${Unix_Tools}

Markus Kuhn

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

http://www.cl.cam.ac.uk/Teaching/2003/UnixTools/

Michaelmas 2003 – Part Ib

A brief history of Unix

→ "First Edition" developed at AT&T Bell Labs during 1968–71 by Ken Thompson and Dennis Ritchie for a PDP 11

→ Rewritten in C in 1973

→ Sixth Edition (1975) first widely available version

→ Seventh Edition in 1979, UNIX 32V for VAX

→ During 1980s independent continued development at AT&T ("System V Unix") and Berkeley University ("BSD Unix")

→ Commercial variants (Solaris, SCO, HP/UX, AIX, IRIX, . . .)

→ IEEE and ISO standardisation of a *Portable Operating System Interface based on Unix (POSIX)* in 1989, later also *Single Unix Specification* by X/Open, both merged in 2001

   The POSIX standard is freely available online: http://www.unix.org/version3/

A brief history of free Unix

→ In 1983, Richard Stallman (MIT) initiates a free reimplementation of Unix called GNU ("GNU's Not Unix") leading to an editor (emacs), compiler (gcc), debugger (gdb), and numerous other tools.

→ In 1991, Linus Torvalds (Helsinki CS undergraduate) starts development of a free POSIX-compatible kernel, later nicknamed *Linux*, which was rapidly complemented by existing GNU tools and contributions from volunteers to form a full Unix replacement.

→ Berkeley University releases a free version of BSD Unix in 1991 after removing remaining proprietary AT&T code. Volunteer projects emerge to continue its development (FreeBSD, NetBSD, OpenBSD).

Free software license concepts

→ **public domain:** authors waive all copyright

→ **"MIT/BSD" licences:** allow you to copy, redistribute and modify the software in any way as long as

   • you respect the identity and rights of the author (preserve copyright notice and licence terms in source code and documentation)

   • you agree not sue the author over software quality (accept exclusion of liability and warranty)

→ **GNU General Public Licence:** requires in addition that

   • any modifications are again covered by the GPL and must be made publicly available as source code

Numerous refinements of these licences have been written. More information on the various types and their philosophies is collected, for example, on http://www.opensource.org/.

Why do we teach Unix Tools?

$\longrightarrow$ Second most popular OS family (after Microsoft Windows)

$\longrightarrow$ Many elements of Unix have became part of common computer science folklore, terminology & tradition over the past 20 years and influenced many other systems (including DOS/Windows)

$\longrightarrow$ Many Unix tools have been ported and become popular on other platforms

$\longrightarrow$ Your future project supervisors and employers are likely to expect you to be fluent under Unix as a development environment

$\longrightarrow$ Good examples for high-functionality user interfaces

This short lecture course can only give you a first overview. You need to spend at least 2–3 times as many hours with e.g. PWF Linux to

$\longrightarrow$ explore the tools mentioned

$\longrightarrow$ solve exercises (which often involve reading documentation to understand important details skipped in the lecture)

Original Unix user interfaces

The initial I/O devices were teletype terminals . . .

Photo: Bell Labs

. . . and later video display terminals such as the DEC VT100, all providing 80 characters-per-line fixed-width ASCII output. Their communications protocol is still used today in graphical windowing environments via "terminal emulator" programs such as xterm.

The VT100 was the first video terminal with microprocessor, and the first to implement the ANSI X3.64 (= ECMA-48) control functions. For instance, "$^{E_{SC}}$[7m" activates inverse mode and "$^{E_{SC}}$[0m" returns to normal, where $^{E_{SC}}$ is the ASCII control character encoded by byte 27.

```
http://www.vt100.net/
http://www.cs.utk.edu/~shuford/terminal/dec.html
http://www.ecma-international.org/publications/standards/Ecma-048.htm
```

Unix tools design philosophy

$\longrightarrow$ Compact and concise input syntax, making full use of ASCII repertoire to minimise keystrokes

$\longrightarrow$ Output format should be simple and easily usable as input for other programs

$\longrightarrow$ Programs can be joined together in "pipes" and "scripts" to solve more complex problems

$\longrightarrow$ Each tool originally performed a simple single function

$\longrightarrow$ Prefer reusing existing tools with minor extension to rewriting a new tool from scratch

$\longrightarrow$ The main user-interface software ("shell") is a normal replaceable program without special privileges

$\longrightarrow$ Support for automating routine tasks

Unix documentation

Most Unix documentation can be read from the command line.
Classic manual sections: user commands (1), system calls (2), library functions (3), devices (4), file formats (5).

→ The man tool searches for the manual page file (→ $MANPATH) and activates two further tools (nroff text formatter and more text-file viewer). Add optional section number to disambiguate:

```
$ man 3 printf     # C subroutine, not command
```

Honesty in documentation: Unix manual pages traditionally include a BUGS section.

→ xman: X11 GUI variant, offers a table of contents

→ info: alternative GNU hypertext documentation system
Invoke with info from the shell of with C-h i from emacs. Use M(enu) key to select topic or [Enter] to select hyperlink under cursor, N(ext)/P(rev)/U(p)/D(irectory) to navigate document tree, Emacs search function (Ctrl-S), and finally Q(uit).

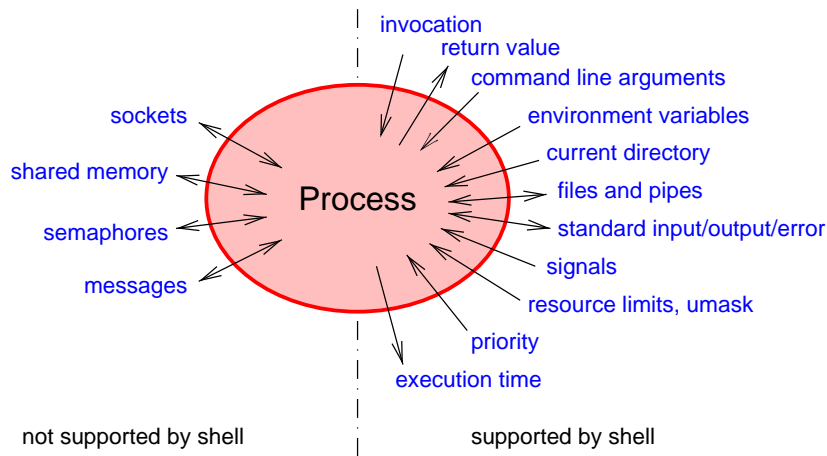→ Check /usr/share/doc/ and Web for further documentation.

Examples of Unix tools

man, apropos, xman, info
  help/documentation browser
more, less
  plaintext file viewer
ls, find
  list/traverse directories, search
cp, mv, rm, touch, ln
  copy, move/rename, remove, renew
  files, link/shortcut files
mkdir, rmdir
  make/remove directories
cat, dd, head, tail
  concatenate/split files
du, df, quota, rquota
  examine disk space used and free
ps, top, free, uptime, w
  process table and system load
vi, emacs, pico
  interactive editors

cc, gcc
  C compilers
make
  project builder
cmp, diff, patch
  compare files, apply patches
sccs, rcs, cvs
  revision control systems
adb, gdb
  debuggers
awk, perl, python, tcl
  scripting languages
m4, cpp
  macro processors
sed, tr
  edit streams, replace characters
sort, grep, cut
  sort/search lines of text, extract
  columns

nroff, troff, tex, latex
  text formatters
mail, pine, mh, exmh, elm
  electronic mail user agents
telnet, ftp, rlogin, finger,
talk, ping, traceroute,
wget, ssh, scp, hostname,
host, ifconfig, route
  network tools
xterm
  VT100 terminal emulator
tar, cpio, compress, zip,
gzip, bzip2
  file packaging and compression
echo, cd, pushd, popd, exit,
ulimit, time, history
  builtin shell commands
fg, bg, jobs, kill
  builtin shell job control
date, xclock
  clocks

which, whereis
  locate command file
clear, reset
  clear screen, reset terminal
stty
  configure terminal driver
xv, display, ghostview,
acroread
  graphics file viewers
xfig, tgif, gimp
  graphics drawing tools
topnm, pnmto, [cd]jpeg
  graphics format converters
passwd
  change your password
chmod
  change file permissions
lex, yacc, flex, bison
  scanner/parser generators

The Unix shell

→ The user program that Unix starts automatically after a login

→ Allows the user to interactively start, stop, suspend, and resume other programs and control the access of programs to the terminal

→ Supports automation by executing files of commands ("shell scripts"), provides programming language constructs (variables, string expressions, conditional branches, loops, concurrency)

→ Simplifies file selection via keyboard (regular expressions, file name completion)

→ Simplifies entry of command arguments with editing and history functions

→ Most common shell ("sh") developed 1975 by Stephen Bourne, modern GNU replacement is "bash" ("Born Again SHell")

Unix inter-process communication mechanisms

invocation
return value
command line arguments
environment variables
current directory
files and pipes
standard input/output/error
signals
resource limits, umask
priority
execution time

sockets
shared memory
semaphores
messages

Process

not supported by shell | supported by shell

Command line arguments, return value, environment variables

A Unix C program is invoked by calling its `main()` function with:

$\longrightarrow$ a list of strings `argv` as an argument

$\longrightarrow$ a list of strings `environ` as a predefined global variable

```
#include <stdio.h>
extern char **environ;

int main(int argc, char **argv)
{
  int i;
  printf("Command line arguments:\n");
  for (i = 0; i < argc; i++)
    puts(argv[i]);
  printf("Environment:\n");
  for (i = 0; environ[i] != NULL; i++)
    puts(environ[i]);
  return 0;
}
```

Environment strings have the form

$$name = value$$

where $name$ is free of "=".

Argument `argv[0]` is usually the name or path of the program.

Convention: `main()` == 0 signals success, other values signal errors to calling process.

File descriptors

Unix processes access files in three steps:

$\longrightarrow$ Provide kernel in `open()` or `creat()` system call a path name and get in return an integer "file descriptor".

$\longrightarrow$ Provide in `read()`, `write()`, and `seek()` system calls an opened file descriptor along with data.

$\longrightarrow$ Finally, call `close()` to release any data structures associated with an opened file (position pointer, buffers, etc.).

The `lsof` tool lists the files currently opened by any process. Under Linux, file descriptor lists and other kernel data can be accessed via the simulated file system mounted under /proc.

As a convention, the shell opens three file descriptors for each process:

$\longrightarrow$ 0 = standard input (for reading the data to be processed)

$\longrightarrow$ 1 = standard output (for the resulting output data)

$\longrightarrow$ 2 = standard error (for error messages)

Basic shell notations

Start a program and connect the three default file descriptors stdin, stdout, and stderr to the terminal:

```
$ command
```

Connect stdout of command1 to stdin of command2 and stdout of command2 to stdin of command3 by forming a pipe:

```
$ command1 | command2 | command3
```

Also connects terminal to stdin of command1, to stdout of command3, and to stderr of all three.

Note how this function concatenation notation makes the addition of command arguments somewhat clearer compared to the mathematical notation command3(command2(command1(arg1), arg2), arg3):

```
$ ls -la | sort -n -k5 | less
```

Execute several commands or entire pipes in sequence:

```
$ command1 ; command2 ; command3
```

For example:

```
$ date ; host linux2
Mon Sep 29 23:52:31 BST 2003
linux2.pwf.cl.cam.ac.uk has address 193.60.95.68
```

Conditional execution depending on success of previous command (as in logic expression short-cut):

```
$ make ftest && ./ftest
$ ./ftest || echo 'Test failed!'
```

Return value 0 for success is interpreted as Boolean value "true", other return values for problems or failure as "false". The trivial tools true and false simply return 0 and 1, respectively.

File redirecting

Send stdout to file

```
$ command >filename
```

Append stdout to file

```
$ command >>filename
```

Send both stdout and stderr to the same file (first redirect stdout to file, then redirect stderr to where stdout goes):

```
$ command >filename 2>&1
```

Feed stdin from file

```
$ command <filename
```

Open other file descriptors for input or output

```
$ command 0<fin 1>fout 2>>log 3<auxin 4>auxout
```

"Here Documents" allow us to insert data into shell scripts directly such that the shell will feed it into a command via standard input. The << is followed immediately by an end-of-text marker string.

```
$ tr <<THEEND A-MN-Za-mn-z N-ZA-Mn-za-m
> Orsber cbfgvat n cbffvoyl ehqr wbxr be fcbvyre gb
> HFRARG, fpenzoyr vg jvgu n Pnrfne pvcure gung
> ebgngrf gur nycunorg ol 13 punenpgref. Cersvk gur
> grkg jvgu n jneavat, gb znxr vg yrff yvxryl gung
> fbzrbar ernqf vg nppvqragnyyl jvgubhg orvat jnearq.
> THEEND
```

Command-line argument conventions

Each program receives from the caller as a parameter an array of strings (argv). The shell places into the argv parameters the words entered following the command name, after several preprocessing steps have been applied first.

Command options are by convention single letters prefixed by a hyphen ("-h"). Unless followed by option parameters, single character flag options can often be concatenated:

```
$ ls -l -a -t
$ ls -lat
```

GNU tools offer alternatively long option names prefixed by two hyphens ("--help"). Arguments not starting with hyphens are typically filenames, hostnames, URLs, etc.

The special option "`--`" signals in many tools that subsequent words are arguments, not options. This provides one way to access filenames starting with a hyphen:

```
$ rm -- -i
$ rm ./-i
```

The special filename "`-`" signals often that standard input/output should be used instead of a file.

All these are conventions that most – but not all – tools implement (usually via the `getopt` library function), so check the respective manual first.

The shell remains ignorant of these "`-`" conventions!

Shell command-line preprocessing

A number of punctuation characters in a command line are part of the shell control syntax

```
| & ; ( ) < >
```

or can trigger special convenience substitutions before `argv` is handed over to the called program:

⟶ brace expansion: `{,}`

⟶ tilde expansion: `~`

⟶ parameter expansion: `$`

⟶ pathname expansion / filename matching: `* ? []`

⟶ quote removal: `\ ' "`

Brace expansion

Provides for convenient entry of words with repeated substrings:

```
$ echo a{b,c,d}e
abe ace ade
$ echo {mgk25,fapp2,rja14}@cam.ac.uk
mgk25@cam.ac.uk fapp2@cam.ac.uk rja14@cam.ac.uk
$ rm slides.{bak,aux,dvi,log,ps}
```

Tilde expansion

Provides convenient entry of home directory pathname:

```
$ echo ~pb ~/Mail/inbox
/home/pb /homes/mgk25/Mail/inbox
```

The builtin `echo` command simply outputs `argv` to stdout and is useful for demonstrating command-line expansion and for single-line text output in scripts.

Parameter and command expansion

Substituted with the values of shell variables

```
$ OBJFILE=skipjack.o
$ echo ${OBJFILE} ${OBJFILE%.o}.c
skipjack.o skipjack.c
$ echo ${HOME} ${PATH} ${LOGNAME}
/homes/mgk25 /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin mgk25
```

or the standard output lines of commands

```
$ which emacs
/usr/bin/emacs
$ echo $(which emacs)
/usr/bin/emacs
$ ls -l $(which emacs)
-rwxr-xr-x    2 root      system    3471896 Mar 16  2001 /usr/bin/emacs
```

Shorter alternatives: variables without braces and command substitution with grave accent (`` ` ``) or, with older fonts, back quote (`'`)

```
$ echo $OBJFILE
skipjack.o
$ echo `which emacs`
/usr/bin/emacs
```

Pathname expansion

Command-line arguments containing ?, *, or [...] are interpreted as regular expression patterns and will be substituted with a list of all matching filenames.

$\longrightarrow$? stands for an arbitrary single character

$\longrightarrow$ * stands for an arbitrary sequence of zero or more characters

$\longrightarrow$ [...] stands for one character out of a specified set. Use "−" to specify range of characters and "ˆ" to complement set. Certain character classes can be named within [:...:].

None of the above will match a dot at the start of a filename, which is the naming convention for hidden files.
Examples:

```
*.bak [A-Za-z]*.??? [[:alpha:]]* [^A-Z] .??* files/*/*.o
```

Quote removal

Three quotation mechanisms are available to enter the special characters in command-line arguments without triggering the corresponding shell substitution:

$\longrightarrow$ '...' suppresses all special character meanings

$\longrightarrow$ "..." suppresses all special character meanings, except for

$ \ `

$\longrightarrow$ \ suppresses all special character meanings for the immediately following character

Example:

```
$ echo '$$$' "* * * $HOME * * *" \$HOME
$$$ * * * /homes/mgk25 * * * $HOME
```

The bash extension $'...' provides access to the full C string quoting syntax. For example $'\x1b' is the ASCII ESC character.

Exercise 1 Write a shell command line that appends :/usr/X11R6/bin to the end of the environment variable $MANPATH.

Exercise 2 Create a new subdirectory and in it five files with unusual filenames that someone unfamiliar with the shell will find difficult to remove. Ask a fellow student to write down for each file the command line that will remove it.

Exercise 3 Given a large set of daily logfiles with date-dependent names of the form log.*yyyymmdd*, write down the shortest possible command line that concatenates all files from 1 October 1999 to 7 July 2002 into a single file archive in chronological order.

Exercise 4 Write down the command line that appends the current date and time (in Universal Time) and the Internet name of the current host to the logfile for the respective current day (local time), using the above logfile naming convention.

Review – what happened so far

$\longrightarrow$ Some historic and philosophical background on Unix

$\longrightarrow$ Inter-process communication facilities

$\longrightarrow$ Where to find documentation
(man, info, /usr/share/doc, -h/--help, Web)

$\longrightarrow$ Unix shell: substitutable central user interface, configuration mechanism, and automation "glue" to connect applications

$\longrightarrow$ piping, file redirection

$\longrightarrow$ command-line meta-characters: |&;()<>{}[]~$*?\'"

$\longrightarrow$ variables

$\longrightarrow$ pitfalls with unusual filenames

Job control

Start command or entire pipe as a background job, without connecting stdin to terminal:

```
$ command &
[1] 4739
$ ./testrun 2>&1 | gzip -9c >results.gz &
[2] 4741
$ ./testrun1 & ./testrun2 & ./testrun3 &
[3] 5106
[4] 5107
[5] 5108
```

Shell prints both a job number (identifying all processes in pipe) as well as process ID of last process in pipe. Shell will list all its jobs with the `jobs` command, where a + sign marks the last stopped (default) job.

Foreground job: Stdin connected to terminal, shell prompt delayed until process exits, keyboard signals delivered to this single job.

Background job: Stdin disconnected (read attempt will suspend job), next shell prompt appears immediately, keyboard signals not delivered, shell prints notification when job terminates.

Keyboard signals: (keys can be changed with `stty` tool)

$\longrightarrow$ Ctrl-C "intr" (SIGINT=2) by default aborts process

$\longrightarrow$ Ctrl-\ "quit" (SIGQUIT=3) aborts process with core dump

$\longrightarrow$ Ctrl-Z "susp" (SIGSTOP=19) suspends process

Another important signal (not available via keyboard):

$\longrightarrow$ SIGKILL=9 destroys process immediately

Job control commands:

$\longrightarrow$ `fg` resumes suspended job in foreground

$\longrightarrow$ `bg` resumes suspended job in background

$\longrightarrow$ `kill` sends signal to job or process

Job control commands accept as arguments

$\longrightarrow$ process ID

$\longrightarrow$ % + job number

$\longrightarrow$ % + command name

Examples:

```
$ ghostview                      # press Ctrl-Z
[6]+  Stopped                 ghostview
$ bg
$ kill %6
```

A few more job control hints

$\longrightarrow$ `kill -9 ...` sends SIGKILL to process. Should only be used as a last resort, if a normal kill (which sends SIGINT) failed, otherwise program has no chance to clean up resources before it terminates.

$\longrightarrow$ The `jobs` command shows only jobs of the current shell, while `ps` and `top` list entire process table. Options for `ps` differ significantly between System V and BSD derivatives, check man pages.

$\longrightarrow$ `fg %-` or just `%-` runs previously stopped job in foreground, which allows you to switch between several programs conveniently.

Shell variables

Serve both as variables (of type string) in shell programming as well as environment variables for communication with programs.

Set *variable* to *value*:

```
variable=value
```

Note: No whitespace before or after "=" allowed.

Make variable visible to called programs:

```
export variable
export variable=value
```

Modify environment variables for one command only:

```
variable1=value variable2=value command
```

"set" shows all shell variables
"printenv" shows all (exported) environment variables.

Some important environment variables

$\longrightarrow$ $HOME — Your home directory, also available as "~".

$\longrightarrow$ $LOGNAME — Your login name.

$\longrightarrow$ $PATH — Colon separated list of directories in which shell looks for commands (e.g., "/bin:/usr/bin:/usr/X11R6/bin").
Should never contain ".", at least not at beginning. Why?

$\longrightarrow$ $LANG, $LC_* — Your "locale", the name of a system-wide configuration file with information about your character set and language/country conventions (e.g., "en_GB.UTF-8"). $LC_* sets locale only for one category, e.g. $LC_CTYPE for character set and $LC_COLLATE for sorting order; $LANG sets default for everything. "locale -a" lists all available locales.

$\longrightarrow$ $TZ — Specification of your timezone (mainly for remote users)

$\longrightarrow$ $OLDPWD — Previous working directory, also available as "~-".

$\longrightarrow$ $PS1 — The normal command prompt, e.g.

```
$ PS1='\[\033[7m\]\u@\h:\W \!\$\[\033[m\] '
mgk25@shep:unixtools 12$
```

$\longrightarrow$ $PRINTER — The default printer for lpr, lpq and lprm.

$\longrightarrow$ $TERM — The terminal type (usually xterm or vt100).

$\longrightarrow$ $PAGER/$EDITOR — The default pager/editor (usually less and emacs, respectively).

$\longrightarrow$ $DISPLAY — The X server that X clients shall use.

Executable files and scripts

Many files signal their format in the first few "magic" bytes of the file content (e.g., 0x7f,'E','L','F' signals the System V *Executable and Linkable Format*, which is also used by Linux and Solaris).

The "file" tool identifies hundreds of file formats and some parameters based on a database of these "magic" bytes:

```
$ file $(which ls)
/bin/ls: ELF 32-bit LSB executable, Intel 80386
```

The kernel recognizes files starting with the magic bytes "#!" as "scripts" that are intended for processing by the interpreter named in the rest of the line, e.g. a bash script starts with

```
#!/bin/bash
```

If the kernel does not recognize a command file format, the shell will interpret each line of it, therefore, the "#!" is optional for shell scripts.

Use "chmod +x file" and "./file", or "bash file".

Shell compound commands

A *list* is a sequence of one or more pipelines separated by ";", "&", "&&" or "||", and optionally terminated by one of ";", "&" or end-of-line. The return value of a list is that of the last command executed.

⟶ (*list*) executes list in a subshell

⟶ { *list* ; } groups a list (to override operator priorities)

⟶ for *variable* in *words* ; do *list* ; done

Expands *words* like command-line arguments, assigns one at a time to the *variable*, and executes *list* for each. Example:

```
for f in *.txt ; do cp $f $f.bak ; done
```

⟶ if *list* ; then *list* ; elif *list* ; then *list* ; else *list* ; fi

⟶ while *list* ; do *list* ; done
   until *list* ; do *list* ; done

⟶ case *word* in
      pattern|*pattern*|...) *list* ;;
      ...
   esac

Matches expanded *word* against each *pattern* in turn (same matching rules as pathname expansion) and executes the corresponding *list* when first match is found. Example:

```
case "$command" in
  start)
    app_server &
    processid=$! ;;
  stop)
    kill $processid ;;
  *)
    echo 'unknown command' ;;
esac
```

The first *list* in the if, while and until commands is interpreted as a Boolean condition. The true and false commands return 0 and 1 respectively (note the inverse logic compared to Boolean values in C!).

The builtin command "test *expr*", which can also be written as "[*expr*]" evaluates simple Boolean expressions on files, such as

-e *file*	is true if *file* exists.
-d *file*	is true if *file* exists and is a directory.
-f *file*	is true if *file* exists and is a normal file.
-r *file*	is true if *file* exists and is readable.
-w *file*	is true if *file* exists and is writable.
-x *file*	is true if *file* exists and is executable.

or strings, such as

string1 == *string2*	*string1* < *string2*
string1 != *string2*	*string1* > *string2*

Examples:

```
if [ -e $HOME/.rhosts ] ; then
  echo 'Found ~/.rhosts!' | \
  mail $LOGNAME -s 'Hacker backdoor?'
fi
```

Note: A backslash at the end of a command line causes end-of-line to be ignored.

```
if [ "`hostname`" == python.cl.cam.ac.uk ] ; then
  ( sleep 10 ; play ~/sounds/greeting.wav ) &
else
  xmessage 'Good Morning, Dave!' &
fi
[ "`arch`" != ix86 ] || { clear ; echo "I'm a PC" ; }
```

Aliases and functions

Aliases allow a string to be substituted for the first word of a command:

```
$ alias dir='ls -la'
$ dir
```

Shell functions are defined with "*name* () { *list* ; }". In the function body, the command-line arguments are available as $1, $2, $3, etc. The variable $* contains all arguments and $# their number.

```
$ unalias dir
$ dir () { ls -la $* ; }
```

Outside the body of a function definition, the variables $*, $#, $1, $2, $3, ... can be used to access the command-line arguments passed to a shell script.

Shell history

The shell records commands entered. These can be accessed in various ways to save keystrokes:

$\longrightarrow$ "`history`" outputs all recently entered commands.

$\longrightarrow$ "`!`*n*" is substituted by the *n*-th history entry.

$\longrightarrow$ "`!!`" and "`!-1`" are equivalent to the previous command.

$\longrightarrow$ "`!*`" is the previous command line minus the first word.

$\longrightarrow$ Use cursor up/down keys to access history list, modify a previous command and reissue it by pressing Return.

$\longrightarrow$ Type Ctrl-O instead of Return to issue command from history and edit its successor, which allows convenient repetition of entire command sequences.

$\longrightarrow$ Type Ctrl-R to search string in history.

Most others probably only useful for teletype writers without cursor.

Readline

Interactive bash reads commands via the `readline` line-editor library. Many Emacs-like control key sequences are supported, such as:

$\longrightarrow$ Ctrl-A/Ctrl-E moves cursor to st**a**rt/**e**nd of line

$\longrightarrow$ Ctrl-K deletes (**k**ills) the rest of the line

$\longrightarrow$ Ctrl-D **d**eletes the character under the cursor

$\longrightarrow$ Ctrl-W deletes a **w**ord (first letter to cursor)

$\longrightarrow$ Ctrl-Y inserts deleted strings

$\longrightarrow$ ESC ^ performs history expansion on current line

$\longrightarrow$ ESC # turns current line into a comment

Automatic word completion: Type the "Tab" key, and bash will complete the word you started when it is an existing $variable, ~user, hostname, command or filename, depending on the context. If there is an ambiguity, pressing "Tab" a second time will show list of choices.

Startup files for terminal access

When you log in via a terminal line or telnet/rlogin/ssh:

$\longrightarrow$ After verifying your password, the `login` command checks `/etc/passwd` to find out what shell to start for you.

$\longrightarrow$ As a login shell, bash will execute the scripts

```
/etc/profile
~/.profile
```

The second one is where you can define your own environment. Use it to set exported variable values and trigger any activity that you want to happen at each login.

$\longrightarrow$ Any subsequently started bash will read `~/.bashrc` instead, which is where you can define functions and aliases, which – unlike environment variables – are not exported to subshells.

Startup files for X Window System access

The "X server" provides access to display, keyboard and mouse for "X client" applications via the "X11 protocol".

Before login, the only client is the X Display Manager (xdm).

After login, xdm will start the script /usr/lib/X11/xdm/Xsession. That invokes the "X clients" (xterm, etc.) that run on your desktop by default. If ~/.xsession exists, this script will be called instead.

Most X clients in Xsession or ~/.xsession are started in background, except for the last one, which is usually a window manager (twm, fvwm2, KDE, etc.). When this last client terminates, and with it the Xsession script, then xdm will reset the X server. This will terminate all X clients and the user is logged out.

You can configure your login screen in ~/.xsession. You can also configure default parameters for many X clients via the xrdb command. See "man X" for details.

Typical `.xsession` file

```
#!/bin/bash
. ~/.profile

# set X defaults and keymaps
userresources=~/.Xdefaults
usermodmap=~/.Xmodmap
if [ -f $userresources ]; then
    xrdb $userresources
fi
if [ -f $usermodmap ]; then
    xmodmap $usermodmap
fi

# start some X clients as background processes
xterm -geometry 80x10+10+5 -C -title "`hostname -s` console" \
      -bg lightgreen &
xclock -geometry 80x80+0-0 -update 1 &
xload -geometry 80x80+90-0 -nolabel &

# start window manager as foreground process
if [ -x /usr/bin/X11/fvwm2 ] ; then
  /usr/bin/X11/fvwm2
else
  twm
fi
```

Exercise 5 Configure your PWF-Linux account, such that each time you log in, an email gets sent automatically to your Hermes mailbox. It should contain in the subject line the name of the machine on which the reported login took place, as well as the time of day. In the message body, you should add a greeting followed by the output of the "w" command that shows who else is currently using this machine.

Exercise 6 Explain what happens if the command "rm *" is executed in a subdirectory that contains a file named "-i".

Exercise 7 Write a shell script "start_terminal" that starts a new "xterm" process and appends its process ID to the file ~/.terminal.pids. If the environment variable $TERMINAL has a value, then its content shall name the command to be started instead of "xterm".

Exercise 8 Write a further shell script "kill_terminals" that sends a SIGINT signal to all the processes listed in the file generated in the previous exercise (if it exists) and removes it afterwards.

Review – what happened so far

$\longrightarrow$ Job control signals and commands suspend, resume, kill, and connect jobs to or disconnect them from terminal

$\longrightarrow$ environment variables are an alternative to command line arguments to supply parameters to applications

$\longrightarrow$ shell scripts, aliases and functions can define new Unix commands

$\longrightarrow$ compound commands for, if, while, case and tests

$\longrightarrow$ editing history

$\longrightarrow$ personalizing the Unix working environment in start-up scripts

sed – a stream editor

Designed to modify files in one pass and particularly suited for doing automated on-the-fly edits of text in pipes. sed scripts can be provided on the command line

```
sed [-e] 'command' files
```

or in a separate file

```
sed -f scriptfile files
```

General form of a sed command:

```
[address,[address]][!]command[arguments]
```

Addresses can be line numbers or regular expressions. Last line is "$". One address selects a line, two addresses a line range (specifying start and end line). All commands are applied in sequence to each line. After this, the line is printed, unless option -n is used, in which case only the p command will print a line. The ! negates address match. {...} can group commands per address.

Regular expressions enclosed in /.../. Some regular expression meta characters:

→ "." matches any character (except new-line)

→ "*" matches the preceding item zero or more times

→ "+" matches the preceding item one or more times

→ "?" matches the preceding item optionally (0–1 times)

→ "^" matches start of line

→ "$" matches end of line

→ "[...]" matches one of listed characters
  (use in character list "^" to negate and "–" for ranges)

→ "\(...\)" grouping, "\{n,m\}" match $n, \ldots, m$ times

→ "\" escape following meta character

Some sed examples

Substitute all occurrences of "Windows" with "Linux" (command: s = substitute, option: g = "global" = all occurrences in line):

```
sed 's/Windows/Linux/g'
```

Delete all lines that do not end with "OK" (command: d = delete):

```
sed '/OK$/!d'
```

Print only lines between those starting with BEGIN and END, inclusive:

```
sed -n '/^BEGIN/,/^END/p'
```

Substitute in lines 40–60 the first word starting with a capital letter with "X":

```
sed '40,60s/[A-Z][a-zA-Z]*/X/'
```

grep, head, tail, sort

→ Print only lines that contain pattern:

```
grep pattern files
```

  Option -v negates match and -i makes match case insensitive.

→ Print the first and the last 25 lines of a file:

```
head -25 file
tail -25 file
```

  tail -f outputs growing file.

→ Print the lines of a text file in alphabetical order: sort file
  Options: -k select column, -n sort numbers, -u eliminate duplicate lines, -r reverse order.

chmod – set file permissions

$\longrightarrow$ Unix file permissions: $3 \times 3 + 2 + 1 = 12$ bit information.

$\longrightarrow$ Read/write/execute right for user/group/other.

$\longrightarrow$ + set-user-id and set-group-id (elevated execution rights)

$\longrightarrow$ + "sticky bit" (only owner can delete file from directory)

$\longrightarrow$ `chmod ugoa[+-=]rwxst files`

Examples: Make file unreadable for anyone but the user/owner.

```
$ ls -l message.txt
-rw-r--r--   1 mgk25    private      1527 Oct  8 01:05 message.txt
$ chmod go-rwx message.txt
$ ls -l message.txt
-rw-------   1 mgk25    private      1527 Oct  8 01:05 message.txt
```

For directories, "execution" right means right to traverse. Directories can be made traversable without being readable, such that only those who know the filenames inside can access them.

find – traverse directory trees

`find directories expression` — recursively traverse the file trees rooted at the listed directories. Evaluate the Boolean expression for each file found. Examples:

Print relative pathname of each file below current directory:

```
$ find . -print
```

Erase each file named "core" below home directory if it was not modified in the last 10 days:

```
$ find ~ -name core -mtime +10 -exec rm -i {} \;
```

The test "-mtime +10" is true for files older than 10 days, concatenation of tests means "logical and", so "-exec" will only be executed if all earlier terms were true. The "{}" is substituted with the current filename, and "\;" terminates the list of arguments of the shell command provided to "-exec".

Some networking tools

$\longrightarrow$ `wget url` — Fetch a file over the Internet via HTTP or FTP.
Option "-r" fetches HTML files recursively, option "-l" limits recursion depth.

$\longrightarrow$ `ssh [user@]hostname [command]` — Log in via compressed and encrypted link to remote machine. If "command" is provided, execute it in remote shell, otherwise go interactive.
Preserves stdout/stderr distinction. Can also forward X11 requests (option "-X") or arbitrary TCP/IP ports (options "-L" and "-R") over secure link.

$\longrightarrow$ `ssh-keygen -t rsa` — Generate RSA public/private key pair for password-free ssh authentication in "~/.ssh/id_rsa.pub" and "~/.ssh/id_rsa". Protect "id_rsa" like password!

Remote machine will not ask for password with ssh, if your private key "~/.ssh/id_rsa" fits one of the public keys ("locks") listed on the remote machine in "~/.ssh/authorized_keys".
On PWF Linux, your Novell-server home directory with ~/.ssh/authorized_keys is mounted only **after** login, and therefore no password-free login for first session.

tar, gzip – packaging and compressing

$\longrightarrow$ `tar` — Convert between a file tree and a byte stream ("tape archiver").

Create archive (recurses into subdirectories):

```
$ tar cvf archive.tar files
```

Show archive content:

```
$ tar tvf archive.tar
```

Extract archive:

```
$ tar xvf archive.tar [files]
```

→ gzip `file` — convert "`file`" into a compressed "`file.gz`" (using a Lempel-Ziv/Huffman algorithm).

→ gunzip `file` — decompress "`*.gz`" files.

→ [un]compress `file` — [de]compress "`*.Z`" files (older tool using less efficient and patented LZW algorithm).

→ b[un]zip2 `file` — [de]compress "`*.bz2`" files (newer tool using Burrows-Wheeler blocktransform).

→ zcat [`file`] — decompress `*.Z/*.gz` to stdout for use in pipes.

→ Extract compressed `tar` archive

```
$ zcat archive.tar.gz | tar xvf -
$ tar xvzf archive.tgz      # GNU tar only!
```

`diff, patch` – **managing file differences**

→ diff `oldfile newfile` — Show differences between two text files as lines that have to be inserted/deleted to change "`oldfile`" into "`newfile`". Option "`-u`" gives better readable "unified" format with context lines. Option "`-r`" compares entire directory trees.

→ patch <`diff-file` — Apply the changes listed in the provided `diff` output file to the old files named in it. The diff file should contain relative pathnames. If not, use option "`-p`n" to remove the first n slashes and preceding characters from pathnames in "`diff-file`".

If the old files found by `patch` do not match exactly the removed lines in a "`-u`" diff output, `patch` will search whether the context lines can be located nearby and will report which line offset was necessary.

Use `diff3` to compare three files and merge the edits from different revision branches.

RCS – Revision Control System

Operates on individual files only. For every working file "example", RCS keeps a revision history database file named "example,v" or (if the RCS/ subdirectory exists) "RCS/example,v".

→ ci example — Move a file (back) into the "example,v" repository as the new latest revision ("check in").

→ ci -u example — Keep a read-only unlocked copy as well. This is equivalent to "ci ..." followed by "co ...".

→ ci -l example — Keep a writable locked copy (only one user can have the lock for a file at a time). This is equivalent to "ci ..." followed by "co -l ...".

→ co example — Fetches the latest revision from "example,v" as a read-only file ("check out"). Use option "-r$n.m$" to retrieve earlier revisions. There must not be a writable working file already.

→ co -l example — Fetches the latest revision as a locked writable file if the lock is available.

→ rcsdiff example — Show differences between working file and latest version in repository (use option "-r$n.m$" to compare older revisions). Normal `diff` options like -u can be applied.

→ rlog example — Show who made changes when on this file and left what change comments.

In a team, keep all the "*,v" files in a shared repository directory writable for everyone. Team members have their own respective working directory with a symbolic link named RCS to the shared directory.

As long as nobody touches the "*,v" files or manually changes the write permissions on working files, only one team member at a time can edit a file and old versions are never lost. The `rcs` command can be used by a team leader to bypass this policy and break locks or delete old revisions. If you have subdirectories or hate locks, use `cvs` instead.

cvs – Concurrent Versions System

cvs is like a layer on top of rcs that manages entire directory trees and provides remote access to repositories.

Create a new repository under ~/repository/CVSROOT/:

```
cvs -d ~/repository init
```

Place content of current directory into that repository as module demo

```
cvs -d ~/repository import demo DEMO START
```

Now best remove the directory you've just imported (to avoid confusion), go where you want to have your working directory and run

```
cvs -d ~/repository checkout demo
```

Inside your new working directory demo/, you can from now use cvs without specifying the location of the repository each time with -d, because this is recorded in the CVS/ subdirectory (a sibling of demo/).

$\longrightarrow$ cvs add *filenames* — Mark a new file to be added to repository

$\longrightarrow$ cvs remove *filenames* — Mark a deleted file to be removed from repository

$\longrightarrow$ cvs commit [*filenames*] — Check any modifications, additions, removals of files that you did into the repository.

$\longrightarrow$ cvs update [*filenames*] — Apply modifications that others committed since the last update to your working directory, unless they are to a file that you edited since then and have not committed yet.

$\longrightarrow$ cvs diff [*filenames*] — Show what you changed so far.

There are no locks. Conflicting changes are merged together automatically and marked for manual intervention in case of overlaps.

Full manual: http://www.cvshome.org/docs/manual/

cc/gcc – the C compiler

Example:

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, World!\n"); return 0; }
$ gcc -o hello hello.c
$ ./hello
Hello, World!
```

Compiler accepts source ("*.c") and object ("*.o") files. Produces either final executable or object file (option "-c"). Common options:

$\longrightarrow$ -W -Wall — activate warning messages (better analysis for suspicious code)

$\longrightarrow$ -O — activate code optimizer

$\longrightarrow$ -g — include debugging information (symbols, line numbers).

gdb – the C debugger

Best use on binaries compiled with "-g".

$\longrightarrow$ gdb binary — run command inside debugger ("r") after setting breakpoints.

$\longrightarrow$ gdb binary core — post mortem analysis on memory image of terminated process.

Enter in shell "ulimit -c 100000" before test run to enable core dumps. Core dump can be triggered by:

$\longrightarrow$ a user pressing Ctrl-\ (SIGQUIT)

$\longrightarrow$ a fatal processor or memory exception (segmentation violation, division by zero, etc.)

Some common gdb commands:

$\longrightarrow$ bt — print the current stack (backtracing function calls)

$\longrightarrow$ p *expression* — print variable and expression values

$\longrightarrow$ up/down — move between stack frames to inspect variables at different function call levels

$\longrightarrow$ b ... — set breakpoint at specified line or function

$\longrightarrow$ r ... — run program with specified command-line arguments

$\longrightarrow$ s — continue until next source code line (skip function calls)

$\longrightarrow$ n — continue until next source code line (follow function calls)

Also consider starting gdb within emacs with "ESC x gdb", which causes the program-counter position to be indicated in source-file windows.

make – a project build tool

The files generated in a project fall into two categories:

$\longrightarrow$ **Source files:** Files that cannot be regenerated easily, such as

- working files directly created and edited by humans
- files provided by outsiders
- results of experiments

$\longrightarrow$ **Derived files:** Files that can be recreated easily by merely executing a few shell commands, such as

- object and executable code output from a compiler
- output of document formatting tools
- output of file-format conversion tools
- results of post-processing steps for experimental data
- source code generated by other programs
- files downloaded from Internet archives

Many derived files have other source or derived files as *prerequisites*. They were generated from these input files and have to be regenerated as soon as one of the prerequisites has changed, and make does this.

A Makefile describes

$\longrightarrow$ which ("target") file in a project is derived

$\longrightarrow$ on which other files that target depends as a prerequisite

$\longrightarrow$ which shell command sequence will regenerate it

A Makefile contains rules of the form

```
target1 target2 ... : prereq1 prereq2 ...
        command1
        command2
        ...
```

Command lines **must** start with a TAB character (ASCII 9).

Examples:

```
demo: demo.c demo.h
        gcc -g -O -o demo demo.c

data.gz: demo
        ./demo | gzip -c > data.gz
```

Call make with a list of target files as command-line arguments. It will check for every requested target whether it is still up-to-date and will regenerate it if not:

$\longrightarrow$ It first checks recursively whether all prerequisites of a target are up to date.

$\longrightarrow$ It then checks whether the target file exists and is newer than all its prerequisites.

$\longrightarrow$ If not, it executes the regeneration commands specified.

Without arguments, make checks the targets of the first rule.

Variables can be used to abbreviate rules:

```
CC=gcc
CFLAGS=-g -O
demo: demo.c demo.h
        $(CC) $(CFLAGS) -o $@ $<

data.gz: demo
        ./$< | gzip -c > $@
```

$\longrightarrow$ $@ — file name of the target of the rule

$\longrightarrow$ $< — name of the first prerequisite

$\longrightarrow$ $+ — names of all prerequisites

Environment variables automatically become `make` variables, for example $(HOME). A "$" in a shell command has to be entered as "$$".

Implicit rules apply to all files with registered suffixes:

```
.SUFFIXES: .eps .gif .jpg $(SUFFIXES)

.gif.eps:
        giftopnm $< | pnmtops -noturn > $@
.jpg.eps:
        djpeg $< | pnmtops -noturn > $@
```

`make` knows a number of implicit rules by default, for instance

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

It is customary to add rules with "phony targets" for routine tasks that will never produce the target file and just execute the commands:

```
clean:
        rm -f *~ *.bak *.o $(TARGETS) core
```

Common "phony targets" are "clean", "test", "install".

Exercise 9 Write down the command line of the single `sed` invocation that performs the same action as the pipe

```
head -12 <input | tail -7 | grep 'with'
```

Exercise 10 Generate a CVS repository and place all your exercise solution files created so far into it. Then modify a file, commit the change, and create a patch file that contains the modification you made. And finally, retrieve the original version of the modified file again out of the repository.

Exercise 11 Add a `Makefile` with a target `solutions.tar.gz` that packs up all your solutions files into a compressed archive file. Ensure that calling `make solutions.tar.gz` will recreate the compressed package only after you have actually modified one of the files in the package.

Exercise 12 Write a C program that divides a variable by zero and execute it. Use `gdb` to determine from the resulting `core` file the line number in which the division occurred and the value of the variable involved.

`perl` – the Swiss Army Unix Tool

$\longrightarrow$ a portable interpreted language with comprehensive library

$\longrightarrow$ combines some of the features of C, `sed`, `awk` and the shell

$\longrightarrow$ the expression and compound-statement syntax follows closely C, as do many standard library functions

$\longrightarrow$ powerful regular expression and binary data conversion facilities make it well suited for parsing and converting file formats, extracting data, and formatting human-readable output

$\longrightarrow$ offers arbitrary size strings, arrays and hash tables

$\longrightarrow$ garbage collecting memory management

$\longrightarrow$ dense and compact syntax leads to many potential pitfalls and has given Perl the reputation of a write-only hacker language

$\longrightarrow$ widely believed to be less suited for beginners, numerical computation and large-scale software engineering, but highly popular for small to medium sized scripts, and Web CGI

perl – data types

Perl has three variable types, each with its own name space. The first character of each variable reference indicates the type accessed:

```
$... a scalar
@... an array of scalars
%... an associative array of scalars (hash table)
```

[...] selects an array element, {...} queries a hash table entry.
Examples of variable references:

```
$days          = the value of the scalar variable "days"
$days[28]      = element 29 of the array @days
$days{'Feb'}   = the 'Feb' value from the hash table %days
$#days         = last index of array @days
@days          = ($days[0], ..., $days[$#days])
@days[3,4,5]   = @days[3..5]
@days{'a','c'} = ($days{'a'}, $days{'c'})
%days          = (key1, val1, key2, val2, ...)
```

perl – scalar values

$\longrightarrow$ A "scalar" variable can hold a string, number, or reference.

$\longrightarrow$ Scalar variables can also hold the special undef value (set with undef and tested with defined(...))

$\longrightarrow$ Strings can consist of bytes or characters (Unicode/UTF-8).
More on Unicode character strings: man perluniintro.

$\longrightarrow$ Numeric (decimal) and string values are automatically converted into each other as needed by operators.
(5 – '3' == 2, 'a' == 0)

$\longrightarrow$ In a Boolean context, the values '', 0, '0', or undef are interpreted as "false", everything else as "true".
Boolean operators return 0 or 1.

$\longrightarrow$ References are typed pointers with reference counting.

perl – scalar literals

$\longrightarrow$ Numeric constants follow the C format:
123 (decimal), 0173 (octal), 0x7b (hex), 3.14e9 (float)
Underscores can be added for legibility: 4_294_967_295

$\longrightarrow$ String constants enclosed with "..." will substitute variable references and other meta characters. In '...' only "\'" and "\\" are substituted.

```
$header = "From: $name[$i]\@$host\n" .
          "Subject: $subject{$msgid}\n";
print 'Metacharacters include: $@%\\';
```

$\longrightarrow$ Strings can contain line feeds (multiple source-code lines).

$\longrightarrow$ Multiline strings can also be entered with "here docs":

```
$header = <<"EOT";
From: $name[$i]\@$host
Subject: $subject{$msgid}
EOT
```

perl – arrays

$\longrightarrow$ Arrays start at index 0

$\longrightarrow$ Index of last element of @foo is $#foo (= length minus 1)

$\longrightarrow$ Array variables evaluate in a scalar context to array length, i.e.

```
scalar(@foo) == $#foo + 1;
```

$\longrightarrow$ List values are constructed by joining scalar values with comma operator (parenthesis often needed due to precedence rules):

```
@foo = (3.1, 'h', $last);
```

$\longrightarrow$ Lists in lists lose their list identity: (1,(2,3)) equals (1,2,3)

$\longrightarrow$ Use [...] to generate reference to list (e.g., for nested lists).

$\longrightarrow$ Null list: ()

$\longrightarrow$ List assignments: ($a,undef,$b,@c)=(1,2,3,4,5); equals $a=1; $b=3; @c=(4,5);

$\longrightarrow$ Command line arguments are available in @ARGV.

perl – hash tables

$\longrightarrow$ Literal of a hash table is a list of key/value pairs:

```
%age = ('adam', 19, 'bob', 22, 'charlie', 7);
```

Using => instead of comma between key and value increases readability:

```
%age = ('adam' => 19, 'bob' => 22, 'charlie' => 7);
```

$\longrightarrow$ Access to hash table %age:

```
$age{'john'} = $age{'adam'} + 6;
```

$\longrightarrow$ Remove entry: `delete $age{'charlie'};`

$\longrightarrow$ Get list of all keys: `@family = keys %age;`

$\longrightarrow$ Use `{...}` to generate reference to hash table.

$\longrightarrow$ Environment variables are available in %ENV.

For more information: `man perldata`

perl – syntax

$\longrightarrow$ Comments start with # and go to end or line (as in shell)

$\longrightarrow$ Compound statements:

```
if (expr) block
  elsif (expr) block ...
  else block
while (expr) block [continue block]
for (expr; expr; expr) block
foreach var (list) block
```

Each `block` must be surrounded by `{...}` (no unbraced single statements as in C).
The optional continue block is executed just before `expr` is evaluated again.

$\longrightarrow$ The compound statements `if`, `unless`, `while`, and `until` can be appended to a statement:

```
$n = 0 if ++$n > 9;
do { $x >>= 1; } until $x < 64;
```

A do block is executed at least once.

$\longrightarrow$ Loop control:

- `last` immediately exits a loop.
- `next` executes the continue block of a loop, then jumps back to the top to test the expression.
- `redo` restarts a loop block (without executing the continue block or evaluating the expression).

$\longrightarrow$ The loop statements `while`, `for`, or `foreach` can be preceded by a label for reference in `next`, `last`, or `redo` instructions:

```
LINE: while (<STDIN>) {
  next LINE if /^#/;    # discard comments
  ...
}
```

$\longrightarrow$ No need to declare global variables.

For more information: `man perlsyn`

perl – subroutines

$\longrightarrow$ Subroutine declaration:

```
sub name block
```

$\longrightarrow$ Subroutine call:

```
name(list);
name list;
&name;
```

A & prefix clarifies that a name identifies a subroutine. This is usually redundant thanks to a prior sub declaration or parenthesis. The third case passes @_ on as parameters.

$\longrightarrow$ Parameters are passed as a flat list of scalars in the array @_.

$\longrightarrow$ Perl subroutines are call-by-reference, that is $_[0], ... are aliases for the actual parameters. Assignments to @_ elements will raise errors unless the corresponding parameters are lvalues.

→ Subroutines return the value of the last expression evaluated or the argument of a `return` statement. It will be evaluated in the scalar/list context in which the subroutine was called.

→ Use `my($a,$b);` to declare local variables $a and $b within a block.

Example:

```
sub max {
  my ($x, $y) = @_;
  return $x if $x > $y;
  $y;
}

$m = max(5, 7);
print "max = $m\n";
```

For more information: `man perlsub`

`perl` – **operators**

→ Normal C/Java operators:

```
++ -- + - * / % << >> ! & | ^ && ||
?: , = += -= *= ...
```

→ Exponentiation: `**`

→ Numeric comparison: `== != <=> < > <= >=`

→ String comparison: `eq ne cmp lt gt le ge`

→ String concatenation: `$a . $a . $a   eq  $a x 3`

→ Apply regular expression operation to variable:
  `$line =~ s/sed/perl/g;`

→ Create reference with `\`, dereference with `$, @, %,` or `&`.

→ `` `...` `` executes a shell command

→ `..` returns list with a number range in a list context and works as a flip-flop in a scalar context (for `sed`-style line ranges)

For more information: `man perlop`

`perl` – **examples of standard functions**

`split /pattern/, expr`
      Splits string into array of strings, separated by pattern.

`join expr, list`
      Joins the strings in `list` into a single string, separated by value of `expr`.

`reverse list`
      Reverse the order of elements in a list.
      Can also be used to invert hash tables.

`substr expr, offset [, len]`
      Extract substring.

Example:

```
$line = 'mgk25:x:1597:1597:Markus Kuhn:/homes/mgk25:/usr/bin/bash';
@user = split(/:/, $line);
($logname, $pw, $uid, $gid, $name, $home, $shell) = @user;
$line = join(':', reverse(@user));
```

`perl` – **more standard functions**

`chop, chomp`
  Remove trailing character/linefeed from string
`pack, unpack`
  build/parse binary records
`sprintf`
  format strings and numbers
`shift, unshift, push, pop`
  add/remove first/last array element
`die, warn`
  abort program with error/warning
`map, grep`
  Iterate over or filter list elements

`lc, uc, lcfirst, ucfirst`
  Change entire string or first character to lowercase/uppercase
`chr, ord`
  ASCII ↔ integer conversion
`hex, oct`
  string → number conversion
`wantarray`
  check scalar/list context in subroutine call
`require, use`
  Import library module

Perl provides most standard C and POSIX functions and system calls for arithmetic and low-level access to files, network sockets, and other inter-process communication facilities.

All built-in functions are listed in `man perlfunc`. A comprehensive set of add-on library modules is listed in `man perlmodlib` and thousands more are on `http://www.cpan.org/`.

perl – regular expressions

$\longrightarrow$ Perl's regular expression syntax is similar to sed's, but (){} are metacharacters (and need no backslashes).

$\longrightarrow$ Substrings matched by regular expression inside (...) are assigned to variables $1, $2, $3, ... and can be used in the replacement string of a s/.../.../ expression.

$\longrightarrow$ The substring matched by the regex pattern is assigned to $&, the unmatched prefix and suffix go into $` and $'.

$\longrightarrow$ Predefined character classes include whitespace (\s), digits (\d), alphanumeric or _ character (\w). The respective complement classes are defined by the corresponding uppercase letters, e.g. \S for non-whitespace characters.

Example:
```
$line = 'mgk25:x:1597:1597:Markus Kuhn:/homes/mgk25:/usr/bin/bash';
if ($line =~ /^(\w+):[^:]*:\d+:\d+:([^:]*):[^:]*:[^:]*$/) {
  $logname = $1; $name = $2;
  print "'$logname' = '$name'\n";
} else { die("Syntax error in '$line'\n"); }
```
For more information: man perlre

perl – predefined variables

$_   The "default variable" for many operations, e.g.

```
print;              =    print $_;
tr/a-z/A-Z/;        =    $_ =~ tr/a-z/A-Z/;
while (<FILE>) ...  =    while ($_ = <FILE>) ...
```

$.   Line number of the line most recently read from any file

$?   Child process return value from the most recently closed pipe or `...` operator

$!   Error message for the most recent system call, equivalent to C's strerror(errno). Example:

```
open(FILE, 'test.dat') ||
  die("Can't read 'test.dat': $!\n");
```

For many more: man perlvar

perl – file input/output

$\longrightarrow$ open *filehandle*, *expr*

```
open(F1, 'test.dat');      # open file 'test.dat' for reading
open(F2, '>test.dat');     # create file 'test.dat' for writing
open(F3, '>>test.dat');    # append to file 'test.dat'
open(F4, 'date|');         # invoke 'date' and connect to its stdout
open(F5, '|mail -s test'); # invoke 'mail' and connect to its stdin
```

$\longrightarrow$ print *filehandle*, *list*

$\longrightarrow$ close, eof, getc, seek, read, format, write, truncate

$\longrightarrow$ "<*filehandle*>" reads another line from file handle FILE and returns the string. Used without assignment in a while loop, the line read will be assigned to $_.

$\longrightarrow$ "<>" opens one file after another listed on the command line (or stdin if none given) and reads out one line each time.

perl – invocation

$\longrightarrow$ First line of a Perl script: #!/usr/bin/perl (as with shell)

$\longrightarrow$ Option "-e" reads code from command line (as with sed)

$\longrightarrow$ Option "-w" prints warnings about dubious-looking code.

$\longrightarrow$ Option "-d" activates the Perl debugger (see man perldebug)

$\longrightarrow$ Option "-p" places the loop

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

around the script, such that perl reads and prints every line. This way, Perl can be used much like sed:

```
sed -e 's/sed/perl/g'
perl -pe 's/sed/perl/g'
```

→ Option -n is like -p without the "print;".

→ Option "-i[*backup-suffix*]" adds in-place file modification to -p. It renames the input file, opens an output file with the original name and directs the input into it.

Example: To make email addresses in your web pages harder to harvest for spammers, the lines

```
perl -pi.bak <<EOT *.html
s/(href=\"mailto:[^@\"]+)@([^@\"]+\")/$1%40$2/ig;
s/([a-zA-Z0-9\.\-\+\_]+)@([a-zA-Z0-9\.\-]+)/$1&#64;$2/ig;
EOT
```

will convert for instance

```
<a href="mailto:jdoe@acm.org">jdoe@acm.org</a>
```

into

```
<a href="mailto:jdoe%40acm.org">jdoe&#64;acm.org</a>
```

For more information: `man perlrun`

perl – a simple example

Generate a list of email addresses of everyone on the Computer Lab's "People" web page, sorted by surname.

Example input:

```
  ...
  <tr><td><a NAME="asa28">asa28</a></td><td>FE04</td><td>63622</td><td></td><td
></td><td><a HREF="/users/asa28/">Abrahams, Alan</a></td></tr>
  <tr><td><a NAME="mha23">mha23</a></td><td>FE22</td><td>63692</td><td></td><td
></td><td><a HREF="/users/mha23/">Allen-Williams, Mair</a></td></tr>
  <tr><td><a NAME="sa333">sa333</a></td><td>GC33</td><td>63680</td><td></td><td
></td><td>Allott, Stephen</td></tr>
  ...
```

Example output:

```
Alan Abrahams <asa28@cl.cam.ac.uk>
Mair Allen-Williams <mha23@cl.cam.ac.uk>
Stephen Allott <sa333@cl.cam.ac.uk>
```

perl – a simple example

Possible solution:

```
#!/usr/bin/perl
$url = 'http://www.cl.cam.ac.uk/UoCCL/people/directory.html';
open(HTML, "wget -O - '$url' |") || die("Can't start 'wget': $!\n");
while (<HTML>) {
  if (/^<tr><td><a name="(\w+)">.*<\/tr>$/i) {
    $crsid = $1;
    if (/<td>(<a href="[^"]*">)?([^<>]*), ([^<>]*)(<\/a>)?<\/td><\/tr>$/i) {
      $email{$crsid} = "$3 $2 <$crsid\@cl.cam.ac.uk>";
      $surname{$crsid} = $2;
    } else { die ("Syntax error:\n$_") }
  }
}
foreach $s (sort({$surname{$a} cmp $surname{$b}} keys(%email))) {
  print "$email{$s}\n";
}
```

Warning: This simple-minded solution makes numerous assumptions about how the web page is formatted, which may or may not be valid. Can you name examples of what could go wrong?

LATEX – a document formatter

LATEX is a sophisticated macro package for the TEX text formatting system. Thanks to its excellent facilities for mathematical typesetting, it has become the de-facto standard for preparing scientific publications in mathematical, physical, computing and engineering disciplines.

Graphical illustrations can be added to TEX in the form of "Embedded PostScript" files, which can be drawn with interactive tools such as "xfig" or "tgif".
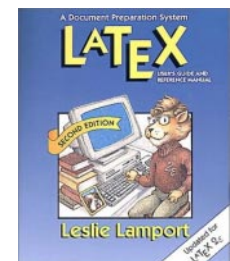
Processing steps:

$$\text{f.tex} \xrightarrow{\text{latex}} \text{f.dvi} \xrightarrow[\text{dvips}]{\text{g1.eps } \text{g2.eps}} \text{f.ps} \xrightarrow{\text{ps2pdf}} \text{f.pdf}$$

Recommended introduction:

Leslie Lamport: LATEX – a document preparation system. 2nd ed., Addison-Wesley, 1994.

TEX Frequently Asked Questions: http://www.tex.ac.uk/cgi-bin/texfaq2html

LATEX example

```
\documentclass[12pt]{article}
\setlength{\textwidth}{75mm}
\begin{document}
\title{\TeX\ -- a Summary}
\author{Markus Kuhn}
\date{28 October 2003}
\maketitle

\section{Introduction}
Mathematical formul\ae\ such as
$e^{i\pi} = -1$ or even
\[ \Phi(z) = \frac{1}{\sqrt{2\pi}}
   \int_0^x e^{-\frac12 x^2}        \]
were a real `pain' to typeset until
\textsc{Knuth}'s text formatter \TeX\
became available \cite{Knuth86}.

\begin{thebibliography}{9}
  \bibitem{Knuth86}Donald E. Knuth:
  The \TeX\ book. Ad\-dison-Wesley, 1986.
\end{thebibliography}

\end{document}
```

TEX – a Summary

Markus Kuhn

28 October 2003

1 Introduction

Mathematical formulæ such as $e^{i\pi} = -1$ or even

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{-\frac{1}{2}x^2}$$

were a real 'pain' to typeset until KNUTH's text formatter TEX became available [1].

References

[1] Donald E. Knuth: The TEXbook. Addison-Wesley, 1986.

TEX input syntax

⟶ TEX reads plain-text *.tex files (e.g., prepared with emacs)

⟶ no distinction is made between space character and line feed

⟶ multiple spaces are treated like a single space

⟶ multiple line feeds (empty lines) are treated as a paragraph separator (just like the \par command)

⟶ command, macro and variable names start with a backslash (\), followed by either a sequence of letters or a single non-letter character (uppercase/lowercase is significant).

   Correct: \par, \item, \pagethree, \LaTeX, \+, \\, \3

   Wrong: \page33, \<>

⟶ space and line-feed characters are ignored if they follow a command/macro/variable name consisting of letters. Use \␣ to add an explicit space (e.g., \TeX\ syntax ⇒ TEX syntax).

Characters with special semantics

In *.tex input files, the characters

```
# $ % & ~ _ ^ \ { }
```

have special functions. Some of these can be included in regular text by writing

```
\# \$ \% \& \_ \^ \{ \}
```

LATEX supports typesetting all ASCII characters via the \verb and \url macros.

% starts a comment

All characters between (and including) a % and the next line feed will be ignored. Append % at the end of a line to avoid interpretation of the subsequent line feed as a space.

[# plus a digit denotes a parameter in macros, ~ is a no-break space, $ delimits inline equations, & is used as a tabulator mark, \\ is a line separator, ^ indicates a superscript and _ a subscript in math mode.]

Blocks

State changes inside a { ... } block last only until the next }:

```
{This is a \bf bold} statement.
```
⇓
This is a **bold** statement.

Commands and macros read for each argument either a single character or a block enclosed by { and }:

```
Typeset \textsl M in \textsl{slanted style}.
```
⇓
Typeset *M* in *slanted style*.

Values of optional LATEX macro arguments are enclosed by [...].

Typewriting versus Typesetting

The ASCII (ISO 646) 7-bit character set with its 94 graphic characters

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

was designed to cover the character repertoire of US typewriters and teletype printers. Some new symbols such as [\]{|}_ were added in the hope that they will be useful for programming.

TeX defines a number of shortcuts and macros to access the full range of "typographic" characters used in high-quality book printing. These still cannot be found on the standard PC keyboard, which was designed for 7-bit ASCII.

Dashes

ASCII provides only a single combined hyphen-minus character, but typesetters distinguish carefully between several dash characters:

-	⇒	-	hyphen
--	⇒	–	en dash
---	⇒	—	em dash
$-$	⇒	−	minus

The hyphen (-) is the shortest of these and is used to combine separate words or split words across line-breaks.

The en dash (–) is often used to denote a range of numbers (as in pages 64–128), or – as in this example – as a punctuation dash.

The em dash is used—like this—as a punctuation dash, often without surrounding space, especially in US typography.

The minus (−) is a mathematical operator, whose shape matches the plus (+), unlike the hyphen or dashes. Compare: -+, –+, —+, −+.

Quotation marks

Typewriters and ASCII offer only undirectional 'single' and "double" quotation marks, while typesetters use 'curly' and "directed" variants.

TeX input files use the single quotation mark (') and the grave accent (`) to encode these, as well the mathematical 'prime' marker and the French accents:

`	⇒	'	left quote
'	⇒	'	right quote
``	⇒	"	left doublequote
''	⇒	"	right doublequote
$'$	⇒	′	prime
\'u	⇒	ú	acute accent
\`u	⇒	ù	grave accent

The apostrophe (it's) is identical to the right single quotation mark.

In some older terminal fonts (especially of US origin), the ` and ' characters have a compromise shape somewhere between the quotation marks ' ' and the accents ` ´.

Non-ASCII Symbols

¡	!`	Å	\AA	¶	\P
¿	?`	ø	\o	†	\dag
œ	\oe	Ø	\O	‡	\ddag
Œ	\OE	ł	\l	©	\copyright
æ	\ae	Ł	\L	£	\pounds
Æ	\AE	ß	\ss	...	\ldots
å	\aa	§	\S		

Combining characters

ó	\'o	ō	\=o	o͡o	\t{oo}
ò	\`o	ȯ	\.o	o̦	\c{o}
ô	\^o	ǒ	\u{o}	ọ	\d{o}
ö	\"o	ǒ	\v{o}	o̲	\b{o}
õ	\~o	ő	\H{o}		

Space – the final frontier

Traditional English typesetting inserts a larger space at the end of a sentence. TeX believes any space after a period terminates a sentence, unless it is preceded by an uppercase letter. Parenthesis are ignored.

This works often: J. F. Kennedy's U.S. budget. Look!
But not always: E.g. NASA. Dr. K. Smith et al. agree.

To correct failures of this heuristic, use

~	$\Rightarrow$	no-break space
\␣	$\Rightarrow$	force normal space
\@	$\Rightarrow$	following punctuation ends sentence

as in

```
E.g.\ NASA\@. Dr.~K. Smith et al.\ agree.
```
$\Downarrow$
E.g. NASA. Dr. K. Smith et al. agree.

Or disable the distinction of spaces with \frenchspacing.

Structure of a LaTeX document

First select a document class and its options, e.g. with

```
\documentclass[12pt,a4paper]{article}
```

Standard classes: `article, report, book, letter, slides`.
Publishers often provide authors with their own class as a `*.cls` file.

Delimit block environments as in

```
\begin{document} ... \end{document}
```

Others: `abstract, center, verbatim, itemize, tabular, ...`

Mark headings with

```
\section{...}            \subsection{...}
\subsubsection{...}       \paragraph{...}
```

and LaTeX will take care of font sizes, numbering, and table of contents.

TeX is a full programming language with macros, variables, recursion, conditional branching, file I/O, and a huge collection of add-ons.

Mathematical typesetting

In TeX, mathematical formulas are formatted in a completely different mode from that used for normal text.

Inline formulas such as a_n (a_n) that appear as part of a normal paragraph have to be surrounded with $...$, while displayed formulas such as

$$F_n = F_{n-1} + F_{n-2} \quad (\backslash[F\_n=F\_{n-1}+F\_{n-2}\backslash])$$

are entered in between \[... \]. In math mode

$\longrightarrow$ space characters are ignored; TeX adds its own space around operators based on heuristics; manually add thinspace with "\,"

$\longrightarrow$ a special math italic font with different inter-character spacing is used, to show single-letter variables better in products

$\longrightarrow$ many additional macros for special symbols are defined

Math italic is very $different$ and never suitable for writing words!

Math symbols – Greek letters

Γ	\Gamma	δ	\delta	π	\pi
Δ	\Delta	ϵ	\epsilon	ϖ	\varpi
Θ	\Theta	ε	\varepsilon	ρ	\rho
Λ	\Lambda	ζ	\zeta	ϱ	\varrho
Ξ	\Xi	η	\eta	σ	\sigma
Π	\Pi	θ	\theta	ς	\varsigma
Σ	\Sigma	ϑ	\vartheta	τ	\tau
Υ	\Upsilon	ι	\iota	υ	\upsilon
Φ	\Phi	κ	\kappa	ϕ	\phi
Ψ	\Psi	λ	\lambda	φ	\varphi
Ω	\Omega	μ	\mu	χ	\chi
α	\alpha	ν	\nu	ψ	\psi
β	\beta	ξ	\xi	ω	\omega
γ	\gamma	o	o		

Binary operations

$\pm$	\pm	$\lhd$	\lhd	$\oplus$	\oplus
$\mp$	\mp	$\cap$	\cap	$\ominus$	\ominus
$\setminus$	\setminus	$\cup$	\cup	$\otimes$	\otimes
$\cdot$	\cdot	$\uplus$	\uplus	$\oslash$	\oslash
$\times$	\times	$\sqcap$	\sqcap	$\odot$	\odot
$*$	\ast	$\sqcup$	\sqcup	$\dagger$	\dagger
$\star$	\star	$\wr$	\wr	$\ddagger$	\ddagger
$\diamond$	\diamond	$\bigcirc$	\bigcirc	$\amalg$	\amalg
$\circ$	\circ	$\rhd$	\rhd	$\unlhd$	\unlhd
$\bullet$	\bullet	$\vee$	\vee	$\unrhd$	\unrhd
$\div$	\div	$\wedge$	\wedge		

$\triangleleft$	\triangleleft	$\triangle$	\bigtriangleup
$\triangleright$	\triangleright	$\triangledown$	\bigtriangledown

Relations

$\leq$	\leq	$\geq$	\geq	$\equiv$	\equiv
$\prec$	\prec	$\succ$	\succ	$\sim$	\sim
$\preceq$	\preceq	$\succeq$	\succeq	$\simeq$	\simeq
$\ll$	\ll	$\gg$	\gg	$\asymp$	\asymp
$\subset$	\subset	$\supset$	\supset	$\approx$	\approx
$\subseteq$	\subseteq	$\supseteq$	\supseteq	$\cong$	\cong
$\sqsubseteq$	\sqsubseteq	$\sqsupseteq$	\sqsupseteq	$\bowtie$	\bowtie
$\in$	\in	$\ni$	\ni	$\propto$	\propto
$\vdash$	\vdash	$\dashv$	\dashv	$\models$	\models
$\smile$	\smile	$\mid$	\mid	$\doteq$	\doteq
$\frown$	\frown	$\parallel$	\parallel	$\perp$	\perp
$\sqsubset$	\sqsubset	$\sqsupset$	\sqsupset	$\Join$	\Join

$\not<$	\not<	$\neq$	\not=	$\not>$	\not>
$\not\leq$	\not\leq	$\not\geq$	\not\geq	$\not\equiv$	\not\equiv
$\not\prec$	\not\prec	$\not\succ$	\not\succ	...	

Arrows

$\leftarrow$	\leftarrow	$\Longleftrightarrow$	\Longleftrightarrow
$\Leftarrow$	\Leftarrow	$\longmapsto$	\longmapsto
$\rightarrow$	\rightarrow	$\hookrightarrow$	\hookrightarrow
$\Rightarrow$	\Rightarrow	$\rightharpoonup$	\rightharpoonup
$\leftrightarrow$	\leftrightarrow	$\rightharpoondown$	\rightharpoondown
$\Leftrightarrow$	\Leftrightarrow	$\leadsto$	\leadsto
$\mapsto$	\mapsto	$\uparrow$	\uparrow
$\hookleftarrow$	\hookleftarrow	$\Uparrow$	\Uparrow
$\leftharpoonup$	\leftharpoonup	$\downarrow$	\downarrow
$\leftharpoondown$	\leftharpoondown	$\Downarrow$	\Downarrow
$\rightleftharpoons$	\rightleftharpoons	$\updownarrow$	\updownarrow
$\longleftarrow$	\longleftarrow	$\Updownarrow$	\Updownarrow
$\Longleftarrow$	\Longleftarrow	$\nearrow$	\nearrow
$\longrightarrow$	\longrightarrow	$\searrow$	\searrow
$\Longrightarrow$	\Longrightarrow	$\swarrow$	\swarrow
$\longleftrightarrow$	\longleftrightarrow	$\nwarrow$	\nwarrow

Misc math symbols

$\aleph$	\aleph	$\prime$	\prime	$\forall$	\forall
$\hbar$	\hbar	$\emptyset$	\emptyset	$\exists$	\exists
$\imath$	\imath	∇	\nabla	$\neg$	\neg
$\jmath$	\jmath	$\surd$	\surd	$\flat$	\flat
ℓ	\ell	$\top$	\top	$\natural$	\natural
$\wp$	\wp	$\bot$	\bot	$\sharp$	\sharp
$\Re$	\Re	$\|$	|	$\clubsuit$	\clubsuit
$\Im$	\Im	$\angle$	\angle	$\diamondsuit$	\diamondsuit
∂	\partial	$\triangle$	\triangle	$\heartsuit$	\heartsuit
∞	\infty	$\backslash$	\backslash	$\spadesuit$	\spadesuit
$\Box$	\Box	$\Diamond$	\Diamond		

...	\ldots	$\cdots$	\cdots	$\vdots$	\vdots	$\ddots$	\ddots

Large operators

$\sum$	\sum	$\bigcap$	\bigcap	$\odot$	\bigodot
$\prod$	\prod	$\bigcup$	\bigcup	$\otimes$	\bigotimes
$\coprod$	\coprod	$\bigsqcup$	\bigsqcup	$\oplus$	\bigoplus
$\int$	\int	$\bigvee$	\bigvee	$\biguplus$	\biguplus
$\oint$	\oint	$\bigwedge$	\bigwedge		

Delimiters

$[$	\lbrack	$]$	\rbrack
$\lfloor$	\lfloor	$\rfloor$	\rfloor
$\lceil$	\lceil	$\rceil$	\rceil
$\{$	\lbrace	$\}$	\rbrace
$\langle$	\langle	$\rangle$	\rangle
$[\![$	[\![	$]\!]$	]\!]
$\langle\!\langle$	\langle\!\langle	$\rangle\!\rangle$	\rangle\!\rangle

Alternative names

$\ne$	\ne	$\{$	\\{	$\ni$	\owns	$\vert$	\vert
$\neq$	\neq	$\}$	\\}	$\wedge$	\land	$\Vert$	\Vert
$\le$	\le	$\to$	\to	$\vee$	\lor		
$\ge$	\ge	$\gets$	\gets	$\neg$	\lnot		

Stacking things

a^b	a^{b}	a_b	a_{b}
$\overline{a-b}$	\overline{a-b}	$\overbrace{a-b}$	\overbrace{a-b}
$\underline{a-b}$	\underline{a-b}	$\underbrace{a-b}$	\underbrace{a-b}

Exercise 13 When editing sentences, users of text editors occasionally leave some word duplicated by by accident. Write a Perl script that reads plain text files and outputs all their lines that contain the same word twice in a row. Extend your program to detect also the cases where the two occurrences of the same word are separated by a line feed.

Exercise 14 Type in the file `example.tex` on slide 93. Call "`latex example`" twice. Preview with "`xdvi example`" the formatted text in the device-independent format (DVI) and convert it with "`dvips -Ppdf example`" to PostScript. View with "`ghostview example.ps`" and convert with "`ps2pdf example.ps`" into the *Portable Document Format*. Finally, call "`acroread example.pdf &`" to inspect the end of this text-format odyssey.

Exercise 15 Read pages 1–64 of the LaTeX book, then write your CV with LaTeX, convert the result into PDF, and put it onto your PWF homepage. See `http://www.cam.ac.uk/cs/pwf/web/` for information on how to set up a homepage under PWF Linux.

Conclusions

$\longrightarrow$ Unix is a powerful and highly productive platform for experienced users.

$\longrightarrow$ This short course could only give you a quick overview to get you started with exploring advanced Unix facilities.

$\longrightarrow$ Please try out all the tools mentioned here and consult the "`man`" and "`info`" online documentation.

$\longrightarrow$ You'll find on

`http://www.cl.cam.ac.uk/Teaching/2003/UnixTools/`

easy to print versions of the `bash`, `make` and `perl` documentation, links to further resources, and hints for installing Linux on your PC.

$\star\,\star$ Good luck and lots of fun with your projects $\star\,\star$