hello world!\n ;
# redirect standard error to append to log.txt
open STDERR, >>log.txt ;
```

File testing

Like BASH, file tests exist in Perl (source: `man perlfunc`):

```
-r  File is readable by effective uid/gid.
-w  File is writable by effective uid/gid.
-x  File is executable by effective uid/gid.
-o  File is owned by effective uid.

-R  File is readable by real uid/gid.
-W  File is writable by real uid/gid.
-X  File is executable by real uid/gid.
-O  File is owned by real uid.
```

```
-e  File exists.
-z  File has zero size (is empty).
-s  File has nonzero size (returns size in bytes).

-f  File is a plain file.
-d  File is a directory.
-l  File is a symbolic link.
-p  File is a named pipe (FIFO), or Filehandle is a pipe.

-S  File is a socket.
-b  File is a block special file.
-c  File is a character special file.
-t  Filehandle is opened to a tty.

-u  File has setuid bit set.
-g  File has setgid bit set.
-k  File has sticky bit set.

-T  File is an ASCII text file (heuristic guess).
```

```
-B  File is a "binary" file (opposite of -T).

-M  Script start time minus file modification time, in days.
-A  Same for access time.
-C  Same for inode change time (Unix, may differ for other platforms)
```

Using file status

You can use file status like this, for instance, as pre-test:

```
while (<>) {
    chomp;
    next unless -f $_;      # ignore specials
    #...
}
```

Using file status

Or you can use them as a post-test:

```
if(! open(FH, $fn))
{
  if(! -e "$fn")
  {
    die "File $fn doesn’t exist.";
  }
  if(! -r "$fn")
  {
    die "File $fn isn’t readable.";
  }
  if(-d "$fn")
  {
    die "$fn is a directory, not a regular file.";
```

```
  }
  die "$fn could not be opened.";
}
```

Subroutines in Perl

You can declare subroutines in Perl with `sub`, and call them with the & syntax:

```
my @list = qw( /etc/hosts /etc/resolv.conf /etc/init.d );
map ( &filecheck ,  @list) ;

sub filecheck
{
    if(-f "$_")
    {
        print "$_ is a regular file\n";
    }
    else
    {
```

```
        print "$_ is not a regular file\n";
    }
}
```

Subroutine arguments

To send arguments to a subroutine, just use a list after the subroutine invocation, just as you do with built-in functions in Perl.

Arguments are received in the @_ array:

```
#!/usr/bin/perl -w
# 2006 10 04 - rdl Script39.pl
# shows subroutine argument lists
use strict;
my $val = max(10,20,30,40,11,99);
print "max = $val\n";
```

```
sub max
{
    print "Using $_[0] as first value...\n";
    my $memory = shift(@_);
    foreach(@_)
    {
        if($_ > $memory)
        {
            $memory = $_;
        }
    }
    return $memory;
}
```

Using `my` variables in subroutines

You can locally define variables for a subroutine with `my`:

```
sub func
{
  my $ct = @_;
  ...;
}
```

The variable `$ct` is defined only within the subroutine `func`.

`sort()` and `map()`

The built-ins functions `sort()` and `map()` can accept a subroutine rather than just an anonymous block:

```
@list = qw/ 1 100 11 10 /;
@default = sort(@list);
@mysort = sort {&mysort} @list;
print "default sort: @default\n";
print "mysort: @mysort\n";
sub mysort
{
  return $a <=> $b;
}
```

```
# yields
default sort: 1 10 100 11
mysort: 1 10 11 100
```

As you can see, `sort()` sends along two special, predefined variables, `$a` and `$b`.

`cmp` and friends

As discussed earlier, <=> returns a result of -1,0,1 if the left hand value is respectively numerically less than, equal to, or greater than the right hand value.

`cmp` returns the same, but uses lexical rather numerical ordering.

grep

A very similar operator is `grep`, which only returns a list of the items that matched an expression (`sort` and `map` should always return a list exactly as long as the input list.)

For example:

```
@out = grep {$_ % 2} qw/1 2 3 4 5 6 7 8 9 10/;
print "@out\n";
# yields
1 3 5 7 9
```

Notice that the block item should return 0 for non-

matching items.

Directory operations

```
chdir $DIRNAME;          # change directory to $DIRNAME

glob $PATTERN;           # return a list of matching patterns
# example:
@list = glob "*.pl";
print "@list \n";
Script16.pl Script18.pl Script19.pl Script20.pl Script21.pl [...]
```

Manipulating files and directories

```
unlink $FN1, $FN2, ...;     # remove a hard or soft link to files

rename $FN1, $FN2;          # rename $FN1 to new name $FN2

mkdir $DN1;                 # create directory with umask default permissi

rmdir $DN1, $DN2, ...;      # remove directories

chmod perms, $FDN1;         # change permissions
```

Traversing a directory with `opendir` and `readdir`

You can pull in the contents of a directory with opendir and readdir:

```
opendir(DH,"/tmp");
@filenams = readdir(DH);
closedir(DH);
print "@filenams\n";
# yields
.s.PGSQL.5432.lock .. mapping-root ssh-WCWcZf4199 xses-langley.joHONt . O
```

Calling other processes

In Perl, you have four convenient ways to call (sub)processes: the backtick function, the `system()` function, `fork()`/`exec()`, and `open()`.

The backtick function is the most convenient one for handling most output from subprocesses. For example

```
@lines = `head -10 /etc/hosts`;
print "@lines\n";
```

You can do this type of output very similarly with `open`, but `open` also allows you do conveniently send input to subprocesses.

`exec()` lets you change the present process to another executable; generally, this is done with a `fork()` to create a new child subprocess first.

The `system()` subroutine is a short-cut way of writing `fork/exec`. Handling input and output, just as with `fork/exec`, is not particularly convenient.

Program development

- emacs (and vi)
- flex and bison
- makefiles
- source level debugging
- diff
- rcs and subversion

- ☞ gprof
- ☞ glade

`emacs`

`emacs` is a superior text-based program development environment over `vi`, and it is easy to install.

Why use `emacs`? The way that `emacs` %

☞ While not "standard", as is `vi`, it is very common and it is generally very easy to install these days.

☞ It is completely programmable. In fact, it takes the idea of programming to a much higher level in that it maps arbitrary sequences of keystrokes to arbitrary functions.

☞ `emacs` lisp is a pleasant programming language. If you like other languages, other versions of `emacs` support: MacLisp, `scheme`, `guile`, Common Lisp, ObjectCaml, even `teco`.

☞ `emacs` has also been called “Eight Megabytes And Continuously Swapping.” Despite that moniker, it is actually reasonably efficient.

The tutorial

Most of the verbatim material here is taken "verbatim" from the Emacs Tutorial. You can use `ctrl-h t` to display this tutorial in `emacs`:

```
The following commands are useful for viewing screenfuls:
        C-v     Move forward one screenful
        M-v     Move backward one screenful
        C-l     Clear screen and redisplay all the text,
                moving the text around the cursor
                to the center of the screen.
```

More of the tutorial

```
                        Previous line, C-p
                                :
                                :
Backward, C-b .... Current cursor position .... Forward, C-f
                                :
                                :
                          Next line, C-n

>> Move the cursor to the line in the middle of that diagram
   using C-n or C-p.  Then type C-l to see the whole diagram
   centered in the screen.
```

A quick summary of most useful “move around” commands

```
C-f     Move forward a character
C-b     Move backward a character
M-f     Move forward a word
M-b     Move backward a word
C-n     Move to next line
C-p     Move to previous line
C-a     Move to beginning of line
C-e     Move to end of line
M-a     Move back to beginning of sentence
M-e     Move forward to end of sentence
M-<     Move to top of the buffer
M->     Move to bottom of the buffer
```

The basic portions of an `emacs` window

The mode line has several parts: the first indicates your coding system (use `c-h C` to find more information about your current one.

It then has some status information: a %% indicate that the buffer is read-only, `**` indicate that the buffer is modified,

The menu bar

If you like menu bars, you can access the one in `emacs` with `m-‘`.

Creating windows

You can split your current window vertically with `c-x 2`.

You can split your current window horizontally with `c-x 3`.

You can jump around windows with `c-o`. You can even scroll another buffer with `c-m-v`

You can `c-x 1` to get rid off all but one window.

Buffer control

You can list your current buffers with `c-x c-b`. You can even use `c-x o` to leap into that buffer and then use the "o" key to go directly to that buffer, or the "k' key to mark the buffer for removal (does not affect the file), and the "x" to do the marked removals.

You can also use `c-x b` to switch buffers.

Finally, `c-x s` will let you save all modified buffers.

Automating `emacs`

You can record simple macros in `emacs` with `c-x (` and `c-x )`.

To play the macro, use `c-x e`

You can give an argument to a function with `c-u NUM`; giving one to a keyboard macro invocation causes that macro to be called that many times.

`vi` UPT 17.1

- ☞ “vi” stands for the VIsual editor.
- ☞ Newest forms such as `vim` and `gvim` are much more featureful than the original barebones editor.
- ☞ It’s “standard” on all Unix machines, and a great way to get `emacs` going!
- ☞ While it doesn’t make automatic backups of files edited, it also doesn’t leave tilde files all over the

place.

☞ It is generally quite efficient.

Calling `vi`

The vi editor is invoked by issuing the command in the following form. The -r option is for recovering a file where the system crashed during a previous editing session. The -t option is to indicate the position within a file the editing should start.

```
vi [-t tag] [-r ] filename
```

Modes in vi

- It has has three main modes:
  - character input mode: where text can be entered
    - insert, append, replace, add lines
  - window mode: where regular commands can be issued
    - basic cursor motions
    - screen control
    - word commands
    - deletions

- control commands
- miscellaneous commands

- line mode: where ex or ed commands can be issued

Character input/output

After invoking `vi`, the user is in the window command mode.

There are a few different commands to enter character input mode.

At that point, a user types in any desired text. The user then uses the ESC key to return back to command mode.

Commands to enter Character Input Mode

```
a    append text after the cursor position
A    append text at the end of line
i    insert text before the cursor position
I    insert text before the first nonblank character in the line
o    add text after the current line
O    add text before the current line (letter O)
rchr replace the current character with ‘‘chr’’
R    replace text starting at the cursor position
```

Basic cursor motion

```
h   go back one character
j   go down one line
k   go up one line
l   go forward one character (space also works)
0    go to the beginning of the line (zero)
$    go to the end of the line
H    go to the top line on the screen
L    go to the last line on the screen
```

Word movement

```
w   position the cursor at the beginning of the next word
b   position the cursor at the beginning of the last word
e   position the cursor at the end of the current word
```

Screen control

```
^U   scroll up one half page
^D   scroll down one half page
^B   scroll up one page
^F   scroll down one page
^L   redisplay the page
```

Deletions

```
dd   delete the current line
D    delete text from the cursor to the end of the line
x    delete character at the cursor
X    delete character preceding the cursor
dw   delete characters from the cursor to the end of the word
```

Searching

`/pattern`	`search forward for "pattern"`
`/`	`search forward for last "pattern"`
`?pattern`	`search backward for "pattern"`
`?`	`search backward for last "pattern"`
`n`	`re-perform the last / or ? command`

Miscellaneous

```
u    undo previous command
U    restore entire line
Y    save current line into buffer
p    put saved buffer after cursor position
P    put saved buffer before cursor position
J    join current line with following line
%    position cursor over matching "(", ")", "{", or "}"
ZZ   save file and exit (same as :wq)
```

Repetition

You can specify how many times a command is to be performed:

```
3dd     delete 3 lines
4w      advance 4 words
7x      delete 7 characters
5n      perform last search 5 times
```

Working with tags

The `ctags` and `etags` programs let you take in a set of source files as input and creates a `tags`/`TAGS` file as output.

The tags file contains for each function and macro

☞ Object name

☞ File in which the object is defined.

☞ Pattern describing the location of the object.

The output of `etags` is also useful with `emacs`.

Using a tags file

You can use the `-t` option when invoking `vi` to find a particular function.

```
vi -t main
vi -t max
```

gvim

There is a graphical version of `vi` called `gvim`.

Multi-level undo in `vim` (not `vi`, though)

Can use the `u` command to undo multiple changes, as opposed to `vi`, which can only undo the last change. Each time you enter `u`, the previous change is undone.

Source level debugging

- ☞ Source level debugging is a nice help when debugging execution problems.
- ☞ To enable source level debugging with gcc/g++, you should use the -g option.

Source level debugging

- The symbol table information includes the correspondence between
  - statements in the source and locations of instructions in the executable
  - variables in the source and locations in the data areas of the executable

GDB: the Gnu debugger

- ☞ GDB is a line oriented debugger where actions are initiated by typing in commands at a prompt.
- ☞ It can be invoked for executables created by gcc and g++.

GDB: the Gnu debugger

- General capabilities
  - Starting and exiting your program from the debugger.
  - Pausing and continuing execution of your program while in the debugger.
  - Examining the state of your program.
  - Changing the state of your program.

Starting and stopping GDB

☞ You can start gdb along these lines

```
gdb YOURPROGRAM [core|pid]
```

☞ If you don't specify a core file or a process id, then you can start a new execution of YOURPROGRAM with the `run` command.

Starting and stopping GDB

- ☞ You can specify whatever arguments you like after `run`, including i/o redirection.

  ```
  run 123 > /tmp/out
  ```

- ☞ You can exit gdb with the `quit` command.

Stopping and continuing execution of your program in gdb

☞ You can set and remove breakpoints.

☞ You can also step through execution, and as well simply continue it.

Setting and removing breakpoints

☞ You can set a breakpoint to stop either when a certain location in the source is reached, or when a condition occurs.

☞ The general form is

```
break [SOMEFUNCTION|SOMELINENUM] [if SOMECONDITIION]
```

☞ Specifying just `break` will set a breakpoint at your current location.

☞ You can remove a breakpoint with

```
delete BREAKPOINT
```

Examples

```
(gdb) break                sets a breakpoint at the current line

(gdb) break 50             sets a breakpoint at line 50 of the current file

(gdb) break main           sets a breakpoint at routine main()

(gdb) break 10 if i == 66     break execution at line 10 if the variable i
                              has the value 10

(gdb) delete 3             remove the 3rd breakpoint

(gdb) delete               deletes all breakpoints
```

Stepping through execution

- You can step to the next statement, or you can step *into* a function.
- The general form is

```
step [N]  # also, "s [N]" is generally defined as "step [N]" for most v
```

where N indicates the number of steps to take, defaulting to 1 if not specified. Execution will not continue through a breakpoint (or program termination.)

Nexting through execution

Often, you don't want to step *into* a function. You can use the `next` command to simply go to the next statement rather than `stepping` into a function specified on the current line.

```
next [N]  # also, "n [N]" is generally defined as the same
```

Continuing execution

You can continue execution up to the next breakpoint found, or program termination.

```
cont [N]  # also, "c [N]" is generally defined as the same
```

N here specifies skip the first N-1 breakpoints.

Continuing execution until the end of a loop

You can use the `until` command to execute your program until it reaches a source line greater than the one that you are currently on. If you are not at a jump back, this is the same as the `next` command. If you are at a back jump such as in a looping construct, then this will let you execute until the point that you have exited the loop.

Examining the state of your program

- ☞ Listing source code.
- ☞ Printing the values of expressions.
- ☞ Displaying the values of expressions.
- ☞ Printing a stack trace.
- ☞ Switching context in a trace.

Listing source code

You can list source code a specified line or function.

The general form is

```
list [[FILENAME:]LINENUM[,LINENUM]]|[[FILENAME:]FUNCTIONNAME]
```

If you don’t specify anything, then you will get 10 lines from the current program location, or 10 more lines if you have already listed the current program location.

Listing source code examples

```
(gdb) list     # list 10 lines from the current location

(gdb) list 72  # list lines 67-76 (the 10 lines around line 72

(gdb) list calc.c:55  # list lines 50-59 of the file calc.c

(gdb) list 80,95  # list lines 80..95 of the current file

(gdb) list somefunc  # list the function somefunc

(gdb) list cal.c:january   # list the january function in cal.c
```

Printing the values of expressions

You can print the value of expressions involving variables based on the state of the execution of the process. You can also specify to some degree the formatting of those expressions, such as asking for hexadecimal or octal values.

```
print[/FMT] EXPRESSION
```

The FMT can be ’o’ for octal, ’x’ for hexadecimal, ’d’ for signed decimal, ’f’ for float, ’u’ for unsigned decimal,

’t’ for binary, and ’a’ for address. If no EXPRESSION is given, the last one is used.

Example `print` commands

```
print i         # prints the value of the variable i
print a[i]      # prints the value of a[i]
print/t a[i]    # prints a[i] in binary
print a[i]-x    # prints the value of a[i] - x
print a         # prints the values in array a
print p         # prints the value of the pointer p
print *p        # prints the value pointed to by p
p i             # prints the value of i
```

Displaying the values of expressions

The `display` command is very similar to the `print` command, but the value is displayed after each `step` or `continue` command.

```
display[/FMT] EXPRESSION
```

Undisplaying expression values

You can use the `undisplay` command to stop displaying expressions.

Printing a stack trace

- ☞ You can print a trace of the activation records of the stack of functions called up until this point.
- ☞ The trace shows the names of the routines called, the values of the arguments passed to each routine, and the line number last executed in that routine.
- ☞ The general form is

```
where [N]
```

If N is positive, then only the last N activation records are shown. If N is negative, then only the first N activation records are shown.

Switching context in the stack

You can up or down in the stack with `up [N]` and `down [N]`.

Changing state in your program execution

You can modify the values of variables while executing in order to avoid making code changes just for the sake of debugging.

For instance,

```
set i = 10      # set the variable i to the value 10
set a[i] = 4    # set a[i] to 4
```

Making impromptu calls to functions

You can call simply invoke a function from the gdb prompt. This can be very useful to call debugging routines that print the values of complex structures that might be difficult to parse with just the gdb `print` command.

```
call FUNCTION(ARGS)
```

Other useful features

One of the most useful things that you can do is to simply run a program that is segfaulting and see where the problem is occurring. Or if you have a core file from a segfaulted program, you can specify to read its state with `gdb PROGNAME CORENAME`.

You can CTL-C when you are in a program that is in an endless loop and actually find out where the loop is.

Command shortcuts

You can create and use aliases, or use the fact that commands only need as many letters as make the command unique (and you can use TAB for completion).

Flex and lexical analysis

From the area of compilers, we get a host of tools to convert text files into programs. The first part of that process is often called lexical analysis, particularly for such languages as C.

A good tool for creating lexical analyzers is `flex`. It takes a specification file and creates an analyzer, usually called `lex.yy.c`.

Lexical analysis terms

- ☞ A token is a group of characters having collective meaning.
- ☞ A lexeme is an actual character sequence forming a specific instance of a token, such as `num`.
- ☞ A pattern is a rule expressed as a regular expression and describing how a particular token can be formed. For example, `[A-Za-z][A-Za-z_0-9]*` is a rule.

☞ Characters between tokens are called whitespace; these include spaces, tabs, newlines, and formfeeds. Many people also count comments as whitespace, though since some tools such as `lint/splint` look at comments, this conflation is not perfect.

Attributes for tokens

Tokens can have attributes that can be passed back to the calling function.

Constants could have the value of the constant, for instance.

Identifiers might have a pointer to a location where information is kept about the identifier.

Some general approaches to lexical analysis

Use a lexical analyzer generator tool, such as `flex`.

Write a one-off lexical analyzer in a traditional programming language.

Write a one-off lexical analyzer in assembly language.

Flex - our lexical analyzer generator

Is linked with its library (`libfl.a`) using `-lfl` as a compile-time option.

Can be called as `yylex()`.

It is easy to interface with `bison/yacc`.

`l file` $\longrightarrow$ $\boxed{\texttt{lex}}$ $\longrightarrow$ `lex.yy.c`

`lex.yy.c and other files` $\longrightarrow$ $\boxed{\texttt{gcc}}$ $\longrightarrow$ `lexical analyzer`

`input stream` $\longrightarrow$ $\boxed{\texttt{lexical analyzer}}$ $\longrightarrow$ `actions taken when rules applied`

Flex specifications

Lex source:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

Definitions

- ☞ Declarations of ordinary C variables and constants.
- ☞ `flex` definitions

Rules

The form of rules are:

```
regularexpression     action
```

The actions are C/C++ code.

Flex regular expressions

s	string s literally
\c	character c literally, where c would normally be a lex operator
[s]	character class
^	indicates beginning of line
[^s]	characters not in character class
[s-t]	range of characters
s?	s occurs zero or one time

Flex regular expressions, continued

```
.        any character except newline
s*       zero or more occurrences of s
s+       one or more occurrences of s
r|s      r or s
(s)      grouping
$        end of line
s/r      s iff followed by r (not recommended) (r is *NOT* consumed)
s{m,n}   m through n occurrences of s
```

Examples of regular expressions in `flex`

```
a*          zero or more a's

.*          zero or more of any character except newline

.+          one or more characters

[a-z]       a lowercase letter

[a-zA-Z]    any alphabetic letter

[^a-zA-Z]   any non-alphabetic character

a.b         a followed by any character followed by b

rs|tu       rs or tu
```

```
a(b|c)d     abd or acd

ˆstart      beginning of line with then the literal characters start

END$        the characters END followed by an end-of-line.
```

Flex actions

Actions are C source fragments. If it is compound, or takes more than one line, enclose with braces ('{' '}').

Example rules:

```
[a-z]+          printf("found word\n");
[A-Z][a-z]*     { printf("found capitalized word:\n");
                  printf("  '%s'\n",yytext);
                }
```

Flex definitions

The form is simply

```
name  definition
```

The name is just a word beginning with a letter (or an underscore, but I don't recommend those for general use) followed by zero or more letters, underscore, or dash. The definition actually goes from the first non-whitespace character to the end of line. You can refer to it via `{name}`, which will expand to `(definition)`.

(cite: this is largely from “man flex”.)

Tattoueba:

```
DIGIT    [0-9]
```

Now if you have a rule that looks like

```
{DIGIT}*\.{DIGIT}+
```

that is the same as writing

```
([0-9])*\.([0-9])+
```

An example Flex program

```
   /* either indent or use %{ %} */
%{
   int num_lines = 0;
   int num_chars = 0;
%}
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
int main(int argc, char **argv)
{
  yylex();
  printf("# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```

Another example program

```
digits       [0-9]
ltr          [a-zA-Z]
alphanum     [a-zA-Z0-9]
%%
(-|\+)*{digits}+        printf("found number: '%s'\n", yytext);
{ltr}(_|{alphanum})*    printf("found identifer: '%s'\n", yytext);
'.'                     printf("found character: {%s}\n", yytext);
.                       { /* absorb others */ }
%%
int main(int argc, char **argv)
{
    yylex();
}
```

Bison and parsing

From the area of compilers, we get a host of tools to convert text files into programs. After lexical analysis, the second part of that process when you are dealing with traditional languages such as C is syntax analysis, which also known as parsing.

A good tool for creating parsers is `bison`. It takes a specification file and creates an syntax analyzer, previously called `y.tab.c` by `yacc` and now is generally just `FILENAME.tab.c`.

Parsing terms

☞ Production rules define a parser. Informally, these can be expressed in BNF/EBNF form.

☞ Production rules are made up a left hand side with a non-terminal, and righthand side made up terminals and non-terminals.

☞ A terminal "represents a class of syntactically equivalent tokens" [Bison manual].

Attributes for terminals and non-terminals

Terminals and non-terminals can have attributes.

Constants could have the value of the constant, for instance.

Identifiers might have a pointer to a location where information is kept about the identifier.

Some general approaches to syntax analysis

Use a compiler-compiler tool, such as `bison`.

Write a one-off recursive descent parser.

Write a one-off parser suited to your program.

Bison - our lexical analyzer generator

Can be called as `yyparse()`.

It is easy to interface with `flex/lex`.

y file → bison → y.tab.c (*.tab.c)

y.tab.c and other files → gcc → syntax analyzer

input stream → syntax analyzer → actions taken when rules applied

Calling Bison

Here's an example of calling Bison (which will be very useful when compiling `assign6`):

```
Assign6-solution.out: Assign6-solution.y Assign6-solution.l
        bison -d --debug --verbose Assign6-solution.y
        flex Assign6-solution.l
        cc -c lex.yy.c
        cc -c Assign6-solution.tab.c
        cc -o Assign6-solution.out Assign6-solution.tab.o lex.yy.o
```

The `-d` option specifies to output an explicit `y.tab.h/*.tab.h` file for flex. Specifying `--debug` and `--verbose` (combined with enabling `yydebug`) make it much easier to debug your parser!

Bison specifications

Bison source:

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

Definitions

- ☞ Declarations of ordinary C variables and constants.
- ☞ `bison` declarations.

Rules

The general form for production rules is:

```
<non-terminal> :  <sequence of terminals and non-terminals> {action} | ..
```

The actions are C/C++ code. Actions can appear in the middle of the sequence of terminals and non-terminals.

Bison declarations

```
%token TOKEN        create a TOKEN type

%union { }          create a Union for llvals.

%right TOKEN        create a TOKEN type that has right associativity

%left TOKEN         create a TOKEN type that has left associativity
```

Bison actions

Actions are C source fragments.

Example rules:

```
variableDeclaration : ID COLON ID SEMICOLON {
                                        printf("emitting var %s of type %
                                            } ;
```

The $3 and $1 refer to the values of the items 3 and 1 in the righthand side of the production rule.

An example of Bison: first, its matching flex file

```
%{
#include <stdlib.h>
#include <string.h>
#include "Assign6-solution.tab.h"
extern int linecount;
%}
%%
program                 return PROGRAM;
end                     return END;
variables               return VARIABLES;
var                     return VAR;
functions               return FUNCTIONS;
define                  return DEFINE;
```

```
statements        return STATEMENTS;
if                return IF;
then              return THEN;
else              return ELSE;
while             return WHILE;
,                 return COMMA;
"("               return LPARENTHESIS;
")"               return RPARENTHESIS;
"{"               return LBRACE;
"}"               return RBRACE;
:                 return COLON;
;                 return SEMICOLON;
[a-zA-Z0-9]+      yylval = (int)strdup(yytext); return ID;
[\n]              linecount++;
[ \t]+
```

An example Bison program

```
%{
#include <stdlib.h>
#include <stdio.h>
int linecount = 0;
void yyerror(char *s)
{
  fprintf(stderr,"file is not okay -- problem at line %d\n",linecount);
  exit(1);
}
int yywrap()
{
  return 1;
}
%}
%token ID
```

```
%token PROGRAM
%token END
%token VARIABLES
%token VAR
%token STATEMENTS
%token IF
%token THEN
%token ELSE
%token WHILE
%token LBRACE
%token RBRACE
%token COLON
%token SEMICOLON
%token FUNCTIONS
%token COMMA
%token DEFINE
%token LPARENTHESIS
%token RPARENTHESIS
%%
program : PROGRAM ID variablesSection functionsSection statementsSection
variablesSection : VARIABLES LBRACE variableDeclarations RBRACE ;
```

```
variableDeclarations :  | variableDeclarations variableDeclaration ;
variableDeclaration : ID COLON ID SEMICOLON {printf("emitting var %s of t
functionsSection : FUNCTIONS LBRACE functionDeclarations RBRACE ;
functionDeclarations : | functionDeclarations functionDeclaration ;
functionDeclaration : DEFINE ID COLON ID LPARENTHESIS argsList RPARENTHES
statementsSection : STATEMENTS LBRACE statements RBRACE ;
statements : | statements statement ;
statement : VAR variableDeclaration | whileLoop | ifStruct | subroutineCa
whileLoop : WHILE LPARENTHESIS subroutineCall RPARENTHESIS LBRACE stateme
ifStruct : IF LPARENTHESIS subroutineCall RPARENTHESIS LBRACE statements
           |
           IF LPARENTHESIS subroutineCall RPARENTHESIS LBRACE statements
subroutineCall : ID LPARENTHESIS callArgsList RPARENTHESIS ;
argsList : | argPair | argsList COMMA argPair ;
argPair : ID ID ;
callArgsList : | ID |  callArgsList COMMA ID ;
%%
int main(int argc, char **argv)
{
  //  yydebug = 1;
  yyparse();
```

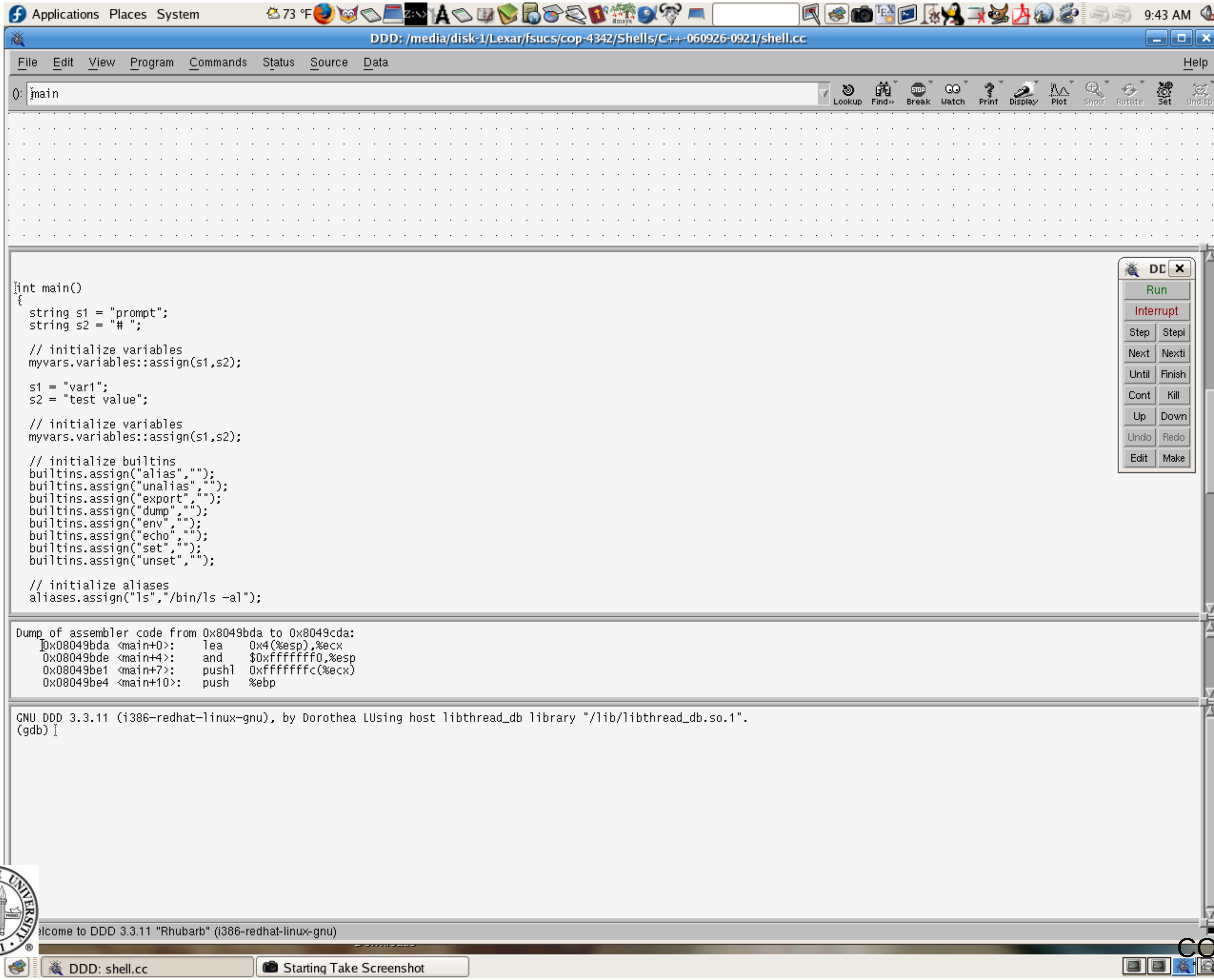
```
  printf("input is okay\n");
}
```

More tools: DDD

The Data Display Debugger (DDD) is a graphical front-end for GDB and other command line debuggers.

From DDD you can execute all of the GDB commands.

It also has a graphical interface which displays GDB commands, shows source code, shows executions, and allows to choose common options for commands.


Applications Places System
DDD: /media/disk-1/Lexar/fsucs/cop-4342/Shells/C++-060926-0921/shell.cc
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
int main()
{
string s1 = "prompt";
string s2 = "# ";
// initialize variables
myvars.variables::assign(s1,s2);
s1 = "var1";
s2 = "test value";
// initialize variables
myvars.variables::assign(s1,s2);
// initialize builtins
builtins.assign("alias","");
builtins.assign("unalias","");
builtins.assign("export","");
builtins.assign("dump","");
builtins.assign("env","");
builtins.assign("echo","");
builtins.assign("set","");
builtins.assign("unset","");
// initialize aliases
aliases.assign("ls","/bin/ls -al");
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
Dump of assembler code from 0x8049bda to 0x8049cda:
0x08049bda <main+0>: lea 0x4(%esp),%ecx
0x08049bde <main+4>: and $0xfffffff0,%esp
0x08049be1 <main+7>: pushl 0xfffffffc(%ecx)
0x08049be4 <main+10>: push %ebp
GNU DDD 3.3.11 (i386-redhat-linux-gnu), by Dorothea LUsing host libthread_db library "/lib/libthread_db.so.1".
(gdb)
lcome to DDD 3.3.11 "Rhubarb" (i386-redhat-linux-gnu)
DDD: shell.cc
Starting Take Screenshot
9:43 AM

DDD features

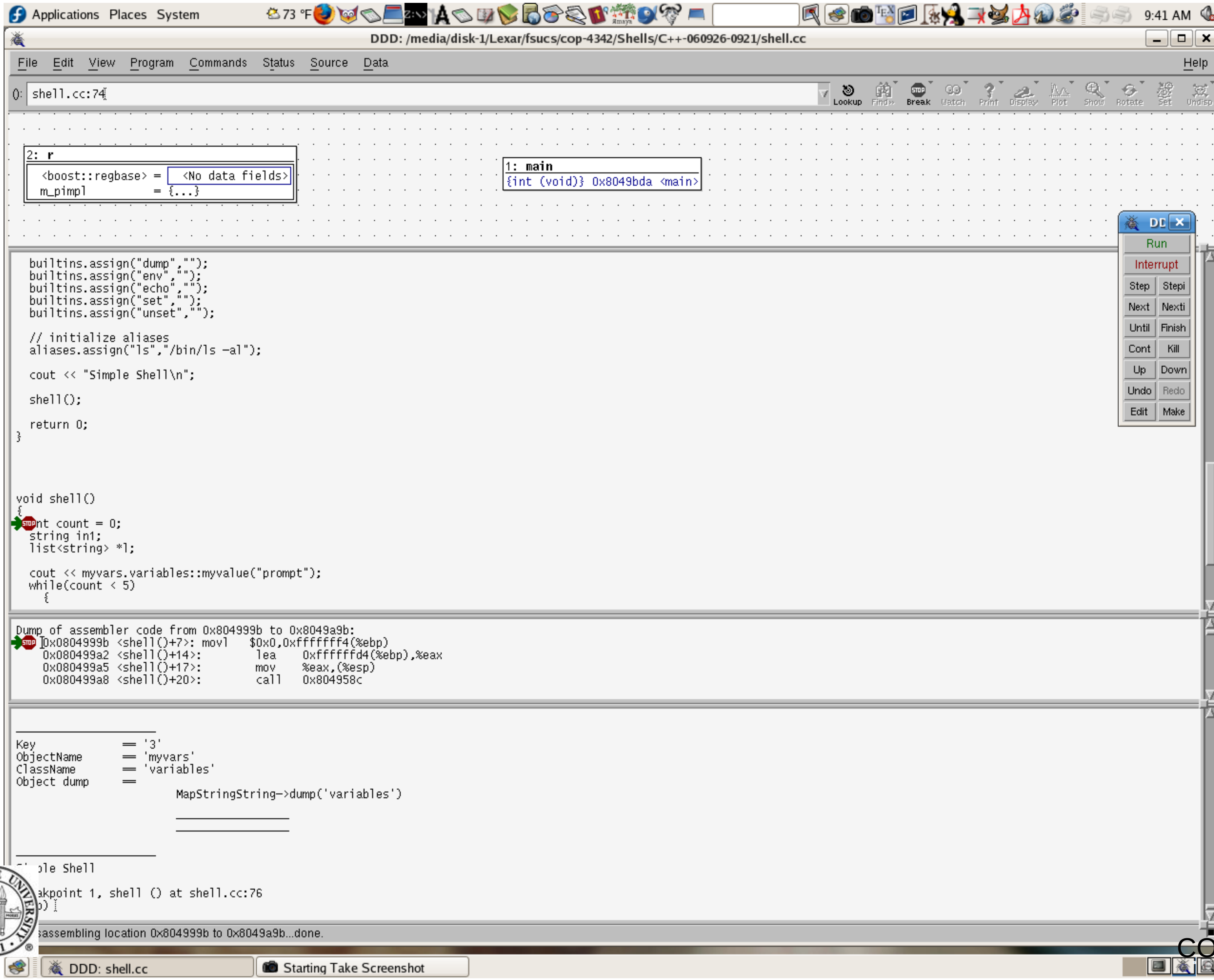
DDD shows four different windows:

- ☞ A data window to display variables.
- ☞ Source window to display source code.
- ☞ Machine code window to display disassembled machine code

DDD features

☞ GDB console where conventional gdb commands can be typed.

DDD also has other panels which include common commands that can be selected with the mouse.


Applications Places System
DDD: /media/disk-1/Lexar/fsucs/cop-4342/Shells/C++-060926-0921/shell.cc
File Edit View Program Commands Status Source Data Help
(): shell.cc:74
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
2: r
<boost::regbase> = <No data fields>
m_pimpl = {...}
1: main
{int (void)} 0x8049bda <main>
builtins.assign("dump","");
builtins.assign("env","");
builtins.assign("echo","");
builtins.assign("set","");
builtins.assign("unset","");
// initialize aliases
aliases.assign("ls","/bin/ls -al");
cout << "Simple Shell\n";
shell();
return 0;
}
void shell()
{
int count = 0;
string in1;
list<string> *l;
cout << myvars.variables::myvalue("prompt");
while(count < 5)
{
Dump of assembler code from 0x804999b to 0x8049a9b:
0x0804999b <shell()+7>: movl $0x0,0xfffffff4(%ebp)
0x080499a2 <shell()+14>: lea 0xffffffd4(%ebp),%eax
0x080499a5 <shell()+17>: mov %eax,(%esp)
0x080499a8 <shell()+20>: call 0x804958c
Key == '3'
ObjectName == 'myvars'
ClassName == 'variables'
Object dump ==
MapStringString->dump('variables')
ple Shell
akpoint 1, shell () at shell.cc:76
b)
sassembling location 0x804999b to 0x8049a9b...done.
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make
DDD: shell.cc
Starting Take Screenshot
9:41 AM

Using the DDD Source Window

Can set a breakpoint by using the right mouse button and positioning the cursor to the left of a source code line.

Can instantly view the value of a variable by placing the mouse over it (look at the very bottom of the display.)

Can highlight a variable and select to print or display its value by using the options at the top.

Using the DDD Data Window

To have a variable with its value appear in the data window as a display:

- ☞ A user can highlight a variable in the source window and then click on the display button.
- ☞ A user can double click on a variable in the source window.

`diff`

The diff Unix utility compares two files and displays the differences between the two files. The differences are displayed with an ed-like notation indicating what changes are needed to modify the first file to make it similar to the second.

`diff` is very useful in shell scripts to detect differences between expected output and actual output.

`diff` Output (UPT 11.1)

- Diff output consists of a list of changes.
- General form consists of a sequence of:

```
commands
lines
```

`diff` Output (UPT 11.1)

- Commands are of the form (a for append, c for change, and d for delete):

  `linenums [acd] linenums`

- Lines from the first file are preceded by <. Lines from the second file are preceded by >.

- `diff -r` can be recursively to compare whole directories trees.

`diff` Example

tmp1.txt:

```
cat
dog
mouse
```

tmp2.txt:

```
cat
mouse
```

tmp3.txt:

```
dog
mouse
cow
```

```
% diff tmp1.txt tmp2.txt
2d1
< dog
% diff tmp2.txt tmp3.txt
1d0
< cat
3a3
> cow
```

```
% diff tmp2.txt tmp3.txt
1c1
< cat
---
> dog
2a3
> cow
```

Patch (UPT 20.9)

Patch is a Unix utility that takes diff output and applies the commands to make the first file have the same contents as the second file. Updates to free software are often accomplished using patch. Often the differences between versions of files is much smaller than the files themselves.

cmp

The cmp Unix utility just returns a status indicating if the files differ.

```
Exit status       Meaning
-----------       -------
     0            Files are identical
     1            Files are different
     2            An error occurred
```

The cmp utility is often used when comparing two binary files since it is typically quicker than diff.

You can also specify `-s` to make `cmp` silent when it finds a difference (by default, it displays the byte and line number where the first difference was found.)

Configuration Management Systems

Definitely not the same as a Content Management System!

Configuration Management Systems are however quite similar to Content Management Systems (CMSs):

☞ Configuration Management Systems always provide a history mechanism, as do most CMSs.

☞ Provides controlled access by different users to shared files.

Configuration Management Systems

- ☞ SCCS – Source Code Control System. This is now deprecated. It kept the original files, and the deltas to get to the current version(s) of code.
- ☞ RCS – Revision Control System. Still popular. It keeps the most recent version(s) of files, and the deltas to take you back to older version(s).
- ☞ CVS – Concurrent Version System. Quite popular. Actually uses RCS underneath.

☞ subversion – Also quite popular, and is a strong competitor with CVS. Directories and file meta-data are also kept under version control. Commits are also truly atomic.

gprof

The gprof Unix utility produces an execution profile of the call graph of a program.

The profile is useful for finding out where most of the time is spent during the execution of your program.

A gmon.out file will be produced as a side effect A developer can use this information to tune the time-consuming portions of a long-running program.

gprof

You can have a program instrumented to collect data that can be processed by gprof by using the `-pg` option when compiling with gcc:

```
gcc -pg -c XYZ.c
```

A gmon.out file will be produced as a side effect of running your program.

You can obtain the profile from the gmon.out file by running the following command:

```
gprof -b
```

`make`

My description of the program `make` is that it

- ☞ takes a set of rules describing dependencies and
- ☞ describing creation of new files

in order to satisfy the requirements for the “creation” of some target.

make

Another description from Chapter 1 of the Gnu Make manual:

```
The make utility automatically determine
which pieces of a large program need
to be recompiled, and issues commands
to recompile them.
```

Invoking `make`

There are several options that are generally useful with `make`:

```
-f MAKEFILE        # specify an alternative makefile to the defaults of
                   # 'GNUmakefile', 'Makefile', or 'makefile'
-k                 # continue for other targets even after an error

-i                 # completely ignore errors

-d                 # print debugging information

-j [N]             # fork off children to handle tasks. If N is
                   # specified, create no more than N children
```

```
-C DIR            # change directory to DIR before starting the make pro

-s                # silent mode, don’t echo commands
```

Makefiles

Makefiles use rules to determine their actions. The rules look like:

```
target: [ prerequisites ]
    -TAB- action
    -TAB- action
    -TAB- ...
```

Targets

Targets usually either specify a file that is to be made via this rule or just identify the rules for execution (often called a “phony” target.)

Targets may also be implicit.

Prerequisites

These generally define the files that the target depends on, and the general idea is that if any of those have a modified (or creation) time later than the target, then actions for the rule will be executed to create a new version of the target (which you should try to make sure has a new modified or created time.)

Actions

These generally define the actions that are needed to create the target from the prerequisites. These actions are largely executions of discrete programs such as `gcc`, `make` (yes, recursion is quite common), `ld`, `bison`, `flex`, and so on. Rules must consist of consecutive lines that start with a TAB character. Since these are usually interpreted as shell commands, you can do things such as multi-lines (but use the backslash to make sure that the “single-linedness” of your construction is clear):

```
for name in dir1 dir2 dir3 \
do ; \
   ${MAKE} $name ; \
done
```

Actions

There are also actual `make` conditionals which are interpreted by `make` and not by the shell; these look like

```
ifeq (ARG1,ARG2)
...
endif

ifdef (ARG1)
...
endif
```

Setting ordinary variables

You can use “=” and “?=” to set ordinary variables:

```
CFLAGS ?= -g -O3                  # conditionally set ${CFLAGS} to
                                  # ‘‘-g -O3’’ iff it is not
                                  # already defined

CC = /usr/bin/gcc ${CFLAGS}       # unconditionally set ${CC} to
                                  # ‘‘/usr/bin/cc’’
```

Pattern rules

One of the nice things that you can do with make is create “pattern rules”.

These are rules that let you abstract a pattern from a set of similar rules, and use that pattern in lieu of explicitly naming all of those rules.

For instance,

```
%.o : %.c
        cc -c $< -o $@      # $@ refers to the
                            # target, $< refers to
                            # the *first* (and only)
                            # prerequisite
```

Automatic variables

```
$@      # the target of the rule

$<      # the first prerequisite

$^      # all of the prerequisites

$?      # all of the prerequisites that are newer than the target file

$*      # the ‘‘stem’’ only; essentially, this is the complement of the st
        # of the target definition... see Makefile-auto
```

Example Makefiles

```
targets: 01-introduction-out.pdf 02-processes-out.pdf \
03-shells1-out.pdf 03-shells2-out.pdf 04-shells3-out.pdf \
05-shells4-out.pdf 06-environment-out.pdf 07-perl01-out.pdf \
08-perl02-out.pdf 09-perl03-out.pdf 10-perl04-out.pdf \
11-perl05-out.pdf 12-perl06-out.pdf 13-perl07-out.pdf \
14-programdevel-out.pdf 15-programdevel02-out.pdf \
16-programdevel03-out.pdf 17-programdevel04-out.pdf \
18-programdevel05-out.pdf 19-programdevel06-out.pdf \
20-programdevel07-out.pdf structure-out.pdf

%-out.pdf: %.tex
        pdflatex $<
        gij -jar pp4p.jar $*.pdf $*-out.pdf
```

Example Makefiles

```
%.c:
        echo $*
```

make

My description of the program make is that it

- ☞ takes a set of rules describing dependencies and
- ☞ describing creation of new files

in order to satisfy the requirements for the “creation” of some target.

make

Another description from Chapter 1 of the Gnu Make manual:

```
The make utility automatically determine
which pieces of a large program need
to be recompiled, and issues commands
to recompile them.
```

Invoking `make`

There are several options that are generally useful with `make`:

```
-f MAKEFILE          # specify an alternative makefile to the defaults of
                     # 'GNUmakefile', 'Makefile', or 'makefile'
-k                   # continue for other targets even after an error

-i                   # completely ignore errors

-d                   # print debugging information

-j [N]               # fork off children to handle tasks. If N is
                     # specified, create no more than N children
```

```
-C DIR              # change directory to DIR before starting the make pro

-s                  # silent mode, don't echo commands
```

Makefiles

Makefiles use rules to determine their actions. The rules look like:

```
target: [ prerequisites ]
      -TAB- action
      -TAB- action
      -TAB- ...
```

Targets

Targets usually either specify a file that is to be made via this rule or just identify the rules for execution (often called a “phony” target.)

Targets may also be implicit.

Prerequisites

These generally define the files that the target depends on, and the general idea is that if any of those have a modified (or creation) time later than the target, then actions for the rule will be executed to create a new version of the target (which you should try to make sure has a new modified or created time.)

Actions

These generally define the actions that are needed to create the target from the prerequisites. These actions are largely executions of discrete programs such as `gcc`, `make` (yes, recursion is quite common), `ld`, `bison`, `flex`, and so on. Rules must consist of consecutive lines that start with a TAB character. Since these are usually interpreted as shell commands, you can do things such as multi-lines (but use the backslash to make sure that the "single-linedness" of your construction is clear):

```
for name in dir1 dir2 dir3 \
do ; \
   ${MAKE} $name ; \
done
```

Actions

There are also actual `make` conditionals which are interpreted by `make` and not by the shell; these look like

```
ifeq (ARG1,ARG2)
...
endif

ifdef (ARG1)
...
endif
```

Setting ordinary variables

You can use “=” and “?=” to set ordinary variables:

```
CFLAGS ?= -g -O3                  # conditionally set ${CFLAGS} to
                                  # ‘‘-g -O3’’ iff it is not
                                  # already defined

CC = /usr/bin/gcc ${CFLAGS}       # unconditionally set ${CC} to
                                  # ‘‘/usr/bin/cc’’
```

Pattern rules

One of the nice things that you can do with make is create “pattern rules”.

These are rules that let you abstract a pattern from a set of similar rules, and use that pattern in lieu of explicitly naming all of those rules.

For instance,

```
%.o : %.c
        cc -c $< -o $@      # $@ refers to the
                            # target, $< refers to
                            # the *first* (and only)
                            # prerequisite
```

Automatic variables

```
$@      # the target of the rule

$<      # the first prerequisite

$ˆ      # all of the prerequisites

$?      # all of the prerequisites that are newer than the target file

$*      # the ‘‘stem’’ only; essentially, this is the complement of the st
        # of the target definition... see Makefile-auto
```

Example Makefiles

```
targets: 01-introduction-out.pdf 02-processes-out.pdf \
03-shells1-out.pdf 03-shells2-out.pdf 04-shells3-out.pdf \
05-shells4-out.pdf 06-environment-out.pdf 07-perl01-out.pdf \
08-perl02-out.pdf 09-perl03-out.pdf 10-perl04-out.pdf \
11-perl05-out.pdf 12-perl06-out.pdf 13-perl07-out.pdf \
14-programdevel-out.pdf 15-programdevel02-out.pdf \
16-programdevel03-out.pdf 17-programdevel04-out.pdf \
18-programdevel05-out.pdf 19-programdevel06-out.pdf \
20-programdevel07-out.pdf structure-out.pdf

%-out.pdf: %.tex
        pdflatex $<
        gij -jar pp4p.jar $*.pdf $*-out.pdf
```

Example Makefiles

```
%.c:
        echo $*
```

File management

- ☞ `gzip` and `gunzip`
- ☞ `tar`
- ☞ `find`
- ☞ `df` and `du`
- ☞ `od`
- ☞ `sftp` and `scp`

`gzip` **and** `gunzip`

- `gzip` compresses the files named on the command line. After compressing them, it renames them with `.gz` suffixes.
- General form:

```
gzip [FILE]*
```

- `gunzip` undoes compression created by `gzip`.
- General form:

```
unzip [FILE]*
```

- ☞ Other programs that have been used for compression: `compact`, `compress`, and `zip/unzip`.
- ☞ You can also use `gzip/gunzip` as filters with the `-c` option, which redirects output to stdout.
- ☞ Finally, you can specify the level of compression; `-1` gives the fastest compression but does not optimize space, and `-9` gives the slowest compression but the best use of space.

`tar`

`tar` is an old utility, and literally stands for “Tape Archiver”. These days, it is used far more often to handle file archives. It is very useful for creating transportable files between systems, such as when you want to mail a group of files to someone else.

tar options

```
-c      # create an archive
-x      # extract from an archive
-t      # shows files in an archive
-f      # specify a file (the default is a tape device!). You can
        # specify stdout with ‘‘-’’ (or use -O)
-C      # change directory
-v      # work verbosely
-z      # use gzip/gunzip; if used with -c, creates a gzip’d file; if use
        # with -x or -t, it uses gunzip to read the existing archive file
-p      # preserve permissions when extracting
```

Using `tar`

Typically, you will do something like this to create a `tar` archive of an existing subdirectory:

```
tar cf DIRNAME.tar [DIRNAME]+
tar czf DIRNAME.tar.gz [DIRNAME]+
tar -c -f DIRNAME.tar [DIRNAME]+
tar -c -z -f  DIRNAME.tgz [DIRNAME]+
```

Using `tar`

Typically, you will do something like this to extract an `tar` archive of an existing subdirectory:

```
tar xf DIRNAME.tar
tar xzf DIRNAME.tar.gz
tar -x -f DIRNAME.tar
tar -x -z -f DIRNAME.tgz
```

`find`

One of the most useful tools with a recondite syntax is `find`. It allows you to search a directory for files matching some subset of a large number of possible criteria.

find [PATH]+ CRITERIA

`find` criteria

```
-name FILENAME        # finds files which match FILENAME, which can conta
                      # wildcards
-iname FILENAME       # same as -name, but also case-insensitive
-size [+/-]N[bck]     # very useful, it finds files by size. using 'b' (o
                      # indicates N is in blocks; using 'c' indicates N i
                      # using 'k' indicates N using kilobytes. Using '+'
                      # the file is greater than N in size; using '-' mea
                      # that it is less than N in size; using neither mea
                      # the file is exactly size N.
-mtime [+/-]N         # find files based on their last modification time,
                      # in days. +N means match files that have been modi
                      # than N days; -N means match files that have been
                      # less than N days; N means match files that have b
                      # exactly N days previous.
-ls                   # show files in {\tt ls} format rather than just th
```

```
-printf                 # lets you specify arbitrary output formats
-exec COMMAND ;         # lets you run COMMAND over every matching file
-okay COMMAND ;         # same as -exec, but queries you for confirmation b
                        # the command
```

`find` logical operators

```
CRIT1 -a CRIT2        # match only if both criteria CRIT1 and CRIT2 hold
CRIT1 -o CRIT2        # match if either criteria CRIT1 or CRIT2 holds
!CRIT1                # match if criterion CRIT1 does not hold
\( EXPR \)            # evaluate EXPR early
```

`find` examples

```
find .                      # walk the current directory and its subdirectories

find /tmp -mtime +6         # find all files in /tmp that have not been
                            # modified in 6 days

find /tmp -name core -exec rm {} \;     # remove files named 'core' from /

find /tmp -name core -o name '*.o' -okay rm {} \;
                            # query to remove files that are named 'core' or en

find /tmp -iname '*.sh' -exec chmod +x {} \;
                            # add execute permission to all files that end in '
                            # '.SH', '.Sh', or '.sH'
```

`df` and `du`

The `df` command displays information about mounted filesystems. If you don't specify any, then all of the mounted filesystems are shown.

You don't have to specify mount points; any file inside of a filesystem is acceptable:

```
[2006-Fall]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda2             75766204  19014760  52902672  27% /
/dev/hda1               101089     40221     55649  42% /boot
none                    251668         0    251668   0% /dev/shm
```

```
/dev/sda1               981192    480508    450840  52% /mnt-tmp
[2006-Fall]$ df /boot/boot.b
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda1               101089     40221     55649  42% /boot
```

`df` and `du`

The du command shows you the usage of disk space. With no options, it walks your current directory and shows you the space in blocks used by each subdirectory. With -s, it just shows you a summary. You can force du to display in 1k blocks with the -k option.

```
[2006-Fall]$ du -sk .
8744        .
[2006-Fall]$ du midterm1
80          midterm1/Questions/Shell
```

```
20        midterm1/Questions/Process
52        midterm1/Questions/Perl
20        midterm1/Questions/Emacs
20        midterm1/Questions/Awk
20        midterm1/Questions/General
980        midterm1/Questions
1636        midterm1
```

od

The od (octal dump) program writes representations of files to stdout. If '-' is specified, then it looks to stdin for input.

For example, the default od output for the current pdf file is:

```
[langley@sophie 2006-Fall]$ od 22-filemanagement.pdf
0000000 050045 043104 030455 031456 032412 030040 067440 065142
0000020 036012 020074 051457 027440 067507 067524 027440 020104
0000040 033133 030040 051040 020040 043057 072151 056440 037040
0000060 005076 067145 067544 065142 034412 030040 067440 065142
```

```
0000100 036040 005074 046057 067145 072147 020150 030465 020064
0000120 020040 020040 020040 027412 064506 072154 071145 027440
 ...
```

`od` and `xxd`

`od` is useful in several ways; for instance, you can find control characters embedded in files that an editor might not display in a reasonable fashion (though `emacs` is pretty good at displaying embedded characters.)

Using od

```
[2006-Fall]$ od -a 22-filemanagement.pdf
0000000   %   P   D   F   -   1   .   3  nl   5  sp   0  sp   o   b   j
0000020  nl   <   <  sp   /   S  sp   /   G   o   T   o  sp   /   D  sp
0000040   [   6  sp   0  sp   R  sp  sp   /   F   i   t  sp   ]  sp   >
0000060   >  nl   e   n   d   o   b   j  nl   9  sp   0  sp   o   b   j
0000100  sp   <   <  nl   /   L   e   n   g   t   h  sp   5   1   4  sp
0000120  sp  sp  sp  sp  sp  sp  nl   /   F   i   l   t   e   r  sp   /
 ....
```

Using od

```
[2006-Fall]$ od -c 22-filemanagement.pdf
0000000   %   P   D   F   -   1   .   3  \n   5       0       o   b   j
0000020  \n   <   <       /   S       /   G   o   T   o       /   D
0000040   [   6       0       R           /   F   i   t       ]       >
0000060   >  \n   e   n   d   o   b   j  \n   9       0       o   b   j
0000100       <   <  \n   /   L   e   n   g   t   h       5   1   4
0000120                          \n   /   F   i   l   t   e   r       /
 ....
```

Using xxd

The program `xxd` adds some functionality to od: specifically, it can `read` a dump and recreate a binary from it. This is very useful for "patching" a binary.

```
[2006-Fall]$ xxd Script12.sh
0000000: 2321 2f62 696e 2f62 6173 680a 0a23 2032  #!/bin/bash..# 2
0000010: 3030 3620 3039 2031 3120 2d20 7264 6c0a  006 09 11 - rdl.
0000020: 666f 7220 6e61 6d65 2069 6e20 2a0a 646f  for name in *.do
0000030: 0a20 2069 6620 5b20 2d66 2022 246e 616d  .  if [ -f "$nam
0000040: 6522 205d 0a20 2074 6865 6e0a 2020 2020  e" ].  then.
0000050: 2065 6368 6f20 2273 6b69 7070 696e 6720   echo "skipping
0000060: 246e 616d 6522 0a20 2020 2020 636f 6e74  $name".     cont
0000070: 696e 7565 0a20 2065 6c73 650a 2020 2020  inue.  else.
```

```
0000080: 2065 6368 6f20 2270 726f 6365 7373 2024   echo "process $
0000090: 6e61 6d65 220a 2020 6669 0a64 6f6e 650a  name".  fi.done.

[[ ... edit file to change 0000013 to ’37’ rather than ’36’ ... ]]

[2006-Fall]$ !! > /tmp/xyz
xxd Script12.sh > /tmp/xyz
[2006-Fall]$ xxd -r /tmp/xyz
#!/bin/bash

# 2007 09 11 - rdl
for name in *
do
  if [ -f "$name" ]
  then
     echo "skipping $name"
     continue
  else
     echo "process $name"
  fi
done
```

`nm`

The `nm` utility lets you print out the namelist of symbols from object files.

This was very useful in finding where a particular variable or function is defined.

[Historical note: Reading the namelist was also a method used “wayback when” to access particular areas of the kernel to make reports on such values as uptime. Literally, the program would parse the namelist of the kernel, find the reference to the variable that it wanted, and read that area of memory from /dev/kernel to find the values it wanted.]

`strip`

The `strip` utility removes optional symbol table, debugging, and line number information from an object file or an executable. `strip` will reduce the amount of space used by object files and binaries.

sftp

You can transfer files securely with the `sftp` program.

```
sftp [USERNAME]@HOSTNAME
```

Common sftp commands

```
ls [NAME]           # show a directory entry for NAME if specified, other fo
                    # the present remove working directory
dir [NAME]          # alias for 'ls'
!                   # start a subshell
!ls                 # show the local directory (via a subshell)
!COMMAND            # run command in subshell
put LOCALFILE [REMOTEFILE]  # put a local file on the remote machine; use
                            # the filename 'REMOTENAME' if specified
get REMOTEFILE [LOCALFILE]  # pull a remote file to the local machine; ca
                            # it LOCALNAME if specified
cd [DIR]            # change directory on the remote side
lcd [DIR]           # change directory on the local side
chmod PERM FILE     # change permissions on remote file FILE
pwd                 # show the current remote directory
lpwd                # show the local directory
mkdir DIR           # create a new directory on the remote side
```

scp

You can also noninteractivley transfer a file or directory with scp:

```
[2006-Fall]$ scp /tmp/xyz langley@www.cs.fsu.edu:/tmp/xyz
                       -=-= AUTHORIZED USERS ONLY =-=-
You are attempting to log into a FSU Computer Science Department machine.
Please be advised by continuing that you agree to the terms of the
Computer Access and Usage Policy of the Department of Computer Science.
                       -=-= AUTHORIZED USERS ONLY =-=-
langley@www.cs.fsu.edu's password: XXXXXXX
[2006-Fall]$ scp -r /etc langley@www.cs.fsu.edu:/tmp/backup-etc
```

Document preparation in Unix

- ☞ TeX and LaTeX
- ☞ graphviz
- ☞ xfig
- ☞ xv
- ☞ spell checkers
- ☞ printing

Word Processors

Word processors, such as Microsoft's Word® and OpenOffice's Writer, use the WYSIWYG model:

☞ Word processors are interactive.

☞ Word processors are relatively easier to learn

☞ Word processors are very useful for those who need to do simple documents occasionally.

Text formatters

Text formatters, such as TEX/LATEX, use the model of “markup”, where text is decorated with markup commands and then processed by a program; output can then be viewed.

Characteristics, then, of text formatting:

☞ It tends to be batch-oriented

☞ Generally better control over the output

- ☞ Output generally looks better
- ☞ Much better for creating longer documents
- ☞ Much better for creating long-life documents
- ☞ Much better for creating series of related documents
- ☞ Having the source in text means that other text tools can be applied to the source.

TeX and LaTeX

TeX was invented in the late 1970s by Donald Knuth. The first generally useful release was probably TeX82 in 1982, though the language wasn't frozen until 1989.

It was created to make nice mathematical documents, with emphasis on mathematical fonts since many of the easily available ones for electronic production were not high quality.

LaTeX was invented in 1985 by Leslie Lamport. It contains higher level support for many constructions such as table of contents, citations, floating tables and figures, and so forth.

Generating a LaTeX document

There are a variety of ways these days to generate a LaTeX document. The most general one is

`*.tex file` → $\boxed{\texttt{latex}}$ → `*.dvi file` → $\boxed{\texttt{dvips/dvipdf}}$ `*.pdf`

The simplest these days combines these two steps:

`*.tex file` → $\boxed{\texttt{pdflatex}}$ `*.pdf`

The idea behind `dvi` files is that they were to be "device independent", and then output would go to a special driver for whatever output device might be available, such as our ancient Imagen printers.

Of course, Adobe invented PostScript® which instituted what was to become an equally device independent mechanism, at least to the level of fonts. The "Portable Document Format" (pdf) then added fonts to the output format. This was a bit of a muddle for TEXsince its model was to create its own fonts with the program Metafont, but these days, TEXalso can read and use other font families seamlessly.

Metafont and MetaPost

Fonts are created by the Metafont program, and graphics can be created with MetaPost.

Generally, you won't have to worry about this; LaTeXwill usually call Metafont seamlessly if it needs to recreate a font.

LaTeX commands

A LaTeX file must contain not only text but also markup commands. Commands consist of a special single characters or a words preceded by the backslash.

%	indicates a comment	~	represents a space
&	is used in making tables	$	is used to indicate math
{	starts an argument list	}	ends an argument list
_	precedes a subsript	^	precedes a superscript
#	used in defining commands		

Generally, these can be printed by preceding them with a backslash, though the safest thing is to use

SPECIAL.

LaTeX comments

A comment begins with % and ends with the line.

This is similar to the C++ // or Ada - - comment.

Document structure with the “Article” class

```
\documentclass[12pt]{article}        % specify class
\usepackage{fancyvrb}                % preamble: use a package
\usepackage{graphics}                % preamble: use a package
\begin{document}                     % start the actual document to layo
\title{}                             % title of the article
\author{}                            % author of the article
\date{\today}                        % you can specify a date, or use to
\maketitle                           % this displays the preceding
\tableofcontents                     % creates a table of contents
\begin{abstract}                     % start an abstract environment
\end{abstract}                       % end an abstract environment
\section{NAME}\label{}               % start a section, create a label f
...
```

```
\section{NAME}\label{}                  % another section
\bibliography{}                         % generate a bibliography
\end{document}                          % finish the document
```

LaTeX document class

The document class defines the way that the document will be formatted.

Popular classes include:

```
article         % short articles such as journal papers
report          % longer works broken into chapters
book            % has chapters, treats odd and even pages differently
slides          % a slide package
foils           % another slide package
letter          % used for writing letters
exam            % used for making exams
```

For instance, to specify an article with an 11 point font, use

```
\documentclass[11pt]{article}
```

LaTeX packages

TeX is a Turing-complete language, and numerous packages have been created to support use of TeX and LaTeX.

You can access these packages with `\usepackage{}`.

For example,

```
\usepackage{graphics}
\usepackage{graphicx}
```

Beginning the document

To end the preamble and actually start creating displayable material (i.e., the “body” of your document), you insert the `\begin{document}` command; to end the document, you use `\end{document}`.

Environments

Environments allow you to specially treat text that environment uniformly. For instance, you might want to enumerate some items. Rather than having to write spacing and enumeration data for each item, you simply point what the items are:

```
\begin{enumerate}
\item This is item 1.
\item This is item 2.
\end{enumerate}
```

The LaTeXarticle heading

The LaTeXarticle header consists of the title, author, and date.

The `\title{TITLE TEXT}` command is used to store the text for the title.

The `\author{AUTHORS}` command is used to store the author information. You can use `\and` to separate multiple authors.

The `\date` command contains the date of the article. If not specified, the current date will be used.

The LaTeX article heading, cont'd

The `\maketitle` command causes the title, author, and date information to be typeset into the article.

Depending on the style, the title might appear on its own page, or on the first page.

For example,

```
\title{Introduction to \LaTeX}
\author{John Doe \\
Florida State University}
\date{October 10, 2006}
\maketitle
```

Document spacing

The Wikipedia has a good description of TEX's input process at http://en.wikipedia.org/TeX. Here's a summary:

```
The system can be divided into four levels: in the first, characters
are read from the input file and assigned a category code (sometimes
called catcode, for short). Combinations of a backslash (really: any
character of category zero) followed by letters (characters of
category 11) or a single other character are replaced by a control
sequence token. In this sense this stage is like lexical analysis,
although it does not form numbers from digits. In the next stage,
expandable control sequences (such as conditionals or defined macros)
```

are replaced by their replacement text. The input for the third stage is then a stream of characters (including ones with special meaning) and unexpandable control sequences (typically assignments and visual commands). Here characters get assembled into a paragraph. TeX’s paragraph breaking algorithm works by optimizing breakpoints over the whole paragraph. The fourth stage breaks the vertical list of lines and other material into pages.

Document spacing

In addition to simple paragraph breaking and setting in pages, LaTeX handles *floating* figures and tables quite well.

Whitespace in the form of blanks and newlines indicate the end of a word. Otherwise it isn't significant.

New paragraphs can be indicated by at least one blank line.

LaTeX abstract environment

Abstracts are created in LaTeX with the `abstract` environment.

Example:

```
\begin{abstract}
This paper goes over the basics of \LaTeX.
\end{abstract}
```

LaTeX sectioning

A LaTeX article is divided with the following commands:

```
\section{NAME}
\subsection{NAME}
\subsubsection{NAME}
```

Section numbers and titles are saved for a table of contents if requested.

For example:

```
\section{The Art of \LaTeX}
```

```
\subsection{\LaTeX’s Picture Environment}
\section{Font Fun in \LaTeX }
```

Labels and References in LaTeX

Sections are often referred to by number within a document. However, writers can and do decide to reorder sections. LaTeXallows writers to give internal names to sections, and then to refer to those names to avoid having to renumber internal references inside of documents.

For example:

```
\section{The Paucity of Comment Markers}
\label{paucity}
```

```
...
As mentioned in section \ref{paucity}, there are no suitable replacements
```

LaTeX font styles

☞ Text shape: you can choose a text “shape” with various “text” commands:

```
\textit{italics text}
\textsl{slanted text}
\textsc{small caps text}
```

italics text

slanted text

SMALL CAPS TEXT

LaTeX font styles

☞ Text weight: you can also choose text "weight" with "text" commands:

```
\textmd{medium weight}\\
\textbf{boldface weight}\\
```

medium weight **boldface weight**

LaTeX font styles

☞ Text families: you can also choose text families with "text" commands:

```
\textrm{Roman family}
\textsf{Sans serif family}
\texttt{Typewriter/teletype family}
```

Roman family

Sans serif family

`Typewriter/teletype family`

LaTeX font styles

☞ Also, you can use `\usepackage{family}` to specify a font family:

```
\usepackage{avant}
\usepackage{bookman}
\usepackage{chancery}
\usepackage{charter}
\usepackage{courier}
\usepackage{newcent}
\usepackage{palatino}
```

Font sizes

You can use the following commands to modify the current font size:

```
\tiny
\scriptsize
\footnotesize
\normalsize
\large
\Large
\LARGE
\huge
\Huge
```

LaTeX tables

LaTeX has two table-related environments: “table” and “tabular”.

The floating “table” enviroment is used to specify location and captioning.

The “tabular” environment is used to format the actual table.

LaTeX tables

```
\begin{table}[t]   %% top placement
\begin{tabular}{c|c|c}   %% center everything
center & center & center \\
\hline                      %% doesn't need a \\
center & center & center \\
center & center & center \\
\end{tabular}
\end{table}
```

Table placement

You can suggest locations for tables, which are "float". You can use the following location suggestions, and you may list them in order of your preference:

- ☞ h – "here". Try to place the table where at this point in the text.
- ☞ `t` – "top". Try to place the table at the top of the current page; if it doesn't fit, try to place it at the top of the next page.

☞ b – “bottom”. Try to place the table at the bottom of the current page; if it doesn’t fit, try to place it at the bottom of the next page.

☞ p – “page”. Place the table on a separate page for tables and figures.

Formatting columns

The `\begin{tabular}{FORMAT}` command allows you to specify column formatting.

```
l     %% column is left-justified
c     %% column is centered
r     %% column is right-justified
|     %% draws a vertical
||    %% draws two vertical lines together
```

Specifying data in the table

Horizontal “data” lines end in “\\”.

Column entries are divided by ampersands (“&”).

Horizontal rules can be drawn with “\hline”.

For example:

```
\begin{tabular}{l|l||l}
Command & Arguments & Explanation\\
\hline
{\tt break} & \verb+[file:]function+ & Sets a breakpoint at function\\
\end{tabular}
```

Figures

LaTeX supports a “figure” environment, where you can place a graphic of some sort (though I think that generally it is best to stick with either encapsulated PostScript®; however, the “png” format generally works fine also.)

Figures

```
\begin{figure}[PLACEMENT]
\includegraphics[OPTIONS]{FILENAME}
\caption{CAPTION}
\label{LABEL}
\end{figure}
```

Figures

Note that the PLACEMENT is an option specified with [ ], not a requirement as with the table environment.

Options

```
width=      %% you can specify a width, such as [width=5in]

height=     %% you can specify a height, such as [height=5in]

scale=      %% you can specify a scaling factor, such as [scale=0.75]

angle=      %% you can specify an angle in degrees, such as [angle=45]
```

Figure example

Figure 1: FSU 1851 logo

```
\begin{figure}[h]
\centering
\includegraphics[width=2.2in]{fsu-1851-trans.png}
\caption{FSU 1851 logo}
\end{figure}
```

Another figure example

Figure 2: FSU 1851 logo

```
\begin{figure}[h]
\centering
\includegraphics[width=1.6in,angle=30]{fsu-1851-trans.png}
\caption{FSU 1851 logo}
\end{figure}
```

Lists in LaTeX

There are many types of lists possible in LaTeX.

For instance, you can use:

☞ `itemize` – bulleted lists

☞ `enumerate` – numbered lists

☞ `description` – customized lists

☞ `dinglist` – a type of customized used on this list

Lists in $\LaTeX$

The general form is

```
\begin{LISTTYPE}
\item
\item
 ...
\item
\end{LISTTYPE}
```

Example of a list

```
\begin{dinglist}{\DingListSymbolA}
\item {\tt itemize} -- bulleted lists
\item {\tt enumerate} -- numbered lists
\item {\tt description} -- customized lists
\item {\tt dinglist} -- a type of customized used on this list (via
\verb+\usepackage{pifont}+, which gives you access to ding characters)
\end{dinglist}
```

Arbitrary text rotation

You can use the package “rotating” to do arbitrarily rotated text:

Rotate this text

```
\usepackage{rotating}
...
\begin{rotate}{30}
Rotate this text
\end{rotate}
```

The verbatim and Verbatim environments; inline verb

With the wide allocation of special characters to default use in LaTeX, it is often convenient go into a mode that explicitly treats special characters as ordinary ones. Since this very useful for displaying program code, these environments generally also are monospaced and, by default, in a teletype font.

☞ `\verb` – you can use the inline `\verb` to specify verbatim while in normal paragraph mode, such as `%@*!)!%$%*!@` with `\verb+%@*!)!%$%*!@+`.

☞ `\begin{verbatim}` – you can use the standard verbatim environment for multiline material

☞ `\begin{Verbatim}` – if you do a `\usepackage{fan` you can include verbatim material in footnotes, modify the font size and font family, and many other effects.

Fancy Verbatim

The output of the following

```
\begin{Verbatim}[fontshape=it,frame=leftline,fontsize=\scriptsize]
Easy to see what is there
When the left line is where
We might care
\ end{Verbatim}
```

is on the next slide...

Fancy Verbatim

```
Easy to see what is there
When the left line is where
We might care
```

Multiple columns

You can also create multicolumn output in the middle of a page with the “multicol” package:

```
\documentclass[12pt]{article}
\usepackage{multicol}
\begin{document}
\setlength{\columnseprule}{1pt}  %% make a one pt rule between columns
Not multicolumn in the beginning, but the next bit is:
\begin{multicols}{3}
This is 3 col material in the middle of a page, instead of for the
whole document. It’s convenient on occasion, but usually the tabular
environment is what you want, not multicol.
\end{multicols}
And then back to single column mode.
\end{document}
```

Bibliographies in LaTeX

You can keep your bibliographic references in a file called `BIBLIO.bib`; this file is to be processed by the program `bibtex`.

The text references in your paper are made with the `\cite` command:

```
\cite{KEY}
```

Bibliographies in LaTeX

You cause the actual generation of the bibliography with:

```
\bibliographystyle{STYLE}
\bibliography{BIBLIO}
```

Creating your bibliography database

Each entry in the database contains predefined information, some general and some specific to various types of publications.

These fields include author, title, journal, volume, number, pages, date, institution, publisher, url.

Creating your bibliography database

The general form of each of the entries in a *.bib file is:

```
@entry_type{key,
    field_name = ‘‘text’’,
    field_name = ‘‘text’’,
     ...
    field_name = ‘‘text’’
}
```

Examples

```
@book{Crandal:2001:PNCP,
  author = "Richard Crandall and Carl Pomerance",
  title = "Prime Numbers: A Computational Perspective",
  year = "2001",
  address = "New York",
  publisher = "Springer-Verlag",
  ISBN = "0-387-94777-9"
}
```

Examples

```
@article{Cipra:1996:SLLN,
  author = "Barry Cipra",
  title = "The Secret Life of Large Numbers",
  year = "1996",
  journal = "What’s Happening in the Mathematical Sciences",
  volume = "3",
  address = "Providence Rhode Island",
  publisher = "American Mathematical Society",
  pages = "90-99",
  ISBN = "0-8128-0355-7"
}
```

Bibliography styles

There are four `\bibliographystyles` recognized:

☞ `plain` – entries are ordered alphabetically and markers are a number inside square brackets

Bibliography styles

- ☞ `unsrt` – entries are ordered by appearance of citation inside the paper
- ☞ `alpha` – same as plain but markers are an abbreviation of the author's name and year

Bibliography styles

☞ `abbrv` – same as plain but bibliographic listing abbreviates first names, months, and journal names

The order of events

In order to have your bibliography compiled into your paper, you run the following sequence of programs:

```
pdflatex BASENAME
bibtex BASENAME
pdflatex BASENAME
```

The order of events

While you can specify suffixes with pdflatex/latex, bibtex is not some accommodating and it is easier to just specify the basename. This is also true inside of your document: at the `\bibliography` command, don't put the `.bib`.

Viewing output

You have a number of choices for viewing various output:

- ☞ `dvi` files – you can use `xdvi` or `evince`.
- ☞ `ps` files – you can use `gv`, `ghostview`, or `evince`.
- ☞ `pdf` files – you can use `xpdf` or `evince`.

Conversions

As mentioned earlier, there are a number of conversions that you might want to do with your LaTeXoutput:

☞ `dvips / dvi2ps` – converts a DVI file to PostScript® (PS).

☞ `ps2pdf` – converts a PostScript file to Portable Document Format (PDF).

Conversions

- `dvipdf` – converts a DVI file to PDF.
- `pdftops` – converts a PDF file to PS.

Conversions

- ☞ `pdftotxt` – converts a PDF file to text.

Diagrams with dot files

The `graphviz` package allows you to use an ordinary text file to automatically create graph visualizations.

As you can see from the examples displayed, it can make some very neat visualizations. You can find more information at `http://www.graphviz.org`.

The dot language

Here's the dot code for the graph in my sendmail paper:

```
// Uses graphviz package from http://www.graphviz.org

digraph MailSplit
{
        "Outside Mailer"  [shape = parallelogram];
        "Incoming Mailer"  [shape = parallelogram];
        "Outgoing Mailer"  [shape = parallelogram];
        "Outside Mailer" -> "Incoming Mailer"
           [label =
             "An Email Message With\n Multiple Recipients In\n Envelope"]
        "Incoming Mailer" -> "Queue Entry for\n Recipient #1"
```

```
    [label = "Recipient #1"];
"Incoming Mailer" -> "Queue Entry for\n Recipient #2"
    [label = "Recipient #2"];
"Incoming Mailer" -> "..." [style = "dotted"];
"Incoming Mailer" -> "Queue Entry for\n Recipient #n"
    [label = "Recipient #n"];
subgraph cluster_0 {
         style = filled;
        color = lightgrey;
        label = "Incoming Queue";
        "Queue Entry for\n Recipient #1"
           [style=filled,color=white];
        "Queue Entry for\n Recipient #2"
           [style=filled,color=white];
        "..." [style=filled,color=white];
        "Queue Entry for\n Recipient #n"
           [style=filled,color=white];
}
"Queue Entry for\n Recipient #1" -> "Outgoing Mailer";
"Queue Entry for\n Recipient #2" -> "Outgoing Mailer";
"..." -> "Outgoing Mailer" [style=dotted];
```

```
        "Queue Entry for\n Recipient #n" -> "Outgoing Mailer";
}
```

Other tools: `xfig`

`xfig` is a menu-driven tool that allows a user to interactively create and manipulate figures. Features include:

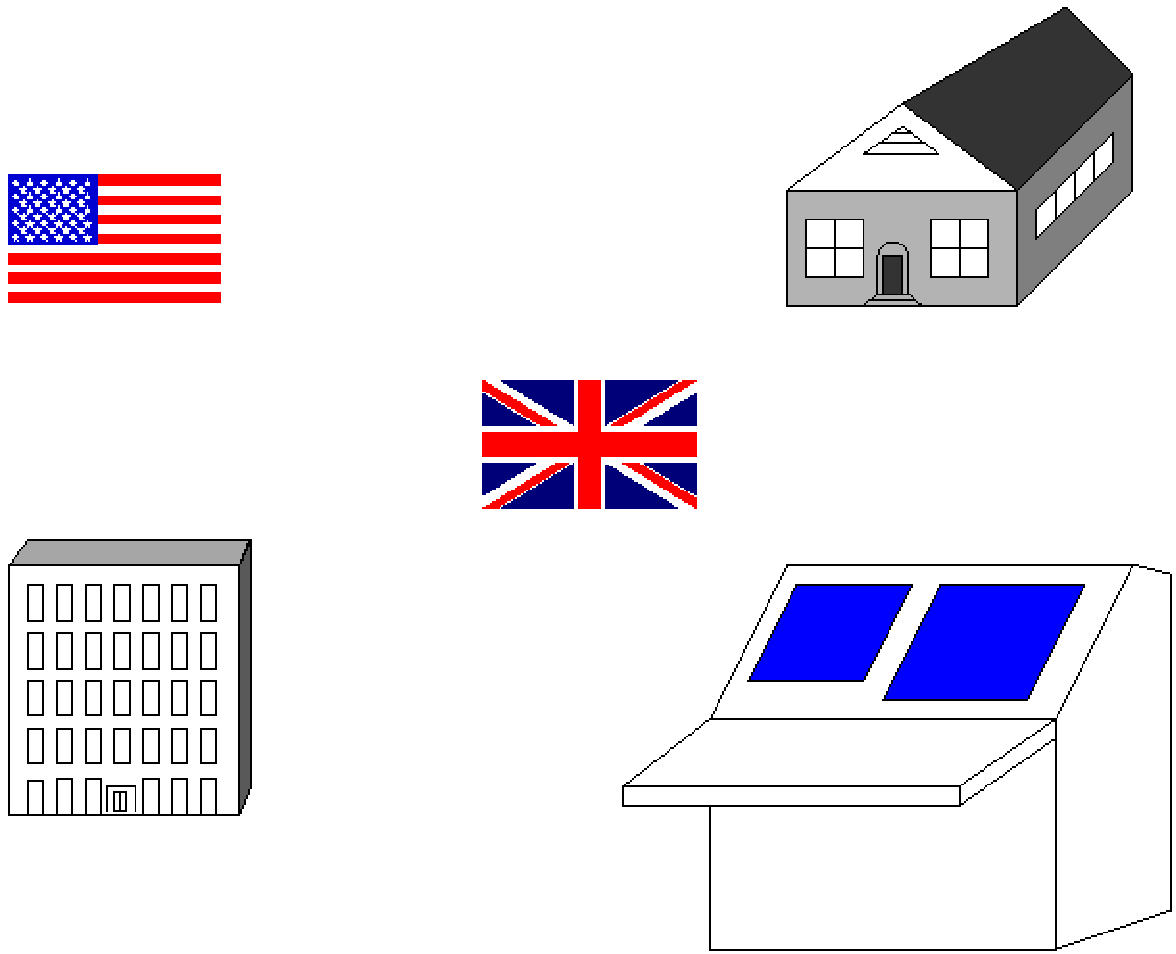
☞ Drawing lines, ellipses, splines, polygons, rectangles, arcs, and arrows.

☞ Entering text and arrows.

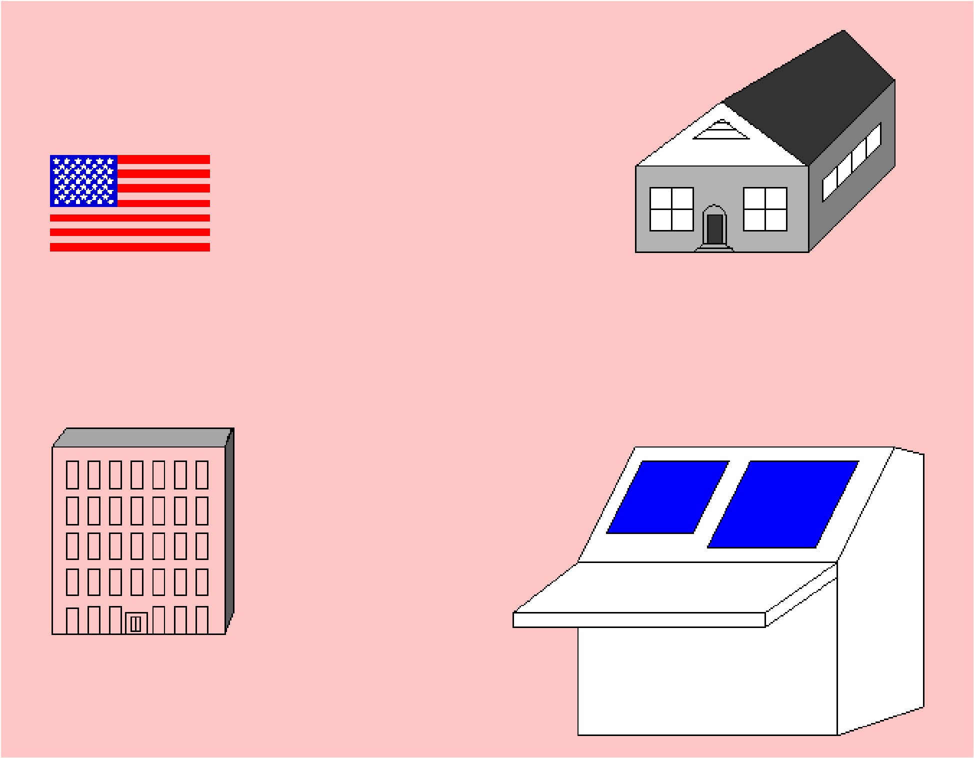
☞ Components can be scaled, moved, copied, deleted, flipped, rotated, and aggregated into larger components.

- ☞ A variety of line styles are supported.
- ☞ Libraries of icons are also supported.
- ☞ Items can also be floodfilled with colors or patterns.

`xfig` example



`xfig` example



Other `xfig` capabilities

- ☞ Can export into different formats (default is fig format, but in this slide presentation, the fig files were exported as png files), including LaTeX picture format, MetaPost, MetaFont, gif, encapsulated PostScript, Portable Document Format, png, and jpeg.
- ☞ Can use a grid to control placement ("snap to grid".)
- ☞ Can change the characteristics of existing objects.

☞ Can perform group operations on aggregations of objects.

xv, gimp, krita and inkscape

There are a number of programs to display or manipulate images. The program xv is one of the oldest; it has steadily gained features over the years.

Another is the gimp, which has as its strongest point manipulation, although many people have criticized its interface.

Recently krita has become quite popular. Like gimp, it also has its strongest manipulation of images.

A different kind of program is `inkscape`, which while it can take in an image graphic, its strong point is creating scalable vector graphics (SVG).

`spell` and `ispell`

The `spell` utility will check a file for spelling problems. It is usually just a script pointing to `aspell/ispell` running in batch mode.

The `aspell` program is a replacement from GNU for `ispell`. Its default mode is interactive. `aspell` is very featureful, and interfaces well with emacs.

Printing control with lpr/lprm/lpq

- `lpr` – The traditional BSD method of queuing print items to printers. Some popular options are:

```
-#NUM       a number of copies
-PQUEUE     specify a print queue by name
-p          run a formatter over the file before its printed so that p
```

- `lpq [-PQUEUE]` – Lets you look at the print jobs for a given queue QUEUE. It gives a job number for each that is useful for deleting items with `lprm`.

- `lprm [-PQUEUE] [-]` – Lets you remove items from a print queue. You can either specify job numbers (determined from `lpq`), or with just “-”, which removes all of your items from a queue.

pr

pr is a common formatter for print jobs that does various tasks, such as placing header/footer information such as page numbers and doublespacing.

Common options:

```
-W NUM       set page width to NUM
-l NUM       set page length to NUM
-h HEADER    specify header rather than the default, which is the filename
-d           doublespace output
-COLUMN      multicolumn output: print with COLUMN number of columns
-w NUM       set page width to NUM for multiple column output
```

a2ps

The program a2ps converts text files to PostScript. It allows you to do things such as printing multiple virtual pages on a single page.

For example:

```
a2ps --print-anyway yes -5 -o termcap.ps /etc/termcap
```

will reformat the /etc/termcap file to five pages per sheet.

Common options for `a2ps`

`-r`	`landscape mode`
`-f #`	`use font size #`
`-o OUT`	`write output to file name OUT rather than printing to ‘‘lpr’’`
`--columns N`	`N columns per page`
`-#`	`prints # pages per sheet of paper`

Building blocks for Unix power tools

Now that we have given a good overview of a lot of the better Unix tools, I want to take some time to talk about our toolset for building Unix programs.

The most important of these are the system calls.

Building blocks for Unix power tools

A Unix system call is a direct request to the kernel regarding a system resource. It might be a request for a file descriptor to manipulate a file, it might be a request to write to a file descriptor, or any of hundreds of possible operations.

These are exactly the tools that every Unix program is built upon.

File descriptor and file descriptor operations

In some sense, the mainstay operations are those on the file system.

File descriptor and file descriptor operations

Unlike many other resources which are just artifacts of the operating system and disappear at each reboot, changing a file system generally is an operation that has some permanence (although of course it is possible and even common to have “RAM” disk filesystems since they are quite fast, and for items that are meant to be temporary anyway, they are quite acceptable.)

Important file descriptor calls

A file descriptor is an int. It provides stateful access to an i/o resource such as a file on a filesystem, a pseudo-terminal, or a socket to a tcp session.

```
open()    -- create a new file descriptor to access a file
close()   -- deallocate a file descriptor
```

Important file descriptor calls

```
dup()      -- duplicate a file descriptor
dup2()     -- duplicate a file descriptor
```

Important file descriptor calls

```
fchmod()  -- change the permissions of a file associated with a file
          -- descriptor
fchown()  -- change the ownership of a file associated with a file
```

Important file descriptor calls

```
fcntl()   -- miscellaneous manipulation of file descriptors: dup(), set
          -- close on exec(), set to non-blocking, set to asynchronous
          -- mode, locks, signals
ioctl()   -- manipulate the underlying ‘‘device’’ parameters for
```

Important file descriptor calls

```
flock()   -- lock a file associated with a file descriptor
```

Important file descriptor calls

```
pipe()     -- create a one-way association between two file
           -- descriptors so that output from
           -- one goes to the input of the other
```

Important file descriptor calls

```
select()  -- multiplex on pending i/o to or from a set of file descriptor
```

Important file descriptor calls

```
read()    -- send data to a file descriptor
write()   -- take data from a file descriptor
```

Important file descriptor calls

```
readdir() -- raw read of directory entry from a file descriptor
```

Important file descriptor calls

```
fstat()   -- return information about a file associated with a fd: inode,
             perms, hard links, uid, gid, size, modtimes
fstatfs() -- return the mount information for the filesystem that the fil
          -- descriptor is associated with
```

Important filesystem operations

In addition to using the indirect means of file descriptors, Unix also offers a number of direct functions on files.

```
access()  -- returns a value indicating if a file is accessible
chmod()   -- changes the permissions on a file in a filesystem
chown()   -- changes the ownership of a file in a filesystem
```

Important filesystem operations

```
link()    -- create a hard link to a file
symlink() -- create a soft link to a file
```

Important filesystem operations

```
mkdir()    -- create a new directory
rmdir()    -- remove a directory
```

Important filesystem operations

```
stat()    -- return information about a file associated with a fd: inode,
             perms, hard links, uid, gid, size, modtimes
statfs()  -- return the mount information for the filesystem that the fil
          -- descriptor is associated with
```

Signals

```
alarm       -- set an alarm clock for a SIGALRM to be sent to a process
            -- time measured in seconds
getitimer   -- set an alarm clock in fractions of a second to deliver eit
            -- SIGALRM, SIGVTALRM, SIGPROF
```

Signals

```
kill        -- send an arbitrary signal to an arbitrary process
killpg      -- send an arbitrary signal to all processes in a process gro
```

Signals

```
sigaction   -- interpose a signal handler (can include special ‘‘default’
            -- ‘‘ignore’’ handlers)
sigprocmask -- change the list of blocked signals
```

Signals

```
wait        -- check for a signal (can be blocking or non-blocking) or ch
waitpid     -- check for a signal from a child process (can be general or
```

Modifying the current process's state

```
chdir        -- change the working directory for a process to dirname
fchdir       -- change the working directory for a process via fd
chroot       -- change the root filesystem for a process
```

Modifying the current process's state

```
execve      -- execute another binary in this current process
fork        -- create a new child process running the same binary
clone       -- allows the child to share execution context (unlike fork(2
exit        -- terminate the current process
```

Modifying the current process's state

```
getdtablesize  -- report how many file descriptors this process can have
               -- active simultaneously
```

Modifying the current process's state

```
getgid      -- return the group id of this process
getuid      -- return the user id of this process
getpgid     -- return process group id of this process
getpgrp     -- return process group's group of this process
```

Modifying the current process's state

```
getpid      -- return the process id of this process
getppid     -- return parent process id of this process
getrlimit   -- set a resource limit on this process (core size, cpu time,
            -- data size, stack size, and others)
getrusage   -- find amount of resource usage by this process
```

Modifying the current process’s state

```
nice          -- change the process’s priority
```

Networking

```
socket      -- create a file descriptor

bind        -- bind a file descriptor to an address, such a tcp port
listen      -- specify willingness for some number of connections to be
            -- blocked waiting on accept()
accept      -- tell a file descriptor block until there is a new connecti

connect     -- actively connect to listen()ing socket

setsockopt  -- set options on a given socket associated with fd, such out
            -- data, keep-alive information, congestion notification, fin
            -- and so forth (see man tcp(7))
getsockopt  -- retrieve information about options enabled for a given con

getpeername -- retrieve information about other side of a connection from
getsockname -- retrieve information this side of a connection from fd
```

Others

```
brk          -- allocate memory for the data segment for the
             -- current process

gethostname  -- gets a ''canonical'' hostname for the machine
gettimeofday -- gets the time of day for the whole machine
settimeofday -- sets the time of day for the whole machine
mount        -- attaches a filesystem to a directory and makes it availab
sync         -- flushes all filesystem buffers, forcing changed blocks to
             -- ''drives'' and updates superblocks
futex        -- raw locking (lets a process block waiting on a change
                to a specific memory location)
sysinfo      -- provides direct access from the kernel to:
                    load average
                    total ram for system
                    available ram
```

```
amount of shared memory existing
amount of memory used by buffers
total swap space
swap space available
number of processes currently in proctable
```

SYS V IPC

```
msgctl      -- SYS V messaging control (uid, gid, perms, size)
msgget      -- SYS V message queue creation/access
msgrcv      -- receive a SYS V message
msgsnd      -- send a SYS V message

shmat       -- attach memory location to SYS V shared memory segment
shmctl      -- SYS V shared memory control (uid, gid, perms, size, etc)
shmget      -- SYS V shared memory creation/access
shmdt       -- detach from SYS V shared memory segment
```

Numerical tools

There are a large number of tools available for Unix machines:

- ☞ Desktop tools such as `bc`, `dc`, and Pari/GP
- ☞ Computer Algebra Systems such as `maxima`
- ☞ Numerical tools library: GMP and Pari/GP
- ☞ Visualization via `gnuplot` and `graphviz`

bc **and** dc

bc is a calculator. Normally, it works with integers, but you can set it the number of decimal places with the `scale` variable:

```
[langley@sophie 2006-Fall]$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
1/6
0
scale=20
1/6
.16666666666666666666
```

bc

You can also do quick base conversions with bc:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
obase=16
ibase=10
16
10
quit
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
```

```
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
ibase=10
obase=16
15
F
quit
```

bc

bc uses traditional infix notation:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
12 + 34
46
12 * 34
408
34 / 12
2
99 - 12
87
```

```
56 % 14
0
3 ^ 3
27
```

bc

bc also allows small programs to be written:

```
a=0
while(a < 10)
{
  a = a+1;
  print a * a , "\n";
}

1
4
9
16
25
36
```

```
49
64
81
100
```

bc

bc supports the following statement types:

- ☞ Simple expressions, such as `3 * 5`
- ☞ Assignment, such `a = a - 1`
- ☞ `if/then`
- ☞ `while`
- ☞ Compound statements between `{ }`

- ☞ C-style `for`: `for(EXP1 ; EXP2 ; EXP3)`
- ☞ `break` and `continue`
- ☞ Function definition and return with `define` and `return`

bc

Math functions available when started with `-l`:

```
s(x)     # sine of x in radians
c(x)     # cosine of x in radians
a(x)     # arctangent of x in radians
l(x)     # natural logarithm of x
e(x)     # e to x
sqrt(x)  # square root of x (doesn't actually need -l option)
```

dc

The program dc is desk calculator much like bc in calculator mode, but is uses Reverse Polish Notation (RPN) rather than infix notation. Unlike bc, dc doesn't support complex statements and programming.

dc

```
[langley@sophie 2006-Fall]$ dc
34 99
f
99
34
55 88
f
88
55
99
34
+
*
*
f
```

```
481338
quit
```

dc

dc commands:

```
p     # print the top value from the stack
n     # print the top value from the stack and pop it off
f     # print the entire stack
+     # adds the top two values from the stack and pushes the result
-     # subtracts the first value on the stack from the second, pops them
      # off, and pushes the result
*     # pops top two values from stack, pushes multiplication result onto
/     # pops top two values from stack, pushes division result back on st
~     # pops top two values from stack, pushes both division and remainde
      # back on stack
```

GP/Pari

GP/Pari is a much featureful calculator than bc. It handles integers, reals, exact rationals, complex numbers, vectors, and more. It does modular arithmetic natively. It can some equation simplification, and it has a number of number theoretical functions such as `gcd()`.

GP/Pari

Starting GP/Pari at a shell prompt is easy:

```
$ gp
                          GP/PARI CALCULATOR Version 2.1.7 (released)
                        i686 running linux (ix86 kernel) 32-bit version
                        (readline v4.3 enabled, extended help available)
                                Copyright (C) 2002 The PARI Group
PARI/GP is free software, covered by the GNU General Public License, and comes WITHOUT A
Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.
   realprecision = 28 significant digits
   seriesprecision = 16 significant terms
   format = g0.28
parisize = 4000000, primelimit = 500000
? simplify((a+1)*(a-1))
%1 = a^2 - 1
? ??
```

You can also start it inside of Emacs with `M-x gp` if the appropriate pari.el file is available on your machine. The details are in the GP/Pari manual which you can pull up with `?? emacs`.

Using gp

gp also uses simple infix notation, like bc:

```
? 12 + 24
%2 = 36
?
```

Using gp

Notice that each result is numbered. You can use that notation to refer to a result:

```
? 12 + 24
%43 = 36
? %43 * 14
%44 = 504
?
```

(You can refer to just % for the previous result.)

Builtin functions in GP

There are a very large number of functions builtin to GP. You can them with ordinary prefix notation:

```
? gcd(10199869192881113131718912319123762991178912371711299102l7,
21986997157187511191111916059095111212170119l107)
%42 = 319
? factor(1001)
%3 =
[7 1]

[11 1]

[13 1]
```

```
? factor(540)
%45 =
[2 2]

[3 3]

[5 1]
?
```

Some useful builtin functions in GP

```
gcd       # greatest common divisor
factor    # factorization
simplify  # simplify a one-variable polynomial
```

Debugging

You can turn on copious debugging in GP with `\g20`:

```
? \g20
   debug = 20
? factor(1209401294012940192034901249012490124014212414124102411241111)
Miller-Rabin: testing base 1000288896
IFAC: cracking composite
        343388776245353031772655989810129300476076601488297 27
IFAC: checking for pure square
OddPwrs: is 3433887762453530317726559898101293004760766014882 9727
        ...a 3rd, 5th, or 7th power?
        modulo: resid. (remaining possibilities)
           211:   79   (3rd 1, 5th 0, 7th 0)
           209:   98   (3rd 0, 5th 0, 7th 0)
```

```
IFAC: trying Pollard-Brent rho method first
Rho: searching small factor of 175-bit integer
Rho: using Xˆ2-11 for up to 4770 rounds of 32 iterations
Rho: time =    100 ms,        768 rounds
Rho: fast forward phase (256 rounds of 64)...
Rho: time =     50 ms,        1028 rounds, back to normal mode
Rho: time =     30 ms,        1280 rounds
Rho: time =     40 ms,        1536 rounds
Rho: fast forward phase (512 rounds of 64)...
Rho: time =    120 ms,        2052 rounds, back to normal mode
Rho: time =     30 ms,        2304 rounds
Rho: time =     30 ms,        2560 rounds
Rho: time =     40 ms,        2816 rounds
Rho: time =     30 ms,        3072 rounds
Rho: fast forward phase (1024 rounds of 64)...
Rho: time =    230 ms,        4100 rounds, back to normal mode
Rho: time =     40 ms,        4352 rounds
Rho: time =     40 ms,        4608 rounds
Rho: time =     20 ms,        Pollard-Brent giving up.
IFAC: trying Shanks’ SQUFOF, will fail silently if input
      is too large for it.
```

```
IFAC: trying Lenstra-Montgomery ECM
ECM: working on 8 curves at a time; initializing for up to 3 rounds...
ECM: time =      0 ms
ECM: dsn =  4,        B1 =  700,        B2 =  77000,        gss =  128*42
ECM: time =    200 ms, B1 phase done, p = 701, setting up for B2
        (got [2]Q...[10]Q)
        (got [p]Q, p = 709 = 79 mod 210)
        (got initial helix)
ECM: time =     10 ms, entering B2 phase, p = 913
ECM: finishing curves 4...7
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 27799)
ECM: finishing curves 0...3
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 27799)
ECM: time =    140 ms
ECM: dsn =  6,        B1 =  900,        B2 =  99000,        gss =  128*42
ECM: time =    260 ms, B1 phase done, p = 907, setting up for B2
        (got [2]Q...[10]Q)
```

```
        (got [p]Q, p = 911 = 71 mod 210)
        (got initial helix)
ECM: time =      0 ms, entering B2 phase, p = 1117
ECM: finishing curves 4...7
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 28001)
        (giant step at p = 81761)
ECM: finishing curves 0...3
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 28001)
        (giant step at p = 81761)
ECM: time =    190 ms
ECM: dsn =  8,        B1 = 1150,        B2 = 126500,        gss =  128*42
ECM: time =    320 ms, B1 phase done, p = 1151, setting up for B2
        (got [2]Q...[10]Q)
        (got [p]Q, p = 1153 = 103 mod 210)
        (got initial helix)
ECM: time =     10 ms, entering B2 phase, p = 1361
ECM: finishing curves 4...7
```

```
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 28277)
        (giant step at p = 82003)
ECM: finishing curves 0...3
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
ECM: time =    110 ms,        p <=  28229,
        found factor = 31705445367881
IFAC: cofactor = 108305930498999029971801302679872746576 7
Miller-Rabin: testing base 768462011
Miller-Rabin: testing base 892785826
Miller-Rabin: testing base 739165157
Miller-Rabin: testing base 1874708212
Miller-Rabin: testing base 1732294655
Miller-Rabin: testing base 1648543222
Miller-Rabin: testing base 659912585
Miller-Rabin: testing base 370113064
Miller-Rabin: testing base 670592259
Miller-Rabin: testing base 481073162
IFAC: factor 1083059304989990299718013026798727465767
```

```
        is prime
Miller-Rabin: testing base 1340817133
Miller-Rabin: testing base 353959964
Miller-Rabin: testing base 1730244551
Miller-Rabin: testing base 1484512990
Miller-Rabin: testing base 1728249361
Miller-Rabin: testing base 22662352
Miller-Rabin: testing base 905839691
Miller-Rabin: testing base 2098523762
Miller-Rabin: testing base 1062164725
Miller-Rabin: testing base 1715475524
IFAC: factor 31705445367881
        is prime
IFAC: prime 31705445367881
        appears with exponent = 1
IFAC: main loop: 1 factor left
IFAC: prime 108305930498999029971801302679872746 5767
        appears with exponent = 1
IFAC: main loop: this was the last factor
IFAC: found 2 large prime (power) factors.
%4 =
```

[5441 1]

[6473 1]

[31705445367881 1]

[1083059304989990299718013026798727465767 1]

?

GP/Pari

Getting help is easy. The most comprehensive help comes from firing up the manual pages with `??`. You can choose a specific topic with `?? TOPIC` such as `?? gcd`.

Plotting with GP

You can also make simple plots with GP, such as

```
? ploth(t=0,Pi*2,[sin(t*17)*13,cos(t*52)],1)
%18 = [-12.99999286243945384, 12.99999286243945384, -0.999997803828127196
?
```

The final “1” indicates that this is plotted as a two-dimensional parametric function, i.e., the x coordinate is $x = sin(17t)$, and the y coordinate is $y = cos(52t)$.

Programming with GP

You can program inside of the gp shell. The basic control structures are

```
while(CONDITION,CODE)

if(CONDITION,THEN-CODE,ELSE-CODE)

for(VAR=A,B,CODE)

forstep(VAR=A,B,STEP,CODE)
```

Examples

```
? for(i=2,10,print(i*i%(i+10)))
4
9
2
10
4
15
10
5
0
```

Examples

```
? forstep(i=1,6,0.5,print(i))
1
1.500000000000000000000000000
2.000000000000000000000000000
2.500000000000000000000000000
3.000000000000000000000000000
3.500000000000000000000000000
4.000000000000000000000000000
4.500000000000000000000000000
5.000000000000000000000000000
5.500000000000000000000000000
6.000000000000000000000000000
?
```

Examples

```
? x = 0
%1 = 0
? while(x < 10, x = x+1; print(x))
1
2
3
4
5
6
7
8
9
10
?
```

Examples

```
? x = 10
  %1 = 10
? while(x > 0, if(x % 2 == 0, x = x / 2 , x = x + 7); print(x))
5
12
6
3
10
5
12
  [ ... ]
```

Defining functions

Function definition syntax:

```
NAME([ARG1, [ARG2, [...]]]) = local([ARG1, [ARG2, [...]]]) ; CODE
```

```
NAME([ARG1, [ARG2, [...]]]) =
{
  local([ARG1, [ARG2, [...]]]) ; CODE
}
```

Examples

```
/* long form */
? first_prime_div(x) =
{
  forprime(p=2,x,if(x % p == 0, return(p)))
}
? first_prime_div(35)
%19 = 5
?

/* short form */
? first_prime_div2(x) = forprime(p=2,x,if(x % p == 0, return(p)))
? first_prime_div2(161)
%20 = 7
```

GMP and pari library programming

Both GMP (Gnu Multi-Precision library) and Pari's library are powerful tools for C programming. Generally, GMP is not as featureful, but it sits very close to the metal. Pari gives you much wider range of basic types and functions on those types.

GMP programming

GMP has three basic types: floating point, integers, and rationals.

Functions are also divided by the same three classes.

GMP programming

The types are identified by the following naming convention:

```
mpz_t    #  type for integers
mpz_*    #  names for integer functions

mpf_t    #  type for floats
mpf_*    #  names for floating point functions

mpq_t    #  type for rationals
mpq_*    #  names for rational functions
```

Writing a GMP program

Writing a C program with GMP is easy if a bit tedious. First, you need to pull in the headers:

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>
```

Writing a GMP program

Next you declare variables:

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>

int main()
{
   mpz_t x, y;    // types are simple to use
}
```

Writing a GMP program

Now you **must** initialize any variables before use:

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>

int main()
{
   mpz_t x, y;

   mpz_init(x);   // critical, otherwise errors are unpredictable
   mpz_init(y);   //
}
```

Writing a GMP program

Compiling and linking is simple:

```
gcc -o prog prog.c -lgmp
```

Writing a GMP program

When creating a subroutine, make sure you clear the variables after you finish using them (despite the static declaration, that's just a pointer to the actual dynamically allocated memory for the variable):

Writing a GMP program

```
#include <unistd.h>    // or stdio.h and stdargs.h should work
#include <gmp.h>

void func()
{
   mpz_t x;
   mpf_t y;

   mpz_init(x);
   mpf_init(y);

   mpz_clear(x);      // otherwise you have a memory leak!
   mpf_clear(y);      //
   return;
}
```

Simple example program

```
#include <unistd.h>
#include <gmp.h>

char *answers[3]  =  { "composite", "probably prime", "prime" } ;

int main(int argc, char *argv[])
{
   int result;
   mpz_t n;
   mpz_init(n);
   mpz_set_str(n,argv[1],10);  // set the value of n from a string in base 10

   result = mpz_probab_prime_p(n,20);  // do a primality test with 20 repetitions
   gmp_printf("%Zd is %s\n",n,answers[result]);
}
```

Integer functions: assignment

```
void mpz_set (mpz_t result, mpz_t op)   # z = z
void mpz_set_ui (mpz_t result, unsigned long int op) # z = uint
void mpz_set_si (mpz_t result, signed long int op) # z = signed int
void mpz_set_d (mpz_t result, double op) # z = double
void mpz_set_q (mpz_t result, mpq_t op) # z = q  (via truncation)
void mpz_set_f (mpz_t result, mpf_t op) # z = f  (via truncation)

int mpz_set_str (mpz_t result, char *str, int base)
   # return 0 means string was completely a number
   # in the indicated base, -1 means that it wasn't

void mpz_swap (mpz_t result1, mpz_t result2) # swap two values
```

Integer functions: arithmetic

```
void mpz_add (mpz_t sum, mpz_t op1, mpz_t op2) # z = z + z
void mpz_add_ui (mpz_t sum, mpz_t op1, unsigned long int op2) # z = z + uint
void mpz_sub (mpz_t diff, mpz_t op1, mpz_t op2) # z = z - z
void mpz_sub_ui (mpz_t diff, mpz_t op1, unsigned long int op2) # z = z - unit
void mpz_ui_sub (mpz_t diff, unsigned long int op1, mpz_t op2) # z = uint - z
void mpz_mul (mpz_t result, mpz_t op1, mpz_t op2) # z = z * z
void mpz_mul_si (mpz_t result, mpz_t op1, long int op2) # z = z * signed int
void mpz_mul_ui (mpz_t result, mpz_t op1, unsigned long int op2) # z = z * uint
void mpz_neg (mpz_t result, mpz_t op) # z = -z
void mpz_abs (mpz_t result, mpz_t op) # z = |z|
```

Rational number functions: arithmetic

```
void mpq_add (mpq_t sum, mpq_t addend1, mpq_t addend2) # q = q + q
void mpq_sub (mpq_t difference, mpq_t minuend, mpq_t subtrahend) # q = q - q
void mpq_mul (mpq_t product, mpq_t multiplier, mpq_t multiplicand) # q = q * q
void mpq_div (mpq_t quotient, mpq_t dividend, mpq_t divisor) # q = q / q
void mpq_neg (mpq_t negation, mpq_t operand) # q = - q
void mpq_abs (mpq_t result, mpq_t op) # q = |q|
void mpq_inv (mpq_t inverted_number, mpq_t number) # q = 1 / q
```

Floating point functions: arithmetic

```
void mpf_add (mpf_t sum, mpf_t op1, mpf_t op2) # f = f + f
void mpf_add_ui (mpf_t sum, mpf_t op1, unsigned long int op2) # f = f + uint
void mpf_sub (mpf_t diff, mpf_t op1, mpf_t op2) # f = f - f
void mpf_ui_sub (mpf_t diff, unsigned long int op1, mpf_t op2) # f = uint - f
void mpf_sub_ui (mpf_t diff, mpf_t op1, unsigned long int op2) # f = f - uint
void mpf_mul (mpf_t result, mpf_t op1, mpf_t op2) # f = f * f
void mpf_mul_ui (mpf_t result, mpf_t op1, unsigned long int op2) # f = f *uint
void mpf_div (mpf_t result, mpf_t op1, mpf_t op2) # f = f / f
void mpf_ui_div (mpf_t result, unsigned long int op1, mpf_t op2) # f = uint / f
void mpf_div_ui (mpf_t result, mpf_t op1, unsigned long int op2) # f = f / uint
void mpf_sqrt (mpf_t root, mpf_t op) # f = sqrt(f)
void mpf_sqrt_ui (mpf_t root, unsigned long int op) # f = sqrt(uint)
void mpf_pow_ui (mpf_t result, mpf_t op1, unsigned long int op2) # f = f ˆ f
void mpf_neg (mpf_t negation, mpf_t op) # f = - f
void mpf_abs (mpf_t result, mpf_t op) # f = |f|
```

Comparison functions

```
int mpz_cmp (mpz_t op1, mpz_t op2) # returns negative if op1 < op2,
                                   # 0 if op1 == op2
                                   # positive if op1 > op2
int mpz_cmp_ui (mpz_t op1, unsigned long int op2) # same for uint
int mpf_cmp (mpf_t op1, mpf_t op2) # same for floats
int mpf_cmp_ui (mpf_t op1, unsigned long int op2) # same
int mpq_cmp (mpq_t op1, mpq_t op2) # same for rationals
int mpq_cmp_ui (mpq_t op1, unsigned long int num2, unsigned long int den2
```

Other useful functions

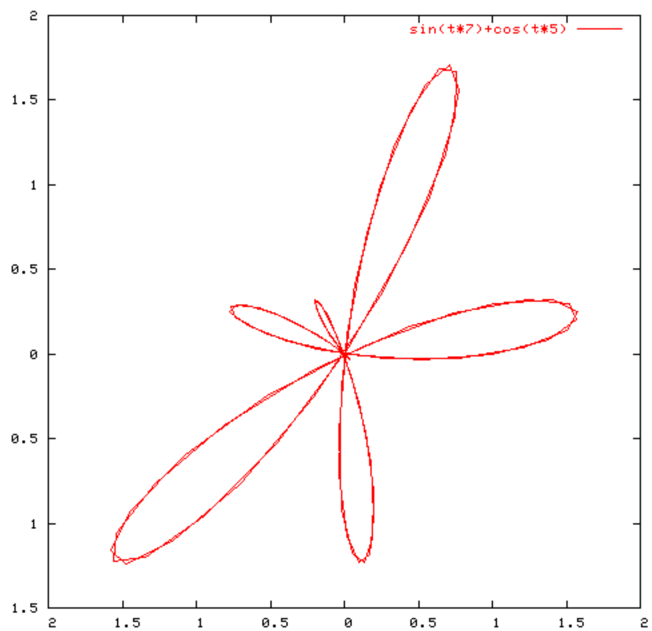
```
int mpz_probab_prime_p (mpz_t N, int repetitions)
          # returns 0 if N definitely composite
          # 1 if probably prime
          # 2 if definitely prime
void mpz_nextprime (mpz_t result, mpz_t N)
          # result is next prime greater than N
void mpz_gcd (mpz_t result, mpz_t op1, mpz_t op2)
          # result is GCD(op1,op2)
int mpz_jacobi (mpz_t a, mpz_t b)
          # jacobi (a/b)    Calculate the Jacobi symbol (a/b). This is defined only for
int mpz_legendre (mpz_t a, mpz_t p)
          # legendre (a/p)
```

Other useful functions

```
unsigned long int mpz_remove (mpz_t result, mpz_t op, mpz_t f)
          # result = divide out all of a given factor f from op
void mpz_fac_ui (mpz_t result, unsigned long int op)
          # result = op!
void mpz_bin_ui (mpz_t rop, mpz_t n, unsigned long int k)
          # computes the binomial coefficient n over k
void mpz_fib_ui (mpz_t fn, unsigned long int n)
          # computes the nth Fibonacci number
```

Gnuplot for plotting

The program `gnuplot` allows you to plot functions and data:



Running `gnuplot`

Most options for running `gnuplot` are invoked from inside `gnuplot`'s shell, so just

```
% gnuplot
```

is enough to get you started.

The basic plotting commands

- ☞ `plot` → operates either in rectangular or polar/paramet coordinates
- ☞ `splot` → lets you plot surfaces and contours
- ☞ `replot` → lets you redo a plot, such as when you change devices

Plotting functions

The basic command to plot a function of one variable is

```
gnuplot> plot f(x)
```

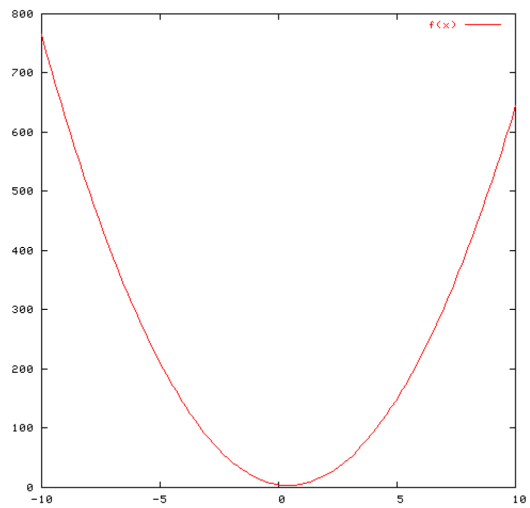
Functions

where $f(x)$ can be user defined or any of the standard math library functions:

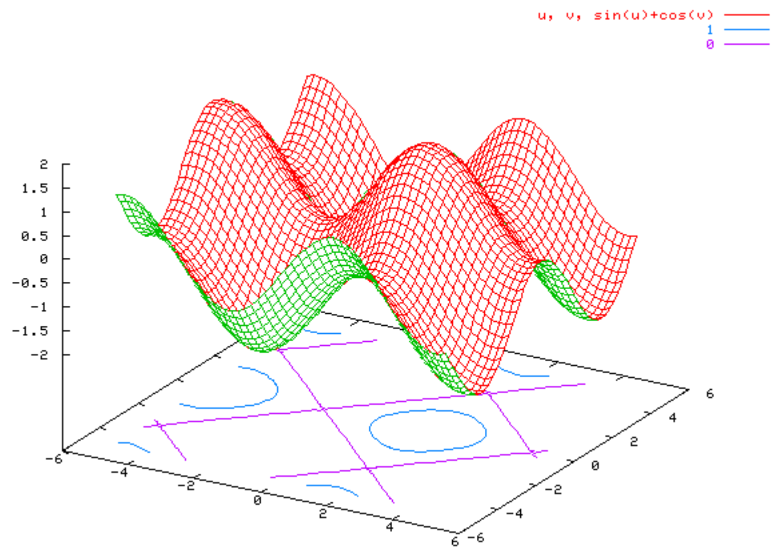
`abs`	`acos`	`acosh`	`arg`
`asin`	`asinh`	`atan`	`atan2`
`atanh`	`besj0`	`besj1`	`besy0`
`besy1`	`ceil`	`column`	`cos`
`cosh`	`erf`	`erfc`	`exp`
`floor`	`gamma`	`ibeta`	`igamma`
`imag`	`int`	`inverf`	`invnorm`
`lgamma`	`log`	`log10`	`norm`
`rand`	`real`	`sgn`	`sin`
`sinh`	`sqrt`	`tan`	`tanh`

Examples of a simple function

```
gnuplot> f(x) = f(x) = 5 + (-6 + 7*x) * x
gnuplot> plot f(x)
```



Example of surfaces and contours



Example of surfaces and contours

```
gnuplot> set parametric        # so we can specify u and v
gnuplot> set hidden3d          # nice looking mode
gnuplot> set contour base      # draw a base projection also
gnuplot> set isosamples 50,50    # lots of sampling
gnuplot> splot u,v,sin(u)+cos(v) # make the plot
```

Network tools: `ssh`

Unix is rich in tools for network connectivity.

One of the most useful is `ssh`. It allows one to execute commands on a remote machine, either one at a time or in a "login" session. Unlike its predecessors `telnet`, `rsh`, and `rlogin`, it provides a secure session, with both encryption for the session and improved authentication security.

ssh

The general form:

```
ssh [-i IDENTITYFILE] [-p PORT] [-x|-X] HOSTNAME | USER@HOSTAME [COMMAND]
```

ssh

If you just specify the hostname, the username will default to your current one. If you specify a command, it will be executed rather than creating a general login shell.

Using -x turns off X11 forwarding. Using -X allows you to forward X11 windows via the encrypted session you are using.

Setting up keys

The general invocation for ssh-keygen is:

```
ssh-keygen -t [dsa|rsa]
```

Setting up keys

For example:

```
[.ssh]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/langley/.ssh/id_rsa): id_rsa3
Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in id_rsa3.
Your public key has been saved in id_rsa3.pub.
The key fingerprint is:
5d:be:5e:50:ab:75:a6:54:bc:16:6e:65:07:9e:ea:f5 langley@machine.cs.fsu.edu
```

Setting up keys

The contents of the resulting ".pub" file are added to the public keys kept in the remote machine's `$HOME/authorized_keys` file:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAq33Tkj7QM68HVK17QB
do8CeyFSTj20Wz89JAJYp4eKD8qDFbDlXg/ngurjIqsuEGRuueIX5Q
h7Re84AaNJdJABYSzZytGR0klO8FFXkBpFEL4bli6ygPAa/vq4cyDV
djmy5S9dulr6afFk/2x3ac4nOgC7LtPSiMh1
UF+N8vpPk= langley@machine.cs.fsu.edu
```

Setting up keys

Once you have added the .pub file to the `authorized_keys` on the remote machine, you need to make sure that you have the corresponding private key in your local `.ssh` subdirectory.

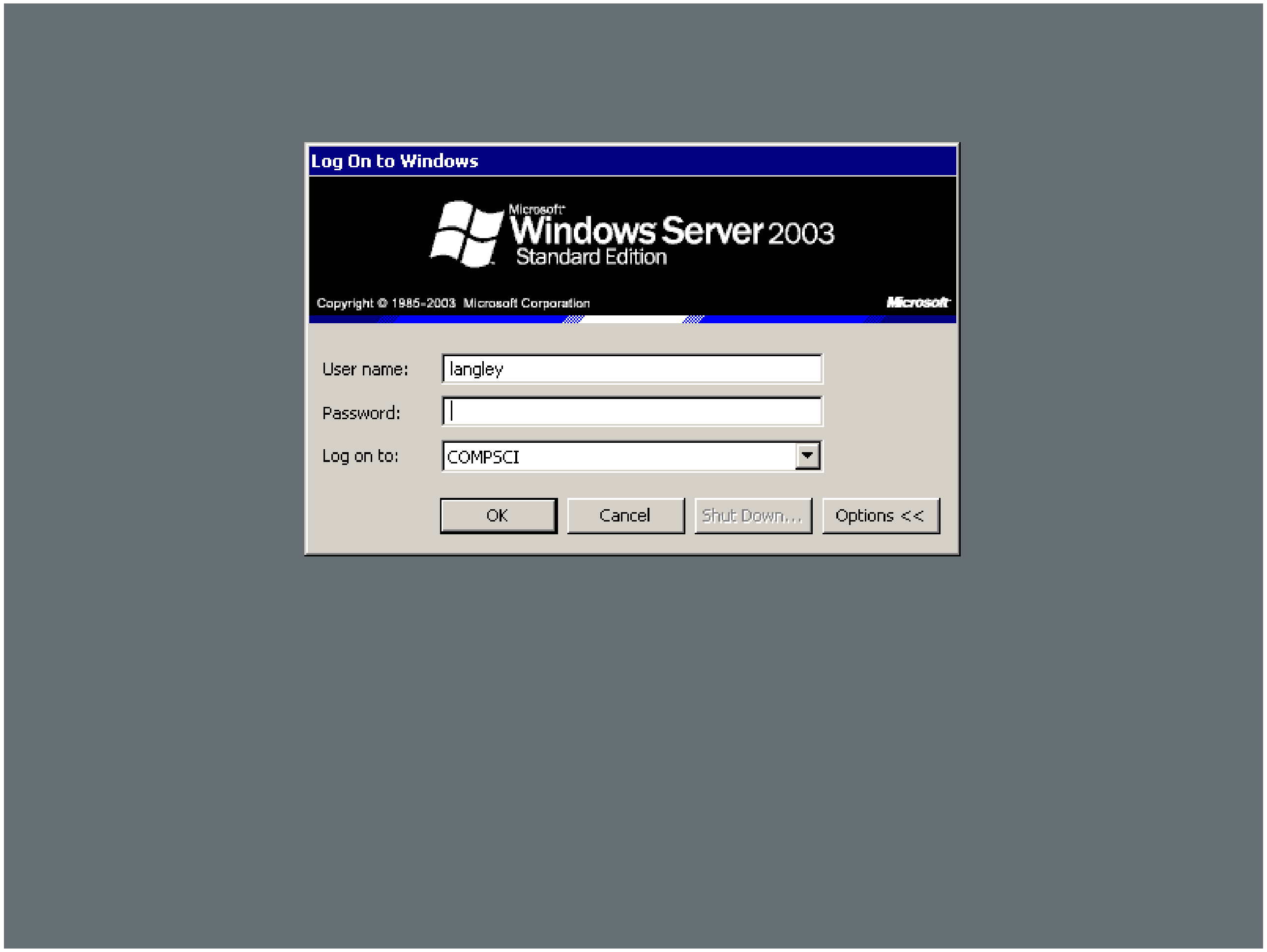
By default, the filenames `id_dsa` and `id_rsa` are used. If you want to login with a private key in a different file, just use the -i option:

```
[.ssh]$ ssh -i id_rsa3 langley@machine.cs.fsu.edu
```

Network tools: `rdesktop`

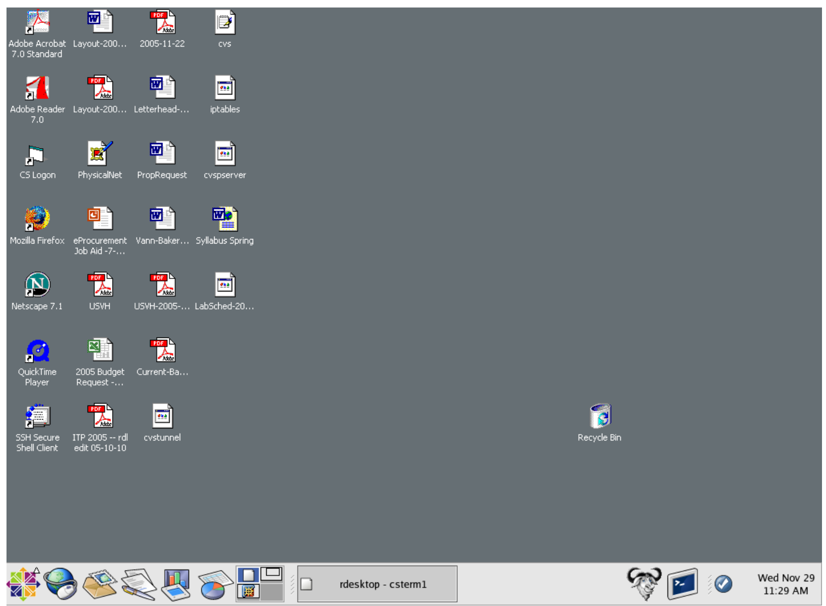
You can, if necessary, access Windows machines running terminal services (or remote desktop) via `rdesktop`.

Network tools: `rdesktop`



Running `rdesktop`

`rdesktop [-f] HOSTNAME`



The -f option puts you in fullscreen mode (`CTRL-ALT-ENTER` to shift back).

Network tools: `ftp`

`ftp` is an older interactive method of transferring files. It is still useful occasionally, though since it is insecure it should only be run within a safely sheltered environment.

Invocation:

```
ftp [-p] HOSTNAME
```

The option -p is not found on every version of `ftp` (modern versions of `ftp` default to this mode), but when it is, it allows you to specify passive mode for data transfers, which can help you use `ftp` going through firewalls.

`ftp` commands

```
cd RDIR               # chdir on the remote machine to RDIR
lcd LDIR              # chdir on the local machine to LDIR
dir [RDIR]            # get a directory of the remote directory RDIR (defaults to .)
get RNAME [LNAME]     # get a single file RNAME from the remote machine, using
                      # LNAME as the local name if specified
put LNAME [RNAME]     # put a single file LNAME from the local machine to the remote
                      # machine, using RNAME as the remote name if specified
mget RNAMEPATTERN     # get multiple files fitting RNAMEPATTERN (expansion is done
                      # remotely)
mput LNAMEPATTERN     # put multiple files fitting LNAMEPATTERN (expansion is done
                      #locally)
hash                  # show a hash mark every time 1k is sent or received
del                   # delete a remote file
mdel RNAMEPATTERN     # delete remote files fitting pattern (expansion is done remotely)
quit                  # exit ftp
![CMD]                # if no CMD is given, start a shell; otherwise, execute the CMD
                      # locally
```

Sending file trees

The easiest way to send a file tree with `ftp` is to use `tar` first, and then `ftp` the tarfile. For example:

```
[2006-Fall]$ tar cfz /tmp/somedir.tgz somedir
[2006-Fall]$ ftp ftp.redhat.com
Connected to ftp.redhat.com.
220 Red Hat FTP server ready. All transfers are logged. (FTP) [no EPSV]
Name (ftp.redhat.com:ftp): ftp
331 Please specify the password.
Password:langley@ftp
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> lcd /tmp
Local directory now /tmp
ftp> put somedir.tgz
```

telnet

Like `ftp`, telnet is an older, insecure program which should be avoided outside of secure environments unless you are using it in a situation for where security is not relevant, such as testing a mail server.

Invoking:

```
telnet HOSTNAME [PORT]
```

Using `telnet`

One of the most useful ways to still use `telnet` is for testing mail servers:

```
[2006-Fall]$ telnet mail.cs.fsu.edu 25
Trying 128.186.120.4...
Connected to mail.cs.fsu.edu (128.186.120.4).
Escape character is ’ˆ]’.
220 mail.cs.fsu.edu ESMTP Postfix
helo machine.cs.fsu.edu
250 mail.cs.fsu.edu
mail from: <langley@cs.fsu.edu>
250 Ok
rcpt to: <langley@cs.fsu.edu>
250 Ok
data
354 End data with <CR><LF>.<CR><LF>
Subject: This is a test

This message is a test message.


.
250 Ok: queued as B01E3F2F50
quit
221 Bye
Connection closed by foreign host.
```

The `r` family

The "r" programs `rlogin`, `rsh`, and `rcp` should all be avoided these days since the "s" programs `ssh` and `scp` are more than adequate replacements.

Web browsers, email clients

There are a large number of web browsers and email clients available on Unix machines.

The traditional line-oriented email client is `mail`; two more recent ones are `pine` and `elm`.

mail

```
[2006-Fall]$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/spool/mail/langley": 2 messages 2 new
>N  1 root@machine.cs.fsu.e  Thu Oct 20 15:54  16/630   "test456"
 N  2 root@machine.cs.fsu.e  Thu Oct 20 15:54  16/627   "test"
& x
```

`mail`

The `mail` program is very lightweight, and you can quickly read mail messages using it.

If you use “q” to quit, the state of your message box will be updated to indicate things such as whether or not you have read a message. If you use “x”, the message box is not updated.

`elm` and `pine`: deprecated

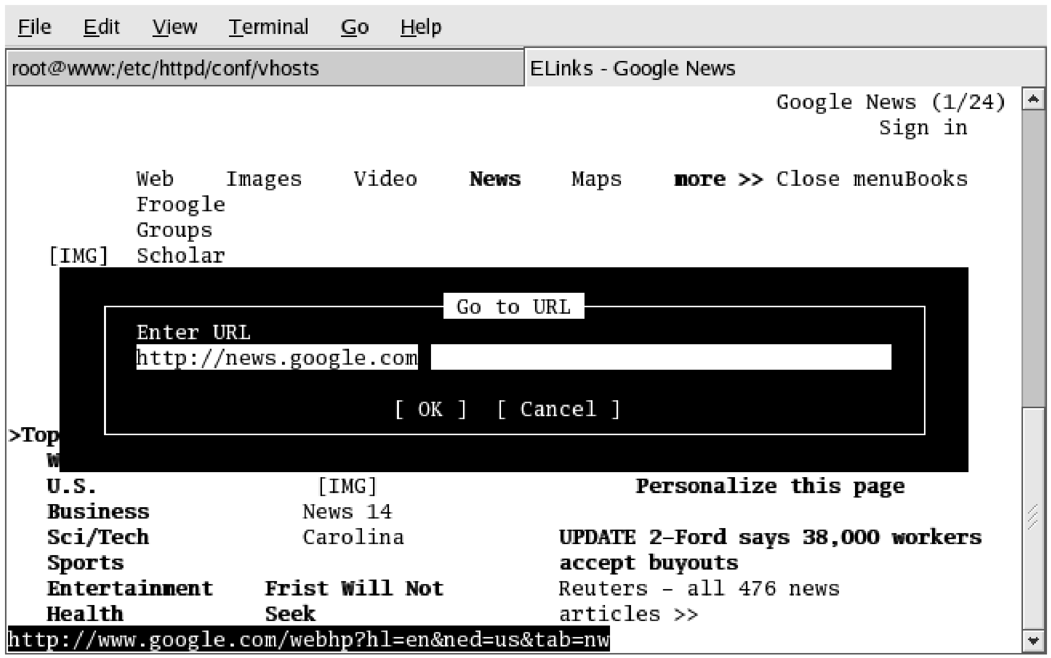
Both `elm` and `pine` are designed as “screen” mailers rather than just a line mailer.

While some people prefer them, they lack many features that other mailers have: `mail` is fast and lightweight, and graphic mailers generally are able to handle `imap` and `pop`, which makes handling multiple mailboxes uniformly very simple.

links (a.k.a. lynx or elinks)

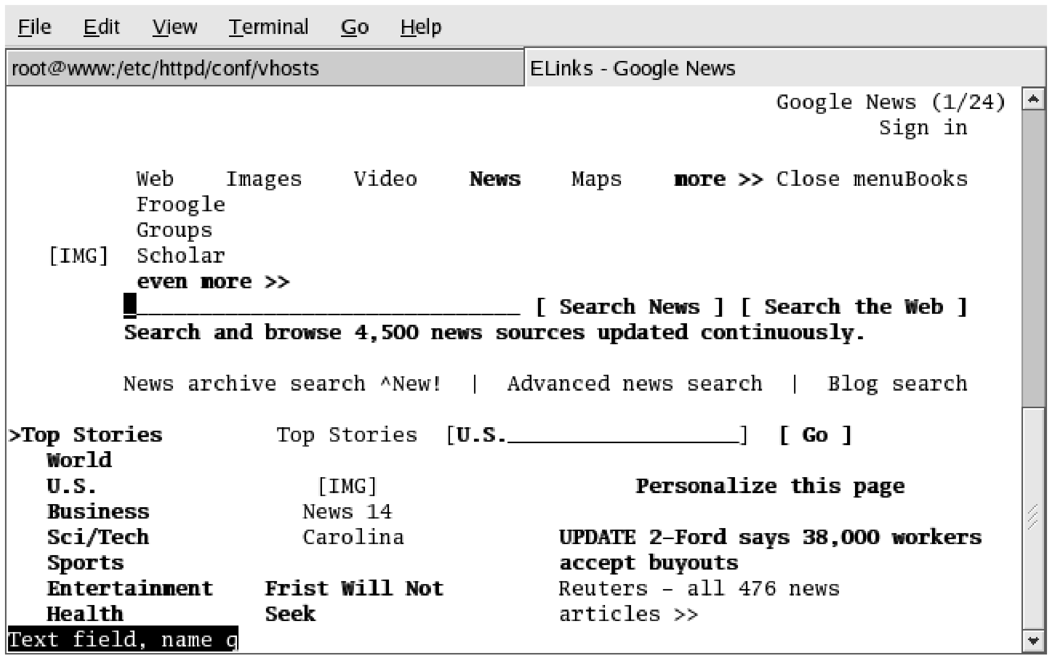
The program links is a nice screen-based webbrowser. While it doesn't handle such as things as flash very well, it is a very responsive webbrowser.

`links` (a.k.a. `lynx` or `elinks`)



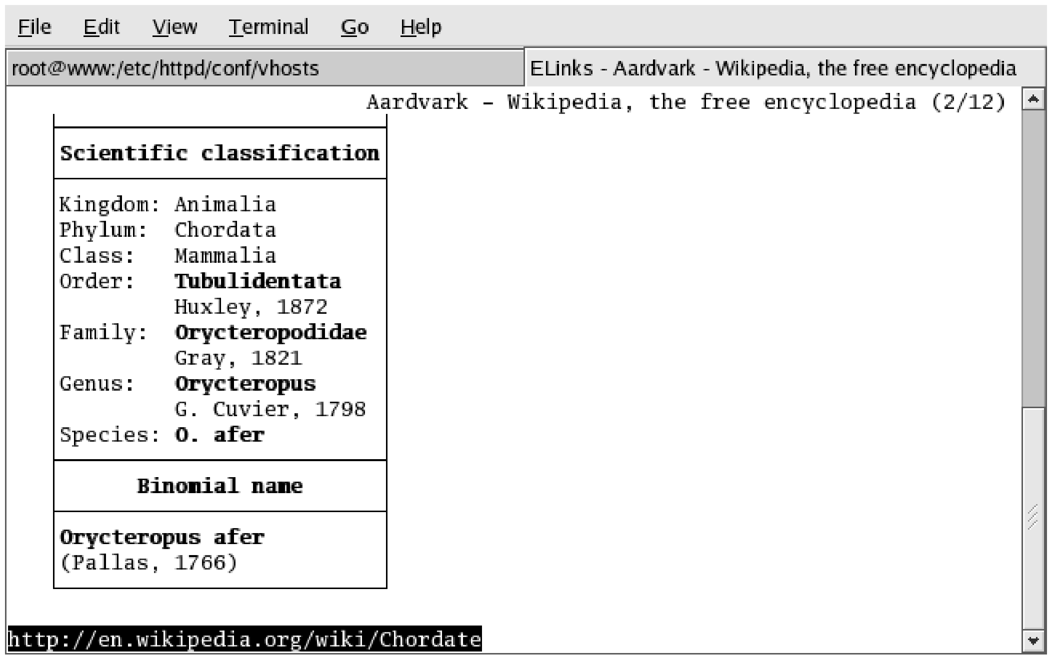
Using the “g” command

`links` (a.k.a. `lynx` or `elinks`)



A typical web page rendered in `links`.

`links` (a.k.a. `lynx` or `elinks`)



A typical web page rendered in `links`.

Default keybindings in `links`

`PageDown`	`page down`
`" "`	`page down`
`PageUp`	`page up`
`b`	`page up`
`Down`	`down`
`Up`	`up`
`Ctrl-C`	`copy clipboard`
`Ctrl-P`	`scroll up`
`Ctrl-N`	`scroll down`
`[`	`scroll left`
`]`	`scroll right`
`Home`	`home`
`Ctrl-A`	`home`
`Ctrl-E`	`end`
`Enter`	`enter`
`Left`	`back`
`d`	`download`
`/`	`search`
`?`	`search back`

n	find-next
Ctrl-R	reload
g	goto url
a	add bookmark
s	bookmark manager
q	quit

Graphic webbrowsing and email

You can now run a variety of graphic webbrowsers and email clients in many Unix/Linux environments.

Browsers:

```
epiphany
firefox
galeon
konqueror
mozilla
```

Graphic webbrowsing and email

Email clients:

```
evolution
mozilla mail
thunderbird
xmail
```

(Another popular option with email is to use a webbrowser reader, such as `squirrelmail` or `openwebmail`.)

Graphic webbrowsing and email

Most graphic email clients can gracefully handle multiple mailboxes on multiple servers. One of the easiest ways to do this is via `imap`, which allows you to leave the mail on the server rather than the `pop` paradigm of pulling it to the local machine.

dd

The dd program is a surprisingly powerful one. It can be used for everything from copying a disk partition to converting ASCII files to EBCDIC.

dd **conversions**

```
ascii   # from EBCDIC to ASCII

ebcdic  # from ASCII to EBCDIC

ibm     # from ASCII to alternated EBCDIC

lcase   # change upper case to lower case

ucase   # change lower case to upper case

swab    # swap every pair of input bytes
```

dd copying

Copying raw block-structured devices is quite easy:

```
dd if=/dev/hda1 of=/dev/hda2
```

dd other tricks

You can also remove bytes from the beginning or the end of a file:

```
dd bs=1 skip=4000    # skip over the first 4000 characters

dd count=10000 bs=1  # copy only the first 10000 characters
```

csplit

`csplit` (context split) lets you split a file by specifying a pattern for each split point.

```
csplit /PATTERN/ /PATTERN/|COUNT
```

csplit

For instance, say you want to split the /etc/termcap file into 1200 separate definitions.

You can easily do this with the single line:

```
csplit /etc/termcap '/^[a-z]/' '{*}'   # the second item is a repeat coun
```

csplit

You can then get 1300+ files, such as

```
[langley@sophie tmp]$ head -1000 xx*
==> xx01 <==
dumb|80-column dumb tty:\
        :am:\
        :co#80:\
        :bl=^G:cr=^M:do=^J:sf=^J:

==> xx02 <==
unknown|unknown terminal type:\
        :gn:tc=dumb:

==> xx03 <==
lpr|printer|line printer:\
        :bs:hc:os:\
        :co#132:li#66:\
        :bl=^G:cr=^M:do=^J:ff=^L:le=^H:sf=^J:
```

```
==> xx04 <==
glasstty|classic glass tty interpreting ASCII control characters:\
        :am:bs:\
        :co#80:\
        :bl=^G:cl=^L:cr=^M:do=^J:kd=^J:kl=^H:le=^H:nw=^M^J:ta=^I:


==> xx05 <==
vanilla:\
        :bs:\
        :bl=^G:cr=^M:do=^J:sf=^J:

==> xx06 <==
ansi+local1:\
        :do=\E[B:le=\E[D:nd=\E[C:up=\E[A:

==> xx07 <==
ansi+local:\
        :DO=\E[%dB:LE=\E[%dD:RI=\E[%dC:UP=\E[%dA:tc=ansi+local1:

==> xx08 <==
ansi+tabs:\
        :bt=\E[Z:ct=\E[2g:st=\EH:ta=^I:
```

```
==> xx09 <==
ansi+inittabs:\
        :it#8:tc=ansi+tabs:

==> xx10 <==
ansi+erase:\
        :cd=\E[J:ce=\E[K:cl=\E[H\E[J:

==> xx100 <==
arm100|arm100-am|Arm(RiscPC) ncurses compatible (for 640x480):\
        :am:ms:ut:xn:xo:\
        :co#80:it#8:li#30:\
        :@8=\E[M:DO=\E[%dB:K1=\E[q:K2=\E[r:K3=\E[s:K4=\E[p:K5=\E[n:\
        :LE=\E[%dD:RA=\E[?7l:RI=\E[%dC:SA=\E[?7h:UP=\E[%dA:\
        :ac=‘‘aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}˜˜:\
        :ae=ˆO:as=ˆN:bl=ˆG:cb=\E[1K:cd=\E[J:ce=\E[K:cl=\E[H\E[J:\
        :cm=\E[%i%d;%dH:cr=ˆM:cs=\E[%i%d;%dr:ct=\E[3g:do=ˆJ:\
        :eA=\E(B\E)0:ho=\E[H:k0=\E[y:k1=\E[P:k2=\E[Q:k3=\E[R:\
        :k4=\E[S:k5=\E[t:k6=\E[u:k7=\E[v:k8=\E[l:k9=\E[w:k;=\E[x:\
        :kb=ˆH:kd=\E[B:ke=\E[?1l\E>:kl=\E[D:kr=\E[C:ks=\E[?1h\E=:\
        :ku=\E[A:le=ˆH:mb=\E[5m:md=\E[1m:me=\E[m\017:mk=\E[8m:\
        :mr=\E[7m:nd=\E[C:rc=\E8:\
        :rs=\E>\E[?3l\E[?4l\E[?5l\E[?7h\E[?8h:\
        :..sa=\E[0%?%p1%p6%|%t;1%;%?%p2%t;4%;%?%p1%p3%|%t;7%;%?%p4%t;5%;%?%p7%t;8%;m%?%p
        :sc=\E7:se=\E[m:sf=ˆJ:so=\E[7m:sr=\EM:st=\EH:ta=ˆI:ue=\E[m:\
```

```
        :up=\E[A:us=\E[4m:tc=ecma+sgr:tc=klone+color:

==> xx1000 <==
ncr260wy60wpp|NCR 2900_260 wyse 60 wide mode:\
        :co#132:\
        :cm=\Ea%i%dR%dC:\
        :is=\Ee6\E~4\E+\Ed/\Ee1\Ed*\Er\EO\E‘1\E‘;\E‘@\E~!\E"\Ee4\Ex@\E‘9\Ee7:\
        :rs=\Ee6\E~4\E+\Ed/\Ee1\Ed*\Er\EO\E‘1\E‘;\E‘@\E~!\E"\Ee4\Ex@\E‘9\Ee7:\
        :tc=ncr260wy60pp:
```

`csplit`

Alternatively, you can also just specify arbitray line numbers:

```
% csplit /etc/termcap 4 10 110
110
107
5023
734959
```

Portable anymaps

Way back, there was a package called “PBM”, the Portable BitMap package. It allowed you to convert files of many different graphic types to other types, and it allowed you to manipulate these files from the command line.

For instance, when I did the window dumps for some of the lectures, I used this package something along these lines:

```
sleep 10 ; xwd > /tmp/xwd.1
```

```
xwdtopnm < /tmp/xwd.1 | pnmtopng > /tmp/rdesktop01.png
```

The conversions

```
# PNM conversions
giftopnm    # GIF to pnm
rasttopnm   # Sun rasterfile to pnm
tifftopnm   # tiff to pnm
xwdtopnm    # X window dump format to pnm

pnmtotiff   # pnm to tiff
pnmtoxwd    # pnm to xwd
pnmtorast   # pnm to Sun rasterfile
pnmtops     # convert to postscript


# PPM conversions
gouldtoppm  # Gould scanner file to ppm
ilbmtoppm   # Amiga format to ppm
```

```
ppmtogif    # gif to ppm
pgmtoppm    # convert pgm to PPM (convert grayscale to color)
```

Manipulations

```
ppmdither    # dither a file (reduce the number of colors used)
ppmdepth     # change the number of planes in an image
ppmquant     # reduce the number of colors used in a file
ppmquantall  # run ppmquant over many files so they share common colormap
ppmforge     # create fractal forgeries of clouds, stars, and planets
pnmcrop      # crop borders from an image
pnmcut       # extract arbitrary rectangle from an image
pnmarith     # add, subtract, multiply, abs(diff) two images
pnmenlarge   # enlarge an image by integer factor
pnmscale     # arbitrary resize an image
pbmreduce    # reduce image by integer factor
pnmsmooth    # smooth a picture (useful after resizing)
pnmfile      # describe file’s image characteristics
pnmflip      # flip an image
pnmgamma
```

`ppmforge` fun

Image generated with ppmforge

```
ppmforge -stars 100 -night -width 200 -height 200  | pnmtopng > /tmp/xyz.png
```