INTRODUCTION TO OPEN COMPUTING

Attila Máté
Brooklyn College of the City University of New York

July 1997
Last Revised: January 2003

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

TABLE OF CONTENTS

Preface **5**

Chapter I. Unix **6**
1. Introduction to Using Unix 6
2. The vi Text Editor 12
3. Permission Codes 17

Chapter II. The Web **19**
4. X Windows and Netscape 19
5. Starting to Build Your Own Web Page 22
6. Hypertext Markup Language 23
7. Empowering Netscape 29

Chapter III. Pascal **36**
8. Introduction to Pascal 36
9. Simple Pascal Programs 40
10. More on Pascal 41
11. Loops 44
12. Conditional Statements 48

Chapter IV. Applications of Pascal **52**
13. The Euclidean Algorithm 52
14. Implementing the Euclidean Algorithm in Pascal 54
15. File Handling in Pascal 57
16. Representing the Greatest Common Divisor as a Linear Combination 59
17. Pascal Program for Finding the Representation the Greatest Common Divisor as a Linear Combination 61
18. The Fundamental Theorem of Arithmetic 67
19. Truth Tables 69
20. Day of the Week 72

Chapter V. Miscellanea **78**
21. Mathematical Typesetting 78
22. Electronic Mail 86
23. Shell Scripts 90
24. More Shell Scripts 100
25. Source Scripts 121
26. Job Control 124
27. Remote Access 126
28. Solution to Problem 1 in Section 16 130

Index **131**

PREFACE

Unix provides a powerful *open* computer operating environment; open here means that Unix is not controlled by a single company; changes to be made to Unix are discussed out in the open, rather than in closed corporate board rooms. In fact, the source code of a version of Unix, called Linux, is available for anyone to study and modify – whereas for many software companies, the source code of programs is a highly valued trade secret. Such an openness is especially important in a college environment, since computing cannot be properly studied if important parts of the computing environment are not available for inspection. Closed, proprietary computing can be useful in a college environment if the computer is strictly a tool to be used, and not studied. Two interesting Web sites where news items and technical issues involving computers can be found is

`http://www.infoworld.com/` and `http://theregister.co.uk/`

(cf. Section 4 on the World Wide Web to see how to look up such Web sites; see p. 20 especially).

In Chapter I, we give an introduction to Unix. This introduction is kept brief, only the necessary prerequisites for necessary for efficient file manipulation, editing, and running programs are discussed. Then, in Chapter 2 we describe how to write a simple Web page. A short introduction to Hypertext Markup Language (HTML) is given, and instructions are given as to how to make a file written in HTML visible for everyone connected to the World Wide Web, provided the network has a properly configured World Wide Web server. Afterwards, in Chapter III, the programming language Pascal is discussed; the implementation in Pascal of the Euclidean algorithm and its extensions are discussed; then applications to print truth tables and to find the day of the week of any date in the Gregorian calendar are given. Chapter IV outline a mixture topics, such as the mathematical typesetting language TEX, electronic mail, some simple shell scripts useful for writing electronic mail, and source scripts useful in avoiding the need of typing long directory names.

These notes were originally written between May 22 and July 13, 1997. The text was revised a number of times between December 1997 and January 2003; many misprints were corrected, and hopefully only a few new ones were introduced. We are grateful to *Nancy McGough* and to *James Rowe* for suggesting numerous changes and corrections. Nancy McGough's Web site is

`http://www.ii.com/`

While effort was made that the notes be as thorough and accurate as possible, further revisions are likely to incorporate many changes. It will be appreciated if suggestions for such changes are brought to the author's attention.

CHAPTER I

UNIX

After a brief description of the basic Unix commands, the *vi* text editor discussed; text editors are needed to create any piece of writing on the computers. Then permission codes are described. Familiarity with permission codes is indispensable for putting documents on the World Wide Web.

1. Introduction to Using Unix

Unix is a powerful operating system initially developed at ATT Bell Laboratories (now Lucent Technologies, Inc.), beginning around 1969. Today, variants of Unix or Unix-like operating systems run on many computers, from supercomputers to workstations to personal computers. They have different names; for example the operating system used on workstations made by Sun Microsystems is called Solaris; an older, but still very good, operating system on these workstations is SunOS 4.x. The operating system on IBM RS6000 workstations is AIX. Many DEC (Digital Equipment Corporation) computers use the Ultrix operating system. Hewlett-Packard workstations run the operating system called HP-UX.

Linux. A Unix-like operating system called Linux is available for personal computers and for some workstations. It is a stable and powerful operating system, and it can be obtained for free, but installing it requires some skill. You can read about it on the Internet at

`http://www.linuxworld.com/`

or at

`http://www.ssc.com/linux/`

While there are many differences in these variants of Unix, they all support the same basic features, and for our purposes, these differences will not be important. For example, things that you learn to do on Sun workstations you will be able to do usually without changes, or only with minor changes, on a personal computer running Linux. Other free Unix operating systems include FreeBSD, OpenBSD, and NetBSD. These are descendants of BSD (Berkeley Software Distribution) Unix developed at the University of California, Berkeley, in the early 1980s. The Web site of FreeBSD is

`http://www.freebsd.org/`

Open Source Software. Linux is a salient example of Open Source Software. In Open Source Software, the source code, that is the code describing how the software was originally created, is available to anyone for inspection. In commercial software,

the source code is usually a tightly protected trade secret, in order to prevent others from creating unlicensed products using the software.

The year 1998 was a turning point in the the history of Open Source Software; this year, many major corporations including Intel, Sun Microsystems, Oracle (the second largest software company in the world), and IBM, announced their support for Linux. The modern Open Source Movement has a long history – it goes back to the establishment of the Free Software Foundation in the late 1970s; see p. 38. You can read about Open Source Software at

`http://www.opensource.org/`

Unix versus NT. The most frequently used operating systems for personal computers are variants of the Windows operating system by Microsoft Corporation, such as Windows 95, its successor Windows 98, and the more powerful Windows NT (currently Windows NT 4.0). The next version of Windows 98 and that of Windows NT 4.0 will be merged into a single product called Windows 2000. Unix is more properly compared to Windows NT than the less sophisticated Windows 98. There are several articles devoted to comparing Unix and Windows NT; for example, see

`http://www.unix-vs-nt.org/kirch/`

An interesting sideline of the comparison between Unix and various Microsoft operating systems is the recent antitrust trial of the US Department of Justice and nineteen states vs. Microsoft. The judge's finding of facts concluding the first phase of the trial can be found at the site

`http://usvms.gpo.gov/findfact.html`

and at the site (in HTML, with links to various parts to of the documents, plus download links to various formats, e.g. PDF, that is, Adobe's Portable Document Format)

`http://www.usdoj.gov/atr/cases/f3800/msjudgex.htm`

The site

`http://www.seattletimes.com/news/technology/html98/`
`mono_19991106.html`

(the site address was scrolled over to the next line, but when entering it into your browser the line must not be broken up), which appeared in the Seattle Times, gives an interesting legal analysis of the ruling.

Logging in. In order to use a computer using Unix, you need a login name, and you need a password. The login name will often be an abbreviation of your name, and it should consist of all lower case letter. The password will usually have 8-12 letters, and it is often required that it should contain a mixture of upper and lower case letter, numbers, and non-letter characters (such as, e.g. `%`, `$`, `#`, etc.). You log in to the computer by typing your login name and, when the computer asks you to do so, your password. When you type your password, it will not appear on the screen; this is a security measure, in case someone is looking over your shoulder.

After you have successfully logged in, the computer displays a short message about itself, and afterwards it displays a few symbols at the beginning of the line; this is called the *prompt* (meaning that the computer prompts, or invites, you to type something), or, more precisely, the *shell prompt*.[1] The prompt may be something as simple as the single character $ or %, or a short message including the name of

[1]The computer understands the commands that you type through a layer, called *shell*, that translates your commands into the computer's own language.

the machine, the directory location (see below), or a line number. In our examples below, we will use $ for the prompt.

Changing your password. If you do not like your password, this is the time to change it to something you can easily remember (but it follows certain rules about containing upper and lower case characters, numbers, etc.).

Be prepared that changing your password can be somewhat disruptive. For example, the Sun Workstations at the Atrium[2] are programmed to log you out and prevent you from logging in for about half an hour after changing your password. Further, these computers are also programmed to require you to change your password when you log in the first time. For this reason, it is best to change your password immediately after you receive your account, so that your work will not be interrupted at a later time.

You can change your password by typing

```
$ passwd
```

The symbol $ at the beginning of the line is the prompt written by the computer, so only the word "passwd" is typed by the user. The computer will respond with something like

```
Password:
```

After typing in your new password, the computer will ask you to repeat the new password. After entering the same new password twice, the password will be changed. The above dialog can be different depending on the system used. For example, it can go something like this:

```
$ passwd
Type current password:
Type new password:
Retype new password:
Password has been changed
```

Here it is assumed that after the colon each time you typed the corresponding password; note, that these passwords will not appear on the screen, for reasons of security.

Files and directories. The information on computers is stored in files. These are connected segments of data – we will refrain from an abstract definition. There are several kinds of files; the two main types are text files and binary files. Basically, text files are files that you can read (if you have permission); binary files will not make sense if you try to read them. Often, text files are called (not quite correctly) ASCII files; ASCII refers to the way these files are encoded. ASCII or ASCII, pronounced *ask-ee*, is an abbreviation of American Standard Code for Information Interchange, and describes the most wide-spread way computers store letters, numbers, punctuation marks, and control characters.

If you are not sure what kind of file is a file called `myfile` is, you can find this out by typing

```
$ file myfile
```

Note here that $ at the beginning of the line is the prompt given by the computer. The computer may give a reply such as

```
myfile:        ascii text
```

[2] "Atrium" in these notes refers to the Atrium Computer Laboratory at Brooklyn College; see `http://acc6.its.brooklyn.cuny.edu/`

Or, you may get a reply such as

```
myfile:         ascii directory
```

Directories are special kind of files that are used to organize the information on the computer. In order to organize the files you may have, related files are grouped together into a single unit, called *directory*; they work much the same way as in a paper filing system, you put related documents in the same folder (in fact, Macintosh computers use the name *folder* rather than directory). To list the contents of (i.e., the files grouped together in) a directory, type

```
$ ls
```

`ls` stands for "list" (but you should always type "ls", and not "list"; the latter is only the origin of the name of the command, and it is not recognized by the computer). The reply you receive may look like

```
edg             euclid_new.p  junk          junk1
pascal          sec1.tex      sec2.tex      sec3.tex
sec4.tex        sec5.tex
```

As you can see here, filenames can contain letters (upper or lower case), numbers, and various other characters, such as period (.) and underscore (_). The reason to use filenames such as `junk` or `junk1` is that you will know that these are not very important files, needed only temporarily. You can use the command `ls` in different ways; for example, typing

```
$ ls -F
```

will give

```
edg*            euclid_new.p  junk          junk1
pascal/         sec1.tex      sec2.tex      sec3.tex
sec4.tex        sec5.tex
```

Here the asterisk after `edg` means that `edg` is an executable file; the slash / after `pascal` means that `pascal` is a directory.

There is much more to the command `ls`. You can find out more about it by looking up the on-line manual page on it by typing

```
$ man ls
```

Initially, the information you receive is more than overwhelming. If you really want to read it you can scroll down by hitting the space bar, scroll backwards by pressing the letter `b`, and advance a single line by pressing the enter key (also called "return key"). At this point, the best choice may be to quit trying to read the information presented; you can do this by pressing the letter `q`. In any case, as you learn more about Unix, you will find the on-line manual pages very helpful if somewhat terse.

If you want to actually see what is in a file, you might try to do this by typing

```
$ cat edg
```

This is safe to do only if you know that the file called `edg` is in fact an ascii file (you can check this by using the `file` command, as described above). You may get something like

```
#! /bin/sh
#This program substitutes string1 for string2 in file3
ed $3 <<%
1,\$s/$1/$2/g
w
%
```

At this point, you are not expected to make sense of this, but you can read each individual letter. This is a so-called shell script, a little program that will help you

accomplish repetitive tasks by automating them. The word `cat` is a shortening of *concatenate* (what this word means is not important at the moment).

If you want to check which directory you are in, you can type

```
$ pwd
```

`pwd` stands for "print working directory". The computer may reply with something like

```
/users1/jsmith/euclid
```

This means that the directory called / (often referred to as the `root` directory) contains a directory `users1`, which contains a directory called `jsmith`, which contains a directory called `euclid`. This last one is the directory you are currently working. If there is a file called sec1.tex in this directory, then that file can be identified together with its location as

```
/users1/jsmith/euclid/sec1.tex
```

This way of referring to a location is called *absolute* reference to the location. There are relative ways to refer to the same file. In fact, in the directory

```
/users1/jsmith/euclid
```

this file can simply be referred to as `sec1.tex`. From the directory `/users1/jsmith` this file can be referred to as `euclid/sec1.tex` (note that you can recognize relative references by the fact that the character / is missing in front). From a *subdirectory* of (i.e., directory contained in) `/users1/jsmith` called `misc`, that is, from the directory

```
/users1/jsmith/misc
```

the above file can be identified as

```
../euclid/sec1.tex
```

Here `..` is the name of the directory immediately above the present working directory; i.e., looked at from `/users1/jsmith/misc`, the directory `..` means the directory `/users1/jsmith`. We add that the the single dot `.` refers to the present working directory; that is

`../euclid/sec1.tex` and `./../euclid/sec1.tex`

refer to the same file. Finally, `~` refers refers to the users home directory. For example, if `/home/cmuser` is the home directory of *cmuser*, then the descriptions

`~/euclid/whole.ps` and `/home/cmuser/euclid/whole.ps`

refer to the same files.

To move between directories, you can use the command `cd`. For example, if you are in the directory `/users/jsmith`, then

```
$ cd euclid
```

changes to the directory `/users/jsmith/euclid`. The command

```
$ cd /users/jsmith/euclid
```

changes to this directory from anywhere. In case you are in the directory `/users/jsmith/misc`, then you can change to the above directory by typing

```
$ cd ../euclid
```

Typing

```
$ cd ..
```

in the same directory will move you to the directory `/users/jsmith`, while typing

```
$ cd .
```

has no effect, since `.` refers to the present working directory. Finally, typing

```
$ cd
```

alone will usually put you into your *home directory*. This in the present example would be `/users/jsmith`, and normally the login name of the user in question would be jsmith. You can find out if this is indeed the case by typing

```
$ whoami
```

the expected reply being

```
jsmith
```

Creating directories. You can create a new directory by the `mkdir` command. For example, if you are in the directory `/users1/jsmith/euclid`, then the command

```
$ mkdir eucl_algorithm
```

will create a subdirectory called `eucl_algorithm` provided no directory or file by this name exists in the directory `/users1/jsmith/euclid`, if such a file or directory already exists, the computer will come back with a message explaining why the new directory could not be created. If you change your mind, and do not want the directory after all, you can remove it by typing

```
$ rmdir eucl_algorithm
```

as long as the directory is empty (i.e., no files or directories have been placed into the directory).

More on files. You can remove a file called `sec2` by typing

```
$ rm sec2
```

We mentioned above how to remove an empty directory by using the `rmdir` command. If the directory is not empty, i.e., if it contains files or other directories, then the command `rm -r` needs to be used. For example, the command

```
$ rm -r eucl_algorithm
```

removes the directory `eucl_algorithm` along with all the files and directories contained in it. This is a dangerous command, since before you use it you must be sure you want to discard all these files and directories, whereas the command `rmdir` is safe, since not much can be lost by removing an empty directory. In Unix parlance, `-r` after the command name `rm` is called an *option*.

You can change the name of the file `sec2` to `section2` by the command

```
$ mv sec2 section2
```

This is the first command that we mentioned that has two arguments: `sec2` and `section2`. There is a danger here; if there already exists a file called `section2`, this command will destroy it (and replace it with the file `sec2`. The `mv` command (which stands for "move", but typing "move" would not work) behaves differently if the second argument is an existing directory. For example, if there is a directory called `pascal_examples` in the current directory, then

```
$ mv sec2 pascal_examples
```

moves the file `sec2` into the directory `pascal_examples`.

The command

```
$ cp sec2 sec2.backup
```
will copy the file `sec2` to `sec2.backup`; if the file `sec2.backup` already exists, its old version will be lost. Assuming that `pascal_examples` is an existing directory in the current working directory, typing
```
$ cp sec2 pascal_examples
```
will copy the file `sec2` into this directory; if the file `pascal_examples/sec2` already exists, its old copy will lost. Finally,
```
$ cp -r pascal_examples pascal_examples.bak
```
will duplicate copy the directory `pascal_examples` along with all its subdirectories onto a new directory `pascal_examples.bak` (if a directory by this name already exists, it will not be destroyed; a copy of `pascal_examples` with all its subdirectories will be copied into this directory as a new subdirectory).

Wildcards in filenames. The asterisk `*` in filenames is called a wildcard, since it matches any string of characters. For example, the command
```
$ ls -d p*s
```
might produce the reply
```
papers   picture.ps   proposal   purchases
```
that is, files (or directories) with names that begin with `p` and end with `s`. The explanation for the option `-d`: if a file in question is a directory, then this option ensures that only the name of the directory is given; without this option, after the name of the directory, all the files in the given directory would also be listed if the name of the directory matches the pattern described in the argument of the `ls` command (in the present case the argument is `p*s`, that is, the string given after the command).

2. The vi Text Editor

Text editors enable one to create text files. Unix has a number of text editors; a widely used among them is called *vi* (for visual). You can start up *vi* by typing
```
$ vi filename
```
Before doing this, make sure that you are in the directory where you want to work in. The whole screen (or window, if you are using a window system – see below) will change, and you will be in *vi command mode*. In order to be able to enter text, you should type *i* and then you can enter text by using the keyboard as on a normal type writer. When you can enter text, the editor is said to be in *vi insert mode*. If you have entered a sufficient amount of text, or you feel you need to make some corrections, you can go back to *command mode* by pressing the *escape key*. If your keyboard does not have an escape key, you can achieve the same effect by typing `Ctl-[`, i.e., by holding down the *control key* and pressing `[`.

Cursor movement. The cursor is a little square that highlights a one-character size portion of the key, usually to indicate where the next keystroke will have its effect. In command mode, you can move the cursor by using the keys `h` (to the left), `j` (downwards), `k` (upwards), and `l` (to the right). The arrow keys to the right of the keyboard can usually be used in the same way, but it is better to use the keys `h`,

`j`, `k`, `l`, especially if you are a good typist, since by taking away your hand from the main part of the keyboard you will lose time. Also, the arrow keys do not always work. Other ways to move the cursor is by typing `w` (to the beginning of the next word), `b` (to go backwards by one word), `t`x (to the next occurrence of the character x on the same line), and so on.

With many of the above commands, you can use a number to perform the command repeatedly. For example, typing `2k` moves the cursor two lines up, typing `13l` moves it 13 places to the right, and typing `3b` moves it three words back.

Basic commands. Assuming you are in command mode to delete the character at the current cursor position, type `x`; to delete the whole line in the cursor position, type `dd`. To delete a word, type `dw`. To replace the character at the current cursor position, type `r` and then the new character. For example `rx` replaces the current character with `x`; typing `rr` will replace it with `r`. Finally, typing `r`$\langle LF\rangle$ replaces it with the line feed character; here $\langle LF\rangle$ stands for the *line feed* character, obtained when pressing the *enter key* (also called *return key*) on the keyboard.[3] That is, if you type `r`$\langle LF\rangle$ over a space character, the result is the break-up of the line into two lines at the given location. To join two lines, type `J` over the first one of them. To change a word, while in command mode, type `cw` with the cursor being on the first character of the word. The word will be deleted, and the editor will change to *insert mode*; that is, the text you type at the keyboard will be added to the file; you would usually type a replacement word and then go back to *command mode* by pressing the *escape key*. Typing `D` will delete the rest of the current line (the part to the right of the current cursor position), while typing `C` will allow you to change the rest of the current line.

You can use numbers with the some of the above commands. Typing `3dd` deletes three lines, typing `4x` deletes four characters. Typing `2dw` deletes two words, typing `d2w` does the same (the latter is perhaps the more logical order, since you can read it as saying "delete two words"); typing `2cw` or `c2w` will allow you to change two words.

To go into *insert mode* before the current cursor position type `i` (for "insert"), after the current cursor position, type `a` (for "add"). To go into insert mode below the current line (so as to insert a new line), type `o`, and above the current line, type `O`.

When in command mode, it is especially important not to press the *capitals lock key* accidentally. Unix is case sensitive, and the above commands have completely different meanings in upper case than in lower case, and typing a few capital letters in command mode instead of the intended lower case letters can cause serious damage to the file.

If you make a mistake, you can correct it by typing `u` when in command mode. This will take you back to the version of the file before the last change was made.[4]

---

[3] In Unix, the line feed character ($\langle LF\rangle$, decimal ascii code 10), is used to break lines. The Macintosh operating system uses the *carriage return* character ($\langle CR\rangle$, decimal ascii code 13) for this, and Windows uses $\langle CR\rangle$ and $\langle LF\rangle$ together.

[4] The editor *vi* behaves differently in Sun Unix and in Linux. In Sun Unix, typing `u` *toggles* between the current and the preceding version of the file; that is, typing `u` twice, takes you back to the current version. In Linux, typing `u` two or more times takes you to yet earlier versions of the file. To get back to the current version, you have to type `Ctrl-r` the same number of times as you typed `u`.

Command-line mode. When typing a colon `:` in command mode, the colon will appear at the bottom of the screen and the cursor moves down to the next place after the colon. The computer waits for you to enter a command on the bottom line; it is said to be in *command-line mode*. After entering the command and pressing pressing the *enter key*, the command in question will be executed and the editor will return to command mode. Typical commands are

```
:set nu
```

which will put line numbers at the beginning of each line. These lines will *not* be part of the file being created; the file itself will not be changed; the line numbers are useful in a long file for certain editing commands. The line numbers can be removed by typing

```
:set nonu
```

Typing

```
:16
```

will move the cursor to line 16. The command

```
:4,7d
```

will delete lines 4–7. Typing

```
:4,7j
```

will join lines 4–7 into a single line. Typing

`:4,7s/`*string1*`/`*string2*`/g`

replaces each occurrence of the characters *string1* with the characters *string2* in lines 4–7; the letter `s` after the range `4-7` stands for *substitute*, meaning that the second string gets substituted for the first one. In fact, this command is called the *substitute command* of `vi`. Here `g` stands for *global*. Typing the same thing without `g`, i.e., entering

`:4,7s/`*string1*`/`*string2*`/`

will replace only the first occurrence of *string1* in each of the lines 4–7. For example, the command

```
:4,7s/exit/exists/
```

will change the first occurrence of the word "exit" to "exists" in each of the lines in the *range* of lines 4–7, while the command

```
:12s/there is/there are/g
```

will change all occurrences of the text "there is" to "there are" in line 12. The command

`:4,7s/^/␣␣/`

will add two spaces at the beginning of lines 4–7. Here the symbol caret `^` indicates the beginning of the line, and the symbol ␣ indicates the space character; on the screen the space character is not shown (except that it occupies a space between two words, say), but it is customary to use the symbol ␣ to emphasize the presence of a space character. The character `^` is called a metacharacter in *vi*. This is because the character is used to denote something other (i.e., the beginning of the line) then the character `^` itself. Typing

```
:4,7s/$/;/
```

will add a semicolon at the end of each of the lines 4–7. This is because the metacharacter `$` denotes the end of the line in *vi*.

You can *escape* metacharacters by quoting them with the aid of the *Unix quote character* `\` (backslash). That is

```
:4s/\^/$/g
```

will replace all occurrences of the character `^` with the character `$` in line for; by placing `\` in front, the character `^` no longer acts as a metacharacter. Notice, that the character `$` in the *replacement text* in this example does not need to be escaped (in the replacement text, `$` never acts as a metacharacter – it would not make much sense to refer to the end of the line in the replacement text).

The command

```
:7s/\$/^/
```

will replace the first occurrence of the character `$` with a caret `^`.

The command

```
:4,7t19
```

will copy lines 4–7 after line 19. To move lines 4–7 after the current line 19, type

```
:4,7m19
```

The command

```
:14r sec2
```

will copy the contents of the file *sec2* after line 14, and the one

```
:0r ../README
```

will copy the contents of the file *../README* at the beginning (i.e., after line 0) of the current file. The contents of lines 23 through the last line can be written out to a new file called *transfer* by typing

```
:23,$w transfer
```

Note here that `$` refers to the last line of a file. If a file named *transfer* already exist, then this command will not do anything except complain. But, typing

```
:23,$w! transfer
```

will replace the current content of the file *transfer* with the contents of lines 23 through the last line of the file being edited (the original content of the file *transfer* will be lost).

```
:w
```

simply writes the current state of the file to the memory. Until you do this, the changes you made to the file are not permanent, and you can quit the editor and discard the changes by typing

```
:q!
```

This will get you back to the command line, and you can reenter *vi* again. Typing

```
:q
```

without the exclamation point will not let you quit the editor if you have made changes in the file without having saved these changes, i.e., without having typed

```
:w
```

afterwards. If you want to quit the editor while keeping the changes you made to the file, you can do this by typing

```
:wq
```

Typing *ZZ* in command mode will have the same effect.

Escaping to the shell. In many Unix programs, the exclamation point `!` is used as the shell escape character. In *vi*, this means the following. In command line mode, one can enter a Unix command after typing an exclamation point. For example, while in command mode, typing

```
:! ls
```

lists the files in the current directory (i.e., in the directory in which *vi* was invoked). Including the space here is optional. That is, typing

```
:!pwd
```

will print the present working directory. These examples also show that typing a space after the exclamation point is optional.

Buffers. You can put various parts of the text into temporary storage, called a buffer. For example, typing `yw` on the first character of a word puts ("yanks") this word into the buffer. To insert this word somewhere else in the text, move the cursor to the intended position; then typing `p` will write ("put") the content of the buffer (i.e., the word just yanked) after the current cursor position, while typing `P` puts the content of the buffer before the current cursor position.

In addition to the single *unnamed buffer*, there are named buffers to each lower case letter of the alphabet. To yank a word into the buffer named s, type `"syw` on the first letter of the word, and to put the content of this buffer into the text, type `"sp` or `"sP` in the appropriate cursor position.

To yank a whole line, type `Y` (or `"cY` to yank it into the buffer named c), to yank the text from the current cursor position to the character, say, `s` on the same line, type `yts` (read: *yank to s*.

You can use numbers with some of the above commands. For example, typing `y3w` will yank (put into the unnamed buffer) three words, and `2Y` will yank two lines.

Capital Lock key. Unix commands, including *vi* are case sensitive. Accidentally pressing the *Capital Lock/* key on the keyboard represent a special danger in *vi*. The commands meant to be typed in lower case cannot by typed in upper case, and typing them in upper case accidentally can cause serious damage to the file in a few key strokes. Therefore you need to exercise special care that you do not press the Capital Lock key inadvertently.

Editing commands. The command `fc` allows you to repeat the last command you typed, possibly in an improved form. Suppose you typed the command

```
$ cp problem
```

you will get an error message. The message is different for various Unix systems; it could be something like

```
cp: missing destination file
Try ‘cp --help’ for more information.
```

in Linux, or

```
cp:  Insufficient arguments (1)
Usage: cp [-f] [-i] [-p] f1 f2
       cp [-f] [-i] [-p] f1 ... fn d1
       cp -r|R [-f] [-i] [-p] d1 ... dn-1 dn
```

in Solaris. The problem, is of course, the missing second argument, since the command should look something like

```
$ cp problem question
```

This command will copy the (content of) the file `problem` to a file called `question` (which may be a new file, which then is created, or an old file, in which case the old content is destroyed).

The mistyped command can be corrected by typing the command

```
$ fc
```

This will invoke the editor `vi` with the mistyped command as the content of a

temporary file being edited.[5] If you make corrections in this file (using the editing commands of `vi`, discussed above), and quit the editor, the corrected command will be executed.

When typing a command and you realize that the command you typed is incorrect, you may rescue the situation by adding the pound sign `#` at the beginning of the line; this will prevent the command from executing. For example, the command

```
$ #cd vi_experiment
```

will do nothing. Then you may type

```
$ fc
```

to correct the mistakes in the command.

How to add the pound sign at the beginning of the current command line depends on the version of Unix shell you use. If you use *Korn shell* (the shell used at the Atrium), after noticing your mistake in the command, you press *escape*, then you press caret `^`, to get to the beginning of the line. This works the same way as in `vi`. Then, as in `vi`, you press `i` to go into insert mode), then you press the pound key `#` to add the latter character at the beginning of the line. Then press the *return key* to get to the next command line (as we mentioned, the command just edited will do nothing, since it starts with the character `#`).

In the *bash shell*, which is common in Linux, after noticing your mistake, you press `Ctl-a` (while holding down the *control* key, press `a`). This gets you to the beginning of the line. Then press the pound key `#` to add it at the beginning of the line. Then press the *return key* to get the the next command line.

3. Permission Codes

In the interest of facilitating file sharing between users of similar category or interest, users in Unix are placed into a number of groups. You can belong to one or more groups of users. You are able to give more privileges to a user that belongs to one of the same groups as you belong to, than to other users. We do not need to worry about user groups, but this much is needed to understand the meaning of permission codes. You may list files in the current directory with their permission codes by typing

```
$ ls -l
```

The answer you get may look like this:

```
total 8
-rwxr-xr--   1 jsmith   core5       83 May 29  1991 edg
-rw-------   1 jsmith   core5      440 May 22 17:27 euclid_new.p
-rw-r--r--   1 jsmith   core5        0 May 28 21:24 junk
dr-x--x-w-   2 jsmith   core5     1024 May 28 11:20 pascal
-rw-r--r-x   1 jsmith   core5     4698 May 25 18:44 sec5.tex
```

jsmith in the middle of the line indicates that these files are owned by the user called *jsmith* and by the group called core5; the file names are given on the far right. The

[5] Unix commands are often customizable, and so `fc` may behave differently; in particular, another editor, such as `emacs`, may be invoked. The editor to be used may be specified as the value of the shell variable FCEDIT

letter *d* at the beginning of the line listing the file *pascal* indicates that this file is a directory; the others are ordinary files. The next nine letters afterwards are the permission codes; the first three (after the very first letter) concerns the user (or, rather, the owner of these files, *jsmith* in this case), the next three concerns users in the same group or groups, and the last three concerns all other users. The letter *r* means the file is readable, *w* means the file is writable, i.e., can be changed or written into, and *x* means the file is executable (what this last one means will be explained later).

For example, the user *jsmith*, the owner of these files, can read, write into, or execute the file *edg*. Users in the same group can read and execute this file, but cannot write it. All other users can read this file.

For directories, the meanings of permission codes are not quite intuitive, so we give some explanation: If you have read permission for a directory, you can list its content by using the

```
$ ls
```

command. If you have write permission for it, then you can create new files in the directory, and delete any files (with the `rm` command) for which you have write permission (if you do not have write permission for a file, you cannot usually[6] delete it even if you have write permission for the directory containing it). Finally, if you have execute permission for it, you can use the `cd` command to change to it.

Permission codes can be changed by the *chmod* (change mode) command. The command

```
$ chmod o+x edg
```

adds execute permission to other users, the one

```
$ chmod g-r edg
```

takes away read permission from users in the same group, while the command

```
$ chmod u-w,g-x edg
```

will take away write permission from the user *jsmith* herself, and takes away execute permission from users in the same group.

Permission codes are sometimes indicated by octal (i.e., base 8) numbers. Take, for example the last item from the above list, and under each permission code write 1 where permission is given and zero where it is not given:

```
total 8
-rw-r--r-x   1 jsmith   core5     4698 May 25 18:44 sec5.tex
 110100101
```

The binary (base 2) number 110100101 can be written in base 8 as 645; if the same permission code is desired for the file *euclid_new.p*, this can be achieved by the command

```
$ chmod 645 euclid_new.p
```

---

[6] If you are owner of the file called, say, `myfile` but you have no write permission for it, upon typing

```
$ ls
```

you will get a reply such as

```
rm: myfile: override protection 444 (yes/no)?
```

(the meaning of the number 444 will be discussed just below) or

```
rm: remove protected file ‘myfile’?
```

If you answer this question by typing `y`, the file will be removed, and if you answer it by typing `n` (or anything else), it will not be removed.

CHAPTER II

THE WEB

After an introduction to the X Windows graphical user interface (GUI), a brief description of Netscape is given. Then we discuss how to set up your own Web page, and the basic features of the Hypertext Markup Language (HTML) are presented.

4. X Windows and Netscape

After logging in at a Sun workstation, you may choose to work directly at the standard Sun console (this is what you get after logging in). You get more flexibility by entering the OpenWindows environment. Openwindows is based on X Windows created at the Massachusetts Institute of Technology in the 1980s. Most Unix systems use a form of X Windows, and so whatever you learn on Sun workstations can be transferred to other implementations of Unix. You can enter the OpenWindows environment by typing

```
$ openwin
```

Several windows will open, and you can do your work in any of these windows by typing commands as if you were working at the console. You can do different work in each of the windows at the same time. You can open new windows as you need them and close unneeded windows.

First you need to learn how to manipulate windows by using the mouse. This is fairly easy, and a few minutes help session with a friend is much more productive than reading any manuals. After learning the basics, you may refine your windowing skills if you wish to do so by reading parts of the manual given elsewhere in these notes. The most important point is that the active window is always the one to which the mouse is pointing; before you can type in a window, you need to move the mouse so that it points to the window in question.

After finishing your session, you must first exit OpenWindows (usually by using the mouse), and then you must log out at the console (some setups will log you out automatically when you exit OpenWindows, but other setups will not, and you can never rely on being logged out automatically).

Netscape and the World Wide Web. There are certain applications that you will only be able to use in the OpenWindows environment, because these applications rely on X Windows resources. Notably among these is Netscape, which will allow you to look at information put on public view on computers all over the world. To open Netscape, type

```
$ netscape &
```

The effect of typing the symbol `&` (ampersand) at the end of the line is to run Netscape in the *background*. What this means in practical terms is that after Netscape is started, the window in which the above line was typed is immediately available for other use (such as editing ordinary files). If one types

```
$ netscape
```

i.e., if one omits the symbol & at the end of the line, Netscape will run in the *foreground*, and the window in which this line was typed will be unavailable until Netscape is closed; other windows will, however, be available.

After learning how to use the mouse in general, it is fairly easy to learn how to use Netscape. After starting Netscape, one can visit the Brooklyn College Web site by first clicking with the left mouse button on the word *File* in the Netscape windows; this will open a small window with a number of choices. By clicking with the left mouse button on the phrase *Open Location* among these choices, another window will open. After positioning the mouse pointer in this window, one can type the Web address of the College:

```
http://www.brooklyn.cuny.edu/
```

Then clicking with the left mouse button on the button labeled *Open* in this window will bring up the Web site. Usually, the Netscape window will not be able to display the whole content of the site, and one can move around by manipulating the scroll bar at the right-hand side of the window (possibly there is also a scroll bar at the bottom). One can do this either by clicking with the left mouse button on various points in the scroll bar, or keeping the left mouse button depressed on the scroll button and moving the mouse pointer; when the display is in the desired position, the mouse button should be released.

In the displayed page, certain locations will be *active* areas. These are usually underlined words in color, or certain parts of some pictures. One can find these active areas by moving the mouse pointer onto them (without depressing any buttons). The mouse pointer will change shape over active areas (from an arrow, it will change into a hand with an index finger pointing). By clicking with the left mouse button over an active area, the *link* contained at that area will be activated, and a new Web page will be displayed (or, possibly, another part of the same Web page will be displayed). The link is said to *point to* this new location.

There are many interesting things you can find on the Web. For example, if you are interested in information about a Unix-like operating system called Linux on your personal computer, you may visit the sites

`http://www.linuxHQ.com/` and `http://www.linuxworld.com/`

A wide range of computer documents about operating systems and program packages and computer use in general can be found at

```
http://riceinfo.rice.edu/Computer/Documents/#HDR19
```

A recent book about the World Wide Web, written by its inventor, is Tim Berners-Lee with Mark Fischetti, *Weaving the Web*, HarperSanFrancisco, 1999.

One can leave Netscape by first clicking with the left mouse button on the word *File* at the top of the Netscape window, and then clicking with the left mouse button on the word *Exit* among the choices displayed. As seen from the above description, the most frequently used mouse button is, at least initially, the left mouse button.

Bookmarks. You can use the *bookmarks* feature of Netscape to save your frequently visited Web sites. By pressing the *bookmarks* button in the Netscape window, you will quickly learn how to use this feature. Unfortunately, however, you

cannot conveniently store more than 30 or 40 bookmarks in this way, since the window showing the bookmarks will become unwieldy if you try to save more bookmarks. You might then be tempted to thin out your bookmarks and save only the most important ones, but there is a better way.

The bookmarks are usually stored in the file `bookmarks.html` in the directory `.netscape`; the latter is normally located directly in you home directory. Note that the directory `.netscape` is a so-called *hidden file* (see p. 88), so to list it you need to type `ls -a` rather than `ls`. What you can do is to make a directory called `bookmarks.dir` in the directory `.netscape`, and copy the `bookmarks.html` file into the former directory. Working in the directory `.netscape`, you would do this by typing, say

```
$ cp bookmarks.html bookmarks.dir/bookmarks5.html
```

if, say, you are doing this the fifth time (so the files `boommarks1.html` through `boommarks4.html` in the directory `bookmarks.dir` already exits. After doing this, you can delete all (except perhaps a few, most frequently used) items from your bookmarks. You should do this by using the *bookmarks* button in Netscape rather than deleting lines from the `bookmarks.html` file, since if you do the latter, Netscape will complain. The fact that Netscape complains can be considered a flaw in Netscape; at best, it is a *Unix-unfriendly* feature in Netscape. Ideally, Netscape should accept any changes made in its `bookmarks.html` file or in its other configuration files if these changes are correctly made, and should not force the user to use Netscape buttons to make configuration changes.

To go to a site stored in the just created `bookmarks5.html` file, you should proceed as if you were going to a Web side, as described above, but after clicking on the *Open Location* button, you should type

`file:`*homedirectory*`/.netscape/bookmarks.dir/bookmarks5.html`

in the window just opened, where *homedirectory* should be replaced with the absolute path to your home directory. E.g., if your login name is *cmuser*, your home directory may be `/home/cmuser`. In this case, you should type

```
file:/home/cmuser/.netscape/bookmarks.dir/bookmarks5.html
```

Note that the slash (`/`) after `file:` is part of the absolute location of the file. By the way, you should save this location as a bookmark. You may not like the name Netscape assigns to this bookmark; in this case, you may edit the `bookmarks.html` file located in the `Netscape` directory manually (i.e., by using, say, the `vi` editor) to give the name *bookmarks5* to this location; it is best to do this while Netscape is not open. Afterwards, Netscape might complain that the bookmarks have been changed, but apparently you can safely ignore this annoying complaint.

Netscape crashes. Sooner or later Netscape will crash on you; that is, it will close unexpectedly. When you next try to open it, it will give you a message saying that Netscape is already running. You are then offered to open a version of Netscape that is not fully functional; most likely, it will not be able to use its *cache*, a directory where items (such as files describing picture) from recently visited Web sites are stored. Rather than using such a dysfunctional version of Netscape, quit Netscape, remove the file called `lock` in the `.netscape` directory mentioned above, and start Netscape again. However, often, when Netscape crashes, the Netscape program (process, cf. p. 103) is still around and it must be dealt with (it must be killed). This is a somewhat more complicated issue, an it is discussed on p. 104.

Clearing Netscape's cache. Depending on the version of Netscape you are

using, it may not be very good about clearing its cache, mentioned just before, and a cache that is too cluttered may actually slow down Netscape. From time to time, you can clear the cache by typing the command

```
$ rm -rf /cache/*
```

in your `.netscape` directory. (See the discussion below why this command is somewhat *dangerous*.) Here the command `rm`, as we saw it above, is used to remove files or directories, option `-r` says that directories should also be removed (without this option directories are not removed), the option `-f` forces removal (more about this below), and the two options can be stated together as `-rf`. Finally, `*` stands for any file or directory (in fact, technically directories are also files), so the above command removes all files or directories located in the directory `cache`.

Normally, one does not need to use the `-f` option with the `rm` command to force removal of files. However, especially for novice users, the command `rm` is often a *shell alias* for the command `rm -i` (shell aliases are discussed on p. 121). If so, whenever you type `rm` on the command line, actually the command `rm -i` is executed. The option `-i` stands for *interactive*. That is, for each file encountered that, you get a question whether you want it removed. If the file is called, say, `whatsit`, you get the question

```
rm:  remove ‘whatsit’?
```

and you have to answer this question with `y` or `n` for *yes* or *no*. If there are too many files to be removed, this may get to be too cumbersome; the option `-f` is used to override the option `-i`.

If you are in the directory `.netscape/cache`, you can type

```
$ rm -rf *
```

instead of the command given above. This command is, however, *very dangerous* since it removes everything in the directory that you happen to be in; if you type this line mistakenly believing that you are in the `.netscape/cache` directory, the result can be disastrous. In fact, the first version of this command, given above is also somewhat dangerous, and it may be altogether safer to use a *shell script* to clear Netscape's cache (see p. 99 in Section 23).

5. Starting to Build Your Own Web Page

If you have an account on the Unix network at the Atrium, you can build your own Web page. To do this, as the first step you need to make your home directory accessible to the Web server by changing its permission code to 711. You can do this by typing

```
$ cd
$ chmod 711 .
```

Here the first line puts you into your home directory; if you already there, it is unnecessary but you can still type it, since then it will do nothing. The second line changes the permission code of your home directory. The dot "`.`" in this line refers to the directory you are currently in (the present working directory), which will be your home directory, since you just changed to it in the preceding line. Permission codes and the command *chmod* were described above. The permission codes assigned

by this will allow anyone in the world to enter your home directory without being able to make any changes in it, and without being able to read its content; that is, someone entering your home directory, and then trying to list the names of the files in it by typing

```
$ ls
```

will get a reply such as

```
.unreadable
```

or something similar.

As the next step, you need to make a directory called *public_html* in your your home directory by typing

```
$ mkdir public_html
```

Here *html* abbreviates *Hypertext Markup Language*; what this is will be explained later. Next, the correct permission codes must be assigned to this directory; this can be done by typing

```
$ chmod 711 public_html
```

As we mentioned above, permission codes assigned by this will allow anyone in the world to enter this directory without being able to make any changes in it, and without being able to read its content. At this point, you should enter this directory by typing

```
$ cd public_html
```

In this directory, a file called *index.html* with permission code 644 must be created; this can be done by the following commands:

```
$ touch index.html
$ chmod 644 index.html
```

Here, the command *touch* creates an empty file, and the command *chmod* afterwards changes its permission code to the code desired. In practice, instead of using the command *touch* one can use a text editor such as *vi* to create the file *index.html*, and then use *chmod* to change the permission code. In fact, it is advisable to check the permission code at the end by typing

```
$ ls -l index.html
```

since some of the manipulations of this file may have changed the permission codes, and, if necessary, to use *chmod* to change the permission code to 644. This permission code allows anyone in the world to read the file *index.html*, but does not allow them to write to this file.

The next task is to actually write the file *index.html*. This can be done by typing

```
$ vi index.html
```

and then typing the appropriate text into the file. In the next section we will discuss what this text ought to be.

6. Hypertext Markup Language

The file *index.html* has to follow the format, that is, *syntax*, of *html*, or *Hypertext Markup Language*. To explain how this is done, consider the Web page displayed on p. 27. The *index.html* file specifying this Web page used to be located at the site

`http://sci.brooklyn.cuny.edu/~mate/`

In July 1998, this site contained the following text (note that the site has since been updated, and some of the links in this file are no longer valid):

```
<HTML>
<HEAD>
<TITLE>Attila Mate's Home Page</TITLE>
</HEAD>

<BODY>

<BR><HR><BR>
<CENTER>
 <H1>Attila M&aacute;t&eacute;'s
        Home Page</H1>
</CENTER>
<BR><HR><BR>

<IMG ALIGN=LEFT BORDER=0 SRC="Matecsm.gif"
    ALT="Picture"
HSPACE=25>
<!--the ALT attribute defines an alternative text to
the picture, used by text browsers-->
<ADDRESS>
<BR>
Attila M&aacute;t&eacute;<BR>
Department of Mathematics<BR>
<A HREF="http://www.brooklyn.cuny.edu/"
>Brooklyn College </A><BR>
2900 Bedford Avenue<BR>
    Brooklyn,</A> NY 11210-2889<BR>
USA<BR>
<A HREF="mailto:mate@sci.brooklyn.cuny.edu">e-mail:
    mate@sci.brooklyn.cuny.edu</A><BR>
office phone: (718) 951-5246<BR><BR>
Photograph by C. J. Mozzochi
</ADDRESS>

<BR CLEAR=LEFT>
<!--The above line moves the text position vertically
down to the next unobstructed left margin-->
<BR>
<HR><BR>
 
 
<!--This is the space character, to create
indentation-->
To see my letter sent to the London Sunday Times in
response to a negative article about Linux,
<A HREF="http://www.ssc.com/linux/suntimes/rebut.html"
```

```
 47  >click here</A>.
 48  To my knowledge the letter has not been published in
 49  the London Times.
 50  You may prefer to read a discussion of the London
 51  Sunday Times article first. To do this,
 52  <A HREF="http://www.ssc.com/linux/suntimes/"
 53  >click here</A>.
 54  This discussion has links both to the original
 55  article, and to my letter.
 56  You can also go directly to the
 57  <A HREF=
 58  "http://www.the-times.co.uk/news/pages/sti/97/04/20/st
iinnsnd01001.html?1007000">The Times Internet Edition</a>
 59  to read the original article first, then come back
 60  here to follow the links to the discussion and to
 61  my letter.
 62  <P>
 63  <BR>
 64  Last updated:
 65  Wed Jul 22 10:06:03 EDT 1998.
 66  </BODY>
 67  </HTML>
```

First note that the numbers on the left are line numbers and they are not part of the file. We included them for easy reference. In particular, line 44 is longer than can be displayed in the editor window, and so it is wrapped around (broken up in the display, but the continuing part does not receive a new line number since, since it does not constitute a new line in the logical sense).

To understand HTML, first note that the text to be displayed is interspersed with short strings of symbols intended to control the way the text is displayed; these strings of symbols are called *HTML tags*. We will discuss the tags occurring in the above text. A systematic list of other tags can easily be found on the World Wide Web (see below). The text begins on line 1 with the tag `<HTML>`, which indicates that the following text is an HTML document; this tag comes with a pair, `</HTML>` on line 67, indicating the end of the HTML document. Some tags come singly, others come in pairs; in the latter case, the second member of the tag is preceded by a slash `/`. Also note that the tags are enclosed between the symbols `<` (less than) and `>` (greater than).

Header information is placed between the par of tags `<HEAD>` and `</HEAD>` (see lines 2 and 4), and as part of the header information, the title of the Web page is placed between the pair of tags `<TITLE>` and `</TITLE>`. The title of the Web page is not part of the information displayed on the Web page itself; it is displayed in the title bar at the top. Note that we used capital letters in tags so that they can be easily identified; however, tags are not case sensitive, i.e., they can just as well be written lower case. The tags `</title>` and `</TITLE>` mean the same thing.

The main part of the Web page is placed between the tags `<BODY>` (on line 6) and `</BODY>` (on line 66). Note that leaving blank lines such as lines 5 and 7 has no effect. The new line character and the space character in the file are treated the same way, and if they occur in the text, several space and new line characters have

the same effect as a single space character (a space between words). Line breaks in the displayed text can be forced by the tag `<BR>`; this tag has no pair – it is a *solitary* tag. The three tags occurring on line 8 are `<BR><HR><BR>`. The first and third of these forces line breaks, resulting in vertical spaces left that correspond to the space taken up by a line. The middle tag, `<HR>`, is a *horizontal rule* (horizontal line); this tag also a solitary tag. The pair of tags `<CENTER>` (on line 9) and `</CENTER>` (on line 12) indicates that the text between them should be centered, and the pair of tags `<H1>` (on line 10) and `</H1>` (on line 11) indicates that the text in between is a heading (set in large type); there are six different levels of headings, `<H1>`, `</H1>` being the largest, and `<H6>`, `</H6>` being the smallest. The letter *á* with the acute accent is produced by the string `&aacute;` (including the semicolon) in line 10, while the letter *é* in the same line is produced by the string `&eacute;` (including the semicolon). The line break at the end of line 10 and the several space characters at the beginning of line 11 appear as a single space in the displayed text.

The tag `<IMG>` in line 15 is used to include an image (picture) in the Web page. The expression `ALIGN=LEFT` involving the *attribute* `ALIGN` and its *value* `LEFT` indicates that the image should be on the left of the page. Tags in HTML describe *elements*; that is, part of the document; *attributes* modify the appearance of these elements in ways specified by their *values*. The attribute `BORDER` allows us to place a border around the image; here the width of the border is specified to be 0, so no border appears; the attribute `BORDER` can be omitted. In fact, attributes can often be omitted; in this case their *default* value (i.e., the value assumed by the computer) are used.

The image itself is described in the file `Matecsm.gif`, located in the directory `public_html` (i.e., in the same directory in which the file `index.html`, being discussed, is located). The file `Matecsm.gif` is a *GIF-file*; GIF stands for "Graphics Interchange Format." The file is produced by scanning a photograph into the computer; it could also have been produced by a digital camera. The attribute `SRC` indicates that this file is the *source* of the image. This attribute is *required*, i.e., it cannot be omitted. The attribute `ALT` in line 16 introduces the text given in quotes (i.e., *Picture*) that is to be displayed by a text-based browser (such as *lynx*), which is unable to display images. Finally, the attribute `HSPACE` in line 17 places horizontal space of the amount specified by the number 25 on both sides of the image; without this horizontal space, the image would appear all the way to the left on the page, and the text on its right-hand side would appear right at the edge of the picture.

In lines 18–19, the *delimiters* `<!--` and `-->` enclose a comment; that is a remark that is not visible in the Web page. Comments are useful for the persons designing and maintaining (modifying, updating) the Web page. The form of a comment is

`<!--`*text*`-->`

The pair of tags `<ADDRESS>` (in line 20) and `</ADDRESS>` (in line 33) enclose the address of the author of the Web page; they make the information placed between them appear in italics. This tag is used more for documentation than for formatting the page.

In lines 24 and 25 the pair of tags `<A>`...`</A>` enclose what is called an *anchor*: part of the text on which you can click (with the left mouse button) to produce some action, such as moving to another Web page.

The tag `<A>` requires at least one *option*; that is, this tag never occurs by itself, it is followed by further information between the symbols `<` and `>`. The most important

Attila Máté's Home Page

Attila Máté
Department of Mathematics
Brooklyn College
2900 Bedford Avenue
Brooklyn, NY 11210-2889
USA
e-mail: mate@sci.brooklyn.cuny.edu
office phone: (718) 951-5246

Photograph by C. J. Mozzochi

To see my letter sent to the London Sunday Times in response to a negative article about Linux, click here. To my knowledge the letter has not been published in the London Times. You may prefer to read a discussion of the London Sunday Times article first. To do this, click here. This discussion has links both to the original article, and to my letter. You can also go directly to the The Times Internet Edition to read the original article first, then come back here to follow the links to the discussion and to my letter.

Last updated: Wed Jul 22 10:06:03 EDT 1998.

among this is the option `HREF`, which introduces a *hyperlink* (or hyper-reference). The form of an anchor with a hyperlink is

`<A HREF="`*link target*`">`*anchor text*`</A>`

The *link target* is the location of the Web page to be transferred to; the *anchor text* is the text that is actually displayed on the page, that is, on which one needs to click in order to get the target of the link. For example, in the anchor given in lines 24–25, if one clicks on the displayed text *Brooklyn College*, then one is transferred to the Web site of Brooklyn College, which is located at

`http://www.brooklyn.cuny.edu/`

Another type of anchor is given in lines 29–30. By clicking on the anchor text

mate@sci.brooklyn.cuny.edu, a window is opened in which one can type a letter to be emailed (that is, sent by electronic mail; more about electronic mail later, in Section 22 on p. 86). The link target

```
mailto:mate@sci.brooklyn.cuny.edu
```

contains the electronic mail address that the letter should be sent to. Note that the link target itself is not displayed; what is displayed is the anchor text, which happens to be very similar; the reason to display the electronic mail address in the anchor text is that some people may prefer to take note of the address and send email on some other occasion, without using the automatic addressing feature provided by this anchor. More about the types of link targets, called *URI*'s or *Uniform Resource Identifiers*, also called *URL*'s or *Uniform Resource Locators*, can be found at

```
http://cie.motor.ru/RFC/1630/index.html
```

In line 35, the tag `<BR>` with the attribute `CLEAR` given the value `LEFT` inserts a vertical space until the left margin is clear, so that the text will continue underneath the picture.

The main text of the Web page begins at line 44. The paragraph ends on line 61, and the beginning of the new paragraph is indicated by the solitary tag `<P>` on line 62. Actually, this is not a solitary tag; the tags `<P>` and `</P>` form a pair, the former indicating the beginning of a paragraph and the latter its end; however, the latter tag is usually omitted since the beginning of a new paragraph also signals the end of the preceding paragraph. Most browsers, including Netscape, display the separation of two paragraphs by adding vertical space corresponding to a blank line.

In order to indent the text starting on line 44, two space characters added at the beginning of the line. A space character has the code "` `";[7] note that the first semicolon is part of the code. On line 63, the line-break tag `<BR>` inserts additional vertical space.

The anchor on line 58 is of special interest. The name of the link target is too long to fit in a single line of the window of the text editor. For this reason, the text editor wraps this line, so that it can be conveniently viewed, but from a logical view point this is a single line, since it contains no line feed character. Line feed character must not be included, since the name of the link target must be reproduced exactly.

As a point of fact, Netscape ignores line feed characters in the name of the link target. Hence line 58 could have been broken up into three lines without changing the appearance of the Web page:

```
44  "http://www.the-times.co.uk/news/pa
45  ges/sti/97/04/20/stiinns
46  nd01001.html?1007000"
```

(we included line numbers – these are not part of the text; in this case, the line numbers of the successive lines have to be increased accordingly). However, this appears to be a feature of Netscape, and the documentation of HTML does not seem to explicitly state that such line breaks are allowed. For this reason, such line breaks should be avoided, since they may cause trouble for some browsers other than Netscape.

We will not touch upon the more complicated features of HTML; there are many books available on HTML. One can also learn more about HTML by consulting various Web sites. The Web site of the World Wide Web Consortium,

[7] One needs to be precise here about what is quoted; hence one semicolon inside and one outside the quotation marks.

`http://www.w3.org/`
is a source of authoritative information. At this site, at the address
`http://www.w3.org/TR/REC-html40/`
one can find the specifications of HTML 4.0.

Legal issues. Please note that putting a page on the Web constitutes publication, and as such there are certain legal issues that need to be kept in mind. First, a publication may be libelous, or may violate copyright laws or obscenity laws. Finally, the College may withdraw computer privileges upon the publication of certain material that violates College's acceptable use policies. It is easy to use common sense to safely stay away from the pitfalls; in order to navigate the fringes, you need to study the issues involved carefully.

Geekspeak. When trying to read Web pages about computers, you will often run into technical jargon that can make your reading difficult. On occasion, you may also run into fashionable "geek" slang that will further obscure your reading material. An example for the latter is YMMV, which stands for "your mileage may vary."[8] For instance, the writer may talk about the installation of a certain software, explaining that you should except such and such a performance, but YMMV, meaning that the result you get may differ, depending on your setup. At the site
`http://online-today.com/geekspeak`
you can find a dictionary of technical jargon and slang that may prove helpful. We would discourage the use of "geek" slang and excessive technical jargon, since it is much better to be understood than to appear "knowledgeable."

Secret Web pages. In order to view a file with Netscape, you need to know the name of the file. So if you put a file called `for_joe3782` in your `public_html` directory, and tell this file name only to your friend Joe, no one else can read this file without somehow first finding out its name. This works because the permission code of the directory `public_html` is 744, that is
`drwx--x---`
(the `d` indicates that `public_html` is a directory, the rest is its permission code), so the general public has only permission to change to this directory, and has no permission to list its content. Of course, as usual, the file `for_joe3782` must have permission code 611. You can think of the number 3728 as a *PIN number* (Personal Identification Number) used by bank cards to protect your money at ATMs (Automatic Teller Machines).

7. Empowering Netscape

There are various ways to make Netscape work for you better. We will give some examples.

[8] This phrase is commonly used by automobile manufacturers when rating the fuel consumption of a vehicle. They might say something like a car rates "35 miles per gallon highway and 25 miles per gallon city, but your mileage may vary, depending on driving conditions."

Bookmarks on the Web. Instead of creating the directory `bookmarks.dir` in the `.netscape` directory as described toward the end of Section 4, you can create it, for example, in your `public_html` directory After assigning the permission codes as described Section 5 (711 for directories and 644 for files), you can then look at bookmarks in question by typing

`http://`*your_web_site*`/bookmarks.dir/bookmarks5.html`

in the *Open Location* window of Netscape if you called the file containing the bookmarks `bookmarks5.html`, as at the end of Section 4. This solution will only work on a computer where you have an accessible web page. In this example, the phrase *your_web_site* should be replaced with the actual address of your Web page. For example, if your login name is *cmuser*, and you have a working Web page at the Atrium, you can access the above bookmarks by typing

```
http://acc6.its.brooklyn.cuny.edu/~cmuser/bookmarks.dir/boo
kmarks5.html
```

the above line needs to be typed in a single line; since it is too long to print it as a single line in these notes, we scrolled its end over into the next line. You should have no problem with typing the line in the *Open Location* window without breaking it up. Here `acc6.its.brooklyn.cuny.edu` is the address of the Web server at the Atrium Computer Laboratory of Brooklyn College. The advantage of this solution is that you can have remote access to your bookmarks. So will everyone else who knows where to look; to prevent this, instead of naming the directory `bookmarks.dir`, you may give it some unlikely name, such as `bookmarks4796.dir`. Note that unauthorized users cannot easily find out the name of this directory, since it is located in the directory `public_html`, and no one else than you has read permission on this directory, since its permission code is 711. So, while all users can change to the directory `public_html`, they cannot list its content by typing `ls`.

Viewing PostScript files. Many documents written in the page definition language PostScript (see p. 79) are available on the Web. PostScript is especially useful to display mathematical writings with complicated formulas. Mathematical documents are often written in the typesetting language TEX(see p. 78) can be routinely converted into PostScript; to convert them into HTML is often difficult, if not impossible.

A free PostScript viewer called *ghostview* is available for Unix systems, it is a standard part for many Linux distributions. A way to view PostScript documents on the Internet is first to download them to the local system and then view them with *ghostview* on the local system.

If a link is given on a Web page to a PostScript document, Netscape can be configured to automatically invoke *ghostview* to view it just by clicking on the link, so that one does not have to go through the separate steps of downloading the document and then invoking *ghostview*. To see whether Netscape is already so configured, in, say Netscape 4.51, click first on *Edit* on the top bar of the Netscape Window, then click on *Preferences* in the small drop-down window. A large window will open up; click on *Navigator* in the left column, and then click on *Applications* among the subitems that appear. In the middle of the window, you will see a number of applications that Netscape can invoke. For example, PostScript might have the following entry:

```
Description                    Handled by
.....................................................
PostScript Document            /usr/X11R6/bin/ghostview %s
.....................................................
```

If an entry like this is already present, there is nothing to do. If not, you need to add it by clicking on the button *New* in the lower part of the window. A window entitled *Netscape: Application* will open up; you will have to make the entries given in the right column:

```
Description:              Postscript Document
   MIMEType:              application/postscript
   Suffixes:              ai,eps,ps
```

In the lower part of the window, under the major heading *Handled By*, click on the diamond next to the word *Application* so as to highlight the diamond, then enter the location of the *ghostview* program, followed by a space and the string `%s`. First you need to find out the location of the *ghostview* program. The best way do this is to type

```
$ which ghostview
```

This command will tell you the location of the program *ghostview* that is invoked when you type the command

```
$ ghostview
```

Alternatively, you can also type

```
$ whereis ghostview
```

but the result is harder to interpret, since `whereis` gives also other information (e.g., the location of the online manual pages). If the answer is

```
/usr/X11R6/bin/ghostview
```

then you make the appropriate entry in the small highlighted area on the right:

```
Application:              /usr/X11R6/bin/ghostview %s
```

Having enabled Netscape to view PostScript files, when using clicking on link to such a file, a separate *ghostview* window will open to display the document. This window can be closed simply by typing a letter `q` anywhere in the main part of the window.

Enabling Netscape for viewing PostScript may have a side effect that is easy to deal with. When clicking on a link representing a PostScript file in a Web page, Netscape tries to download the file unless Netscape is enabled to view PostScript files as described above. If you enabled Netscape to do this, and you still want to download the file rather than view it, you need to shift-click on the link; that is, depress the *Shift* key on the keyboard, and then click on the link. This will prevent Netscape from viewing the file. If you do not know whether, after clicking on a link Netscape would display the item or download the file, it is alway safe to shift-click to make sure that Netscape will download it rather than display it.

Data Compression. The idea of data compression goes back to *Claude E. Shannon*, the founder of *Information Theory*, who came up with the idea in the late 1940s. In the ASCII code, a letter is represented by an eight bit code. In everyday English, the letter e occurs significantly more often than the letter x; for this reason, the number of bits used to represent an English text can be lowered if the letter e is described by a shorter code than the letter the letter x. This idea can be generalized

arbitrary binary files: frequently occurring bit sequences can be represented by a shorter code than rare bit sequences.

The compression of images is based on a different idea. A picture is represented in a computer by storing all the *pixels* (*pic*ture *el*ements, i.e., the brightness and color information for each point of the picture). Since nearby pixels usually differ very little (in a human face, nearby points are usually of nearly the same color, for example), instead of storing all the pixel values, for most pixels it is more efficient to store how it differs from a nearby pixel that is stored in full.

There are various mathematical algorithms for compressing images. The requirements for image compression are less stringent than for compressing a file containing English text. The compression of a text file should be *lossless*; that is, from the compressed file one must be able to reproduce the original text exactly. For image compression, it is often permissible to be *lossy*; it may not matter if the original image data cannot be reconstructed if the human eye cannot tell the difference between the original image and the image reconstructed from compressed data. *GIF* is a lossless image compression scheme, while *JPEG* is a lossy one. The advantage of a lossy image compression scheme is that usually greater compression can be achieved.

Compressed Web pages. The file compression program *gzip* and its sister *gunzip* used for uncompressing files compressed by *gzip* is available free for Unix systems, and it is included with most Linux distributions. You can compress your *index.html* file by typing

```
$ gzip index.html
```

The compressed file will be called *index.html.gz* while the *index.html* file itself will be deleted. In general, *gzip* will attach *.gz* to the filename to name the compressed file. You can uncompress the file *index.html.gz* to get back the original file *index.html* by typing

```
$ gunzip index.html
```

Note that nothing is gained by compressing the same file twice. The purpose of compressing files is to save disk (storage) space, and, even more importantly, to save time needed for transmission of files, especially through slow telephone lines. You can put a compressed copy of your *index.html* file in the *public_html* directory for faster retrieval of your Web page; but note that you ought to retain a copy of the file *index.html* as well, since most browsers are not enabled to read compressed files. As *gzip* deletes the original copy of the files, you need to make special arrangements to retain a copy. This is easy to do. For example, while working in the directory *public_html*, you can accomplish this as follows:

```
$ cp -p index.html index.html_temp
$ gzip index.html
$ chmod 611 index.html.gz
$ mv index.html_temp index.html
```

Here the command `cp -p`, i.e., the *-p* option of the command *cp*, is used to copy a file while preserving its attributes (such as permission codes, the time of the creation of the file, etc.). The *chmod* on the third line is necessary, since one cannot be sure that the permission codes of the compressed file will be 611.

One can enable Netscape to view compressed files in a way similar to how we enabled to view PostScript. After clicking on *Navigator* and *Applications* in the *Preferences* window, as before, check if *compression* is enabled. If not, then click on

the button *New* as above, and make the following entries:

```
Description:          gzip compressed data
   MIMEType:          application/gzip
   Suffixes:          gz
Application:          /bin/gunzip %s
```

Now if at the Web site

`http://sci.brooklyn.cuny.edu/~mate/`

there is also a compressed file *index.html.gz* in the directory *public_html*, then you can load it into a *gunzip*-enabled Netscape by going to the site

`http://sci.brooklyn.cuny.edu/~mate/index.html.gz`

However, this is of limited usefulness, since only the text of the *index.html* file is compressed this way, and text does not take that much time to load anyway, unless very long. What really takes time is the loading of the images linked in the file *index.html*. There is no use to compress these images with *gzip* since their format, most often *GIF* or *JPEG* (the latter stands for Joint Photographic Expert Group, the original name of the committee that wrote this compression standard), is already compressed, and there is no sense in compressing a compressed file, since in general no further reduction in the file size can be achieved by repeated compression. In any case, Netscape cannot handle images compressed by *gzip* that are linked to automatically load into a Web page even if you set up Netscape to handle compression.[9]

On the other hand, if you enabled Netscape to handle both PostScript files and compressed files, it will also be able to handle compressed PostScript files, as one would expect. This is very useful, since PostScript files are not compressed, and a significant reduction in their size can be achieved through compressing them by *gzip*.

Enabling Netscape to handle compressed files has a side effect similar to the one we mentioned above in connection with PostScript. When clicking on a link to a file with a name ending in `.gz`, Netscape tries to view it rather than download it. This may be a problem, since many such files are not viewable at all. To handle this problem, it is always safe to shift-click on such a link rather than click on it. If you accidentally clicked on such a link to a file called, say

`ftp://acc5.its.brooklyn.cuny.edu/pub/mate/somefile.tar.gz`

stand by; you do not need to stop Netscape.[10] Netscape will download the file `somefile.tar.gz` into it *chache* (see p. 21), and when the download finishes (which may take some time if the file is large), netscape will display an error window saying something like unknown suffix; this refers to the suffix `tar`[11] in the file name. If

[9] That is, it cannot handle a file called *picture.jpg.gz* obtained by compressing the JPEG file *picture.jpg* if it is included in a Web page using the HTML tag `<IMG>`, described on p. 26. Compressing a JPEG file with *gzip* does usually result in a minor reduction of file size.

[10] The letters `ftp` stand for *File Transfer Protocol*, one of the ways to transfer files between different computers

[11] The suffix `tar` stands for Tape Archive, a Unix utility originally used for making tape backups. However, it has many other uses. Documents with a complicated directory structure can be put into a single file using the `tar` command. Such *tar* files, often called *tarballs*, often compressed with `gzip`, form one of the favorite ways to store software on the Internet.

you click on the button *OK* in this window, the window will go away. The you can properly download the file by shift-clicking on the link. This time, however, Netscape will not spend time with downloading the file again: it will take the file from its *cache*.

Command line control of Unix Netscape. By typing

```
$ netscape http://linuxtoday.com/ \
> http://www.theregister.co.uk/ &
```

two Netscape windows will open, one at each of the Web sites listed. Note here that the `$` sign in the first line is the prompt. The backslash at the end of the first line is the *Unix quote character*. Its role is the quote the *end-of-line character* (the character making the display start at the beginning of the next line). The end-of-line character (usually created by pressing the *Enter* key on the keyboard), normally signifies the end of a command; but if the end-of-line character is quoted, the computer displays the *greater than* sign `>` at the beginning of the next line (that is, the sign `>` is *not typed* by the user), and the command than can be continued on the second line. The character `&` at the end of the second line makes sure that the process runs in the background, so that the window in which we typed the above command will remain usable (this was explained in p. 20 after we first discussed running Netscape – see Section 26 for more information about running jobs in the background). The above command could have been typed on a single line, but, since it is quite long, it is convenient to break it up.

The character `>` at the beginning of the second line in the command above is the *secondary shell prompt*, while the dollar sign `$` at the beginning of the first line is the *primary*, or main, *shell prompt*. Both prompts can be changed by the user, so the symbol at the beginning of the second line may be different.[12]

One can also give commands from the command line to Netscape when the latter is already running. For example, the command

```
$ netscape -remote \
> 'openURL(http://www.linuxworld.com/,new-window)'
```

opens a new window at the site

`http://www.linuxworld.com/`

in an already running Netscape. Note, that, as above, we used two lines to write this command. Also observe that this command does not need to be run in the background.

If one wants to go to the same Web site in an already running Netscape, but wants to use an already open window rather than to open a new one, one can type

```
$ netscape -remote 'openURL(http://www.linuxworld.com/)'
```

More about controlling an already running Netscape in Unix can be found at the Web site

`http://home.netscape.com/newsref/std/x-remote.html`

[12]The primary prompt is the value of the shell variable (see p. 92) `PS1`, and the secondary prompt is the value of the shell variable `PS2`. The default values for these variables are usually `$` and `>`.

Mozilla. A new offshoot of the browser wars, in which Microsoft made Internet Explorer its standard browser and thereby much reduced the use of Netscape, is the browser Mozilla; see

http://www.mozilla.org/

In the late 1990s, the code for Netscape was completely rewritten, and the result of this effort is the open source browser Mozilla and the new version of Netscape (with version number 6 or higher). An important virtue of these browsers is their strict compliance with the standards of the World Wide Web Consortium, so as the prevent the fracturing of the Web. The command line controls of Unix Netscape described above also work with Mozilla, if one replaces the word `netscape` with `mozilla` (provided Mozilla is installed on the system).

CHAPTER III

PASCAL

We give an introduction to the Pascal programming language, and explain how to compile and run Pascal programs in Unix. Loops and conditional statements and the tracing of programs are discussed.

8. Introduction to Pascal

The language of computers, called *machine language* uses certain strings (sequences) of 0's and 1's. This language is very cumbersome for people to work with. It was known from the theoretical work of the British mathematician Alan Turing that computers are in a sense *universal machines*: any systematic calculation that can be done at all can in principle be done on a computer (subject to space and time limitations). This gradually led to the realization that instructions for a computer can be written in a way that are easier for people to comprehend than strings of 0's and 1's, and the computer itself can be used to translate these instructions into its own machine language. One can envisage a "sufficiently smart" computer that can take instructions in English, and translate the instructions into its own machine language. However, there are several difficulties with using English. First of all, the rules of English are too complicated, secondly, the meaning of English is often ambiguous. A compromise needs to be made: the language needs to be reasonably easy for people to use, it needs to be reasonable simple for the computer, and it certainly needs to be unambiguous. Among the first such high level languages was FORTRAN (Formula Translator), proposed in the early 1950's by John Backus.

The programming language Pascal, named after the seventeenth century French mathematician Blaise Pascal, was developed in the late 1960's under the leadership of Niklaus Wirth. In the early 1980s it was one of the most popular programming languages, and it is still one of the most elegant languages. Recently its popularity has given way to the language called C. While Pascal is easier to learn, and mistakes in Pascal programs are usually easier to find, C is closely associated with the Unix operating system. It is unlikely, however, that the world will soon agree on a single programming language. For example, FORTRAN is still important, since it handles large numerical calculations more efficiently than C or Pascal. Then there are many computer languages used in specialized areas; for example, we met HTML before (see Section 6 on p. 23), and we are going to meet the mathematical typesetting language TEX (see Section 21 on p. 78).

Whatever the future of Pascal, after learning to program in Pascal it is much easier to learn other programming languages.

One of the simplest Pascal program is the following:

```
1  program hello(output);
2  begin
3    writeln('Hello world!')
4  end.
```

The numbers on the left are not part of the program; they are line numbers that allow us to more easily explain what this program does.

When *run*, i.e., *executed*, or made to do its work, the only visible action of this program will be to write

```
Hello world!
```

on the screen. Before we explain the meaning of the lines of this program, we will describe how to run this program.

The first step is to create a file called, say `first_program.p` containing the above text. It is traditional in Unix to end names of Pascal programs with `.p`. In DOS parlance, `.p` is called file name extension (because in the old operating system DOS for personal computers, the file name consists of a main part, and a period . followed by what is called a file name extension). In Unix, it is not correct to talk about file name extension; simply, Unix allows periods in file names, and `.p` is just the last part of the file name; some Unix setups require that file names of Pascal program end this way, others do not (in DOS, the required file name extension for Pascal programs is usually `pas`).

As a matter of good work organization, it is better not to create the file `first_program.p` in your home directory. Instead, you can make a new directory called, say, `pascal_programs` by typing

```
$ mkdir pascal_programs
```

on the command line, and create the above file in this directory. Recall that you do not type the `$` at the beginning of the line; it is the prompt provided by the computer to indicate that you may type a command. If you already created the file `first_program.p` in your home directory, you can use the `mv` command, described in the introductory section on Unix (see p. 11 in Section 1), to move it into the newly created directory. All your subsequent Pascal programs should be put in this directory (when you have too many Pascal programs, you may create a number of different directories to organize them).

As the second step, this program must be *compiled*, that is, translated into the computer's own machine language. This is done by invoking the *Pascal compiler*, usually called `pc`

```
$ pc -s first_program.p -o first_program.comp
```

That is, the Pascal compiler `pc` is invoked with the `-s` (standard) option to compile the program called `first_program.p`, and to produce a compiled program (i.e., executable machine language program) called `first_program.comp` (you may chose a different name; if you are not a good typist, you may want to choose a simple short name such as, say, `ex1` for the compiled program). The option `-o` in the above line stands for *object file* and indicates that the object file produced by the compiler, i.e., the executable program, should be given the name following this option. The command can also be typed without the `-o` option:

```
$ pc -s first_program.p
```

In this case the computer chooses the default filename `a.out` as the name of the compiled program. Your computer may not have a Pascal compiler installed, or

else it may have have several Pascal compilers having different names; a well-known one is `gpc`, which stands for *GNU Pascal Compiler*, written by the Free Software Foundation. You may read about the Free Software Foundation at

```
http://www.gnu.ai.mit.edu/
```

In the above line we used the `-s` option to indicate that we want the compiler only to accept standard Pascal; this will be helpful when you want to *port* (i.e., move to an environment different from the Sun computers at the Atrium) your programs, since no extensions unknown to some Pascal compilers will be accepted. Actually, the real world is more complicated than described here, and there are different levels of compliance with standard Pascal. To find out more, you may want to read the on-line manual pages of the Pascal compiler by typing

```
$ man pc
```

However, Unix on-line manual pages are notoriously hard to read, so we do not recommend you to do this unless you are really dedicated.

To run the compiled program, you simply have to type its name on the command line, prefixed by the symbol `./`:

```
$ ./first_program.comp
```

If everything was done correctly, you should get the output

```
Hello world!
```

On some systems the command

```
$ first_program.comp
```

may work equally well. This is an issue of system configuration and security that will be discussed in Section 23 on p. 98 in the Subsection *The present working directory in the search path*.

Next we will outline how the above program works. First note that everything in the above program is written in lower case, except the letter `H` in the phrase `Hello world!`, which is quoted (i.e., put between single quotation marks). Certain words in the program are *key words*, i.e., reserved words that have specified meanings. These are `program`, `begin`, `writeln`, and `end` ("key word" is somewhat colloquial as far as the formal description of the Pascal programming language is concerned; the proper name for key words is *word symbol*; key words are also called *reserved words*, indicating that they they cannot be used outside their strictly defined roles). The word `program` introduces the *program heading* (the first line), where the program is given the name `hello`, and it is indicated by the word `output` that the program will write to the *output* file (which at present is the window in which you are working). The program heading must be followed by a semicolon; hence the semicolon at the end of the first line. The word `hello` on this line is called an *identifier*. Identifiers are names given to various things to be discussed below; they must start with a letter and they may contain letters and digits. Upper and lower case letters in an identifier are considered equivalent (but some compilers may insist that you only use lower case letters only – this is not the case for the compiler at the Atrium).

The *body* of the program is in lines 2–4. Between the key words `begin` and `end`, the action to be taken by the program is described; there is only a single *statement* here, given on line 3. The word `writeln` indicates that the program should write to the output file (the screen); and in parentheses between single quotes, the text to be written is given; note that both quote sings are right-quote signs. The period on line 4 indicates the end of the whole program.

The various parts of the program are kept apart by the use of *separators*. These

are spaces and *line feed* (obtained when pressing the return key) characters. For example, the space between `program` and `hello` is a separator. Without it, the key word `program` would not be recognized; the word `programhello` would be taken to be an identifier. One needs at least one separator to keep different parts of the program apart, but the use of more than one separator is also allowed. For example, there are three separators after the word `begin` in line 2 (one end-of-line character, and two space characters). Compilers usually accept the *tab* character. but the Pascal standard specified by the ISO (International Standards Organisation) does not mention the tab character as a separator.

The above program could also have been written as

```
program hello(output);begin writeln('Hello world!')end.
```

(we omitted the line number here, which in any case is never a part of the file containing the program). Indeed, the required separators are present in this version of the program, and the compiler will not complain. However, the first version is easier to read for people, and so it is preferable. In particular, indentation is used to make it easier for people to recognize various parts of the program.

Correcting errors in Pascal programs. Errors in Pascal programs come in two kinds. *Syntax errors* – these are errors in the form of the program, i.e., errors in following the grammar of the language (in which case, usually, the computer will not be able to "understand" what is intended by the program), and *semantic error*, or *logic errors* – there are errors in the meaning of the program, i.e., the computer is not is able to "understand" the action described in the program, but the action described is not what is intended.

The compiler usually finds and lists the syntax errors, though the error messages given by the compiler are often hard to understand. Logic errors are hard to find; they can often be found only after extensive testing, and subtle logic errors are often found only after long use of the program, if at all. Since a program cannot even be compiled if it has syntax errors, the first task is to correct the syntax errors, and only then is it possible to deal with logic errors.

A good way of correcting syntax errors is to work with two windows, one for compilation, another for editing the *source program*; this is the file `first_program.p` in the example above, that is, the program as written (the compiled program, `first_program.comp` in the above example, is called the *object program*). In the first window, the program is compiled repeatedly, in the second window, some corrections suggested by the compiler error messages should be made. When doing this, the file containing the program should be opened only once by the editor, say, `vi`, in the second window, and after each correction the changes should be written out to the file (using the `vi` command `:w` without quitting the editor), and then, moving to the first window, the program should be compiled again. Between each compilation, it is sufficient to make the corrections suggested only by one or two error messages (it is usually best to act on the first error message, since a single error may generate a number of error messages). When there are no more error messages, one may quit the editor (by using the `vi` command `:q` for this, rather than `:wq`, since at this point, presumably, there are no new changes to be written out to the file).

9. Simple Pascal Programs

We are going to explore the Pascal programming language by considering a somewhat more complicated program:

```
1  program example2(output);
2  var num : integer;
3  begin
4    num := 15;
5    writeln('The number is ', num:1)
6  end.
```

This program is still quite useless from a practical point of view. All it does (after compiling and running it) is to write the message

```
The number is 15
```

on the screen. A new feature in this program is the *declaration* contained on line 2: this line says that the identifier *num* denotes a *variable* of the type *integer*. Integers are positive or negative whole numbers within a certain range; the range depends on the particular Pascal implementation. The Pascal compiler at the Atrium allows integers n in the range

$$-2^{31} \leq n \leq 2^{31} - 1 = 2,147,483,647.$$

The reason that a power of two is chosen here is that the computer usually does all its calculations in binary (base 2) arithmetic; but in order to communicate with people on their terms, it converts the results of all calculations to the decimal (base 10) system (unless otherwise instructed). The observation that computers should use binary arithmetic is not a trivial one. In the 1940s the first electronic computer called ENIAC (Electronic Numerical Integrator And Calculator) containing 18000 vacuum tubes used decimal arithmetic. Binary arithmetic was used in subsequent computers, as a result of a proposal by the mathematician *John von Neumann*, who noted that conversion for binary and decimal arithmetic is an easy task for the computer to perform. He also worked out the architecture of the modern *stored program computer*, the architecture adopted by later computers (this architecture is now called *von Neumann architecture*).

The role of this declaration is to reserve a location in the computers memory for the variable *num*. The declaration must be followed by a semicolon. As mentioned before, line 1 constitutes the header of the program; the rest constitutes its body. The part of the program on lines 4–5 is the action part of the program, enclosed between the key words *begin* and *end*. The parts of the program that cause the computer to take action are called statements. Lines 4 and 5 each contain a statement. The statement on line 4 is called an *assignment*. What it does is to give, i.e., *assign*, the value 15 to the variable called *num*. The assignment symbol `:=`, i.e. a colon followed by an equation sign can be read as "is to be replaced with"; i.e., the current value of *num* is to be replaced with the value of the expression on the right-hand side. The symbol `:=` must be distinguished from the equation sign `=`, which is also used in Pascal programs (see later).

We already met the *writeln* statement. Here the statement writes two items to the output file (the screen). The first item is quoted, and is written exactly as indicated

between the single quotation marks (both of them right quotes). The second item, separated from the first item by a comma, to be printed out is the value of the variable *num*. In the string `num:1` the 1 after the colon indicates that the minimum *width* of the space used for printing the variable should be 1 characters (more space is used if it is needed to print this value). The number indicating the width can be omitted. That is, instead of line 5 one can write the line

```
writeln('The number is ', num)
```

but in this case a *default* number, i.e., a number decided upon by the computer (or, rather, by the compiler), will be used as width; and you may not like this default. If, for example, the default is 5, the printout would look like this:

```
The number is    15
```

Here three spaces are added, and these three spaces and the two characters in the number 15 make up the width of 5.

Note the role of the space before the quotation mark in line 5; this space ensures that the word *is* and the number 15 is separated by a space (that is, the space is printed as part of the quoted text). If one omits this space, i.e., if one replaces line 5 with the following

```
writeln('The number is', num:1)
```

then the printout will look like

```
The number is15
```

If one wants a period at the end of the line printed out, i.e., if one wants the printout

```
The number is 15.
```

then a third item needs to be added in the *writeln* statement; that is, line 5 needs to be modified to read

```
writeln('The number is ', num:1,'.')
```

Every statement must be followed by a semicolon, except the statement directly preceding the key word *end*; hence there is a semicolon at the end of line 4, but there is no semicolon at the end of line 5. In fact, putting a semicolon at the end of line 5 would cause no harm. Placing a semicolon at the end of line 5 will add an *empty statement* after the semicolon and before the separator preceding the key word *end*. To write the empty statement one uses no characters (that is, the empty statement is really nothing when written down), and it has no effect – its presence is indicated only by the presence of semicolon. This remark does not imply that one can add semicolons at any place at will; a lot of trouble is caused by misplaced or missing semicolons in Pascal programs.

10. More on Pascal

The following program is now easy to understand:

```
1  program example3(output);
2  var num1, num2, sum : integer;
3  begin
4    num1 := 15;
5    num2 := 12;
6    sum := num1 + num2;
```

```
7    write('The sum of ', num1:1, ' and ');
8    writeln(num2:1, ' is ', sum:1)
9  end.
```

On line 2, three variables called *num1*, *num2*, and *sum* are declared to be of type integer. The spaces in this line, except for the first one, are added only for easier readability; they can be omitted, and the line can also be written as

```
var num1,num2,sum:integer;
```

(we omitted the line number, which is not part of the file containing the program). The first space is, of course, required in order to separate the key word *var* from the identifier *num1*.

In line 6, the assignment statement includes a simple arithmetic expression. Arithmetic expressions can be written in much the same way as they are in mathematics. The arithmetic operators allowed are `+` for addition `-` for subtraction and sign inversion (i.e., for changing the sign of a number, as in the expression `-5`, changing the number 5 to -5), `*` for multiplication (that is 5 times 4 is written as `5*4` rather than the usual mathematical notation $5 \cdot 4$ – the reason for this is the limitations imposed by the keyboard), `/` for division; however, even if performed on numbers of the type integer, the result will be a number of the type *real* (the type real has not yet been discussed; it is much like a decimal fraction written in scientific notation – e.g., $2.3798 \cdot 10^2$, but we will not give the precise definition here; division by zero will result in an error), `div` for integer division (that is, the quotient resulting from the division of two integers is given, and the remainder is discarded; the result is an integer; for a precise definition, one needs to carefully discuss the case when one or both of the operands are negative, but we will not be interested in that case; division by zero will result in an error) `mod` gives the remainder of the division of two integers (again, special explanations must be given if one or both of the operands are negative; division by zero is again not allowed). The rules for evaluating arithmetic expressions are much the same as in mathematics. For example, the expression

```
4*(2+3)+3*2
```

is evaluated to be 26, since the sum $2 + 3$ in parenthesis has to be evaluated first, giving 5, then $4 \cdot 5$ makes 20; next, the multiplication $3 \cdot 2$ must be evaluated, giving 6, before the addition (since multiplication has *priority* over addition); finally, the two partial results 20 and 6 must be added, resulting in 26.

In line 7, the *write* statement is new. It works exactly like the writeln statement, except that it does not print an end-of-line character; i.e., it does not start a new line after finishing printing. The program will print the following line:

```
The sum of 15 and 12 is 27
```

If we change line 8 to

```
write(num2:1, ' is ', sum:1)
```

then the end-of-line character is not printed. The effect of this is that after finishing the program, the prompt for entering a new command appears on the same line as the printout of the program. Assuming that the prompt is `$`, the printout will now look like this:

```
The sum of 15 and 12 is 27$
```

(as we explained, `$` is not part of the printout; it is the prompt, which would normally be printed on the next line; in actual fact, the prompt you get will usually be different).

Lines 7 and 8 can be replaced with a single *writeln* statement:

```
writeln('The sum of ', num1:1, ' and ',
     num2:1, ' is ', sum:1)
```

That is, we listed all items to be printed in a single *writeln* statement; since putting all items in a single line would have made the line too long, we broke up the line after the third item.

The next program is a slightly modified version of the one above:

```
 1  program example4(input,output);
 2  var num1, num2, sum : integer;
 3  begin
 4    write('Enter the first number: ');
 5    readln(num1);
 6    write('Enter the second number: ');
 7    readln(num2);
 8    sum := num1 + num2;
 9    write('The sum of ', num1:1, ' and ');
10    writeln(num2:1, ' is ', sum:1)
11  end.
```

The word input in the program heading (line one) indicates that the program will read from the input file (which at present is the window in which you are working). The meaning of the *readln* statement in line 5 is that when the execution of the program arrives at this statement, it will wait for input to be typed at the keyboard (the window in which the program is running must be the active window, i.e., the mouse pointer must be located in this window). It will accept input until an end-of-line character is typed, and the characters typed before the end-of-line character will be used to form the content of the variable *num1* (indicated in parenthesis in the *readln* statement in question). The sequence of characters typed must form a (positive or negative) integer, since the variable *num1* was declared to be of the type integer.

Without the *write* statement on line 4, the person running this program would not know why the program waits upon arriving at line 5 (in fact, she would not even know that the program arrived at line 5; she would only notice that the program is not doing anything, as if it had stopped). As the *write* statement prints the message

```
Enter␣the␣first␣number:␣
```

on the screen (here we indicated the space characters printed on the screen by ␣ to call attention to the the fact that a space character is printed at the end of the line; normally, we will not indicate space characters this way, since their presence is clear from the space left between words). After this, the *readln* statement on line 5 will wait for a number to be typed. If the number 153 is typed and return key is pressed, the variable *num1* will be assigned the value 153, and the screen will look as like this:

```
Enter the first number: 153
Enter the second number: □
```

The second line was printed by the *write* statement on line 6; the position of the cursor (indicated by □ here – we normally do not show the cursor in these notes) indicates that a space was printed at the end of the line. If the number 22 is entered now, then the program finishes, and its printout plus the characters typed by the used will appear as follows:

```
Enter the first number: 153
```

```
Enter the second number: 22
The sum of 153 and 22 is 175
```

11. Loops

Next we will consider the following program.

```
 1  program example5(input,output);
 2  var i, sum, n : integer;
 3  begin
 4    write('Type n (n must be a positive integer): ');
 5    readln(n);
 6    i := 1;
 7    sum := 0;
 8    while i <= n do
 9    begin
10      sum := sum + i;
11      i := i + 1
12    end;
13    write('The sum of integers from 1 through ');
14    writeln(n:1,' equals ', sum:1)
15  end.
```

This program contains several new features. The first one to mention is the *compound statement* in lines 9–12. A compound statement is a sequence of statements enclosed between the key words *begin* and *end*; as we mentioned before, there must not be a semicolon before the key word *end*. The purpose using a compound statement here is to make a sequence of statements appear as a single statement. Note that the statement part of the body of the program, i.e., the content of lines 3–15 can also be considered as a single compound statement. Normally, a compound statement is made up from more than one statements. However, in our first program (the one printing `Hello world!`), the body of the program was a compound statement made up of a single statement.

The key new feature of this program is, however, the *while* ... *do* statement in line 8 (in fact, as we will see in a moment, this statement occupies lines 8–12). The *syntax* (i.e., the form) of the *while* ... *do* statement is as follows:

`while` *condition* `do` *statement*

In the present case, the condition is `i <= n`, and the statement is the compound statement in lines 9-11; here the meaning of `<=` is $\leq$, that is *less than or equal to*; the keyboard does not allow one to write $\leq$.

Its *semantics* (i.e., its meaning) can be explained as follows. When the execution of the program arrives at the key word *while*, the condition is checked. If the condition is satisfied, the statement is executed and the sequence of the execution of the program continues at the key word *while*. If the condition is not satisfied, then the statement is not executed, and the execution program continues with the next statement in the sequence (in the present case, this is the statement in line 13).

As we will see, under normal conditions the *while* statement in lines 8–12 is executed repeatedly a number of times; statements that are supposed to be executed repeatedly a number of times under normal conditions are called *loop* statements or *loops*. The *while* statement is only one example of loop statements; there are a number of other loop statements. Loop statements are also called *repetitive statements*.

To understand what the above program does, one must *trace* (i.e. follow through) its execution. Rather than doing this by pencil and paper, the computer can be used to do this. All one needs to do to accomplish this is to insert writeln statements at key points in the program:

```
 1   program example5(input,output);
 2   var i, sum, n :  integer;
 3   begin
 4     write('Type n (n must be a positive integer): ');
 5     readln(n);
 6     i := 1;
 7     sum := 0;
 7a    writeln('line  7: ','sum =',sum:3,'  i =',i:2);
 8     while i <= n do
 9     begin
 9a      writeln('line  9: ','sum =',sum:3,'  i =',i:2);
10       sum := sum + i;
10a      writeln('line 10: ','sum =',sum:3,'  i =',i:2);
11       i := i + 1;
11a      writeln('line 11: ','sum =',sum:3,'  i =',i:2)
12    end;
12a   writeln('line 12: ','sum =',sum:3,'  i =',i:2);
13    write('The sum of integers from 1 through ');
14    writeln(n:1,' equals ', sum:1)
15  end.
```

We inserted lines 7a, 9a, 10a, 11a, and 12a. The purpose of numbering the lines this way is to not change the numbering of the other lines – the line numbers are again not part of the file containing the program. Note, however, that you will not be able to get the editor to display the line numbers this way; the best you can do is to write the content of line 7a as a continuation of line 7 (and similarly for lines 9a, 10a, ...) to keep the same line numbers; this will not change the meaning of the Pascal program. Using line numbers 7a, 9a, ..., is convenient for us in talking about the inserted statements. Observe that it is not permissible to insert a *writeln* statement directly after line 8, since line 8 must be followed by a single statement, which is the compound statement in lines 9–12. If we run this program, then it will wait for us at line 5 so that we can enter a number. If we enter the number 6 (and then press the return key), the printout will be as follows:

```
21  Type n (n must be a positive integer): 6
22  line  7: sum =  0  i = 1
23  line  9: sum =  0  i = 1
24  line 10: sum =  1  i = 1
25  line 11: sum =  1  i = 2
26  line  9: sum =  1  i = 2
```

```
27   line 10: sum =  3  i = 2
28   line 11: sum =  3  i = 3
29   line  9: sum =  3  i = 3
30   line 10: sum =  6  i = 3
31   line 11: sum =  6  i = 4
32   line  9: sum =  6  i = 4
33   line 10: sum = 10  i = 4
34   line 11: sum = 10  i = 5
35   line  9: sum = 10  i = 5
36   line 10: sum = 15  i = 5
37   line 11: sum = 15  i = 6
38   line  9: sum = 15  i = 6
39   line 10: sum = 21  i = 6
40   line 11: sum = 21  i = 7
41   line 12: sum = 21  i = 7
42   The sum of integers from 1 through 6 equals 21
```

Again, the numbers on the left are line numbers, and are not part of the printout. We numbered the lines of the printout beginning with line 21, so that we can mix references to lines in the program and the printout, without having at every time to explicitly say whether we refer to the program or to the printout.

First observe that the widths indicated on the inserted *writeln* statements are needed to ensure the proper alignment of the printout. The width specification `sum:3` ensures that the width of three characters are reserved for printing the value of *sum*; the reserved space of three characters are used to print the the number *right justified* (i.e., as far to the right as allowed by the reserved space) and the spaces not used up by digits (or by a possible minus sign) are filled with spaces. Thus the printed values of *sum* in lines 33 and 33 (of the printout) are properly aligned.

Note that the inserted *writeln* statements do not change the action of the program at all (except for the extra lines of printout obtained). Thus the printout gives a fairly complete trace of the program. First note that line 21 (of the printout) would have been printed by the original program, and the number 6 at the end of the line was in fact typed by the user and not printed by the program. As line 22 tells us, after line 7 (of the program), the value of i is 1, so the condition in line 8 is satisfied. Hence the loop in lines 8–12 is entered. As line 25 (of the printout) shows, the first time line 11 is reached the value of i is 2; the condition in line 8 is again satisfied, so the execution continues in line 9. Lines 28, 31, 34, 37 show the successive values of i in line 11; since this value is less than or equal to 6, the value of n, the condition in line 9 is satisfied and the loop is reentered. Finally, line 40 shows that the next time the value of i on line 11 is 7; the condition in line 9 is tested again and it is found that the condition is not satisfied (since $7 \leq 6$ us false). The loop is not reentered, and the execution of the program continues in line 13. Line 41 indicates the values of *sum* and i right after the loop is exited. (Of course, only the original version of the program continues in line 13; the modified, "tracing" version of the program continues in line 12a. But the tracing statements inserted in lines 7a, 9a, 10a, 11a, and 12a do not affect the working of the program – except for the tracing printout obtained.) Finally, line 42 shows the printout of the original version of the program produced in lines 13 and 14.

As seen form the above trace, the variable i is initially given the value 1 in line 6, and its value is *incremented* (i.e., increased) by 1 in line 11 on each execution of the loop; that is, assuming $n = 6$, i runs through the values 1, 2, 3, 4, 5, 6, 7. The variable *sum* is initially given the value 0 in line 7, and its value is increased by the current value of i in line 10 on each execution of the loop. That is *sum* runs through the values

$$\begin{aligned} 0 & \\ 1 &= 0 + 1 \\ 3 &= 1 + 2 \\ 6 &= 3 + 3 \\ 10 &= 6 + 4 \\ 15 &= 10 + 5 \\ 21 &= 15 + 6 \end{aligned}$$

The last value of i is 7, but after i is given this value in line 11, the loop condition on line 8 will fail, and so line 10 will not be executed; thus, the last value of i that is added to *sum* is 6. Thus in the end we have

$$sum = 0 + 1 + 2 + 3 + 5 + 6 = 1 + 2 + 3 + 4 + 5 + 6.$$

In general, depending on the value of n, we have

$$sum = 1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i.$$

In words, the final value of *sum* is the sum of positive integers 1 through n. This explains the text printed out in lines 13–14 of the program.

Problems

1. Trace through the execution of the following program. That is, indicate how the variables i, *sum*, and n change through the execution of the program, and what is printed by the program.

```
 1  program example(output);
 2  var i, sum, n : integer;
 3  begin
 4    n := 7;
 5    i := 3;
 6    sum := 0;
 7    while i <= n do
 8    begin
 9     sum := sum + i;
10     i := i + 2
11    end;
12    writeln('The total equals ', sum:1)
13  end.
```

2. Trace through the execution of the following program.

```
 1  program example(output);
 2  var i, sum, n : integer;
 3  begin
 4    n := 7;
 5    i := 3;
 6    sum := 3;
 7    while i < n do
 8    begin
 9     i := i + 2;
10     sum := sum + i
11    end;
12    writeln('The total equals ', sum:1)
13  end.
```

Compare the trace of this program to the trace of the program given in Problem 1.

3. Write a Pascal program that adds up the all the even integers 2, 4, 6, ..., up to a positive integer n entered at the keyboard, and then prints out the result (if the number n entered is even, n itself should also be added, and if it is odd then it should not be added, since we are adding only even integers).

4. Write a Pascal program that adds up the all the odd integers 1, 3, 5, ..., up to a positive integer n entered at the keyboard, and then prints out the result (if the number n entered is odd, n itself should also be added, and if it is even then it should not be added, since we are adding only odd integers).

12. Conditional Statements

Try to run the first program of the preceding section with the value $n = 65536$ typed in when the program asks you to type a value for n (do not try this with the second version, with the *writeln* statements for tracing inserted; the printout will be unmanageable; the tracing version should only be run with small values for n, perhaps not larger than 20). The printout, at least with the version of Pascal installed at the Atrium will be as follows:

```
The sum of integers from 1 through 65536 equals -2147450880
```

This result certainly cannot be correct, since the sum of positive numbers must be positive. In fact, the correct value of the sum is

$$\sum_{i=1}^{65,536} i = 2,147,516,416$$

Now the reason for the the incorrect answer is clear. As we saw earlier, the largest integer that can be represented by the Pascal compiler at the Atrium is

$$2^{31} - 1 = 2,147,483,647,$$

and the value of the above sum is somewhat larger than this integer. On the other hand,

$$\sum_{i=1}^{65,535} i = 2,147,450,880$$

is somewhat smaller that the largest integer representable by the Pascal compiler at the Atrium; and, indeed, if one runs the program with $n = 65535$, the correct printout is obtained:

```
The sum of integers from 1 through 65535 equals 2147450880
```

Thus, the above program should be only run with values $n \leq 65535$. It is desirable to write the program in such a way that it will not run if a larger value of n is entered. Otherwise, the wrong result may inadvertently be accepted. In fact, if one runs the program with $n = 99000$, the printout

```
The sum of integers from 1 through 99000 equals 605582204
```

which is wrong, but it may not be recognized as wrong upon a casual glance. The correct answer is in fact

$$\sum_{i=1}^{99,000} i = 4,900,549,500$$

The following modification of the program prevents such errors from happening:

```
 1  program example5(input,output);
 2  var i, sum, n : integer;
 3  begin
 4    write('Type n (n must be a positive integer): ');
 5    readln(n);
 6    if n > 65535 then
 7      writeln('n is too large!')
 8    else
 9      begin
10        i := 1;
11        sum := 0;
12        while i <= n do
13        begin
14          sum := sum + i;
15          i := i + 1
16        end;
17        write('The sum of integers from 1 through ');
18        writeln(n:1,' equals ', sum:1,'.')
19      end
20  end.
```

The new feature this program is the *if* statement starting in line 6. In fact, this statement takes up all of line 6–16. The syntax of the *if* statement can be described as

`if` *condition* `then` *statement*$_1$ `else` *statement*$_2$

The semantics of this statement is straightforward: if *condition* is satisfied then *statement*$_1$ is executed, otherwise (i.e., if if *condition* is not satisfied) *statement*$_2$ is executed.

The *else* part can in fact be omitted. The syntax of this version if

`if` *condition* `then` *statement*

According to the semantics of this version, if *condition* is satisfied then *statement* is executed, nothing is done otherwise. The *if* statement is a type of *conditional* statement; another type of conditional statement is the *case* statement, discussed later, in Section 20 on p. 74.

The above program contains the first version of the if statement in lines 6–16. Line 6 contains the condition, line 7 contains statement$_1$, and lines 9–19 constitute statement$_2$; as can be seen, in the present case statement$_2$ is a compound statement. Also note the close similarity between this program and the first program of the preceding section. In fact, lines 4–5 and lines 9–19 were copied from that program. A minor modification was made in line 18: a fourth item, `'.'`, was added in the *writeln* statement. The effect of this is to print a period at the end of the line.

If this program is run with $n = 50000$, then the condition in line 6 fails, and so the statement in lines 9–19 will be executed; the printout will be as follows:

```
Type n (n must be a positive integer): 5000
The sum of integers from 1 through 50000 equals 1250025000.
```

(here the number 5000 is in fact typed by the user, and not printed by the computer). If the program is run with $n = 99000$, then the condition in line 6 is satisfied, and so the statement in line 7 is executed. The printout will be as follows:

```
Type n (n must be a positive integer): 99000
n is too large!
```

Maintenance and portability issues. Since writing computer programs is very time consuming, it is desirable to make programs easily *portable*, i.e., transferable from one computer to another. Even when you do not intend to port your program to a different computer, you may still want to *maintain* it. That is, the environment on the computer you are using may change. A newer version of the operating system or an improved version of the Pascal compiler may be installed (sometime, a newer version, thought to be an improvement, turns out to be worse than the older version). In any case, a program that ran perfectly in the old environment, may have to be changed (that is, *maintained*) to suit the new environment. Here is a version of the above program that takes these considerations into account:

```
 1  program example5(input,output);
 2  {This program calculates the sum of integers
 3  between 1 and n. n is entered at the keyboard,
 4  and it must be positive. If n is larger than
 5  the constant largestn, the program notifies
 6  the user that the calculation cannot be done
 7  by the program. The value of this constant
 8  is implementation-dependent, and it may have
 9  to be changed when porting the program.}
10  const largestn = 65535;
```

```
11  var i, sum, n : integer;
12  begin
13    write('Type n (n must be a positive integer): ');
14    readln(n);
15    if n > largestn then
16      writeln('n is too large!')
17    else
18      begin
19        i := 1;
20        sum := 0;
21        while i <= n do
22        begin
23          sum := sum + i;
24          i := i + 1
25        end;
26        write('The sum of integers from 1 through ');
27        writeln(n:1,' equals ', sum:1,'.')
28      end
29  end.
```

This program is essentially identical to the earlier version, except for two new features. First, we included a *comment* in lines 2–9; a comment is any text placed between the braces { and }. A comment does not affect the execution of a program in any way except that a comment acts as a separator (the other separators, namely spaces and end-of-line characters, were discussed above); thus, one cannot place a comment inside the name of an identifier, the same way one cannot place a space there; at any place where spaces are allowed, comments are also allowed (and, in fact, one can use comments instead of spaces as separators). The purpose of comments is to document the program, that is, to explain its important features (what it does, how it does it, how it may be changed, who wrote it, who changed it and when, etc.). Often, the person who wrote a program is no longer accessible, or does not remember how the program works, and someone else has to maintain or port the program. Comments are very helpful in this respect.

The second feature is the declaration of a *constant* in line 10. In the earlier version, the value of this constant was explicitly written in the condition that is now in line 15. In porting the program, this constant may need to be changed; it is much easier to find values that may need to be changed by looking through the constants in the declaration part of the program than to scan the whole program for values that need to be changed.

The declaration of the constant starts with the key word `const`; the value of the constant must be given in the declaration, using the equation sign `=` rather than the assignment symbol `:=`. The value of a constant cannot be changed in the program (i.e., a constant must not appear on the left-hand side of an assignment statement).

CHAPTER IV

APPLICATIONS OF PASCAL

A mathematical discussion of the Euclidean algorithm and its extension to represent the greatest common divisor of two numbers as a linear combination is given. Pascal implementations of the algorithms presented are described, and a proof of the Fundamental Theorem of Arithmetics (asserting the unique prime factorization of integers) is included. Then Pascal programs are presented to print truth tables and to determine the date of the week in the Gregorian calendar.

13. The Euclidean Algorithm

The common divisors of two numbers are the numbers that are divisors of both of them. For example, the divisors of 12 are 1, 2, 3, 4, 6, 12. The divisors of 18 are 1, 2, 3, 6, 9, 18. Thus, the common divisors of 12 and 18 are 1, 2, 3, 6. The greatest among these is, perhaps unsurprisingly, called the *greatest common divisor* of 12 and 18. The usual mathematical notation for the greatest common divisor of two integers a and b are denoted by (a, b). Hence,

$$(12, 18) = 6.$$

The greatest common divisor is important for many reasons. For example, it can be used to calculate the *least common multiple* of two numbers, i.e., the smallest positive integer that is a multiple of these numbers. The least common multiple of the numbers a and b can be calculated as

$$\frac{ab}{(a, b)}.$$

For example, the least common multiple of 12 and 18 is

$$\frac{12 \cdot 18}{(12, 18)} = \frac{12 \cdot 18}{6}.$$

Note that, in order to calculate the right-hand side here it would be counterproductive to multiply 12 and 18 together. It is much easier to do the calculation as follows:

$$\frac{12 \cdot 18}{6} = \frac{12}{6 \cdot 18} = 2 \cdot 18 = 36.$$

That is, the least common multiple of 12 and 18 is 36. It is important to know the least common multiple when adding two fractions. For example, noting that $12 \cdot 3 = 36$ and $18 \cdot 2 = 36$, we have

$$\frac{5}{12} + \frac{7}{18} = \frac{5 \cdot 3}{12 \cdot 3} + \frac{7 \cdot 2}{18 \cdot 2} = \frac{15}{36} + \frac{14}{36} = \frac{15 + 14}{36} = \frac{29}{36}.$$

Note that this way of calculating the least common multiple works only for two numbers. We will not discuss the case of more than two numbers; our main interest here is the greatest common divisor. First note that we usually consider only positive divisors. Thus the common divisors of -12 and -18 are 1, 2, 3, 6, i.e., the same as those of 12 and 18. So their greatest common divisor is also 6; i.e., $(-12, -18) = 6$. Similarly, $(-12, 18) = 6$.

Every positive integer is a divisor of 0. Thus the common divisors of 0 and 18 are just the divisors of 18. The greatest common divisor of 0 and 18 is thus the greatest divisor of 18, i.e., 18 itself: $(0, 18) = 18$. Similarly, $(12, 0) = 12$. Note that $(0, 0)$ is undefined, since any positive integer is a divisor of 0 and 0. On the other hand, while it may be uninteresting to consider the greatest common divisor $(12, 12)$, the definition is meaningful in this case, and we have $(12, 12) = 12$.

A way of determining the greatest common divisor of two integers was described by Euclid about 2400 years ago. This method, called the *Euclidean algorithm*, even though it was probably not invented by Euclid himself, or its minor modifications, is still the best method to determine the greatest common divisor of two numbers. While the greatest common divisor of 12 and 18 can be easily guessed, the situation is different when larger numbers are involved.

We illustrate the Euclidean algorithm by determining the greatest common divisor of 546 and 422.

Step 1. Divide 422 into 546; the remainder is 124. We replace 546 by this remainder. That is, in place of the original numbers 546 and 422, we will work with 422 and 124.

Step 2. Divide 124 into 422; the remainder is 50. We replace 422 with this remainder. That is, in place of 422 and 124, we will work with 124 and 50.

Step 3. Divide 50 into 124; the remainder is 24. We replace 124 with this remainder. That is, in place of 124 and 50, we will work with 50 and 24.

Step 4. Divide 24 into 50; the remainder is 2. We replace 50 with this remainder. That is, in place of 50 and 24, we will work with 2 and 24.

Step 5. Divide 2 into 24; the remainder is 0. The remainder being 0, we stop. The greatest common divisor is the last nonzero remainder, that is, 2.

Why does this procedure work? Before we answer this question, note that the procedure will always terminate. In fact, in each step we work with smaller numbers, so in the end we must reach 0 as a remainder, if no sooner, then at the point when the last divisor is 1.

To explain why the method works it is sufficient to note that each of the pairs we work with have the same greatest common divisor:

$$(546, 422) = (422, 124) = (124, 50) = (50, 24) = (24, 2) = (2, 0).$$

We never actually worked with the last pair, 2 and 0, since there is no reason to go on, because it is clear that $(2, 0) = 2$. It is easy to justify any of these equalities; we will illustrate this with the discussion of the equality

$$(422, 124) = (124, 50).$$

To see this, note that 50 was obtained as the remainder when dividing 124 into 422; in fact,

$$422 = 3 \cdot 124 + 50.$$

This equation shows that any common divisor of 124 and 50 is also a divisor of 422. if one writes this equation as

$$50 = 422 - 3 \cdot 124,$$

then one can see that any common divisor 422 and 124 is also a divisor of 50. Therefore, the common divisors of the numbers 422 and 124 are the same as the common divisors of the numbers 124 and 50; i.e., the greatest common divisor of 422 and 124 is the same as the greatest common divisor of 124 and 50.

14. Implementing the Euclidean Algorithm in Pascal

To describe more how the greatest common divisor of 422 and 546 was determined, in each step write a for the smaller number and b for the larger number. That is,

$$\begin{aligned}
&\text{in Step 1,} \quad a = 422, \quad b = 546\\
&\text{in Step 2,} \quad a = 124, \quad b = 422\\
&\text{in Step 3,} \quad a = \ 50, \quad b = 124\\
&\text{in Step 4,} \quad a = \ 24, \quad b = \ 50\\
&\text{in Step 5,} \quad a = \ \ 2, \quad b = \ 24\\
&\text{in Step 6,} \quad a = \ \ 0, \quad b = \ \ 2
\end{aligned}$$

Earlier, we only had five steps; here we included a sixth step to indicate that in the last step we had to determine the greatest common divisor $(2, 0)$. It is easy to describe how the new values of a and b are obtained in each step. Consider, for example, Step 2. The current values of a and b in this step are given as

$$a = 124 \quad \text{and} \quad b = 422.$$

Denote by a_{new} and b_{new} the new values of a and b, i.e., the values of a and b that will be used in the next step (i.e., in Step 3). We have

$$a_{\text{new}} = 50 \quad \text{and} \quad b_{\text{new}} = 124.$$

It is easy to write the formulas that calculate the new values. Write $x \bmod y$ for the remainder when y is divided into x; in fact, this is the notation used in Pascal. For example, $26 \bmod 3 = 2$. We have

$$a_{\text{new}} = b \bmod a \quad \text{and} \quad b_{\text{new}} = b \bmod a.$$

In Pascal, one would be tempted to write the following program segment to implement this:

```
a := b mod a;
b := a
```

This, however, will not work. If one starts with the values $a = 124$ and $b = 422$, then the effect of the first line will be $a = 422 \bmod 124$, that is $a = 50$. Then the result of the second line will be $b = 50$, i.e., b is assigned the new value of a, whereas it needs to be assigned the old value of a. Therefore, before a is changed, its value needs to be preserved; for this we introduce a new variable called *temp*. The correct program segment will therefore be

```
temp := a;
a := b mod a;
b := temp
```

In the Euclidean algorithm this step has to be repeated until a zero remainder is obtained. The Pascal program accomplishing this can be easily written. At the beginning, lines were added to read in the data, and at the end, a line was added to print out the result. The whole program is as follows:

```
 1  program euclid(input, output);
 2  var a, b, temp : integer;
 3  begin
 4    write('Type a (a must be a positive integer): ');
 5    readln(a);
 6    write('Type b (b must be a positive integer): ');
 7    readln(b);
 8    while a <> 0 do
 9    begin
10      temp := a;
11      a := b mod a;
12      b := temp
13    end;
14    write('The greatest common divisor of the above');
15    writeln(' numbers equals ', b:1)
16  end.
```

The numbers at the beginning of each line are not part of the program; they are added only so that we can easily refer to each line when describing how this program works. To give some explanation, the numbers a and b are read in in lines 5 and 7. The statements in lines 14 and 15 could have been written as a single *writeln* statement, but then line 14 would have been too long; it is easier to break up the statement.

Even though in the above description we assumed that the number b is always larger than a during the run of this program, it is important to note that the program runs correctly even if one enters a larger value for a than for b. If, for example, one enters $a = 546$ and $b = 422$, it is easy to check that after the first execution of the statements in lines 10–12 one will have $a = 422$ and $b = 546$, that is the values of a and b will be interchanged. This is because the remainder of the division $422 \div 546$ is 422 (and its quotient is 0). One can change the above program in a way that this interchange is done explicitly:

```
1  program euclid(input, output);
```

```
 2  var a, b, temp : integer;
 3  begin
 4    write('Type a (a must be a positive integer): ');
 5    readln(a);
 6    write('Type b (b must be a positive integer): ');
 7    readln(b);
 8    if a > b then
 9    begin
10      temp := a;
11      a := b;
12      b := temp
13    end;
14    while a <> 0 do
15    begin
16      temp := a;
17      a := b mod a;
18      b := temp
19    end;
20    write('The greatest common divisor of the above');
21    writeln(' numbers equals ', b:1)
22  end.
```

In lines 8–13, a and b are interchanged in case we have $a > b$ initially. In line 10, the old value of a is stored in a variable called *temp*, so that in line 12 this old value can be assigned to b; note that the old value of a is destroyed in line 11. This modification of the program will not significantly improve its efficiency; we included this version only as an illustration.

Problems

1. Trace through the execution of the two programs in this section when a) the values $a = 422$ and $b = 546$ are entered at the keyboard; b) the values $a = 546$ and $b = 422$ are entered at the keyboard.

2. In the following program, two positive integers are entered at the keyboard, and the program is supposed to print out first the larger number, and then the smaller number. That is, if a happens to be larger than b, the numbers need to be interchanged on lines 10–12. *Fill in these missing lines*. Semicolons must be correctly placed.

```
 1  program example(input, output);
 2  var a, b, temp : integer;
 3  begin
 4    write('Type a (a must be a positive integer): ');
 5    readln(a);
 6    write('Type b (b must be a positive integer): ');
 7    readln(b);
 8    if a > b then
```

```
 9    begin
10
11
12
13    end;
14    write('The larger of the two numbers');
15    writeln(' entered is ',b:0);
16    write('The smaller of the two numbers');
17    writeln(' entered is ',a:0)
18  end.
```

15. File Handling in Pascal

We are going to show how to rewrite the Pascal program for the Euclidean algorithm in a way that the input is taken from a file and the output is written to a file. Only slight modifications are needed in the first program presented in the preceding section on p. 55. The following program reads two positive integers from the file called `infile` and prints the result into the file called `outfile`. Since this program is very much like the above-mentioned program carrying out the Euclidean algorithm, we will explain only these modifications after the program.

```
 1  program euclid(infile, outfile);
 2  var a, b, temp : integer;
 3  infile, outfile : text;
 4  begin
 5    reset(infile);
 6    read(infile,a);
 7    read(infile,b);
 8    while a <> 0 do
 9    begin
10      temp := a;
11      a := b mod a;
12      b := temp
13    end;
14    rewrite(outfile);
15    write(outfile,'The greatest common divisor');
16    write(outfile,' of the above numbers');
17    writeln(outfile,' equals ', b:1)
19  end.
```

In line 1 of this program, the filenames `infile` and `outfile` are mentioned; these are files where the input will be read from and the output will be written to, respectively. These filenames were chosen arbitrarily. The filenames `input` and `output` are no longer used; these latter names are reserved filenames: `input` refers to the *standard input* (usually the terminal keyboard) and `output` refers to the *standard output*

(usually the terminal display screen). However, the present program does not take input from the keyboard, and it does not write to the terminal display screen.

In line 3, the files `infile` and `outfile` are declared to be text files; more precisely, they are declared to be a variable of type `text`. That is, the key word `var` on line 2 introduces the variables to be declared and lines 2 and 3 contain the names of these variables (`a`, `b`, `temp` are declared to be of type `integer` on line 2, and `infile` and `outfile` are declared to be of type `text` on line 3. The statement

```
reset(infile)
```

in line 5 prepares the file `infile` for reading; it is necessary to include this statement before the first `read` or `readln` statement referring to this file. (Its actual role is to make sure that the file is going to be read from its beginning; a file to be read must be initialized, i.e., it must be clarified from where it should be read, in the same way as an integer variable must be initialized, i.e., be given a value.)

In line 6, the statement

```
read(infile,a)
```

will read an item from the file called `infile` and makes this item to be the value the variable `a`. The variable `a` having been declared to be of type `integer`, the item to be read must be an integer. Similarly, the statement in line 7 reads the next item from the file `infile`. Items in text files must be separated by one or more spaces or end-of-line characters. That is, the file `infile` for this program may look like as follows:

```
422␣546
```

where the only space character occurring in the file is explicitly indicated as ␣, but one may include extra space and end-of-line characters; that is, the following file would also be acceptable as `infile`

```
␣␣422␣␣
␣␣␣
546
```

If we compare this program with the original Pascal program doing the Euclidean algorithm on p. 55, we find that lines 4 and 6 of the earlier program were omitted here; indeed, these lines were used to warn the interactive user to enter numbers at the keyboard, and there is no interactive user at present. Furthermore, the earlier program used `readln` instead of the statement `read` at present. The `read` statement is more flexible, since it accepts spaces and end-of-line characters as separators between items, whereas `readln` always demands an end-of-line characters after the item(s) that it reads. For this reason, if we change lines 6 and 7 in the present program to

```
readln(infile,a);
readln(infile,b);
```

then the first version of the file `infile` will not work, because it does not contain an end-of-line character after the first item to be read. In fact, if we use the first version `infile` with the modified version of the present program (i.e., the version containing the `readln` statement on lines 6 and 7) we get an error message when running the program, indicating what is called a *run-time error* (i.e., the program has compiled correctly, that is, we do not have a *compile-time error*, but we run into a problem when running the program). This run-time-error may have more or less serious consequences; with some compilers the consequence is simply that the program will not perform correctly, and indicates this with a harmless message. On

the Atrium Sun computers, when running the program the way described, we get the following message:

```
Trace/BPT trap(coredump)
```

This is a message that you will have to deal with. You probably will not want to study the meaning of Pascal error messages at this point, so we will not discuss what the message "Trace/BPT" means. The meaning of the word "coredump" is, however, important. The computer put the entire content of the memory related to the running of the program into a file called `core` so that the cause of the error could be analyzed. This may be important when trying to find the cause of the error; however, analyzing the file `core` requires some knowledge of programming that is beyond the scope of this book. What is important for now is that the file `core` is quite large, usually over 8 megabytes, and unless you are prepared to use it for analyzing the cause of the error, you should remove it as soon as possible by typing

```
$ rm core
```

to avoid wasting memory space.

Returning to the description of the above program (as originally presented, i.e., with `read` rather than `readln` on lines 6 and 7), lines 8–13 work the same way as the corresponding lines did in the original program performing the Euclidean algorithm presented on p. 55. In line 14, the statement

```
rewrite(outfile)
```

open the file called `outfile` for writing (as the first step, if this file already exists its contents will be erased, and if it does not exist it will be created). In lines 15–17 the `write` and `writeln` statements work as usual, except that the first argument `outfile` of these statements indicate that the writing should be done into the file called `outfile` rather than to the standard output (i.e., the terminal display screen).

16. Representing the Greatest Common Divisor as a Linear Combination

The greatest common divisor of any two integers can be written as a sum of multiples of these integers. Instead of sum of multiples, we will say *linear combination*, since this, perhaps more intimidating, term is the one commonly accepted in the mathematical literature. For example, the greatest common divisor of 422 and 546 is 2, and we have

$$2 = 422 \cdot 22 + 546 \cdot (-17).$$

This is such an important observation that we will formulate it as a theorem:

THEOREM. *For any two integers a and b at least one of which is assumed to be different from 0, there are integers x and y such that*

$$(a, b) = ax + by.$$

It is important to assume that not both of the numbers a and b are zero, since, as we mentioned above, the greatest common divisor $(0, 0)$ is meaningless.

The reason this is true is that in fact all the numbers occurring in the course of the Euclidean algorithm are so expressible. Indeed, we using the above example of the numbers 422 and 546, we have

$$\begin{aligned}
546 &= 422 \cdot 0 + 546 \cdot 1 \\
422 &= 422 \cdot 1 + 546 \cdot 0 \\
124 &= 422 \cdot (-1) + 546 \cdot 1 \\
50 &= 422 \cdot 4 + 546 \cdot (-3) \\
24 &= 422 \cdot (-9) + 546 \cdot 7 \\
2 &= 422 \cdot 22 + 546 \cdot (-17)
\end{aligned}$$

The first two equations here are obvious. Each of the other equations can be derived from the preceding two equations. To derive the fifth equation, for example, recall that 24 is the remainder when 50 is divided into 124. This is expressed by the equation,

$$124 = 2 \cdot 50 + 24,$$

saying that 2 is the quotient and 24 is the remainder of the division $124 \div 50$. 24 can be expressed from here as

$$24 = 124 - 50 \cdot 2,$$

where we found in convenient to change the order of the factors in the product. Taking the expression for 124 from the third and the expression for 50 from the fourth equation, we can now write this as follows:

$$24 = \big(422 \cdot (-1) + 546 \cdot 1\big) - \big(422 \cdot 4 + 546 \cdot (-3)\big) \cdot 2.$$

This can be algebraically simplified as

$$24 = 422 \cdot \big((-1) - 4 \cdot 2\big) + 546 \cdot \big(1 - (-3) \cdot 2\big).$$

If we carry out the operations indicated in the parentheses, we obtain

$$24 = 422 \cdot (-9) + 546 \cdot 7.$$

Thus we obtained the fifth equation from the third and fourth equations, as we promised we would do. The Euclidean algorithm performed with these additional calculations so that the greatest common divisor is obtained as a linear combination of the original numbers is called the *extended Euclidean algorithm.*

Problem

1. Using the extended Euclidean algorithm, find the greatest common divisors of the following pairs of numbers, and represent them as linear combinations of the given numbers: *a)* 14 and 25, *b)* 384 and 488, *c)* 645 and 242, *d)* 122 and 48, *e)* 735 and 141, *f)* 452 and 984.

17. Pascal Program for Finding the Representation the Greatest Common Divisor as a Linear Combination

In order to write a Pascal program to find the representation of the greatest common divisor as a linear combination, we have to carry out the calculations described in the preceding section in a completely symbolic way. We recall that in the example described there, we showed how the equation

$$24 = 422 \cdot (-9) \quad + 546 \cdot \quad 7$$

can be obtained from the equations

$$\begin{aligned} 124 &= 422 \cdot (-1) \quad + 546 \cdot \quad 1 \\ 50 &= 422 \cdot \quad 4 \quad\quad + 546 \cdot (-3). \end{aligned}$$

In this derivation we also used the equation

$$24 = 124 - 50 \cdot 2.$$

For the symbolic discussion, write $a_0 = 422$ and $b_0 = 524$ (the original numbers whose greatest common denominator is being taken); at the step in the Euclidean algorithm we are considering, we have $a = 50$, $b = 124$. The equation that $q = 2$ is the quotient and $r = 24$ is the remainder when 50 is divided in 124 can be written as

$$124 = 2 \cdot 50 + 24;$$

or, as we rearranged it above,

$$24 = 124 - 50 \cdot 2.$$

Symbolically, this equation can be written as

$$r = b - aq.$$

The equations

$$\begin{aligned} 124 &= 422 \cdot (-1) + 546 \cdot \quad 1 \\ 50 &= 422 \cdot \quad 4 \quad + 546 \cdot (-3) \end{aligned}$$

can be symbolically written as

$$\begin{aligned} b &= a_0 x \quad + b_0 y \\ a &= a_0 x_1 + b_0 y_1, \end{aligned}$$

where $x = -1$, $y = 1$, $x_1 = 4$, and $y_1 = -3$. We can then write

$$r = b - aq = (a_0 x + b_0 y) - q(a_0 x_1 + b_0 y_1) = a_0(x - q x_1) + b_0(y - q y_1).$$

That is, if the equation

$$24 = 422 \cdot (-9) + 546 \cdot 7$$

is symbolically written as

$$r = a_0 x_{1,\text{new}} + b_0 y_{1,\text{new}}$$

then we have

$$x_{1,\text{new}} = x - q x_1 \quad \text{and} \quad y_{1,\text{new}} = y - q y_1.$$

Now, in the next step of the Euclidean algorithm, we will have $b_{\text{new}} = a\,(= 50)$ and $a_{\text{new}} = r\,(= 24)$. For the next step, the old equations

$$\begin{aligned} 124 &= 422 \cdot (-1) + 546 \cdot \quad 1 \\ 50 &= 422 \cdot \quad 4 \quad + 546 \cdot (-3), \end{aligned}$$

or, symbolically, the equations

$$\begin{aligned} b &= a_0 x \quad + b_0 y \\ a &= a_0 x_1 + b_0 y_1, \end{aligned}$$

will have to be replaced with (updated to) the new equations

$$\begin{aligned} 50 &= 422 \cdot \quad 4 \quad + 546 \cdot (-3) \\ 24 &= 422 \cdot (-9) + 546 \cdot \quad 7, \end{aligned}$$

which can now be symbolically written as

$$\begin{aligned} b_{\text{new}} &= a_0 x_{\text{new}} \quad + b_0 y_{\text{new}} \\ a_{\text{new}} &= a_0 x_{1,\text{new}} + b_0 y_{1,\text{new}}, \end{aligned}$$

The update equations for all the new variables have already been given, except those for x_{new} and y_{new}. These update equations are easy to obtain; namely, it is clear that the first equation

$$b_{\text{new}} = a_0 x_{\text{new}} + b_0 y_{\text{new}}$$

of the present group is the same as the second equation

$$a = a_0 x_1 + b_0 y_1$$

of the earlier group. That is, we must have

$$x_{\text{new}} = x_1 \qquad \text{and} \qquad y_{\text{new}} = y_1.$$

We will now summarize these update equations. Noting that we calculated r as $r = b \bmod a$, we have

$$\begin{aligned} a_{\text{new}} &= r = a \bmod b \\ b_{\text{new}} &= a \\ x_{\text{new}} &= x_1 \\ y_{\text{new}} &= y_1. \end{aligned}$$

In Pascal, b div a denotes the quotient when a is divided into b. Thus we will write $q = b$ div a to calculate q in the equation $b = aq + r$. Thus the rest of the update equations can be written as

$$\begin{aligned} q &= b \text{ div } a \\ x_{1,\text{new}} &= x - qx_1 \\ y_{1,\text{new}} &= y - qy_1. \end{aligned}$$

In order to accomplish this update in Pascal, we need to use certain temporary variables; even in the original program performing the Euclidean algorithm we needed a temporary variable. We write

```
 1  while a <> 0 do
 2  begin
 3    temp := a;
 4    q := b div a;
 5    a := b mod a;
 6    b := temp;
 7    tempx := x1; tempy := y1;
 8    x1 := x - q*x1; y1 := y - q*y1;
 9    x := tempx; y := tempy
10  end;
```

We surrounded the whole passage with a loop starting in line 1 and ending in line 10; in the same way as the original program for the Euclidean algorithm, the passage in lines 3–9 has to be repeated as long as $a \neq 0$; in fact lines 3, 5, and 7 are the same as the lines in the loop of the Euclidean algorithm. To start off this loop, consider the initial equations in the method representing the greatest common divisor as a linear combination:

$$\begin{aligned} 546 &= 422 \cdot 0 + 546 \cdot 1 \\ 422 &= 422 \cdot 1 + 546 \cdot 0. \end{aligned}$$

This is the place in the algorithm when we use the first two equations to derive the third equation. At this point, the above equations correspond to the algebraic equations

$$\begin{aligned} b &= a_0 x \ + b_0 y \\ a &= a_0 x_1 + b_0 y_1, \end{aligned}$$

That is, our initial equations are

$$\begin{aligned} a &= a_0, & b &= b_0, \\ x &= 0, & y \ &= 1, \\ x_1 &= 1, & y_1 &= 0. \end{aligned}$$

In Pascal, these equations can be written as follows:

```
a0 := a;  b0 := b;
```

```
x := 0;  y := 1;
x1 := 1;  y1 := 0;
```

In the equations of first line we interchanged the sides; i.e., we assigned the value of the variable a to a_0 and b to b_0 rather than the value of a_0 to a and b_0 to b, as one would expect on the basis of the equations above. The only reason for this is that we would like to keep the similarity between the original program for the Euclidean algorithm and the program being designed at present. That is, the values entered at the keyboard will first be read into a and b, and then these values be also assigned to a_0 and b_0, rather than reading these values first into a_0 and b_0 and then assigning them to a and b. Putting all these together, a preliminary version of the program can now be written as follows:

```
 1  program euclid(input,output);
 2  {The greatest common divisor d of two positive
 3   integers, a and b, entered at the keyboard, is
 4   calculated and is represented in the form d=ax+by,
 5   where x and y are integers.}
 6  var a, b, a0, b0, q, x, y, x1, y1, temp,
 7        tempx, tempy : integer;
 8  begin
 9    write('Type a (a must be a positive integer):  ');
10    readln(a);
11    write('Type b (b must be a positive integer):  ');
12    readln(b);
13    a0 := a; b0 := b;
14    x := 0; y := 1;
15    x1 := 1; y1 := 0;
16    while a <> 0 do
17    begin
18      temp := a;
19      q := b div a;
20      a := b mod a;
21      b := temp;
22      tempx := x1; tempy := y1;
23      x1 := x - q*x1; y1 := y - q*y1;
24      x := tempx; y := tempy
25    end;
26    {Error check, commented out in the final version:
27    if b = a0*x+b0*y then
28      writeln('The result is correct.')
29    else
30      writeln('The result is not correct.');}
31    writeln(' ',b:1,'=',a0:1,'*',x:1,'+',b0:1,'*',y:1)
32  end.
```

When testing this program, it is worth including lines 27–30. At the end of the program, the variable b should hold the greatest common divisor of a_0 and b_0, and the equation

$$b = a_0x + b_0y$$

should be satisfied. After testing the program sufficiently, there is no further need for these lines, and so these lines should not be part of the final program. However, rather than deleting the lines, it is a good idea to keep them, but to comment them out. If on a later occasion the program is modified, then these lines can be useful for further testing, and they can easily be reinstated (uncommented out, to use an awkward but useful phrase). Of course, even if this test is successfully passed, the correctness of the above program is not guaranteed, and other tests are necessary to be convinced that the program does what it is required to do.

When we run this program and enter the inputs 422 and 546 for a and b, respectively, the printout will be as follows:

```
2=422*22+546*-17.
```

This does not look right, since -17 should be in parentheses. One can easily be convinced that, of the numbers x and y in the equation

$$(a_0, b_0) = a_0 x + b_0 y$$

one is positive and the other is negative. In the final version of the program, we check which of them in negative, and then put that number in parentheses, so as to obtain the nicer printout

```
2=422*22+546*(-17).
```

Here is the program:

```
 1  program euclid(input,output);
 2  {The greatest common divisor d of two positive
 3   integers, a and b, entered at the keyboard, is
 4   calculated and is represented in the form d=ax+by,
 5   where x and y are integers.}
 6  var a, b, a0, b0, q, x, y, x1, y1, temp,
 7        tempx, tempy : integer;
 8  begin
 9    write('Type a (a must be a positive integer):  ');
10    readln(a);
11    write('Type b (b must be a positive integer):  ');
12    readln(b);
13    a0 := a; b0 := b;
14    x := 0; y := 1;
15    x1 := 1; y1 := 0;
16    while a <> 0 do
17    begin
18      temp := a;
19      q := b div a;
20      a := b mod a;
21      b := temp;
22      tempx := x1; tempy := y1;
23      x1 := x - q*x1; y1 := y - q*y1;
24      x := tempx; y := tempy
25    end;
```

```
26    {Error check, commented out in the final version:
27    if b = a0*x+b0*y then
28      writeln('The result is correct.')
29    else
30      writeln('The result is not correct.');}
31    if x < 0 then
32      writeln(' ',b:1,'=',a0:1,'*(',x:1,')+',
33        b0:1,'*',y:1)
34    else
35      writeln(' ',b:1,'=',a0:1,'*',x:1,'+',
36        b0:1,'*(',y:1,')')
37  end.
```

The *writeln* statements in lines 32–33 and 35–36 were broken up because they are too long to conveniently fit in one line; note that the breakup point must be outside the range of the quote signs.

Problem

1. What is printed by the following program? (You must be precise; you must indicate spaces and you must not include a space where there is no space. You may indicate a space by the symbol ␣.)

```
 1  program example(output);
 2  var b, a0, b0, x, y : integer;
 3  begin
 4    a0 := 856; b0 := 962;
 5    x := -118; y := 105;
 6    b := 2;
 7    if x < 0 then
 8      writeln(' ',b:1,'=',a0:1,'*(',x:1,')+',
 9        b0:1,'*',y:1)
10    else
11      writeln(' ',b:1,'=',a0:1,'*',x:1,'+',
12        b0:1,'*(',y:1,')')
13  end.
```

2. Write a program that performs the extended Euclidean algorithm as above, but instead of only printing the equation representing the greatest common divisor as a linear combination, it represents all the remainders that come up in the Euclidean algorithm as a linear combination. That is, if one enters the numbers 546 and 422 as inputs to this program, each of the numbers 546, 422, 124,, 50, 24, and 2 is represented as a linear combination of 422 and 546 (see p. 60 for these equations). *Hint*: The easiest way to write this program is to take the last program of the main text above, and moving the statement carrying out the printing inside the loop. (Note that after moving this statement, the indentation needs to be adjusted so that the program will look nice to the human reader.)

3. If one follows the hint at the end of the preceding problem, and one enters 546 and 422 (in this order), the equations representing each of the numbers 546, 422, 124,, 50, 24, and 2 as a linear combination of 422 and 546 will be printed. However, if one enters the same numbers in the order 422 and 546, the equation representing 546 will not be printed. How can this situation be corrected? *Hint:* The easiest way is to insert a conditional statement in the program after the input statements so that, if necessary, the numbers are interchanged to make the variable a represent the larger number and the variable b, the smaller number. This can be done the same way as in Problem 2 of 14 (see p. 56). *Explain* why this idea works and *write* the program.

4. If the program written in Problem 3 is run with the inputs 546 and 422, there is still a minor problem. The first two equations printed look as follows:

```
546=546*1+422*(0)
422=546*0+422*(1)
```

The parentheses in these lines are not needed, and the output would look better if they were not printed. Explain why the parentheses are printed, and modify the program in such a way that these parentheses are not printed.

18. The Fundamental Theorem of Arithmetic

An integer is a prime if it is greater than one and its only divisors are one and itself. The first few primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,.... It is not hard to see that every positive integer greater than one can be written as a product of primes. For example,

$$2599443 = 3 \cdot 3 \cdot 7 \cdot 11 \cdot 11 \cdot 11 \cdot 31.$$

The product on the right-hand side is called the *prime factorization* of the number on the left. The only other ways to write the number on the left as a product of primes to write the same primes in a different order; e.g.,

$$2599443 = 7 \cdot 3 \cdot 31 \cdot 11 \cdot 3 \cdot 11 \cdot 11,$$

but since changing the order of factors will not change the product, this prime factorization is not essentially different. The fact that a positive can be written only one way as a product of primes, aside from the order of factors is called the Fundamental Theorem of Arithmetic. It can formally be stated as

The Fundamental Theorem of Arithmetic. *The prime factorization of every integer greater than one is unique, aside from the order of factors.*

The key result needed in order to prove this is the following lemma, described by Euclid. Euclid, however, does not mention the Fundamental Theorem of Arithmetic itself; the Fundamental Theorem was first formulated and proved by the German mathematician Karl Friedrich Gauss in 1801, even though it had been used long before.

LEMMA. *Let a and b be integers and let p be a prime. If ab is divisible by p then either a is divisible by p or b is divisible by p.*

Note that in mathematics, "or" is always used in the inclusive sense; that is, the possibility that both a and b are divisible by p is allowed. We will give a formal proof.

PROOF. We will suppose that a and b are both positive (the case when a or b equals 0 can easily be dealt with, and when a or b is negative, it is harmless to consider their absolute values instead). Assume that a is not divisible by p; we will then have to show that b is divisible by p. With this assumption, we have $(a, p) = 1$. Indeed, p being a prime number, its only divisors are 1 and p. As we assumed that p is not a divisor of a, the greatest common divisor of a and p can only be one. As we saw above, this greatest common divisor can be represented as a linear combination. That is, we have integers x and y such that

$$1 = ax + py.$$

Multiplying both sides by b, we obtain

$$b = abx + pby.$$

Both terms on the right-hand side are divisible by p; indeed, our assumption was that ab is divisible by p. Hence the left-hand side, that is, b, is also divisible by p. This is what we wanted to show.

Note that it is easy to conclude from the above lemma a similar statement for the product of more than two integers. For example, if, for some integers, a, b, and c, and for some prime number p, the product abc is divisible by p, then either a or b or c is divisible by p. To see this, write $abc = (ab)c$. Then, by using the lemma, we can see that either ab or c is divisible by p. If now ab is divisible by p, then, using the lemma again, we can see that either a or b is divisible by p. Similarly for the product of four or more integers.

We can now use the above lemma, or, rather, its extension to a product of more than two factors, to present the

PROOF OF THE FUNDAMENTAL THEOREM OF ARITHMETIC. Assume, on the contrary, that there are positive ,integers with two different prime factorizations. Let n be the smallest such integer, and let its two different prime factorizations be

$$n = p_1 p_2 \dots p_k = q_1 q_2 \dots q_l.$$

Note first that each of the primes p_i must be different from each of the primes q_j. Indeed, if, for example, we had $p_1 = q_1$ then the number

$$n/p_1 = p_2 p_3 \dots p_k = q_2 q_3 \dots q_l$$

would be a number smaller than n with two different prime factorizations, in contradiction with our assumption.

On the other hand, the equation above shows that the prime number p_1 is a divisor of the product $q_1 q_2 \dots q_l$. So, by the lemma above, or, rather, by its extension to more than two factors, at least one of the numbers q_1, q_2,... q_l, say q_j for a certain

j with $1 \leq j \leq l$, must be divisible by p_1. As q_j is a prime, its only divisors are 1 and itself. As p_1, being a prime, is different from 1, we must have $p_1 = q_j$. This is a contradiction, since, as we stated above, p_1 must be different from q_j. This contradiction shows that our initial assumption was wrong; that is, there are no integers with two different prime factorization.

The above method of proof, involving the consideration of the least positive integer with a certain property (i.e., the existence of two different prime factorizations, in the above example) to be disproved is called the *method of infinite descent*. It is a variant of *mathematical induction*.

Problems

1. You should be able to answer this question without doing any calculations(in fact, the numbers given are probably too big even for your calculator): Given that you know that

$$\begin{aligned} &7,324,472,493,949 \cdot 4,057,012,961,706 \\ &\quad = 13 \cdot 2,285,806,141,970,012,665,670,538 \end{aligned}$$

and

$$4,057,012,961,706 = 13 \cdot 312,077,920,131 + 3,$$

decide whether $7,324,472,493,949$ is divisible by 13. Give reasons; the correct reason is just as important as the correct answer. Do not waste your time by dividing 13 into $7,324,472,493,949$, since the answer can be given much more quickly by reasoning.

2. Using the second equation in Problem 1, determine the greatest common divisor 13 and $4,057,012,961,706$.

19. Truth Tables

Pascal has propositional variables, i.e., variables that can take values true and false, and it also implement certain operations of propositional logic such as negation, disjunction, and conjunction. One can use these features to create truth tables of propositional formulas. In Pascal, a propositional variable is declared to be of type *boolean*, in recognition of the British mathematician George Boole, who formalized propositional logic.

```
1  program example9(output);
2  var a, result : boolean;
3  begin
4    writeln('A','      not A');
5    writeln;
6    a := true;
7    repeat
```

```
 8      result := not a;
 9      writeln(a:1, '        ',result:1);
10      a := not a
11    until a
12  end.
```

This program prints the following:

```
21  A      not A
22
23  T        F
24  F        T
```

The numbers on the left are line numbers and are not part of the program or the printout. We numbered the lines in the printout beginning with 21, so we can easily refer to lines both in the program and in the printout. In line 2 (of the program), the variables *a* and *result* are declared to be of type *boolean*; these are variables that can assume only the values *True* and *False*. The first line, line 21, of the printout is printed in line 4; the number of spaces in quotes have to be carefully adjusted to get a nice printout. The effect of the *writeln* statement in line 5 is to print a *blank* (i.e., empty) line; this is line 22. The *repeat* statement in lines 7-11 is similar to the *while* statement considered above, but it works somewhat differently. Its syntax is

`repeat` *list of statements* `until` *condition*

Here "list of statements" refers to one or more statements separated by semicolons; there should be no semicolon after the last statement. At present, the list of statements consist of the three statements in lines 8-10, and the condition is the boolean variable *a*; since a boolean variable can be either true or false, the condition will be satisfied if this variable is true.

The meaning of the *repeat* statement is as follows. The list of statements is executed and then the condition is tested. If it is found that the condition is false, the list of statements is executed again; this is repeated until it is found that the condition is true, when the execution continues on the next line. The *repeat* statement is also a loop statement.

In the present program, the loop in line 7 is entered with *a* being false, then in line 8 *b* is set to be the negation of *a* (the operator `not` denotes logical negation). In line 9, the current values of *a* and *b* are printed out; the second item in the *writeln* statement prints an appropriate number of spaces for good alignment of the printed table. The width 1 specified here results in printing `T` or `F` false for the values of *a* and *b*; if the width were specified to be 5, `True` and `False` would be printed.[13] In line 10, the value of *a* is changed to be false, so the second time the loop in line 7 is entered with *a* false. In line 10, *a* is changed to be true, and in line 11 the loop is exited (since the condition is satisfied).

The following program prints out the truth table of conjunction.

```
1  program example(output);
2  var a, b, result : boolean;
```

[13]This behavior is observed with the GNU Pascal Compiler `gpc`; Sun Pascal behaves differently. In Sun Pascal, `TRUE` and `FALSE` are printed. The International Standards Organisation (ISO) Pascal Report supports the interpretation given by GNU Pascal – see Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, third edition, Springer-Verlag, New York–Berlin, 1985, §§12.3.4–12.3.5 on pp. 198–199.

```
 3  begin
 4    writeln('A ','     B ','   result');
 5    writeln;
 6    a := true;
 7    b := true;
 8    repeat
 9      repeat
10        result := a and b;
11        writeln(a:1,'      ',b:1,'      ',result:1);
12        b := not b
13      until b;
14      a := not a
15    until a
16  end.
```

The printout looks as like this:

```
21  A      B    result
22
23  T      T      T
24  T      F      F
25  F      T      F
26  F      F      F
```

The above program contains two nested *repeat* loops (i.e., one loop is inside the other loop). The outside loop in lines 7–15 is entered with a true and in the same way as in the first program, this loop is executed twice, first with a true, and the second time with a false. Similarly, for each execution of the outside loop, the inside loop on lines 9–13 is executed twice, first with b true and then with b false. In line 10, the conjunction of a and b is calculated (the operator `and` denotes logical conjunction), and in line 11 one line of the truth table is printed.

One can use the same idea to print truth tables containing more than two variables. The following program prints out the truth table of the formula $(A \wedge b) \vee (\sim C)$, where $\wedge$ stands for conjunction, $\vee$, for disjunction, and $\sim$, for negation (the operator for disjunction in Pascal is `or`):

```
 1 program example(output);
 2 var a, b, c, result : boolean;
 3 begin
 4   write('A','  B');
 5   writeln('  C',' result');
 6   writeln;
 7   a := true;
 8   b := true;
 9   c := true;
10   repeat
11     repeat
12       repeat
13         result := (a and b) or (not c);
14         write(a:1,'  ',b:1,'  ',c:1);
15         writeln('   ',result:1);
```

```
16         c := not c
17       until c;
18       b := not b
19     until b;
20     a := not a
21   until a
22 end.
```

The printout of this program is as follows:

```
31 A  B  C result
32
33 T  T  T   T
34 T  T  F   T
35 T  F  T   F
36 T  F  F   T
37 F  T  T   F
38 F  T  F   T
39 F  F  T   F
40 F  F  F   T
```

Problems

1. As we mentioned in the footnote above, the description of the printing of Boolean variables given above only applies to GNU Pascal. In Sun Pascal, `TRUE` and `FALSE` are printed even if the Boolean variables are specified with width 1. This results in misalignments of the tables printed by the programs in the present section when using Sun Pascal. Solve the misalignment problem by printing the Boolean variables in the above programs with width 5; width 5 is needed, since the word `TRUE` is contains four letters while the word `FALSE` contains 5. In order to get a neat alignment in the tables, the number of quoted spaces in the `write` or `writeln` statements in these programs must be adjusted.

2. The misalignment problem in Sun Pascal can also be solved by replacing the `writeln` statements by longer conditional statements of the type

```
if a then
  write('T')
else
  write('F')
```

Solve the misalignment problem in Sun Pascal by using such conditional statements.

20. Day of the Week

Next we are going to consider a somewhat more complex program that will calculate the day of the week give when given a date such as

```
3 17 1997
```

meaning March 17, 1997. In the declaration part of the program we use the concept of an *array*. The declaration, given below in line 12 is as follows:

```
lengthofmonth:array[1..12] of integer;
```

The identifier `lengthofmonth` here describes the data having data type

`array[1..12] of integer`.

This means that there are 12 locations, denoted as

```
lengthofmonth[1]
lengthofmonth[2]
lengthofmonth[3]
.................
lengthofmonth[12]
```

allocated to store data of the type integer. These will be used to indicate the number of days in each of the month January through December; in actual fact, since multiples of seven can be neglected in calculating the day of the week, instead of 31 for January, we will store 3, instead of 28 for February, we will store 0, and so on.

The *char* data type has not been used before; this is used to store *characters*, i.e., letters, numbers, and some other symbols. Line 11 contain the following declaration:

```
dayofweek:packed array[1..9] of char;
```

This reserves 9 locations of the data type char called

```
dayofweek[1]
dayofweek[2]
dayofweek[3]
............
dayofweek[9]
```

These will be used to store the letters in the name of the day of the week (Monday, Tuesday, ...) to be printed out. The meaning of the word "packed" in the declaration is that characters normally take up less room in the memory than integers and other data types; so a sequence of characters can be placed in the memory more tightly than a sequence of integers, for example. Packed arrays take up less room, but access to individual members of the array is often more costly than if the array was not packed (however, we will not need access to individual members of this array; we will only be interested in the whole word spelled out by the array).

A kind of loop statement not discussed before, the *for* statement, is used in lines 50–51:

```
for i:=1 to month-1 do
  daystocurrent:=daystocurrent+lengthofmonth[i]
```

In this statement, the statement in the second line is repeatedly executed, starting with the initial value (currently 1) of the control variable (i at present), incrementing this value by 1 for each new execution; the statement is executed the last time with final value (currently $month - 1$) of the control variable. The variable is usually an integer (it is allowed to be any *ordinal* data type, but we will not discuss what these are), and the initial and terminal values can be expressions; these expressions are

evaluated only once, when the loop is entered. It is not permissible to change the value of the control variable inside the loop. It is easy to see that this loop adds the values of *lengthofmonth*[*i*] for i with $1 \leq i \leq month - 1$ to the starting value of the variable *daystocurrent*.

A new kind of conditional statement is found in lines 70–77, called the *case* statement:

```
case daycount of
  0: dayofweek:='Sunday   ';
  1: dayofweek:='Monday   ';
  2: dayofweek:='Tuesday  ';
  3: dayofweek:='Wednesday';
  4: dayofweek:='Thursday ';
  5: dayofweek:='Friday   ';
  6: dayofweek:='Saturday '
end
```

In this statement, *daycount* is a variable, called the *selector*, of type integer, and depending on its value, the statement in the corresponding line is executed. For example, if $daycount = 3$, the line

```
dayofweek:='Wednesday'
```

is executed. The selector in a case statement could be an expression of the type integer, and not just a variable.

Next we give the listing of the whole program. The comments in the program give a fairly detailed explanation of how this program works, but after the listing we will add more in the way of explanations.

```
 1  program daydate(input,output);
 2    {This program is calculates the day of the week
 3     for each date in the Gregorian calendar; that
 4     is after, but not including, the year 1582 in
 5     Western Europe, and after 1752 in Great Britain
 6     and its American colonies}
 7  var
 8    day,month,year,daystocurrent,
 9      leapyearcount,yearcount,daycount,i:integer;
10    dayofweek:packed array[1..9] of char;
11    leapyear:boolean;
12    lengthofmonth:array[1..12] of integer;
13  begin
14    write('Enter the date by typing the number ');
15    writeln('of months,');
16    write('days, and years.  E.g.  for September ');
17    writeln('15, 1990');
18    writeln('enter 9 15 1990:');
19    readln(month,day,year);
20    {decide if the current year is a leapyear:}
21    if ((year mod 4 = 0) and (year mod 100 <> 0))
22                  or (year mod 400 = 0)
23    then
```

```
24      leapyear:=true else leapyear:=false;
25    {initialize the lengthofmonth array; this will
26     store the remainder of the number of days in
27     each month divided by seven:}
28     lengthofmonth[1]:=3;
29     lengthofmonth[2]:=0;
30     lengthofmonth[3]:=3;
31     lengthofmonth[4]:=2;
32     lengthofmonth[5]:=3;
33     lengthofmonth[6]:=2;
34     lengthofmonth[7]:=3;
35     lengthofmonth[8]:=3;
36     lengthofmonth[9]:=2;
37     lengthofmonth[10]:=3;
38     lengthofmonth[11]:=2;
39     lengthofmonth[12]:=3;
40     {the current setting of February is only correct
41      if the year is not a leap year; this will be
42      remedied:}
43     if leapyear then lengthofmonth[2]:=1;
44     {now calculate how many days have elapsed in
45      the year up to and including the current day;
46      actually, multiples of seven will be ignored
47      in this calculation}
48      daystocurrent:=day;
49      for i:=1 to month-1 do
50        daystocurrent:=daystocurrent+lengthofmonth[i];
51      {each year has 365=52*7+1 days, and each leap
52       year has 366=52*7+2 days; hence the number of
53       years before the current year is counted, and
54       the number of leap years is added (so each
55       leap year is counted twice); the current year
56       is not counted}
57       yearcount:=year-1;
58       leapyearcount:=yearcount div 4 +
59              yearcount div 400 - yearcount div 100;
60       yearcount:=yearcount+leapyearcount;
61      {now we can calculate the daycount, which is
62       essentially the date of the week}
63      daycount:=(yearcount+daystocurrent) mod 7;
64      {now daycount is related to the actual date of
65       the week; the following table can be set up by
66       printing out daycount for a date where the date
67       of the week is known}
68      case daycount of
69        0: dayofweek:='Sunday   ';
70        1: dayofweek:='Monday   ';
71        2: dayofweek:='Tuesday  ';
```

```
72        3: dayofweek:='Wednesday';
73        4: dayofweek:='Thursday ';
74        5: dayofweek:='Friday   ';
75        6: dayofweek:='Saturday '
76      end;
77      writeln(dayofweek)
78  end.
```

The Gregorian calendar, named after Pope Gregory XIII (1502–1585; pope 1572–1585) is the calendar currently in use. According to it, a year consists of 365 days, except for leap years consisting of 366 days. A leap year is a year whose number is divisible by 4, except for years whose number is divisible by 100; among these years only those are leap years whose number is divisible by 400. This is an improvement over the the Julian calendar, introduced by Julius Caesar in 46 B.C., according to which every fourth year was a leap year. The above program works only for the Gregorian calendar.

In line 8 the variables *day*, *month*, *year* will be used to store the day, month, and year in the date about which the day of the week is to be established. This date will be typed at the keyboard. The variable *daystocurrent* will be used to calculate the number of days to and including to the day in question in the given year; in this calculation, multiples of 7 may be neglected. In line 9, in the variable *leapyearcount* the number of leap years are counted from the year 1 A.D. as if the Gregorian calendar had been in effect from that date. Even though the Gregorian calendar has not been effect for that long, this will not cause problems in comparing recent dates for which the Gregorian calendar is valid. In the variable *yearcount* the number of years are counted since year 1 A.D. with leap years counting double; the reason for this is that 365 divided by 7 gives a remainder 1, while 366 divided by 7 gives a remainder 2. In the variable *daycount* the number of days from January 1 of the year 1 A.D. are counted, neglecting multiples of 7. This is done only formally, since, as we already mentioned, the Gregorian calendar has not been effect that long; for comparing days of the week for recent dates, this is, however, of no consequence. As a point of interest, note that there was no year 0 A.D.: year 1 B.C. was directly followed by year 1 A.D.; since we only consider recent dates, this is again of no consequence for us. We already discussed the array declarations in lines 10 and 12; in line 11, the boolean variable *leapyear* will be true if the given year is a leap year; this will be important in calculating the number of days to the date in question in the given year. In lines 14–18, a message is written to prompt the user to enter the date; in line 19, the date entered is read. The *readln* statement in this line will assign the first number entered to the variable *month*, the second one, to the variable *day*, and the third one, to the variable *year*.

In lines 21–22, it is decided if the year in question (called "current year" in the comments) is a leap year. If it is, the boolean variable is assigned the value *true*, otherwise it is assigned the value *false*. In lines 28–39, the length of each month is entered, while multiples of 7 are neglected; that is, for example, 2 is entered for September in line 36, since September has 30 days, and the remainder of 30 divided by 7 is 2. The length of February is considered to be 28 days; the case of leap years will be taken into account later, in line 43. In lines 48–50, the number of days in the year in question is calculated, and this value is assigned to the variable *daystocurrent*; see the discussion of the *for* statement above. As we mentioned several times before,

multiples of 7 are neglected here. In lines 57–59, the number of leap years from the year 1 B.C. is calculated (this calculation uses the fictitious assumption that the Gregorian calendar has been used since that date). The variable *yearcount* calculated in line 60 counts the number of days (with multiples of 7 neglected) from January 1, 1 A.D. to the year in question; for this, one simply has to add the number of leap years to the number of the year in questions, so that leap years are counted twice, while regular years are counted only once (this is because 365 gives a remainder of 1 when divided by 7, while 366 gives a remainder of 2). In line 63 the number of days elapsed in the year in question is added, and the remainder when dividing by 7 is taken. This remainder corresponds to the day of the week – to figure out which remainder corresponds to which day, one needs to run the incomplete version of the program with a recent date where the date of the week is known. Having done this, one can make the correspondence between the value of the remainder and the days of the week; this is done in lines 68–76. On line 77, the day of the week in question is printed out.

CHAPTER V

MISCELLANEA

A short glimpse of the mathematical typesetting language TEX is given; in order to write about mathematics today, the tool of choice is TEX, rather than any of the standard word processors. A basic introduction to using "plain vanilla" Unix electronic mail is given, then some shell scripts are discussed that facilitate the writing and sending of electronic mail. Next, source scripts are discussed for changing the Unix environment, e.g., changing directories on changing shell variables. Finally, a brief section on job control is given.

21. Mathematical Typesetting

The best way to write about mathematics on computers is to use the typesetting language TEX (pronounced *tek*) designed and implemented by Donald Knuth. TEX is now widely accepted in the mathematical community. It is extensively used in producing mathematics books and journals. It is also used in other sciences such as physics, chemistry, and astronomy, where professional typesetting of mathematical formulas is important. It is excellent for typesetting even texts not containing mathematical formulas. These notes were typeset using TEX. The letters TEX are in fact the Greek upper case letters tau, epsilon, and chi, except that the epsilon is shifted to produce the unique appearance. The letters $\tau\epsilon\chi$, which is the lower case version of TEX, form the beginning of the word technology in Greek.

Before you start experimenting with TEX, it is best to make a new directory called something like *texfiles*, and do all the work in this directory, rather than mix works on TEX, Pascal, and HTML in the same directory. At this point, it is best to use an X Windows environment, such as OpenWindows on Sun Workstations, because this will be needed for previewing the typeset version of work being created. To start out, using the text editor *vi*, create a file called, say, *first.tex* (the ending *.tex* of the file name is often required – the first part of the name is chosen by the user at will) with the following content:

```
This is \TeX!
\bye
```

The character \, called backslash, plays a special role in TEX; it introduces what are called *control sequences*. In the above file, there are two control sequences: `\TeX`, which produces the tex logo TEX; the other one is `\bye`, indicating the end of the tex file. To typeset this file, type

```
$ tex first
```

assuming that the name of the above file was *first.tex* (as usual, the `$` sign at the beginning of the line stands for the prompt). If everything goes well (and often it will not, as we will discuss soon), after the *tex* program finishes, two files will be produced: *first.dvi* and *first.log*. Not surprisingly, the first file is usually called *dvi* file (pronounced *D-V-I*), and the second one, *log* file; the file *first.tex* is usually called tex file (tex is pronounced *tek*). Here DVI stands for *device independent*, which refers to the fact that it contains information that any kind of printing device, supplied with appropriate computer software, should be able to use the dvi file to print the document. The log file contains a log, i.e., list, of actions taken by the TEX program; this is usually an extended version of what it writes on the screen. If you become familiar with TEX, you will find the information in the log file very useful; at this point, it may be somewhat confusing.

The next step is to preview the printout on the screen; Assuming that you are in an X Windows environment such as OpenWindows, this can be done by using the program *xdvi*; this allows you to view the result produced by the dvi file. Type

```
$ xdvi -s 4 -thorough first &
```

Here the option *-s* introduces the *shrink factor* 4; a larger number will produce a smaller display, and the option *-thorough* influences the behavior of the *xdvi* program on color displays (if you are really interested in the details, you can read the on-line manual pages of *xdvi*). The value 4 may be replaced by something else depending on the set up and on the size of the display window required. Finally, the symbol `&` at the end of the line ensures that *xdvi* will run in the *background*. The effect of this, that you will receive a prompt in the window in which you typed the above line, and so you can continue to use this window even while *xdvi* is running (i.e., while its display window is open).

Typing the above line will result in the opening of a display window. One can close this window by typing `q` in it (i.e., typing *q* while the mouse pointer is in this window). If the document extends to several pages, one can go to the next page by typing `n` (or `f` for "forward") in the display window; typing `3n` or `3f` will go three pages forward. Typing `4b` or `4p` will be back four pages (`p` for previous), while `n` and `p` will go back one page. Typing `7g` will take you to p. 7, while typing `g` alone will display the last page.

The display at present will be a single line:

> This is TEX!

The letters in the display will perhaps be smaller than desired; we will explain later what to do about this. This display window may be left open. If changes are made in the file *first.tex*, and the file is repeatedly typeset by typing

```
$ tex first
```

then the display can be updated simply by clicking in the display window with the left mouse button.

To print the file, one can type

```
$ dvips first -o
```

This will produce a file called *first.ps*. *dvips* stands for dvi-to-PostScript, that is, it translates the dvi file *first.dvi* to PostScript file called *first.ps*. PostScript is a page-definition language of Adobe Corporation; the PostScript file can be sent directly to the printer by typing

```
$ lpr first.ps
```

This will send the file to the *default* printer; that is, the computer will guess which printer you want to use. If you want to use a specific printer, you need to know the name of the printer; assume, for example, that the name of the printer you want is *hwp*. Then you can specify that you want the printout on this printer by typing

```
$ lpr -Phwp first.ps
```

Along with your printout, *header page* is usually printed. This is the first page, and it contains information about the job you are printing, but it is not part of what you want to be printed. If you are the only one using the printer at the given time, your printout will not be mixed up with other people's printouts. So, to save paper, you can avoid printing the header page by typing

```
$ lpr -h first.ps
```

After printing, the files *first.dvi* and *first.ps* may (and perhaps should) be removed by typing

```
$ rm first.dvi first ps
```

in order to save disk space; these files can be easily be re-made if the file *first.tex* is kept.

Often there are mistakes in the tex file, and TEX waits, giving the user a chance to correct the mistake; this was especially important when computers were much slower than nowadays. With faster computers, it is usually simpler to stop TEX, correct the mistake, and then run TEX again. One can stop TEX by holding down the control key, and pressing the key c; briefly, this action is described as typing Control-`c`, or `Ctl-c` (often one says `Ctl-C`, but it is important to keep in mind that lower case `c` is typed, i.e., the shift-key is not held down). Typing `Ctl-c` will halt, or more precisely *kill* or *terminate*, most programs (a "halted" or "stopped" program may be resumed or continued – a "killed" or "terminated" program cannot be resumed or resurrected). If this does not work then some other action needs to be taken; we will give some details.

In the file

```
This is \Tex!
\bye
```

there is a mistake; the control sequence `\TeX` is misspelled as `\Tex`, which is enough to seriously upset tex. If the name of the above file is *first1.tex*, then typing

```
$ tex first1
```

will produce the message

```
This is TeX, Version 3.1415 (C version 6.1)
(first1.tex
!  Undefined control sequence.
l.1 This is \Tex
                .
? □
```

The cursor, indicated as □, will appear after the question mark in the last line. Typing `x` will terminate TEX. Typing `Ctl-c` will have the same effect on some systems, while it may have no effect at all on other systems.

Another type of mistake is contained the following file, called, say, *first2.tex*; namely, the control sequence `\bye` at the end of the file is missing:

```
This is \Tex!
```

When one runs TEX on this file, the following message is produced:

```
This is TeX, Version 3.1415 (C version 6.1)
(first2.tex)
*□
```

The asterisk * indicates that TEX can accept input from the keyboard. The missing control sequence `\bye` can now be typed; this will make TEX terminate gracefully.

A third type of mistake appears in the following file, called *first3.tex*:

```
\input mastex
This is \TeX!
\bye
```

The first line of this file instructs TEX to take input from a file called *mastex.tex* (located either in the present directory or in some other directory that TEX "knows about"); there is, however, no file called *mastex.tex*. The following message is produced:

```
This is TeX, Version 3.1415 (C version 6.1)
(first3.tex
! I can't find file `mastex'.
l.1 \input mastex

Please type another input file name: □
```

Here one could type the name of the file that is actually desired; it may happen, however, that there is no such file (for example, it is anticipated that the file will be created later). For this purpose there is a file called *null.tex* somewhere in TEX's own area; this file contains nothing at all, its only purpose is to use its name in such emergency situations. Typing `null` will allow TEX to terminate.

As a simple experiment to see how TEX works, using the text editor *vi*, create a file called *fermat.tex* with the following content (at this point, it is best to use an X Windows environment, such as OpenWindows on the Suns, because this will be needed for previewing the typeset version of the file):

```
 1  \input amstex
 2  \documentstyle{amsppt}
 3  \magnification=\magstep1
 4
 5  \head
 6  Fermat's Last Theorem
 7  \endhead
 8
 9  Around 1637,
10  the French lawyer and amateur mathematician,
11  Pierre de Fermat (1601-1665), wrote on the
12  margin of his copy of a book by Diophantus,
13  the Greek mathematician of the third century
14  {\smc a.d.}, a sentence that can be rendered
15  with modern mathematical notation as follows:
16
17  ``There are no positive integers $n>2$, $a$,
18  $b$, and $c$ such that
```

```
19     $$a^n+b^n=c^n$$
20  holds. I have found a marvelous demonstration
21  of this but there is not enough space on the
22  margin to write it.''
23
24  This statement is now called
25  {\it Fermat's Last Theorem}.
26  Observe that for $n=2$ the
27  above equation has many solutions;
28  the simplest one is
29  $3^2+4^2=5^2$.\par
30  It is generally assumed that Fermat
31  did not have a proof of his Last
32  Theorem.
33  In the first place, he never
34  announced the general statement publicly,
35  but he repeatedly made public statements
36  about the special cases $n=3$ and $n=4$ as
37  late as 1659. Probably he found a mistake
38  in his original proof and was able to prove
39  only special cases. He had no reason to retract
40  a statement he never made in public.
41
42  Fermat's Last Theorem was finally established
43  by Andrew J. Wiles of Princeton University
44  % The history of Wiles's theorem, with a
45  % in the first version is interesting, and
46  % maybe should discussed in an extended
47  % version of this essay.
48  in 1994. You can read more about its history
49  at
50
51  \centerline{\tt
52  http://www.coe.uncc.edu/cas/flt.html}
53
54  \bye
```

The numbers on the left are line numbers, and not part of the file. Before we explain the main points about how to prepare the above file, we include here the typeset version of the above text:

Fermat's Last Theorem

Around 1637, the French lawyer and amateur mathematician, Pierre de Fermat (1601-1665), wrote on the margin of his copy of a book by Diophantus, the Greek mathematician of the third century A.D., a sentence that can be rendered with modern mathematical notation as follows:

"There are no positive integers $n > 2$, a, b, and c such that

$$a^n + b^n = c^n$$

holds. I have found a marvelous demonstration of this but there is not enough space on the margin to write it."

This statement is now called *Fermat's Last Theorem*. Observe that for $n = 2$ the above equation has many solutions; the simplest one is $3^2 + 4^2 = 5^2$.

It is generally assumed that Fermat did not have a proof of his Last Theorem. In the first place, he never announced the general statement publicly, but he repeatedly made public statements about the special cases $n = 3$ and $n = 4$ as late as 1659. Probably he found a mistake in his original proof and was able to prove only special cases. He had no reason to retract a statement he never made in public.

Fermat's Last Theorem was finally established by Andrew J. Wiles of Princeton University in 1994. You can read more about its history at

`http://www.coe.uncc.edu/cas/flt.html`

Next we explain line by line the meaning of the text in the above file. In line 1, the resources of *amstex* (also written as $\mathcal{AMS}$-TEX) are requested. $\mathcal{AMS}$-TEX is the version of TEX used by AMS, the American Mathematical Society. It supports the typesetting of more mathematical formulas than "regular" TEX (called *Plain* TEX); another popular package is *Latex*. Line 2 specifies that that the style file *amstppt*, which stands for "AMS Preprint Style" is to be used. This is a widely available style file. Books or mathematical periodicals may have their own style file, so that the same input file can be made to appear different when printed, depending on the style file used. The third line specifies a magnification by a factor of 1.2 over the regular size, which would be

```
\magnification=\magstep0
```

If the magnification is not specified, magstep0 is assumed; in the first example, we did not specify the magnification, and this is why the letters in the printout were so small.

In lines 5 and 7 the pair `\head` and `\endhead` of control sequences enclose the title of the passage. These control sequences are specific to $\mathcal{AMS}$-TEX; the way this title was rendered is specific to AMS Preprint Style (*amsppt*); using another style, it might have been rendered in, say, small capitals: FERMAT'S LAST THEOREM.

Note that the line breaks in the tex file have nothing to do with the line breaks in the way the text is typeset; that is, even though line 9 is very short, the corresponding text in the typeset version appears normal. Line breaks and spaces only generate spaces between words in the typeset text (in fact, two or more spaces in the text file appear in the typeset text as a single space between words). The places for the line breaks are calculated by the tex program, and, if necessary, the tex program will even break up (with a hyphen) a long word that does not quite fit at the end of the line. On line 14, `\smc` changes the type face to "small capitals"; that A.D. is printed instead of a.d.; note that A.D. would not look as good. The braces { and } are used as delimiters, and they do not appear in the typeset text. Delimiters have many uses; here they indicate that the effect of the control sequence `\smc` applies only between these delimiters (the meaning of this control sequence is: typeset everything that follows in small capital typeface).

Line 16 is a blank line; a blank line starts a new paragraph, so the text that begins on line 17 starts a new paragraph. Single quotes are obtained by typing ` and ', while double quotes are obtained by typing `` and '' (i.e., by typing two

single quotes, rather than using the double-quote key " on the keyboard); thus the text on lines 17–22 will be typeset within double quotes. Mathematical formulas are enclosed within dollar signs. On line 17, `$n>2$` is typeset as $n > 2$, `$a$` is typeset as a. Note that mathematical symbols have a different type face from regular text; the type face is called *mathematical italics*, and it is a little different from regular *italics* (i.e., "slanted" typeface; the word slanted, however, should not be used here, since slanted is a little different from italics: *this is slanted* and *this is italics*). Mathematical formulas that are between double dollar signs will be displayed in the middle of the line. Thus the formula `$$a^n+b^n=c^n$$` is printed as

$$a^n + b^n = c^n.$$

Note that we put the formula in line 19 in a separate line by itself; however, this did not make any difference, and the typeset version would have been the same even if instead of lines 18 and 19 we would have written

```
$b$, and $c$ such that $$a^n+b^n=c^n$$
```

or, for that matter,

```
$b$, and $c$ such that $$a
^n+b^n=c^n$$
```

The reason we put the formula in line 19 in a line by itself because this makes it easier to read the tex file. After some practice, it is possible to learn how to read a tex file without typesetting it, and this allows one to work much faster. However, in order to read the tex file fast, it is important to format the tex file properly, even if formating has no effect on the typeset file. The situation is somewhat similar with the use of indentation in Pascal programs; indentation has no effect on the way the program runs, but it is useful for people wanting to read the program. The difference is that there are fairly strict rules on how to indent a Pascal program, whereas there are really no accepted rules as to how to format a tex file.

In the formula in line 19, the caret symbol ^ introduces superscripts (exponents). Thus `$a^n$` is printed as a^n. Subscripts are introduced by underscore _. Thus, to typeset a_2, one needs to type `$a_2$`. Double superscripts are not allowed, so one cannot type `$3^3^3$`. In order to typeset 3^{3^3}, one needs to type `$3^{3^3}$`; that is, the delimiters { and } must be used to keep things together.

In line 25, the control sequence `\it` indicates italics type font; the effect of this control sequence is confined between the delimiters { and }. Slanted typeface is indicated by `\sl` and bold face (thick letters) by `\bf`. In line 29, the control sequence `\par` indicates the beginning of a new paragraph. Again, the line break has no effect, and a new paragraph would be started even if lines 29–30 were replaced with

```
$3^2+4^2=5^2$.\par It is generally
assumed that Fermat
```

As mentioned above, new paragraphs can also be indicated by blank lines; in fact, using a blank lines instead of the control sequence `\par` would have been better, because that would give a better appearance to the tex file (it would be easier to ascertain where the paragraphs are). We used this control sequence here only as an illustration of a different way of accomplishing the same thing. Writing `\par\par` and even adding blank lines has no new effect; the result will be the same as if `\par` had been written only once.

Lines 44–47 begin with the percent sign `%`; the percent sign introduces a comment, and everything in the line following a percent sign will be ignored by the tex program (i.e., it will not appear in the printout). Here the comments in lines 44–47 were written by to author to indicate how the text may need to be revised. If you really want a percent sign in the text, you can type `\%`; this percent will be typeset as % and will not be interpreted as a symbol introducing a comment; similarly, to get a dollar sign, you need to type `\$`; this will be typeset as $, and will not be interpreted as a dollar sign used to enclose mathematical formulas.

In line 57, the control sequence `\centerline` will center the text between the delimiters { and }. The control sequence `\tt` indicates that this text should be set in *typewriter font*; that is, the letters look like they were typed on a typewriter, rather than printed. The text to be centered should be a separate paragraph; this is why lines 50 and 53 were left blank. The control sequence `\bye` in line 54 signifies the end of the tex file.

There is much more to know about TEX, and in order to learn it, you need to consult books written about it. To find out more, you can look up TEX-related material at the Web site of the TEX Users Group:

```
http://www.tug.org/interest.html
```

An introduction to Latex can be found at

```
http://riceinfo.rice.edu/Computer/Documents/
Classes/Unix/class/class.html
```

We had to break the line here, but note that there are no line breaks (or spaces) in the above address.

TEX and the Unix directory tree. We mentioned above the meaning of the

```
\input somefile
```

statement: instead of proceeding with the working on the present file, the content of the file called `somefile.tex` is going to be processed. In place of taking a simple file located in the present directory, most up-to-date Unix installations of TEX can refer to anywhere in the Unix directory tree. For example, one can write the following line:

```
\input ../texinputs/mytex
```

This takes inputs from the file `mytex.tex` located in the subdirectory `texinputs` of the parent directory of the current directory.

The operating system DOS would apparently have some difficulty with using such references to locations in the directory tree, since DOS uses backslash (\) instead of slash (/) in describing the directory tree. The line

```
\input ..\texinputs\mytex
```

would not work in TEX, since backslash is a special character in TEX.

Setting up your own `texinputs` directory. The `\input` control sequence, in addition to recognize the Unix directory tree, as discussed just before, can also take files certain special directories without having to state the directory tree. For example, when you type

```
\input amstex
```

(cf. p. 81), in Red Hat Linux 6.0 the file

```
/usr/share/texmf/tex/amstex/base/amstex.tex
```

is used. That is, when you type

```
\input somefile
```

the file `somefile.tex` is taken from your working directory, or if there is no file by that name, then it is taken by certain directories specified by the by the TEX program. You can add to the directories searched by the TEX program to find this file. In order to do this, you may add the lines

```
TEXINPUTS=:$HOME/texinputs
export TEXINPUTS
```

in your `.profile` (or `.kshrc`) file in Korn shell (`ksh`) or in your `.bash_profile` (or `.bashrc`) file in the Bourne-again shell (`bash`). The first line assigns a value to the shell variable `TEXINPUTS` and the second line makes this value available to various programs (cf. the footnote on p. 98).

With this change, the `\input` control sequence will make the TEXprogram look for the file `somefile.tex` in all the places it used to look for it, but now it will also look for it in the directory `texinputs` in your home directory.

22. Electronic Mail

Electronic mail is sometimes called email; more often, it is called e-mail. We are convinced by Donald Knuth's essay *Email (let's drop the hyphen)*, which can be found at

`http://www-cs-staff.Stanford.EDU/~knuth/email.html`

that the former version should be used.

There are various mail interfaces to Unix, such as pine, but these are not installed on every system; it is available at the Atrium, and you are welcome to find out about if interested. Here we will explain how to use Unix mail itself without such "easy user interface."

Reading email. Reading email is simple. First, it is desirable to create a directory, such as `mail_received`, where you want to save the letters you receive. The system usually notifies you when you have email. In any case, you can find out whether you have mail by typing[14]

```
$ mail
```

here `$` is the prompt given by the computer. if you have no mail, the computer will answer by something like

```
No mail for jsmith
```

where `jsmith` stands for the login name of the person logged in to the computer. If you have mail, the computer will list the letters received, numbering them as 1, 2,

[14] Unix often has different versions of the mail programs. If you do not get the behavior described below, try typing

```
$ Mail
```

instead. In Linux, you may also try

```
$ mail -n
```

which inhibits the reading of the file `/etc/mail.rc`, since the unwanted behavior of the mail program may be caused by the content of this file. This latter command does not work in all versions of Unix.

3, etc., and indicating who the mail is from and the subject of the email (which is specified by the sender – see below). Then the mail program gives you a prompt:

```
& □
```

(here □ indicates the cursor). We used `&` for the mail prompt, but sometimes other symbols are used; often `>` is the mail prompt. If you are not interested in reading the letters now, you can type

```
& x
```

(that is, you type `x`; `&` is the mail prompt); this will exit the mail program (`x` is for e*x*it) without doing anything; you can come back and read the letters at a later date. If you want to read, say, the 2nd letter, you can type

```
& 2
```

Sometimes you receive a long letter; you can scroll down in this letter by pressing the space bar or the return key (the former usually scrolls by a whole window content, while the latter only scrolls down a single line). If the letter is too long and you do not want to read it all the way, you can quit reading it by typing `q`; you have to be careful, however, because you do not want to type `q` after the letter scrolled down all the way, since this will cause the mail program to quit (see below). You can tell whether or not this is the case: when the letter has scrolled down all the way, the mail prompt will reappear; if it has not, then you can usually see the sign

```
--More--
```

highlighted (if your window has a white background, then the highlighted word will probably appear written in black into a little black rectangle) If you want to save this letter under the file name `letter.msmith.5`, you can type

```
& s 2 letter.msmith.5
```

If there is no file by this name, it will be created; if there is a file by this name, letter #2 will be appended to the end of this file.

You probably want to save all your letters in the directory `mail_received`, created for this purpose. If you do not remember whether you are in this directory, you can find it out by typing

```
& !pwd
```

Here `&` is the mail prompt, written by the computer, and you only type the rest. The exclamation point `!` is the so called "shell escape character"; *shell* is the regular environment where you type Unix commands. That is, after typing `!`, you temporarily leave the mail environment, and you can type any Unix command. We met the command `pwd` above; it stands for "print working directory", and it will tell you in which directory you are. If you are not where you want to be, you should exit the mail program by typing `x`, change to the directory you want to be in, and then reenter the mail program again by typing `mail`. After this, you can save the letter in question. You can delete letter #2 by typing

```
& d 2
```

you can delete more than one letters;

```
& d 2,3,5
```

will delete letters 2, 3, 5, and

```
& d 2-6
```

will delete letters 2, 3, 4, 5, and 6.

```
& q
```

will quit the mail program; quitting and exiting is different. If you exit the mail program by typing

```
& x
```

then you can read all your letters again; even if you have deleted some letters, they will be "undeleted". If you quit the mail program, the deleted letters will not be available again; the letters that have not been saved or deleted will usually be put in a file called `mbox` in the user's home directory (this action is dependent on how the system is set up; in fact, at least some accounts at the Atrium are not set up this way).

Sending email. In your home directory, there usually is a file called `.mailrc`; if there is not, you should create one by using the `vi` text editor (or some other editor you like). If you type

```
$ ls
```

you will not find this file listed; the files whose names start with a period `.` are *hidden files*, but you can easily list them by typing

```
$ ls -a
```

In any case, whether or not you have this file, look at it or create it by typing

```
$ vi .mailrc
```

You can add *aliases* for long mail addresses in this file; that is, you can add a line like this:

```
alias short user@long.email.address.at.some.university.edu
```

This will allow you to refer to the email address on the right by the abbreviated name (alias) short.

To send out email efficiently, also add the line

```
alias null "| ( cat > /dev/null )"
```

This line has to be typed exactly as is, including the spaces. The character `|` (vertical line) is called *pipe* in Unix, and is usually found on the same key as backslash `\`.[15]

For letters to be sent, create a directory called `mail_sent`, and write all letters to be sent out in this directory. A typical letter would be a file called `cmuser5`, the 5th letter sent to `cmuser`, with the following content (created by an editor such as `vi`):

---

[15] The casual user does need to know how this alias works; pipes are described in p. 105. For those with some knowledge of Unix and Unix mail, here is an explanation. An alias is needed that will cause no action; as the on-line manual of mail indicates, pipes are allowed as mail aliases. Here the effect of the pipe is to `cat` the content of the file to be mailed into /dev/null; doing this may waste a little computer time, but has no other perceptible effect. In Linux, the alias `""` works just as well (in fact, it works even better); on Suns, it does not work properly. It will send out the letter; but it will also complain:

```
"null": not a group
```

where *null* is the alias for `""`. It will then save the letter into a file called `dead.letter` (usually in the user's home directory); this totally defeats the purpose of the above alias, so on Suns the mail alias `""` is not usable. The on-line manual for aliases(5) gives a partial answer to this difference in behavior between Linux and the Sun's operating systems SunOS and Solaris.

Actually, on Suns the alias

```
alias null "| cat > /dev/null"
```

behaves better than the one given above. Another possible version is

```
alias null "| ( cat - > /dev/null )"
```

The way this and the alias given above misbehaves is by adding an addressee listed in the mail header such as

```
|@sending.machine.domain.name ( cat > /dev/null )
```

while the version without parentheses does not add this addressee on the Suns; the version without parentheses complains about `Unbalanced '>'` in Linux.

```
 1  ~t cmuser@math.some.school.edu
 2  ~s Your suggestion
 3  Dear Charley,           Sat Jun 21 21:06:51 EDT 1997
 4     I think I can make your suggestion work. I have
 5  to think about it a little more, however.
 6
 7                  Regards,
 8
 9                  J. F. Smith
10                  jsmith@math.myschool.edu
11                  http://www.math.myschool.edu/~jsmith/
12  ~.
13  Letter sent to Charles K. User.
```

The numbers on the left are, as usual, line numbers, and are not part of the file. Lines 1, 2, 12 contain so-called tilde escapes, i.e., lines whose first character is a tilde `~` (tildes occurring at positions other than the first characters of some line have no special effect). In line 1, `~t` adds the address on that line to the recipients of the letter. There can be more than one address on a line; addresses should be separated by a space. There can be several lines with the `~t` escape. In line 2, `~s` states the subject of the letter; the subject can be left empty, but an accurate description of the subject helps the recipient in deciding when to read the letter. In line 13, `~.` indicates the end of the letter; what is written after this point will not be transmitted. That is, line 13 is a comment for the sender of the letter to remind her what the letter is about (several lines of comments can be included). The recipient will get the part of the letter between lines 3–11, plus a mail header added by the mail program, describing where the letter came from, how it was transmitted, etc. There are other tilde escapes: `~c` stands for carbon copy; the addresses stated on lines introduced by this escape will get a copy of the letter. The only difference between recipients of carbon copies and recipients of the letter is what the mail header says about them; lines beginning with the escape `~b` introduce the addresses of blind carbon copy recipients; these addresses will receive a copy of the letter, but the mail header will not mention them as recipients.

To send out the above letter, you can type

```
$ mail null < cmuser5
```

Here `$` is the usual shell prompt (i.e., the regular prompt given by the computer; the reason it is called the shell prompt is that when the computer accepts commands it runs a program called shell), and is not typed by the user; `null` is the alias created above (in the `.mailrc` file), and `cmuser5` is the file containing the letter to be sent (just described). Instead of the above line, you can type

```
$ mail -v null < cmuser5
```

where the option `-v` is the verbose option; in this case the computer will write a detailed log on the screen as to what happened to the email you just sent; it might take a few seconds before this log appears, since it might take some time before the receiving computer is contacted. Actually, in Linux, one must type

```
$ mail -I null < cmuser5
```

or

```
$ mail -vI null < cmuser5
```

instead of the above lines. The option `-I` forces the so-called *interactive mode*. Actually, this term is misleading, since mail there is nothing interactive about the way we are sending email; the only significance of interactive mode is that, in Linux, interactive mode is needed for the mail program to accept the tilde escapes described above (this may not be true for all version of Linux; check the online manual pages for mail to find out).

To explain these commands, note that in these examples the mail command considers the argument called `null` the address where the mail is to be sent, and the argument `cmuser5` the letter to be sent out. Both arguments are required (though there are also other ways to use the mail program). However, the "address" `null` basically tells the mail program to send the letter to no one, while the letter `cmuser5` to be sent out contains information about the disposition of the mail (where to send it, what to send – that is, only parts of the file, not the tilde escapes and the comment). The letters that were sent out can be saved in a directory. Since each letter contains the addresses where it was sent, at a later day it can be ascertained to whom a certain letter was sent out.

23. Shell Scripts

More on sending email. Often one has to repeatedly execute a sequence of simple commands; rather than having to type the same thing at the terminal on repeated occasions, one can put these command in a file, and execute the file as what is called a *shell script*. As the first example, we describe how to deal with a flaw in the way the sending of email was organized in the preceding section: While all the information concerning an email letter, such as the content of the letter, who it was sent to, what the specified subject was, was contained in the letter itself, everything will go wrong if after carefully writing the letter one forgets to send it out. In fact, for a person liable to this kind of forgetfulness, it will be impossible to tell weeks later whether the last crucial step of sending out the letter was carried out.

One way to deal with this situation is never to send out a letter by typing the appropriate *mail* command directly. Instead, one can create the following file called *email*:

```
1  #! /bin/sh
2  mail -v "| ( cat > /dev/null )" < $1
3  cat $HOME/marks/sentmark >> $1
4  date >> $1
```

The[16] first step is to make this file executable by typing

[16] In Linux, the shell script

```
1  #! /bin/sh
2  mail -vI "" < $1
3  cat $HOME/marks/sentmark >> $1
4  date '+%a %b %-d %r %Z %Y' >> $1
```

```
$ chmod u+x email
```

We will say a few words about where this file should be located; for the moment it can reside in the directory where the email letters to be sent out are located (there are things that can go wrong here; we will discuss the execution search path below to indicate how to fix possible problems). To send out the email letter in the file *cmuser5*, you will be able to type

```
$ email cmuser5
```

but just not yet. One more thing will have to be taken care of, as you will see. Next we will explain how the *shell script* email works. In line 1, it is specified that the version of shell contained in the file `/bin/sh` is to be used. In Unix there are a number of command interpreters, called shells. The one in `/bin/sh` is called the Bourne shell, and it is mainly used for writing shell scripts, since there are newer shells that are preferred for interpreting commands entered on the command line. Such newer shell are the C shell at */bin/csh*, or its enhancement at */bin/tcsh*, the Korn shell at */bin/ksh*, and the Bourne-again shell at */bin/bash*. Not all Unix computers will have all these shells; C shell is used mainly on the command line, and is not recommended for shell scripts, while the Korn shell and the Bourne-again shell are used both on the command line and in shell script. There are yet other shells that we will not mention here; not all shells can be found on every Unix computer.

In general, computer languages can be *compiled* or they can be *interpreted*. The former means that a program written in the given language, called the *source program* is first translated in to the computer's on language, and a copy of this translated version is kept for later use; the translated (or compiled) program is usually called the object program. The latter means that no translated version of the program is being kept. The program is translated and executed at nearly the same time. The advantage of compilation is that it has to be done only once and the object program can be executed many times; on the other hand, interpreting a program is an easier task than compiling it.[17]

The way line 1 specifies the shell to be used is interesting: `#` is the shell comment character, and everything after `#` is ignored by the shell. Line 1 is a slight exception. Here the fact that `#` is immediately followed by the explanation point `!` tells indicates that after a space character the shell to be used will follow; the syntax of this first line has to be followed rigorously.

In line 2 the mail program is invoked in the way it was invoked in the preceding section. The first argument of `mail` is

---

works better. The disadvantage of using the string `"| ( cat > /dev/null )" < $1` is that, when using the reply option on some systems to respond to emails sent by the above shell script, the system tries to send a copy of the email to the address described by this string; this address is, however, a "nonsense" address.

In line 2 in this script, the option `-vI` is used for Linux instead of `-v`, as explained at the end of the preceding section. In line 4, the string after the word "date" formats the date for a 12 hour clock rather than the 24 hour clock provided without the use of this string.

[17]This description is only a rough outline of the difference between compilation and interpretation of a language. An interpreter reads the source program statement by statement, and causes the computer to perform the required action, whereas a compiler processes the whole source program and translated it into a different language that is easier for the computer to process. For example, the computer language Perl is a high-level language that is first processed by a compiler into a simpler language, and then and interpreter causes the execution of the program described in this simpler language. The output of the compiler is, however, not kept for later use.

```
"| ( cat > /dev/null )"
```

which is the way the `null` alias was specified in the file *.mailrc*; here we cannot use this alias, since aliases in *.mailrc* are not recognized in commands in shell scripts; it is easy enough to write out this alias once when writing the script *email*. The second argument in line 2 is `$1`; this refers to the first argument used when running the shell script; at present the shell script was run as the command

```
$ email cmuser5
```

The first argument here was `cmuser`. The arguments used on the command line when running the shell script can be referred to inside the shell script as `$0`, `$1`, `$2`, ... `$0` refers to argument number 0, which is the name of the command itself (`email` at present), and the others are the successive arguments used. Since only one argument was used, the arguments `$2`, `$3`, `$4`, ... will at present be undefined within the shell script (actually, their value will be the empty string). It is important to note that the aliases in *.mailrc* are not understood in line 2, those contained in the argument `$1` (`cmuser5` at present) are understood; so one can still use mail aliases in the letter to be sent out.

In line 3, the command `cat` appends the content of the file

```
$HOME/marks/sentmark
```

at the end of the file described `$1`, which, as we indicated above, at present means the file *cmuser5*. Here the symbol `>>` is obtained just by typing two `>` symbols. Furthermore, `$HOME` is a *shell variable* that refers to the users home directory. Technically, the name of this variable is `HOME`; by writing `$HOME` the value of this shell variable is indicated; in any case, `$HOME` will expand to something like

```
/home/jsmith
```

or whatever is used as the home directory of a user having, say, *jsmith* as login name. The directory `$HOME/marks` that is, say `/home/jsmith/marks` has to be created, and in this directory a file called *sentmark* needs to be created with the single line

```
Letter sent on date:
```

This was the missing step we referred to above, and after completing this step, the shell script *email* will work properly. It can now be seen that the effect of the command *cat* in line 3 is to write the line

```
Letter sent on date:
```

at the end of the file *cmuser5*. Finally, in line 4, the command *date* generates the current time and date, and it writes it at the end of the argument referred to as `$1`, i.e., the file *cmuser5*.

To sum up, the above shell script will mail out the file *cmuser5* and append information at the end about the act of mailing it out. In the end, the file *cmuser5* described in the preceding section will look something like this:

```
1  ~t cmuser@math.some.school.edu
2  ~s Your suggestion
3  Dear Charley,           Sat Jun 21 21:06:51 EDT 1997
4     I think I can make your suggestion work.  I have
5  to think about it a little more, however.
6
7                   Regards,
8
9                   J. F. Smith
```

```
10                    jsmith@math.myschool.edu
11                    http://www.math.myschool.edu/~jsmith
12  ~.
13  Letter sent to Charles K. User.
14  Letter sent out on date:
15  Sat Jun 21 21:19:23 EDT 1997
```

As can be seen, lines 14 and 15 were added at the end of the file by the shell script *email* that we were discussing. It might look better if lines 14 and 15 were written in a single line, rather than in two lines, as they are now; in this case the word "date" and the colon afterwards should also be omitted. To accomplish this, one needs to create the file `$HOME/marks/sentmark` with a slightly different content:

```
Letter sent on␣
```

Aside from the missing colon, first, a space character needs to be added after the text (we indicated this space as ␣; we did not indicate the other space characters in the line, whose presence is clearly seen by the gap left between words). Second, there must not be an end-of-line character at the end of the line. If this can be accomplished, then line 4 in the the above shell script will not put the date on a separate line. Since the editor *vi* automatically writes the end-of-line character at the end of the line, *vi* does not seem to be suitable for creating this file. On the other hand, the file can easily be creating by compiling and running the following Pascal program:

```
1  program writesentmark(sentmark);
2  var sentmark : text;
3  begin
4    rewrite(sentmark);
5    write(sentmark,'Letter sent out on ')
6  end.
```

This program does not use the files *input* and *output*, so these are not indicated. On the other hand, it will write to a file called *sentmark*, so this is indicated in line 1. In line 2, *sentmark* is declared a text file. In line 4, the meaning of the *rewrite* statement is to destroy the previous content of the file *sentmark* and to open it for writing. In line 5, the *write* statement is used to write into this file. When the Pascal statement *write* writes to a file rather than to the screen, this file has to be identified as the first argument of the *write* statement. As we mentioned earlier, the *write* statement does not add an end-of-line character at the end; so the file written by this program will be exactly as desired.

If the shell script *email* is now run with the modified version of the *sentmark* file, instead of lines 14–15 above we obtain a single line:

```
Letter sent out on 15  Sat Jun 21 21:19:23 EDT 1997
```

The next question to be solved is how to create the file *cmuser5*, i.e., the letter that was sent out. Some parts of this file have to be typed at the time the letter is written, but there are parts of it that are common to all letters sent out, so there should be a way to create this common part of the letter once and for all. In fact, this is accomplished by the following shell script we call *letter*:

```
1  #! /bin/sh
2  cat $HOME/marks/regards >> $1
```

```
 3  ed $1<<%
 4  0a
 5  Dear $2␣␣␣␣␣␣␣␣␣␣␣
 6  .
 7  1r !date
 8  1,2j
 9  \$a
10  ~.
11  .
12  0a
13  ~t
14  ~s
15  .
16  w
17  q
18  %
19  vi $1
```

Here, in line 5 we indicated the space characters at the end of the line (there are 11 spaces at the end of the line); at other places the space characters were not indicated, since their presence is clear from the gaps left in the text. After making this file executable with the *chmod* command, one can create the skeleton of the file *cmuser5* by typing

```
$ letter cmuser5 Charley,
```

If the file *cmuser5* does not exist, then the the *vi* editor will open up with the following content:

```
1  ~t
2  ~s
3  Dear Charley,           Sat Jun 21 21:06:51 EDT 1997
4
5                   J. F. Smith
6                   jsmith@math.myschool.edu
7                   http://www.math.myschool.edu/~jsmith
8  ~.
```

the date displayed will, of course, be the actual date. If the file *cmuser5* already exists and has the following content

```
1     I think I can make your suggestion work. I have
2  to think about it a little more, however.
```

then the content displayed by *vi* after running the above shell script will be

```
1  ~t
2  ~s
3  Dear Charley,           Sat Jun 21 21:06:51 EDT 1997
4     I think I can make your suggestion work. I have
5  to think about it a little more, however.
6
7                   Regards,
8
```

```
 9                    J. F. Smith
10                    jsmith@math.myschool.edu
11                    http://www.math.myschool.edu/~jsmith
12  ~.
```

To finish the letter, the address in line 1 and the subject in line 2 has to be added; and a remark may be added after line 12. Next we will explain how the above shell script produces these results. The line numbers below will refer to the line numbers in this script, i.e., in the file *letter* above. Line 1 specifies that the Bourne shell is to be used. Line 2 adds the content of the file `$HOME/marks/regards` at the end of the file referred to as `$1`, i.e., the first command line argument of the shell script being discussed; this argument at present is *cmuser5*; if this latter file is empty, then its content will be identical to that of the former file; if it is not, the new file will have things added at the end; here, as mentioned above, `$HOME` refers to the user's home directory. In order for this command to work, the file `$HOME/marks/regards` will have been created with the following content:

```
1
2                     Regards,
3
4                     J. F. Smith
5                     jsmith@math.myschool.edu
6                     http://www.math.myschool.edu/~jsmith
```

The name of this file *regards*, refers to the way this letter is signed. Other "signature" files called *best*, *sincerely*, *truly*, etc. can also be created.

Lines 3–18 contains what is called a *here document*. The commands in these lines are not commands to the shell; they are commands to a text editor called *ed*; *ed* is a basic line editor in Unix (its commands are somewhat similar to some of the *vi* commands in command and command-line mode; in fact, *vi* is based on the line editor *ex*, which is a successor of *ed*). The above shell script could have also been written by using `ex` instead of `ed`, but `ed` appears to be more widely available than `ex`.

Line 3 specifies that the subject of this here document is the program program `ed` with the argument `$1`; this latter refers to the first argument on the command line running the script `letter`, i.e., `$1` refers to the file `cmuser5`. That is, lines 4–17 work as if they were issued on the command line, after having typed

```
$ ed cmuser5
```

on the command line (with important qualifications concerning lines 5 and 9, to be mentioned below). The marks `%` in lines 3 and 18 indicate the beginning and the end of the here document. Other marks (consisting of a single character or whole words) could be used;[18] an important point, however, is, that the mark used cannot occur inside the here document (because as soon as that mark occurs, the here document ends). The symbol `<<` (obtained by typing two `<` signs) indicates that what follows is a here document, and the symbol following it will be the delimiter of the here document. In line 4, the ed command `0a` indicates that the text following in the next few line(s) are to be added at line 0 (i.e., at the beginning) of the file being edited; the period by itself in line 6 marks the end of the text to be added; i.e., at

[18] A here document where a whole work used as a mark is given in p. 101.

present the text to be added is only the text in line 5. Here `$2` is to be interpreted as the second argument of the shell command

```
$ letter cmuser5 Charley,
```

i.e., `Charley,` (including the comma). You may want to address this person as `Charley:` or as `Professor Cmuser:`; accordingly, you may want to type the command lines

```
$ letter cmuser5 Charley:
$ letter cmuser5 'Professor Cmuser:'
```

In the last example, the quote character will be stripped by the shell program; i.e., you will properly obtain the line

```
Professor Cmuser:
```

rather than

```
'Professor Cmuser:'
```

Instead of single quotes, double quotes (shift–`'`) can also be used, i.e., the command line could have been typed as

```
$ letter cmuser5 "Professor Cmuser:"
```

In line 7, the output of the *date* command is placed into the file being edited after line 1. In this line, 1 is the line number of the file being edited, `r` means read into this file, the exclamation point `!` indicated that what is going to be read is the output of a Unix command, and `date` is the Unix command in question. The output of `date` is the current date and time. Line 8 says that lines 1 and 2 have to be joined into a single line.

In line 9, the letter `a` indicates that the text in the next few lines will have to be added; again, only one line, line 10 is added, since the single period in line 11 indicates the end of the lines to be added. In line 9, `$` refers to the last line of the file being edited; that is the text in line 10 has to be added after the last line. In *ed*, as in *vi* command-line mode, `$` refers to the last line of the file being edited. The backslash `\` is the shell quote character; this means, that the dollar sign `$` in line 9 is quoted as far as the shell is concerned; the shell strips the quote character `\`, and *ed* will receive only `$a` instead of the present form of line 9. If we had written `$a` for line 9, that is, if it had been written without the quote character, then the shell would have interpreted `$a` as indicating the value of a shell variable; if this variable is not defined (as is likely to be the case at present), this value will be the empty string, and *ed* would receive the empty string in line 9 instead of `$a`. Lines 12–14 add the content of lines 13–14 at the beginning of the file being edited (since these are written in separate lines, two separate lines will be added). The letter `w` instructs *ed* to write the result of the editing out to the file being edited (this works in much the same way as the command `w` works in *vi* in command-line mode). Finally, `q` in line 18 quits *ed* and, the end of the here document is marked in line 18. Line 19 is now a command to be interpreted by the shell; it will open up the file in *vi* for further editing.

We already commented between the similarities of *ed* and *vi*. To mention further similarities, versions of the command `a` in lines 4, 9, and 12 in *vi* are somewhat analogous to pressing `a` in *vi* command mode, which causes a change to insert mode (see p. 13), and the period in lines 6, 11, and 15 is similar to pressing the escape key in *vi* insert mode, which causes a change to command mode (see p. 13).

Locations for executable files. When executing a program, one can state the name of the program with the full directory path to the program's name. For

example, to start *vi*, you can type

```
$ /bin/vi
```

Actually, on some systems, *vi* is located at a different place, so you may have to type

```
$ /usr/ucb/vi
```

However, as you saw above, you can start up *vi* simply by typing

```
$ vi
```

The reason is that the system searches at certain locations for executable files, and it will execute the file at the first location it finds it. The locations it searches are determined by the shell variable PATH (in Korn shell, Bourne shell, or Bourne-again shell; in C shell, the variable may be called `path` – the machines at the Atrium run Korn shell as its default shell, so most of our comments will apply to the Korn shell). You can find out its value by typing

```
$ echo $PATH
```

The first dollar sign in this line is the shell prompt; the second one is used to get the value of the shell variable. The response you get may be something like

```
/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbi
n:/usr/bin/X11:/usr/X11R6/bin:/root/bin:/usr/local/sbin
```

The directory name *bin* is short for binary, and it is customary to store executable files in a directory called *bin*, even though many of these files are not binary files (they may be text files, such as shell scripts). Note that the above display is a single line, but it wraps around two lines because it is too long; that is, there are no end-of-line characters in this display. The colons are used to separate locations. When you type the name of an executable file, the system will search in turn the directories `/bin`, `/usr/bin`, `/usr/sbin`,..., in this order, until it finds the name you typed. The value of the `PATH` variable is called the *search path*, or execution search path, since it contains a list where executable programs are searched.

If the command you typed is something that is not found, there will be a message telling you this; for example if you want to list the files in a directory and you type

```
$ list
```

the answer most likely will be

```
ksh: list: not found
```

where `ksh` refers to the shell running (Korn shell); the command to list the files is `ls`, and not `list`.

You may want to add other directories to be searched for executable files. You can do this by adding an entry in your `.profile` file (in Korn shell), located in your home directory; note that the dot at the beginning of the file name indicates that this is one of those hidden files; by typing `ls`, the name of this file will not be listed – you need to type `ls -a`. Korn shell uses `.profile`, C-shell uses `.login`, and the Bourne-again shell uses `.bash_profile`. Anyway, if you are using the Korn shell you may make the following entry in your `.profile` file:

```
PATH=$PATH:$HOME/bin:.
export PATH
```

The second line may already be there; in that case, you do not need to add it, but you have to make sure that the first line is added before the occurrence of the second line. The first line modifies the value of the PATH variable; it adds the directory `.`, which refers to the present working directory, and the `bin` directory in your home

directory in the search path. The second line makes this change effective.[19] You only need to make these changes if the PATH variable does not contain these directories.

While it is convenient to have the present working directory . in the execution search path, it is somewhat of a security risk; this is discussed below (see p. 98). You may place some shell scripts that you wrote into the `bin` directory in your home directory; for example, this is where the scripts `email` and `letter` discussed above belong.

Occasionally, different programs with the same name may be found at different locations. This may cause confusion. It may happen that you are used to running some program called `useful_program`, and one day the behavior of this program changes; possibly, you are running a different program by the same name. You may check which program program you are running by typing

```
$ which useful_program
```

(cf. p. 31 for the `which` command). The answer may be something like

```
/usr/local/bin/useful_program
```

The present working directory in the search path. Sometimes the present working directory . is included in the `PATH`. In this case, the command

```
$ echo $PATH
```

may produce the reply

```
/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbi
n:/usr/bin/X11:/usr/X11R6/bin:/root/bin:/usr/local/sbin:.
```

Here the dot . at the end indicates the present working directory. In this case, if you have a program called, say `my_program` in your in your present working directory, you can run it simply by typing

```
$ my_program
```

Even if . is not listed in the `PATH` variable, you can run this program in your present working directory by typing

```
$ ./my_program
```

This works since, rather than just giving the name of a program, it gives a path to it, as the file with the name `my_program` in the present working directory (referred to as .).

However, there are security considerations that make in inadvisable to put your working directory in the seach path, as discussed at the Web site

```
http://www.chicagotribune.com/tech/developer/article/0,2669,2-4
4389,FF.html
```

(this is a single line, scrolled over). Basically the reason is that an intruder can put executable programs in your some of your directory, and you can run these inadvertently. Imagine that an intruder puts a script called `ls` into a directory of yours that will erase all your files in that directory when run. So when you type

```
$ ls
```

in that directory, instead of listing the content of the directory, all the files got erased. This would only happen, though, if . precedes the directory `bin` (the directory that usually contains the program `ls`) in the execution path. For this reason, even if you include . the execution path, it must be the last item there.

Clearing Netscape's cache. At the end of Section 4, we discussed that the simple command

[19] It makes the change available to subshells.

```
$ rm -rf *
```

when executed in the directory `.netscape/cache` can be used to clear Netscape's cache; on the other hand, executed in the wrong directory, this command can cause disaster. It is much safer to use a single line shell script called, say, `netscape-clear`, containing the single line

```
rm -rf $HOME/.netscape/cache/*
```

to perform this task.

Compiling Pascal programs. A simple shell script can save typing time when compiling Pascal programs:

```
1  #! /bin/sh
2  pc -s $1.p -o $1.comp > errors
3  cat errors
```

Calling this script `compile`, one can compile a Pascal program called `example.pc` by typing

```
$ compile example
```

This will produce an executable file example.comp provided the compilation is successful, and the possible errors that occur during the compilation are first written in the file called `errors` and then written on the screen. There are clear advantages of writing the errors into a file, since they are preserved for later viewing, whereas the text displayed on the screen is not.

Putting files on the Web. Suppose you worked on a Pascal program called, say, `example.p`, and you want to display it on your Web page. You can do this by copying it into your `public_html` directory and setting its permission code to 644 (see Section 5, p. 23). Rather than doing this manually, you can do this by the following shell script, called, say, `putonweb`:

```
1  #! /bin/sh
2  cp -f $1 $HOME/public_html
3  cd $HOME/public_html
4  chmod 644 $1
```

To use this shell script to put the Pascal program `example.p` on the Web, you need to change to the directory where this file is located, and then type

```
$ putonweb example.p
```

There are two things to note about this shell script. If there is already a file called `example.p` in the directory `public_html`, the above command will destroy it (that is, replace it with the new version of `example.p`; this is because we used the `-f` (force, i.e., do the copying even if it is destructive) option of the copy command `cp`. Second, in order for this to work, your Web page has to have been set up already, as explained in Section 5.
If the address of your Web site is, say,

`http://www.website.someuniv.edu/~cmuser/`

then you can find the above file at the address

`http://www.website.someuniv.edu/~cmuser/example.p`

Note that even though the file `example.p` is not written in HTML, it can still be displayed on the Web. In fact, what is displayed is the file itself, whereas in case of an HTML file, what is displayed is what is meant to be seen and not the file itself;

in fact, the latter contains formating commands describing what is to be seen. If you want to see an HTML file itself on the Web by using Netscape, you need first to press on the word *View* in the top row of the Netscape window with the left mouse button, and then to press on phrase *Page Source* among the choices displayed, again with the left mouse button.

CLI versus GUI. Currently there is much debate whether computers should have *Command Line Interface (CLI)* or *Graphical User Interface (GUI)* (pronounced *goo-ee*). GUI allows non-dedicated users to quickly learn how to do sophisticated things on the computer. CLI allows them to automate repetitive tasks by using shell scripts; at least at present, shell scripts do not allow you to automate sequences of mouse-clicks used to work with GUI.

If a network administrator has to add 25 new users to a Unix network, he/she can write a shell script that can do this task fast, without human intervention. In Windows NT, adding a user may take about two minutes, with a few mouse clicks. Adding twenty five users, however, cannot be speeded up, the administrator has to go through the same sequence of mouse clicks 25 times (this may change in Windows 2000, the next edition of Windows NT).

Current trends for making everything GUI in certain operating systems are certainly not ideal. It would help at least the experienced users if many GUI commands would have well-documented CLI counterparts. For example, it would be nice if Netscape could be invoked from the command line to collect the ingredients of a Web page without actually displaying it, so a number of Web pages could be collected for later perusal at a convenient time, without having to wait for slow downloads.

24. More Shell Scripts

Using Netscape to view HTML files. *Warning:* The first few examples presented in this section have security problems on account of using temporary files. In a later example, we will describe a more secure way of handling temporary files, and all these script should be rewritten in conformity with the remarks made on p. 113 on account of the *symlink attack* (see there).

Suppose you have an HTML file in the directory

```
$HOME/html_files/example.html
```

where `$HOME` refers to your home directory (see p. 95 above). If your home directory is

```
/home/jsmith
```

then you can view the above file by going to a Netscape window, clicking on *File*, then on *Open Location*, and then typing

```
file:/home/jsmith/html_files
```

Alternatively, you can create the following shell script called `viewhtml` as follows:

```
 1  #! /bin/sh
 2  NN=file:$HOME/html_files/$1
 3  rm -f junknetscape >& /dev/null
 4  touch junknetscape
 5  ed junknetscape<<heremark
 6  0a
 7  netscape -remote 'openURL($NN,new-window)'
 8  .
 9  w
10  q
11  heremark
12  . ./junknetscape
13  rm -f junknetscape
```

Assuming that Netscape is already running, you can run this script by typing

```
$ viewhtml example.html
```

on the command line, so as to view the above file.

What makes the above script complicated is that the command

```
$ netscape -remote \
> 'openURL(file:$HOME/html_files/example.html,new-window)'
```

(here we broke up the command in two lines, as explained on p. 34) does not work, since the `netscape` command does not understand the shell variable `$HOME`; instead, one needs to specify the full path, as in the command

```
$  netscape -remote \
> 'openURL(file:/home/jsmith/html_files/example.html,new-window)'
```

What the above shell script does is to use the text editor `ed` to write the latter command above into the file `junknetscape` and then run the command in this file. A filename such as `junknetscape` is used deliberately, so as to avoid conflicts with names of useful files.

In what follows we will explain in detail how the above script works. Line 1, as usual, specifies the shell to be used. Line 2 introduces a new shell variable called `NN` (it is customary, but not necessary, to use capital letters for shell variables; Unix is case, sensitive, so, while one can use the string `nn` for a shell variable, this would be a variable different from `NN`), and assigns it as value the URL to be used in the `netscape` command (mentioned in line 7). In this value,

```
file:$HOME/html_files/$1
```

`$HOME` and `$1` are themselves shell variables. The value of the former is the home directory of the user,

```
/home/jsmith
```

in our present example, and the value of the second one is the first argument of the shell script we are in the process of describing. That is, if the above shell script is used in the command

```
$ viewhtml example.html
```

then `example.html` is the value of `$1` – `viewhtml` itself is the name of the shell script. In line 3, the file `junknetscape` is removed if it exists, since a leftover file by the name `junknetscape` could cause trouble. The option `-f` of the command `rm`

forces removal, so the user is not prompted if he or she indeed wants to remove the file. The string `>&` is used for output redirection. Normally, the command would write its output to the screen, but this the symbol `>&` makes this output go to the null device /dev/null instead of the screen (the null device is a sort of bottomless garbage can – whatever is put in it just disappears.

Unix has two main kind of outputs: *standard output* and *standard error output*. The two kind of outputs are usually indistinguishable from each other, since both are simply written to the screen. Error messages of Unix commands are usually written to the standard error output. For example, if there is no file called `junknetscape`, the command

```
$ rm junknetscape
```

writes the following message to the screen:

```
rm: cannot remove ‘junknetscape’: No such file or directory
```

writes on the screen (in actual fact, to the standard error output). Such a message would be confusing when running the above shell script. The command

```
$ rm junknetscape >& /dev/null
```

writes nothing onto the screen, since the standard error output is discarded (redirected to `/dev/null`). The symbol `>` is used the redirect the standard output, and the one `>&`, to redirect the standard error output. Unix also has a *standard input*; the standard input is usually what is typed at the keyboard.

Line 4 creates an empty file called `junknetscape` and lines 5–11 is a *here document* (described in p. 95) using the editor `ed` to edit the file `junknetscape`. There we used the percent symbol `%` to indicate the beginning and the end of the here document; here we used the string `heremark` (in lines 5 and 11) instead (any string can be used, but the percent sign is often favored, since it usually does not have other uses; we could have used the percent sign here, but we wanted to show that there are other possibilities). That is, lines 6–10 work as if commands issued to the editor `ed` after typing the command

```
$ ed junknetscape
```

The `ed` command `0a` in line 6 indicates that the next line is to be added after line 0 of the file being edited, namely, of the file `junknetscape`. In this line, the shell variable `$NN` is evaluated to its value assigned on line 2, i.e., to the whole line will become

```
$ netscape -remote 'openURL(file:/home/jsmith/html_files/examp
le.html,new-window)'
```

Note that this is a single line of the file, but since it is too long to display it in a single line, the end of the line is scrolled into the next one (but there is no *end-of-line* character in this text except at the very end).

The lone period in line 8 indicates the end of things to be added; `w` in line 9 instructs the editor `ed` to write the result of the editing to the file being edited (`junknetscape` in the present case), `q` in line 10 makes `ed quit`, and the percent sign `%` in line 11 signifies the end of the here document.

The meaning of the first period in line 12 is that the commands contained in the file mentioned next in the line are to be executed as if they were written out explicitly. The file `junknetscape` is referred to as `./junknetscape`; here the period refers to the present working directory, so the two ways indicated describe the same file (but in some situations, using the latter way has certain advantages).[20] That is,

[20]If the present working directory is not listed in the `PATH` variable, then in some Unix systems

the Netscape command to display the file `example.html` is executed at this point. The use of the first period in line 12 to take commands from the file mentioned in this line is called *sourcing* the file; more about this is explained on p. 122. The reason for sourcing this file rather than using it as a shell script is that this way we do not have to add execute permission to the file. In line 13, the file `junknetscape` is removed, as it is no longer needed. Executing a temporary file in a shell script has some hazards, and the above script should be considered more as a script used to learn how to write shell scripts than an example for a useful script. A more secure way of handling temporary files is to put them into a single temporary directory with appropriately chosen permission codes. This is discussed on p. 113.

Next we will present a slightly improved version of the above shell script.

Unique names for temporary files in shell scripts. Unix is a multitasking system. You can simultaneously have several login sessions on the same machine; this is especially easy to do if you the machine is networked, since you can log into the same machine from different points on the network. Or you and a close friend may use your account at the same time.

This can cause problems with the shell script `viewhtml` above. If two instances of `viewhtml` are run at the same time (and from the same working directory), the script can get confused, since both of them try use the same filename `junknetscape`. Fortunately, it is fairly easy to resolve this problem.

In Unix, the instance of a running program is called a *process*. We use the word instance, since in Unix, the same program can be run simultaneously several times, and each of these runs, or instances, is called a process. Each process is assigned a unique number called *process-id* or *PID*. In a shell script, the shell variable `$$` has the value of the the process-id of the shell current instance of running the shell script. This can be used to generate unique filenames.[21] Instead of the filename `junknetscape` in the above shell script `viewhtml` we can use the filename `junknetscape$$`. If we run `viewhtml`, and the process-id of the current instance `viewhtml` is, say, 1364, then this filename will be resolved as `junknetscape1364` – and so it will be unique to the current instance of `viewhtml`. Here is the improved version of `viewhmtl`, called, say, `viewhtml0`, using this idea:

```
1  #! /bin/sh
2  NN=file:$HOME/html_files/$1
3  rm -f junknetscape$$ >& /dev/null
4  touch junknetscape$$
5  ed junknetscape$$<<%
6  0a
7  netscape -remote 'openURL($NN,new-window)'
8  .
9  w
```

---

the command

```
$ . ./junknetscape
```

will be executed, while the command

```
$ . junknetscape
```

will not – see p. 98.

[21] There are safer ways to make unique filenames; in Linux, one can do this by using the `mktemp` program. See the online manual pages for `mktemp` in Linux to see why this is safer than using the process-id. In Solaris, mktemp is available only as a C library function.

```
10  q
11  %
12  . ./junknetscape$$
13  rm -f junknetscape$$
```

This identical to the script above, except that we used the filename `junknetscape$$` instead of `junknetscape$$`.

Killing Netscape. On occasion one encounters a Web site that was not prepared to Netscape's liking, and it causes Netscape to crash (in which case the Netscape window suddenly disappears without any explanation) or freeze (in which case the Netscape window stays but no amount of clicking can make it do anything, and Netscape becomes totally nonresponsive). Even if the Netscape window disappears, the Netscape program, or process, as it is called, is probably still around.

As we mentioned above, in Unix, the instance of a running program is called a process (seep. 103). To find out what processes are running, one can use the command `ps`, with various options. For example, typing

```
$ ps x
```

where the option `x` explained in the Unix online manual pages[22] as "selects all processes" might produce the following list:

```
  PID TTY          TIME CMD
    1 ?        00:00:04 init
    2 ?        00:00:35 kflushd
.......................................
  425 ?        00:00:00 nfsd
  426 ?        00:00:00 nfsd
  427 ?        00:00:00 nfsd
.......................................
  644 tty4     00:00:00 mingetty
.......................................
23258 ?        00:00:00 xterm
 6421 pts/13   00:00:39 netscape-commun
 6431 pts/13   00:00:00 netscape-commun
 6531 pts/12   00:00:00 sleep
 6534 pts/12   00:00:00 ps
```

The list is quite long; in fact, it may contain more than one hundred lines, so we did not present the whole list; the lines with a series of dots each represent a number of lines that we omitted. In fact, if you type the above command, you will most likely not see the output properly, because it will run past you in the output window two quickly before you can read it, and you will only be able to see the last twenty of so lines of the output. One way to deal with this problem is to *pipe* it through the `more` command:

[22] This is a so-called BSD-style option, where one types `ps x` rather than `ps -x`. This works in Linux. Unfortunately, different versions of Unix use different forms of the `ps` command. in Solaris, one needs to use `ps -A`, `ps -a`, or `ps -e`; you need to consult the online manual pages to find out the exact form needed in various forms of Unix. You can get the online manual pages on the command `ps` by typing

```
$ man ps
```

– see p. 9 for `man` command.

```
$ ps x | more
```

To explain how this work, we first have to discuss the `more` command. Given an ascii file called, say, `textfile`, you can view its content by typing

```
$ more textfile
```

(you can also use an editor, such as `vi`, to view the content of this file). Upon typing this command, you can see the beginning part of the file `textfile`, in fact, exactly as much of it as fits your window. By pressing the space bar on the keyboard, you can see the next windowful. By pressing the return key, your view of the file will advance by a single line. By typing `b`, you can go back by one windowful (this may not work on older systems). Once you get to the end of file, you get back to the Unix command line. You can get back to the command line earlier by typing `q`.

The symbol `|` we used the `more` command above is called a *pipe*. Its meaning is to take the output of the command on the left, and use it as the input on the right – more precisely, the standard output (see pp. 57 and 102) of the first command is taken as the argument (input file) of the second command. That is, the command

```
$ ps x | more
```

is equivalent to the two commands

```
$ ps x > tempfile
$ more tempfile
```

(except that the version with the pipe does not leave the temporary file called `tempfile` behind). Here, in the first line, the greater-than sign `>` redirects the standard output into the file `tempfile`, and in the second line `more` uses this file as its argument.

The command

```
$ grep netscape tempfile
```

writes those line of the file `tempfile` to the screen that contain the word `netscape`. Thus the command

```
$ ps x | grep tempfile
```

will give only part of the output of the command

```
$ ps x
```

described above; that is, we will get only those lines of the output that contain the word `netscape`:

```
6421 pts/13   00:00:39 netscape-commun
6431 pts/13   00:00:00 netscape-commun
```

After these preliminaries, we will discuss a script that can be used to kill all Netscape processes. We will chose the name `killns` for this script; that is, the script can be invoked by typing

```
$ killns
```

without arguments. We will comment on the name `killns` below. Here is the script (in the next subsection we will discuss how a much shorter script can be written to do the same task):

```
1  #! /bin/sh
2  ps x | grep netscape > junkkill0$$
3  # cat junkkill0$$
4  awk '{ print $1 }' junkkill0$$ > junkkill$$
5  # cat junkkill$$
6  ed junkkill$$<<% 2> /dev/null
```

```
 7  0a
 8  kill -s 9
 9  .
10  1,\$-1s/\$/ \\\/
11  w
12  q
13  %
14  # cat junkkill$$
15  . ./junkkill$$ 2> /dev/null
16  rm -rf $HOME/.netscape/cache/*
17  rm -f $HOME/.netscape/lock
18  rm -f junkkill$$ junkkill0$$
```

First notice that the filenames `junkkill$$` and `junkkill0$$` are unique names for temporary files, as discussed above (see p. 103). The pound sign `#` is the shell comment character. Its role in line 1 has already been discussed (cf. p. 91). Its role at the beginning of lines 3, 5, and 14 is to make these lines ignored. These lines are not needed in the shell script, but they are useful for debugging: that is, when writing the script, these lines are initially not commented out (that is, the pound sign `#` is not put at the beginnings of these lines initially), since, when writing the program, it is useful to know what is written in the files `junkkill$$` and `junkkill$$0`.

For example, if one runs this script while Netscape is running without lines 3, 5, and 14 commented out (i.e., with the pound signs `#` removed), in addition to its killing Netscape, one might get the following output (the line numbers at the beginning are not part of the output; we added them so that we can discuss the contents of this output):

```
1 10522 pts/16   00:00:01 netscape-commun
2 10532 pts/16   00:00:00 netscape-commun
3 10522
4 10532
5 kill -s 9 \
6 10522 \
7 10532
```

Lines 1–2 in this output are produced by line 3 of the script `killns`, lines 3–5 are produced by line 3–4 by line 5, and lines 5–7 by lines 5–7. Consideration of this output will be helpful in understanding how the script works.

Line 2 of the script writes the first two lines of the output; the command would write to the standard output (screen), but the standard output is redirected to the file `junkkill0$$` (so nothing gets written to the screen by line 2 of the script). This output is similar to the output of the command

```
$ ps x | grep tempfile
```

discussed above, but some things are different. Lines 1 and 2 in the output are each divided into four fields; the fields are separated by one or more spaces. What these fields mean can be seen from the headings produced by the command

```
$ ps x
```

discussed above (see p. 104). As seen from there, first field is the process-id (PID), the second one is TTY, which is short for teletype, but nowadays it refers to the

terminal (that is to the terminal where the process has been initiated the various names such as `tty4`, `pts/12`, etc., are names for various terminal names in the output given on p. 104, and the question mark indicates that the process has not been initiated at a terminal), the third field is the time the process has been active for, and the fourth field is the command name associated with the process.

Note that the items in the first and third field are different in the present in lines 1–2 in the present output from the items involving `netscape` in the earlier output – this is because these commands were issued at different times, resulting in different process-id's and different activity times.

Line 4 of the script involves the `awk` command, whose name is made of the initials of its designers, Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. The command `awk` is very powerful, and books have been written about it; in fact, `awk` is really a programming language rather than just a Unix command. Here we have to confine ourselves to explain its specific use given in line 4 of the script. The command

```
$ awk '{ print $1 }' junkkill0$$ > junkkill$$
```

(we added the `$` on the left as a prompt, to explain how this command works when issued on the command line) has as its (first and only) argument as the file `junkkill0$$`. The `awk` command goes through this file line-by-line, and with each line it takes the desired action. The desired action here is described by the string

```
'{ print $1 }'
```

(the syntax must be exactly as given above, with the right-quote signs included). The action is to take the first field of each line and write it to the standard output. In the above command, the standard output is redirected to the file `junkkill$$`. Thus, the content of the file `junkkill0$$` being

```
10522 pts/16   00:00:01 netscape-commun
10532 pts/16   00:00:00 netscape-commun
```

when the `awk` command is issued (as seen in the output of the script `viewhtml` described above), as a result of this command, the file `junkkill$$` will contain the first field of each of these lines:

```
10522
10532
```

that is, the process-id's associated by the above processes.[23]

Lines 6–13 of the above script is a *here document* (see p. 95). In line 6, the string

```
2> /dev/null
```

is used to redirect the standard error output of this command to `/dev/null` (which is nothing at all; the purpose of this is to keep this output from cluttering the screen). Above, we used the symbol `>&` to redirect the standard error output (see p. 102); here we used the symbol `2>`. Both works equally well. The explanation for the latter symbol is that the standard error output is called *output stream 2* (the standard output is called *output stream 1* – and `1>` instead of `>` can be used to redirect the standard output).

In the here document in lines 6–13 the editor `ed` (see 95) is used to edit the file `junkkill$$`. Line 6 of the script opens the editor at line 0 of the file for insertion. That is line 8 of the script gets inserted after line 0 (i.e., as a new line 1, or a new

[23]The Unix command `pidof` can be used to find out the process-id of a process directly. The command is available for example in Red Hat Linux, but it is not available in Solaris.

first line) of the file `junkkill$$` (there is no line 0; line 0 is a way to refer to the beginning of a file). The sole period `.` in line 9 indicates the end of the inserted text (that is, the inserted text consists solely of line 9 of the script). Line 10 edits the range `1,\$-1s/` of the file by using the `substitute command` of `ed` – this command works much the same way as the substitute command of `vi` (see p. 14). Here `1` of course refers to line 1 of the file, and `$` its last line; `$-1` refers to the line before the last one. Finally, instead of `$-1` we wrote `\$-1`; the shell quote character `\` is needed, otherwise the dollar sign would be interpreted as introducing the value of a shell variable (as in `$HOME` being the value of the shell variable `HOME`). The shell strips away this quote character, and when the editor `ed` processes line 10 of the script, it sees the range `1,$-1` rather than `1-\$-1`.

The editing done in this range is the replace the string `\$` with the string `␣\\\` (here ␣ indicates a space character; usually we indicate a space character just by leaving a space for it – but in the present case we need to write the symbol ␣ for the space character to be clear). In the first string, the shell quote character `\` is stripped away by the shell as above, so the first string is actually `$`, which just as in `vi`, refers to the end of the line. That is, the second string gets added at the end of each line of the file `junkkill$$` except for the last line. In the second string, `␣\\\` there are three backslash characters `\`. The first one of this is stripped away by the shell as a quote character, the second one is stripped away by the editor `ed`, again as a quote character. Only one backslash `\` remains in the end. That is, what is added at the end of each line of the file `junkkill$$` except for the last one is the string `␣\`. That is, after this action the file `junkkill$$` looks as as lines 5–7 of the script `killns` being discussed:

```
kill -s 9 \
10522 \
10532
```

In fact, the file `junkkill$$` does not yet look like this – so far, all the editing took place temporary file created by the editor `ed`, and only the command `w` in line 11 writes this into the file `junkkill$$`, similarly to the way the command `:w` works in `vi` (see p. 15). The command `q` in line 12 quits the editor, and the percent sign `%` in line 13 terminates the `here document`.

Line 15 of the script executes the command in the file `junkkill$$` (i.e. sources – cf. p. 122). The command executed is the same as

```
$ kill -s 9 10522  10532
```

would be on the command line, since the quoting of the end-of-line character in the file `junkkill$$`

```
kill -s 9 \
10522 \
10532
```

with the shell quote character `\` at the end of the line make this command act as if it were typed on the single line. That is, the command in line 15 kills the processes involving Netscape (there are two such processes in the present example, but there may be fewer or more such processes in an actual situation – `kill -s 9` is a sure kill; see the footnote on p. 125). The messages of the command in this line are suppressed by redirecting the standard error output to `/dev/null`.

Line 16 clears Netscape's cache (see pp. 99 and 21), and line 17 Netscape's `lock`

file (this must be done, since otherwise Netscape cannot be restarted later; the `lock` file of Netscape prevents the user from starting up Netscape again if it is already running). Finally, line 18 removes the temporary files `junkkill$$` and `junkkill0$$`.

We called the above script `killns`. The name `killnetscape` is not suitable, since then the command on p. 2 of the script would include the information about the process `killnetscape` itself in the file `junkkill0$$`. Then the process `killnetscape` itself would be killed in line 15; thus lines 16 and 17 would not be executed at all.

It is not advisable to use the script `killns` to quit Netscape under normal circumstances. The reason is that this may prevent Netscape from taking certain necessary actions. For example, when you bookmark an item in Netscape, the item gets added to your file

```
$HOME/.netscape/bookmarks.html
```

However, this action is not taken immediately. First, Netscape just "remembers" the item (so you can use the added bookmark immediately), and it adds it to the above file with some delay amounting to perhaps 30 seconds or even some minutes. If you kill Netscape in the meanwhile, the `bookmarks.html` file does not get updated, while if you leave Netscape the regular way, the file does get updated.

A more concise script to kill Netscape. Unix is known for its terseness. In line with this reputation, the above shell script `killns` can be rewritten in a much more concise way.

```
1  #! /bin/sh
2  kill -s 9 \
3    `ps x | grep netscape | awk '{ print $1 }' | cat` \
4    2> /dev/null
5  rm -rf $HOME/.netscape/cache/*
6  rm -f $HOME/.netscape/lock
```

To avoid any confusion with the earlier script, call this new script `killns0`.

First notice that this shell script contains a single command, since the first two end-of-line characters of this three-line command are quoted). This command has the following syntax:

```
$ command1 `command2`
```

(to indicate that we are discussing a command typed on the command line, we added the prompt `$` on the left). Here *command1* stands for

```
$ kill -s 9
```

which is a sure kill, and *command2* stands for

```
$ ps x | grep netscape | awk '{ print $1 }' | cat
```

This command uses a repeated pipe, and writes to the standard output the same information as lines 2–4 of the script `killns` above writes to the file `junkkill$$` (see p. 105), such as (using the example given in p. 107):

```
10522
10532
```

That is, disregarding the commented out line, instead of lines 2 and 4 we could have written the following single line in the script `killns`:

```
ps x | grep netscape | awk '{ print $1 }' | cat > junkkill$$
```

The semantics (meaning) of the construction

```
$ command1 `command2`
```

can be described as follows: *command1* is executed with the argument or arguments that result as the output of *command2*. That is, the output of command may contain several *fields* separated by one or more *separators*. Here separators can be spaces, end-of-line characters, or tab characters. Fields are strings of characters not containing separators. This construction is called *command substitution*, in that the result of the execution of a command is substituted as an argument of another command.

To illustrate this construction, consider the following example

```
$ echo ‘cat argfile‘
```

where `argfile` is a file with the following content

```
1␣2
 3<tab>␣4
5␣␣6
```

Here <*tab*> stand for the *tab* character, and as usual, ␣ stands for the space character. The `echo` command was discussed on p. 97. What it does it simply to repeat its arguments to the standard output, while evaluating shell variables. In case of several arguments, these arguments are separated by single spaces. In the above command,

```
cat argfile
```

send the content of argfile above to the standard output; there are six fields in this output, and these six fields will be the arguments of the `echo` commands. These six fields will be sent to the standard output, separated by single spaces. That is, the output of the command

```
$ echo ‘cat argfile‘
```

in the present example is

```
1 2 3 4 5 6
```

With these considerations one can see that the command

```
kill -s 9
```

in lines 2–3 in the shell script `killns0` above is executed with the arguments `10522` and `10532`, i.e., with the fields in the output of the command between two left quotes ‘ in line 3 of the script. That is, the effect running this script is the same as executing the command

```
$ kill -s 9 10522 10532
```

instead (of course, the two process-id numbers on the right change with each instance of running the script). That is, the processes associated with Netscape will be killed. The effect of redirecting the standard error output in line 4 will be that harmless error messages will be suppressed. If we were to delete line 4 from the script `killns0`, then running the script might result in the following message:

```
kill: (10532) - No such pid
```

This is because the second process associated with Netscape, the one with process-id 10532 is dependent on the first one, i.e., the one with process-id 10522. Once the first process is killed by the `kill` command, the second one dies a natural death; so when the `kill` command tries to kill the second process, it no longer exists, so we get the above error message. Since the error message would only serve to confuse the user, it is best to suppress it, as we did it in line 4 of the script.

As before, line 5 clears Netscape's cache, and line 6 removes its `lock` file.

Using Netscape to view HTML files, revisited. At the beginning of this section, we presented a shell script called `viewhtml` that can be used to view HTML files via Netscape, but the script worked only if Netscape was already running. Here we present a script that performs the same function even if Netscape is not running:[24]

```
 1  #! /bin/sh
 2  NN=file:$HOME/html_files/$1
 3  if test -z \
 4    `ps x | grep netscape | grep -v grep | awk '{ print $1 }'`
 5  then
 6    rm -rf $HOME/.netscape/cache/* 2> /dev/null
 7    rm -rf $HOME/.netscape/lock 2> /dev/null
 8    netscape $NN &
 9  else
10    rm -f $HOME/tmp/junknetscape$$ 2> /dev/null
11    touch $HOME/tmp/junknetscape$$
12    ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
13  0a
14  netscape -remote 'openURL($NN,new-window)'
15  .
16  w
17  q
18  heremark
19    . $HOME/tmp/junknetscape$$
20    rm -f $HOME/tmp/junknetscape$$
21  fi
```

If we call this script `viewhtml1`, then the command

```
$ viewhtml1 special_files/example.html
```

will use Netscape to display the file

```
html_files/special_files/example.html
```

in your home directory. If you use this command, the shell variable `$NN` in line 2 of the script evaluates to

```
file:$HOME/html_files/special_files/example.html
```

In lines 3–21 we have a conditional command of the syntax

`if` *condition*
`then`
list$_1$ *of commands*
`else`
list$_2$ *of commands*
`fi`

(in shell scripts, commands are separated by end-of-line characters); one can use semicolons instead, i.e., the following syntax would also work:

`if` *condition*`;then;`*list*$_1$ *of commands*`;else;`*list*$_2$ *of commands*`;fi`

Here list of commands is a number of commands (separated by end-of-line characters or semicolons – a separator is required even after the last command). The semantics

[24]See the footnote in p. 104 discussing the command `ps x`. This works in Linux, but other Unices (the plural of Unix) use different options of `ps`

(or meaning) of this command is as follows: if *condition* it true then the first list of commands, that is, *list*$_1$ *of commands* is executed, and if *condition* it false then the second list of commands, that is, *list*$_2$ *of commands* is executed. The condition in this command is calculated by the `test` command in lines 2–4, used to check whether Netscape is running.

We are about to explain how this command works. The command

`$ test -z` *string*

checks whether *string* in the argument is empty (has zero length); that is; this command returns the value *true* if the string is empty, otherwise it returns the value *false*. The string for this command is provided by line 4 of the script. This line is similar to line 3 in the script `killns0` on p. 109, but there is a key difference. As we saw it above, line 3 in the script `killns0` returns a list of the process-id's of the processes having `netscape` in their name (as we mentioned above, because of the left quotes ‘ around the command, the the process-id's are separated by spaces, even though without the left quotes, the command would return a list of process-id's separated by end-of-line characters – see p. 110.

However, one processes having `netscape` in its name is the command "`grep netscape`," and the process-id of this command is sometimes (but not always) returned by line 3 in the script `killns0`. There this caused no trouble; but here we want to find out whether Netscape itself is running. In line 4 of the present script, the `grep` command is used, among others, with the `-v` option. The command

`$ grep -v` *string file*

selects those lines of *file* which do not contain *string*. Thus, in the command

```
$ ps x | grep netscape | grep -v grep
```

`ps x` selects all processes (one line of output for each process, then the second member of the pipe, `grep netscape`, selects those lines in the output that contain the word `netscape`, and the third member of the pipe, `grep -v grep`, selects those lines among these that do not contain the word grep. Thus, the line corresponding to the process "`grep netscape`" is discarded, and the lines that are kept are genuinely associated with Netscape.

In case Netscape is not running (or, rather, in case there is no process other than "`grep netscape`" having `netscape` in its name), the list of these process-id's will be the empty string, so lines 6–8 of the conditional command in lines 3–21 will be executed. In case Netscape is running, the list will not be the empty string, so lines 10–20 will be executed.

Note that we named the above script `viewhmtl1`. Whatever else we might have named it instead, no name containing the string `netscape` is allowed. Otherwise the process-id of the script itself would be picked up in line 4, so the test in lines 3–4 would always return *false*.

If Netscape crashed or froze, the process associated with Netscape would probably be still around, so the test in lines 3–4 would return *false*; however, Netscape might not be in a position to accept any commands (such as the command given to in line 19); so the only option in this case would be to kill Netscape (see pp. 104 and 109).

A new feature of this script is the place the temporary files in a special directory, namely

`$HOME/tmp`

that is, instead of `junknetscape$$`, we used the temporary file

`$HOME/tmp/junknetscape$$`

It is important to make sure that this directory exists (if it does not, it needs to be created), and has read, write, and execute permissions by the user. In fact, for security reasons, this directory should have no other permissions; that is the permission code of this directory should be 700 (see p. 18 in Section 3). The main reason for this is the so-called *symlink* attack; cf.

`http://www.egr.msu.edu/archives/public/linux-user`
`/2000-March/000051.html`

(note that the above address should not be broken up, and should have no spaces – we had to break it up here since it does not fit in one line). In the symlink attack, temporary executable files created by user programs are replaced by malicious programs placed there by an attacker (symlink is short for *symbolic link*). To prevent this from happening, no outsider should have permission to place anything into the above directory.

In earlier scripts, we simply put temporary files into the present working directory; the possibility of a symlink attack make these scripts less secure than the most recent script, where a separate `tmp` directory is used for temporary files. Another problems with the earlier scripts is that these scripts will fail if one tries to run them in a directory that lacks one of these permissions.

Otherwise, there is very little need to comment on the rest of the script. Line 6 removes Netscape's cache, line 7 removes its lock file (we want these commands to be silent, so we send the standard error message to `/dev/null`, so the command will not complain even if there is nothing to remove), and line 8 starts up Netscape in the background. All this is done in case Netscape has not already been running.

Lines 10–20 are much the same as lines 3–13 of the script `viewhtml` on p. 101. The differences are that in the present script we used unique filenames (see p. 103) and put the temporary files in a special directory; in line 10 we used `2>` to redirect the standard error output instead of `>&` used in line 3 of the earlier script (the two have the same meaning), in line 12 we suppressed the standard error output of the `ed` command, while in the earlier script we did not do so in line 5. Finally, many os lines 10–20 are indented, to indicate to the reader (the computer does not care) that these lines are part of the conditional command in lines 3–21. However, lines 13-18 of the *here document* starting on line 12 cannot be indented. In fact, line 18 cannot be indented, since the mark signifying the end of the here document must start at the beginning of the line; lines 13–17 must not be indented, since in the commands given to the editor `ed`, every character in a line is meaningful. For example, for the `ed` command in line 15 to work, there must be only a single period in the line (so as to put the editor into command mode from insert mode); indenting line 15 would disable the script. Indenting lines 13, 14, 16, and 17 would not disable the script; adding two spaces at the beginning of line 14 would add these two spaces at the beginning of the single line of the file `junknetscape$$` being edited, but these spaces would cause no harm for the command in line 19.

As we mentioned above, some of the above scripts may fail when invoked from a directory where the user has no read, write, or execute permission. For example, the script `viewhtml0` on p. 103 may fail for this reason. The correction is simple, however. Instead of making changes similar to the ones we made in the script `viewhtml1` above (p. 111) to specify a path for the file `junknetscape$$`, we can simply add the line

```
cd $HOME/tmp
```

before line 3 in the script `viewhtml0`. This will ensure that the temporary file `junknetscape$$` will be created in the directory `$HOME/tmp`. Note that this directory change is effective only inside the shell script. After quitting the shell script, one will be back in the same directory where the shell script was invoked.

Going to a Web site given in a file.. The next script will go to a Web site given listed in the file given in the argument. Calling this script `netfread`, and assuming that the file tt website in the present wording directory contains the line

```
http://www.egr.msu.edu/
```

(the Web site of Michigan State University College of Engineering), the command

```
netfread website
```

will open a new Netscape window (whether or not Netscape is already running) and go to the above site. This script might be useful if a friends sends you a Web address in an email. You can copy the address to a temporary file, and use the script to access the site:

```
 1 #! /bin/sh
 2 if test \ -s = \ $1
 3 then
 4   echo "go to URL given in file listed as argument"
 5   exit 0
 6 fi
 7 if !( test -f $1 )
 8 then
 9   echo "file $1 does not exist"
10   exit 1
11 fi
12 rm -f $HOME/tmp/junknetscape$$ 2> /dev/null
13 touch $HOME/tmp/junknetscape$$
14 if test -z \
15   `ps x | grep netscape | grep -v grep | \
16    awk '{ print $1 }' | cat` 2> /dev/null
17 then
18   rm -rf $HOME/.netscape/cache/*
19   rm -rf $HOME/.netscape/lock
20   ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
21 0a
22 netscape
23 .
24 1r $1
25 2,\$j
26 w
27 q
28 heremark
29 else
30   ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
31 0a
32 netscape -remote 'openURL(
33 ,new-window)'
34 .
```

```
35 1 r $1
36 2,\$-1j
37 w
38 q
39 heremark
40 fi
41 ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
42 2s/,/%2C/g
43 w
44 q
45 heremark
46 ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
47 2s/ //g
48 w
49 q
50 heremark
51 ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
52 1,\$j
53 w
54 q
55 heremark
56 . $HOME/tmp/junknetscape$$ &
57 touch $HOME/tmp/junknetscape$$
58 rm -f $HOME/tmp/junknetscape$$ > /dev/null
```

Next we will explain how this script works. In line 2, it is tested if the first (and only) argument of the script is `-s`; this is done by the `test` command, which, in the form

`test` *string1* `=` *string2*

compares if the two strings *string1* and *string2* are identical. The problems is that this cannot be used directly to compare the string -s to another string, since the command `test` with the option `-s` has a special meaning. Namely, the command

```
test -s filename
```

if the file by the name `filename` exists. To go around this problems, a space is added in front of each string; that is the strings "`\ -s`" and "`\ $1`" are compared (here `\` placed before the space character quotes the space character, so the latter is interpreted as part of the string rather as separator). If $1 is the string `-s`, that is if the command entered is

```
netfread -s
```

then the command on line 4 simply writes the line

```
go to the URL given in file as argument
```

written on the screen, and the command line 5 exits the shell script. The argument `0` of the command `exit` means that the shell script will be terminated with the "exit value" 0. The exit value 0 means that the script terminated without error – other exit values are used to indicate various error conditions that caused the termination of the script.

In line 7, the condition

```
test -f $1
```

is true if the file named in the argument `$1` exists, and the exclamation point `!` in the same line negates this condition. That is, if the file does not exist, then line 9 reports this fact, and line 10 makes the shell script terminate with exit value 1; we use this exit value, since to run this script while the file in question does not exist should count as an error.

As before, line 12 removes the file `junknetscape$$` (if it exists), and line 13 creates a new empty file by this name. Lines 15 and 16 test if Netscape is already running (the condition is true if Netscape is not running, and is false otherwise – see the discussion of line 4 of the script `viewhtml` in page above). If Netscape is not running, in line 18, Netscape's cache is cleared, and in line 19 its `lock` file is removed (in case if was left over as a result of a crash or improper termination of Netscape).

Lines 20–28 an here document using the editor `ed` is used to edit the (currently empty) file `junknetscape$$`. Line 22 puts the word `netscape` in the first line, line 24 reads the content of the file `$1` (the argument to the script) beginning line 2 of the the file `junknetscape$$`. It is permissible to have the Web address contained in the file `$1` to be broken up into several lines, therefore, in line 25, lines 2 through the last line (`$` refers to the last line of the file being edited) of the file `junknetscape$$` are joined (so the file will now have exactly two lines). Note that the `$` sign in line 25 is quotes (i.e., preceded by the Unix quote character \); the shell will remove the quote character, so the editor `ed` will see the `$` sign without the \ preceding it; if it were not quoted, the shell would interpret `$j` as the value of a shell variable called `j`, before the editor `ed` would get at it (we have met this kind of situation in here documents before). Line 26 writes out the content of the editors buffer to the file being edited, and line 27 quits the editor. Note that all this happened in a region of a conditional statement that was executed in case Netscape was not running.

Line 29 introduces the alternative when Netscape is already running, and in line 30 the editor `ed` is opened to deal with this situation. First lines 32 and 33 are added as first and second lines, respectively, to the currently empty file `junknetscape$$`; line 35 inserts the content of the argument file `$1` after line 1 (i.e., beginning line 2) of the file being edited (so the line that is currently the second line will move further down). Line 36 joins the first through the last but one line of the file being edited; the last line, the one that was entered as the second line in line 33 will not be joined. Line 37 and 36 writes out to the file and quites the editor, and the word `fi` on line 37 marks the end of the the conditional shell command; that is, the later part of the script deals with the situation independently whether or not Netscape is already running.

The here document in lines 41–45 changes all occurrences of a comma to the string `%2C` in line 2 of the file `junknetscape$$`. This line contains the Web address; in a Web address, a character can always be replaced by the the hexadecimal ascii code of the character, preceded by the percent sign `%`; `2C` happens to be the hexadecimal ascii code of the comma.

Note that the file `junknetscape$$` will contain a Unix command – a comma in a Unix command would be misinterpreted; since commas do occur in some Web addresses, the command would fail without the above substitution. Most Web addresses do not contain commas, so in most cases, lines 41–45 do not do anything. This is somewhat of a problem, since a substitute command (as in line 42) in `ed` that does not do anything (fails to find the pattern to be substituted) will cause

the editor `ed`, when called in a here document, to quit. For this reason, if only one editing task can be entrusted to an `ed` here document if that task can result in failure. This is why the editor `ed` is invoked several times in the remaining part of the script.

The here document in lines 46–50 deletes all spaces in line 2 of `junknetscape$$`, the file containing the Web address. Of course a Web address cannot contain spaces. One way spaces could have intruded into line 2 of the file `junknetscape$$` was when we joined several lines in the file (in line 25 or 36), since joining lines usually replaces the newline character with a space. The here document in lines 50–55 joins all lines the file `junknetscape$$` (currently, the file contains two or three lines, according as Netscape was not or was running at the time of invoking the script). In line 56, the script `junknetscape$$` is invoked in the background, and in line 58, the file `junknetscape$$` is removed. The reason for running the command in line 56 in the background is that this returns the Unix command line (where the command running the whole script currently being discussed was entered) sooner – without running it in the background, one might have to wait a few seconds before one can enter the next command. However, running the command in the background occasionally introduces some timing problems; the file `junknetscape$$` is removed by the command on line 58 before the command on line 56 is able to make use of this file. To prevent this, the command on line 57 was added; this command has no other effect; it updates the time of last change of the file `junknetscape$$`, but this has no importance, since the file is going to be removed anyway.

Going to a Web site given in a file – a variant. A variant, let us call it `netfread0`, or the above shell script is as follows:

```
 1 #! /bin/sh
 2 if test \ -s = \ $1
 3 then
 4   echo "go to URL given in file listed as argument"
 5   exit 0
 6 fi
 7 if !( test -f $1 )
 8 then
 9   echo "file $1 does not exist"
10   exit 0
11 fi
12 if test -z \
13   `ps x | grep netscape | grep -v grep | \
14    awk ' print $1 ' | cat` 2> /dev/null
15 then
16   rm -rf $HOME/.netscape/cache/*
17   rm -rf $HOME/.netscape/lock
18   rm -f $HOME/tmp/junknetscape$$ 2> /dev/null
19   touch $HOME/tmp/junknetscape$$
20   ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
21 0a
22 netscape
23 .
24 1r $1
```

```
25 2,\$j
26 2s/\$/,/
27 2s/,/%2C/g
28 2s/%2C\$//
29 2s/^/ /
30 2s/ //g
31 1,2j
32 w
33 q
34 heremark
35 else
36   rm -f $HOME/tmp/junknetscape$$ 2> /dev/null
37   touch $HOME/tmp/junknetscape$$
38   ed $HOME/tmp/junknetscape$$<<heremark 2> /dev/null
39 0a
40 netscape -remote 'openURL(
41 ,new-window)'
42 .
43 1 r $1
44 2,\$-1j
45 2s/\$/,/
46 2s/,/%2C/g
47 2s/%2C\$//
48 2s/^/ /
49 2s/ //g
50 1,3j
51 w
52 q
53 heremark
54 fi
55 . $HOME/tmp/junknetscape$$ &
56 touch $HOME/tmp/junknetscape$$
57 rm -f $HOME/tmp/junknetscape$$ > /dev/null
```

We will not go through in detail as to how much this script works, because it has a lot of similarities with the preceding script. The main point that is new here is to deal with the error handling of a here document using the editor `ed`. We recall that this problem, which makes `ed` exit prematurely if it receives a substitute command that never finds a match to substitute, caused us to use several here documents in the preceding script. Here, line 26 in the above script, inside a here document, substitutes each comma in the second line of the file `junknetscape$$` with the string `%2C`; we recall that `2C` is the hexadecimal ascii code of the comma (see p. 116 above). This substitution, however, would fail if the line in question contains no comma. Therefore, line 26 adds a comma to the line as its last character, and line 28 removes the string `%2C` resulting from the substitution of this comma. In a similar manner, line 29 adds a space as the initial character to line 2 of the file being edited, and line 30 removes all spaces from this line – line 29 guarantees that the substitution in line 30 will not fail. Lines 45–47 and lines 48–49 use similar tricks.

Perl. Many of the programs described above can be more written more elegantly in the computer language Perl. See

`http://www.perl.com`

or the book Larry Wall, Tom Christiansen, and Jon Orwant, *Programming Perl*, O'Reilly & Associates, Inc. Sebastopol, CA;

`http://www.oreilly.com`

Previewing TEX files. As explained in Section 21, if you have a TEX file called `whole.tex`, you can typeset it with

```
$ tex whole
```

and preview it with, say,

```
$ xdvi -s -thorough whole &
```

This will open a preview window in which the typeset file can be seen (see p. 79). If you change the file `whole.tex`, then you need to type the former command to typeset it again. You do not, however, need to type the latter command again; if you simply click on the preview window, you can see the new version of the typeset file.

Things can go wrong, however, if the file `whole.tex` takes long to typeset. If you click on the preview window before the new version of the file `whole.dvi` is written, the preview window will display nothing useful. It will probably display the lines

```
DVI file corrupted
```

This is because the old version of the file `whole.dvi` has already been removed, and the new version is being written. Since the typesetting of a long TEX file can take as long as ten seconds, this may be quite inconvenient. The following shell script provides a simple workaround this problem.

```
1 #! /bin/sh
2 cp $1.tex $1_junk.tex
3 tex $1_junk
4 cp $1_junk.dvi $1.dvi
5 rm $1_junk.*
```

If you call this file `texxing`, then you can typeset the file `whole.tex` by typing

```
$ texxing whole
```

The point about this script is that the file `whole.dvi` is only corrupted while the command `cp` in line 4 is executed, and that takes only a small fraction of a second. Line 5 removes the files `whole_.tex`, `whole_.log`, and `whole_.dvi`; the latter two were created by the TEX program; in the file name, the asterisk `*` is matched by any string; i.e., any file whose name starts out as `whole_junk.`, and has some additional characters at the end will be removed (see p. 12 for a discussion of the wild card character asterisk `*` in filenames).

Note that the fact that the above script works is an idiosyncrasy of the behavior of the program `xdvi`. For example, the scripts

```
1 #! /bin/sh
2 cp $1.tex $1_junk.tex
3 tex $1_junk
4 cp -f $1_junk.dvi $1.dvi
5 rm $1_junk.*
```

and

```
1 #! /bin/sh
2 cp $1.tex $1_junk.tex
3 tex $1_junk
4 mv $1_junk.dvi $1.dvi
5 rm $1_junk.*
```

do not work; that is, if we repeatedly change the file `whole.tex` and then typeset if by these scripts, the display window opened by `xdvi` will only be updated if we use the first of the above scripts for typesetting, and not if we use the second or the third script. A perhaps somewhat simplistic explanation might be that the file `whole.dvi` produced by line 4 (when the script is run as above, with the argument `whole`) of the first of the above three scripts is recognized by `xdvi` as a new version of the file `whole.dvi`, while the files produced by the second and the third scripts are considered new files by `xdvi` that (accidentally) bear the same name `whole.dvi`. This behavior is easier to understand in case of the third script than in case of the second script, and may, in fact, not be the same for different versions of Unix. The behavior of these scripts were tested only in Red Hat Linux 6.0.

Removing left-over files. The following script removes unwanted files that may have been left over accidentally:

```
1  #! /bin/sh
2  unalias rm 2> /dev/null
3  find . '(' -name core -o -name 'junk*' \
4           -o -name '*.junk' -o -name 'a.out ')' \
5           -user $USER -type f \
6    -exec rm '{}' \;
```

The command `find` is a very versatile, and we cannot explain its main uses here. Briefly, it finds the files described in the pattern, and acts on them the way indicated. We will give a more detailed description of the above script. The second line unaliases the `rm` command; that is, it makes it act the normal way it is supposed to act. Shell aliases are explained in the beginning of the next section; here we will only say that on occasion, the meanings of certain commands are changed, often to protect new users; such a change of meaning is called *aliasing* the command. `rm` is one of these commands; in a way, it is a dangerous command, since it removes files, and once you executed it it is too late to think that you really wanted to keep the file. For this reason its meaning is often changed; see the next secion for details. The `unalias` command removes the alias from the `rm` command, but it only does this for the duration of the shell script, so no permanent change is made. In case the command `rm` is not aliased, the `unalias` command will send an error message to the standard error output (see above); since you do not want to see this error message, the error message is discarded by the redirection `2> /dev/null`.

In lines 3–6 the `find` command is invoked; the backslash `\` at the end of lines 3–5 quotes the newline character at the end of these lines, to indicated that the command does not end with the end of these lines. The `find` command starts its search in the current directory (indicated by the period `.`), and searches this directory and all its subdirectories (and their subdirectories; the usual expression is to say that the command recursively searches the subdirectories) for files whose name matches the pattern given after the option `-name`; the patterns are placed between single right quotes, e.g., `'a.out'`. The `-o` means *or*, that is, search for files whose name

matches the pattern `core`, or `junk*`, or `*.junk`, or `a.out`. These files are unneeded, and will be removed by the command (the file `core` is usually left over by a crashing program (see Section 15), and `a.out` is the default name for a compiled program, and is usually unwanted (because if you wanted to keep the compiled program, you would give it your own name, rather than the default name – see Section 8). The parentheses on lines 3 and for delimit the range of the disjunction of the patterns described by `-o`; these parentheses must be placed between single right quotes, i.e., we have `'('` and `')'`.

The option `-user $USER` on line 5 means that only search for files owned by the `$USER`, which is the value of the shell variable `USER`, naming the person who signed onto the computer (so that you will not search for files owned by others). The option `-type f` means that you only look for files that are properly files (and not directories).

Finally, line 6 specifies what to do with the files that are found; namely, execute the `rm` command on them. That is, all the files that have been found will be removed; the unaliasing of the `rm` command will make sure that you will not be asked about each file whether you really want to remove it. The quoted semicolon `\;` indicates the end of the command named by the `-exec` option.

25. Source Scripts

Shell aliases. As we saw above, if you want to run a sequence of commands repeatedly, you can put them in a shell script and run the shell script instead. There, are, however, some notable things shell scripts cannot do; namely, they cannot change directories, and they cannot change shell variables; that is, they can, inside the script, but after the shell script finishes, the *status quo ante* will be restored. That is, the shell script

```
cd /usr/bin
ls
```

will change to the directory `/usr/bin`, and then list its content, but after it finishes you will be in the directory where you started the shell script, and not in the directory `/usr/bin`. So what are you supposed to do if you often have to examine the content of a directory such as

```
/usr/lib/texmf/texmf/tex/amstex/base
```

Perhaps you want to do this since certain things do not work properly on your system and you want to understand why.

One thing you can do is to define an alias. User defined aliases are usually placed in the `.kshrc` file (this stands for Korn shell root commands) in the user's home directory (C shell uses `.cshrc` and Bourne-again shell uses `.bashrc`. The line you would place in this file may look like

```
alias cdtexbase='cd /usr/lib/texmf/texmf/tex/amstex/base'
```

This will allow you to type

```
$ cdtexbase
```

instead of having to type

```
$ cd /usr/lib/texmf/texmf/tex/amstex/base
```

These aliases are not effective immediately; they will take effect after first logging out, and then logging in again. There are many instances that such aliases are useful. For example, we discussed the command

```
$ xdvi -s 4 thorough first &
```

to display preview the typeset tex file `first.dvi` above. By adding the alias

```
$ alias texview='xdvi -s 4 thorough'
```

to the file `.kshrc`, we will be able to type the shorter command

```
$ texview first &
```

You can find out what aliases are in effect during your session by typing

```
$ alias
```

The reply you receive may be something like this

```
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
alias cdtexbase='cd /usr/lib/texmf/texmf/tex/amstex/base'
alias texview='xdvi -s 4 thorough'
```

The first three among these are there for your protection. They specify the interactive option for the commands `cp`, `mv`, `rm`, since each of these may destroy files (by copying or moving something else over them, or by removing them); so, before a file is lost, you are queried (asked) by the computer if you really want to lose the file. If you do not want to be protected this way, you might delete these aliases, or, better yet, comment them out by placing the *pound sign* `#` at the beginning of the line (recall that `#` is the shell comment character).

Source scripts. As an alternative to aliases, you can write a command into a file and *source* the comment. That is, if you have a file called `cdbase` with the content

```
cd /usr/lib/texmf/texmf/tex/amstex/base
```

then typing

```
$ . ./cdbase
```

will get you to the directory in question. In fact, on typing the last line, the computer takes the commands from the file `cdbase`, and executes them in order as if these commands *were written on the command line*; that is, changes of directory or shell variables will take effect. On some systems, the command

```
$ . cdbase
```

also works (cf. 38), but it may or may not take the file `cdbase` from the present working directory – it may instead take it from the first directory it finds among those listed in the `PATH` shell variable, and the latter command may not work at all if the present working directory is not included in the `PATH` variable (cf. 98). Hence the first version is definitely preferable.

The above command works in the Korn shell, in the Bourne shell, and in the Bourne-again shell (bash); in C shell, to accomplish the same thing, one should type

```
$ source cdbase
```

Hence the expression *sourcing* a file. The problem with the command

```
$ . cdbase
```

is that it works only in the directory the file *cdbase* is located. This is clearly not practical, since you may want to change to the above directory with the long name

from wherever you are. To do this, you can put the following aliases into the *.kshrc* file (in Korn shell; in bash, you put it in the file *.bashrc*):

```
alias sc='pwd > $HOME/lib/sources/sc.junk;cd $HOME/lib/s
ources;.'
```

Note that this is a single line that wraps over into the next line, and not two lines. In fact, for this line to work, it must not be broken up.[25] The meanings of the commands in this alias are fairly straightforward, their intention is not so. We will explain how this works later; here we only mention that the semicolon is used to separate two commands (in fact, even on the command line, you can write several commands separated by semicolons). The file *cdbase* has to be put into the directory `$HOME/lib/sources`, and it has to be changed so that its content should be

```
mv sc.junk sc1.junk
cd /usr/lib/texmf/texmf/tex/amstex/base
```

This source script will work in cooperation with the following script called *return*, placed also in the directory `$HOME/lib/sources`, with the following content:

```
cd ‘cat sc1.junk‘
```

It is important to note that the quote signs here are left-quote sings (using right quotes will not work). The meaning of the command displayed is as follows: execute the command between the left quotes, and take the result of this execution as the argument of the command `cd`.

Now we can explain how all this scheme works. When typing[26]

```
$ sc ./cdbase
```

the present working directory is saved into the file

```
$HOME/lib/sources/sc.junk
```

Assume this directory is, say, `/usr/lib/X11`. The second command in the alias defining *sc*, separated from the first one with a semicolon, changes to the directory

```
cd $HOME/lib/sources
```

and the third command runs (“sources”) the file listed on the command line (*cdbase* at present; this file must be present in the above directory).

The first action of sourcing the file cdbase is executing the first command in this file, i.e, moving the file `sc.junk` to `sc1.junk`, and the second action is changing to the directory with the long name described in the second line. If, after doing some work, one wants to return to the directory where one was before changing directories, one can type

```
$ sc ./return
```

This will cause the command in this file, i.e., the command

---

[25] This is not quite true; in Korn shell, it can be broken up as

```
alias sc='pwd > $HOME/lib/sources/sc.junk; \
    cd $HOME/lib/sources;.'
```

while in bash it can be broken up as

```
alias sc='pwd > $HOME/lib/sources/sc.junk;
    cd $HOME/lib/sources;.'
```

The Korn shell line break will not work in bash, and the bash line break will not work in the Korn shell. The C shell version is completely different:

```
alias sc 'pwd >! $HOME/lib/sources/sc.junk;cd $HOME/lib/sources;source'
```

I have not experimented with how to break this up.

[26] Depending whether or not `.` is included in the PATH variable and on other factors (such as the shell used, etc.), the command

```
$ sc cdbase
```

might also work, but the command given in the main text is preferable.

```
cd ‘cat sc1.junk‘
```

to be executed; the result of the command between the left quotes is the content of the file *sc1.junk* (in the directory `$HOME/lib/sources`), which is the directory where we started out; this was assumed to be `/usr/lib/X11`. so the command *cd* changes back to this directory.

Note the reason for the line

```
mv sc.junk sc1.junk
```

in the file `cdbase`. When we type

```
$ sc ./return
```

the first command in the *sc* alias would have overwritten the file *sc.junk* and its content would have been lost, had this file not been moved to *sc1.junk*.

Observe that, in Korn shell, and in bash, you can change back to the previous directory by typing

```
$ cd -
```

This would not work here, because the way the *sc* alias was set up, the previous working directory would be `$HOME/lib/sources`.

There are other uses of the *sc* alias than changing directories. The following two files in the `$HOME/lib/sources` directory might be useful for changing the shell variable `PATH`. The file *changepath* would have the content

```
oldPATH=$PATH
PATH=.:$HOME/bin:$PATH
export PATH oldPATH
```

and the file *restorepath* would contain

```
PATH=$oldPATH
export PATH
```

This may be a solution preferred to putting these changes in the *.profile* file in case these changes are required only rarely.

26. Job Control

We already talked about running programs in the background. The typical example was to run Netscape in the background by typing

```
$ netscape &
```

The ampersand `&` at the end of the line indicates that the program is to be run in the background; the reason for running the program in the background is to free up the window for other command to be typed: if a command is run the usual way, i.e., in the foreground, then one cannot type another command in the same window until the command finishes; yet Netscape will not finish until one closes the Netscape window by exiting Netscape. Another instance of running a program in the background is to open a new window by typing

```
$ xterm &
```

If you forget to run this program in the background, i.e., if you type

```
$ xterm
```

the new window will still open, but the prompt will not be returned in the original window. There are several ways to remedy this situation.

First, one can type

```
$ exit
```

in the new window. This will close the window. The same effect can be achieved by typing `Ctl-c` (i.e., holding down the control key, and typing `c` at the same time) in the original window (i.e., in the window in which the command `xterm` was typed). `Ctl-c` usually kills (terminates) a running program, although not every program will respond to `Ctl-c`.

As the third action, one can stop the program by typing `Ctl-z`. The computer will respond with something like this:

```
[2]+  Stopped                 xterm
```

Here `[2]` indicates the number of the job, `+` refers to the current job (and `-` refers to the previous job.

One can restart this job in the foreground by typing

```
$ fg %2
```

there would be little purpose of doing this, since the program was running in the foreground before it was stopped, and it was undesirable to have `xterm` running in the foreground. Instead, one can restart this job in the background by typing

```
$ bg %2
```

This is the proper solution, and its effect is the same as if the program `xterm` had been started in the background in the first place. To check which jobs run in the background, one can type

```
$ jobs
```

The answer might something like

```
[1]   Running                 netscape &
[2]-  Stopped                 xterm
[3]+  Running                 xterm &
```

The ampersand `&` at the end of the line indicates that the program is running in the background. One can terminate, say, the third of these programs by typing[27]

```
$ kill -9 %3
```

The command `kill` has a number of options, `-9` is the most effective (but it might be preferable to use a more gentle kill – we will not discuss this issue here, but you can study the on-line manual of `kill`). Both stopped programs and programs running in the background can be killed this way. One should only kill a program when a more gentle option is not available (for example, one can terminate `xterm` by typing `exit` in the `xterm` window one wants to close – except when the window is stuck or hung, in other words, *locked up* (i.e., is non-responsive to keyboard input). It is important to know about killing jobs, because often the computer will not allow you to log out if there are stopped jobs. It may happen that when you want to log out by typing

```
$ logout
```

the computer responds with

```
There are stopped jobs
```

[27] The command

```
$ kill -s 9 %3
```

works exactly the same way.

By killing these jobs, one will be able to log out. Usually, however, even without killing these jobs, one can log out by typing `logout` again, and the stopped jobs will be flushed (killed).

27. Remote Access

Remote shell. Computers at the Atrium can be accessed on the Internet. In Unix, the command to reach a machine on the Internet would be

`$ rsh -l` *login_name machinename*`.its.brooklyn.cuny.edu`

Here *login_name* needs to be replaced by the login name of the user, and *machinname*, by the name of the machine you want to be connected to; this is usually all lower case letters, and you must have found out the name of a machine by having typed

```
$ hostname
```

while connected to the machine on an earlier occasion. The login name in the above command can be omitted, i.e., one can type

`$ rsh` *machinename*`.its.brooklyn.cuny.edu`

instead. Here `rsh` stands for *remote shell*. If both the remote and the local machine run Unix, you can even display X windows graphics created by the remote machine on your local machine. To do this, you need to type

`$ xhost +` *machinename*`.its.brooklyn.cuny.edu`

before logging in (or in a separate, local window). For this, you must have opened X windows on the local machine. If you are permanently connected to the Internet and have your own *host name* (the name of your machine by which it is known on the Internet) and *domain name*, the name (such as *its.brooklyn.cuny.edu*) by which your machine or your local network is found on the internet, then, after logging in you can type

`$ DISPLAY=`*hostname.domainname*`:0.0`

```
$ export DISPLAY
```

where, of course, *hostname* and *domainname* need to be replaced by the actual host name and domainname.

When you are connected to the Internet from your home, it is more likely that you do not have your own host name and domain name. In this case, after logging in, on the remote machine you need to type

`$ DISPLAY=`*local_IP_address*`:0.0`

```
$ export DISPLAY
```

where *local_IP_address* is the IP address (Internet Protocol address) of your local machine. It is something like 148.243.14.91 (but, of course, not this number); we will soon indicate some ways as to how to find out this number. If you connected to an Internet service provider, your IP address will probably change each time you are connected to the Internet (this is called dynamic IP address allocation; an IP address that always remains the same is called static IP address).

Finding out your IP address. The way you find out your IP address may depend on what operating system you are using at home. For example, assume

you are using a machine running Red Hat[28] Linux 6.0 and have an account with the Internet service provider Mindspring (we mentioned the version of Linux for the sake of being specific, since the files we will mention may be located at slightly different places in different versions of Unix). As a first step, connect to your Internet service provider Mindspring, using the protocol ppp; ppp works through phone lines and modems, and will make your computer a part of a larger network. Note that not all Internet service providers allow ppp connections (for example, America Online does not), so when using Linux, you are restricted to certain Internet service providers that support it. You can find Mindspring at

```
http://www.mindspring.net/
```

At the time the ppp connection is made, your computer writes your local IP address into the file

```
/var/log/messages
```

Since the file `/var/log/messages` is not readable by a regular user, you first need to become a *superuser* to read this file.[29] The superuser is a special user with the highest privileges on a Unix machine. You can become superuser by typing

```
$ su -
```

whereupon the computer responds with the word

```
Password:
```

At this point, the you need to type the superuser password (which you would have to know; but if you have a computer running Unix at home, you will have to know the superuser password). After becoming superuser, the prompt will change from the regular user prompt `$` (or whatever the actual prompt is) to the superuser prompt `#` (or to a more complicated prompt displaying the sign `#`). The reason for this is that being a superuser is dangerous, and the different prompt is a reminder that one has to be extremely careful. One must not do any work on the computer as superuser unless it is absolutely necessary, since serious damage can result if mistakes are made. Some administration functions can, however, only be performed as superuser.

Once becoming superuser, you can read the file `/var/log/messages` by using *vi*. The best way to do this is to type

```
# cd /var/log
# view /var/log/messages
```

Here `#` indicates the superuser prompt, as mentioned just before. The command *view* is the same as *vi* to look at text files, but it prevents you from changing the file; more precisely, it will allow you to make changes in the file, but it will prevent you from saving these changes. If you inadvertently made changes while using *view*, you cannot quit the editor in a normal way. To quit, in command mode, you need to type either

```
:q!
```

to force a quit without saving a changes, or

```
:wq!
```

to force a quit while saving the changes; see the discussion of command line mode in *vi* on p. 15.

---

[28] A book about Red Hat coauthored by Red Hat's CEO *Young*, is Robert Young–Wendy Goldman Rohm, *Under the Radar*, Coriolis, 1999.

[29] In Solaris (SunOS 5.5.1), the same file is called `/var/adm/messages`, and is readable by every user.

The problem you have to deal with when using *vi* to find out your IP address is that the file `/var/log/messages` may also contain your IP addresses from past ppp sessions, and you need to look at your last local IP address given in this file. Using the *tail* command makes this problem easy to handle. This command *tail* writes the last few lines of the file on the screen, and its `-f` option writes keeps writing new lines as lines they are being added to end of the file. So if you type the commands (as superuser)

```
# cd /var/log
# tail -f messages
```

just before connecting to your Internet provider, your local IP address will be written on the screen by this command at the time the connection is established. Thus, in addition to finding out your IP address, you can use this command to monitor whether your connection is properly established. After you have seen enough, you must stop the the *tail* command by typing Control-`c`. To quit being superuser, you need to type

```
# exit
```

Remote login and telnet. Instead of `rsh` one can use *remote login* to similar effect:

`$ rlogin -l` *login_name machinename*`.its.brooklyn.cuny.edu`

Telnet works essentially the same way (but uses a different protocol) to connect to a remote machine:

`$ telnet` *machinename*`.its.brooklyn.cuny.edu`

Specifying terminal type. When remotely connecting from a Unix machine to another Unix machine, the connection is usually established "smoothly," without needing additional intervention. When connecting from a non-Unix machine to a remote Unix machine, often the remote machine has to be told of the "terminal type" of the window used on the local machine. What this exactly means is unimportant; the important point is that without doing this the command *vi* used to edit files will not work correctly on the remote machine. Most often, the local machine runs a terminal type called *vt100*. To do this, after logging in on the remote machine, one needs to type the following in the window connected to the remote machine:

```
$ TERM=vt100
$ export TERM
```

The first line assigns the value `vt100` to the *shell variable* `TERM`, and the second line makes the value of this variable to the subshells of the main shell (the command interpreter, i.e., the program that makes the computer react to what you are typing on the command line, after the prompt `$` – for another use of the command *export* see p. 97).

Connecting with non-Unix machines. Some of the above ways to connect to a remote machine can be used even if the local machine is not a Unix machine. Telnet is perhaps to most widely understood protocol to do this. The ways different operating systems invoke telnet depends on the configuration of the local machine; one can often use Netscape to start up telnet. To do this, first click with the left mouse button on the word *File* in the Netscape windows; this will open a small window with a number of choices. By clicking with the left mouse button on the phrase *Open Location* among these choices, another window will open (see p. 20).

After positioning the mouse pointer in this window, one can type the command to start telnet:

`telnet://`*machinename*`.its.brooklyn.cuny.edu`

Telnet is probably the most widely supported protocol on non-Unix machines, but one may also try using other protocols. E.g., using Netscape as above, instead of telnet, one may be able to connect to a remote machine by typing

`rlogin://`*machinename*`.its.brooklyn.cuny.edu`

Using `rsh` instead of `rlogin` will probably not work. In the way described above, one may be able to connect to the Atrium from the computers provided for Web-browsing in the New York Public Library by using telnet or, perhaps, rlogin. When logging in to a computer from a public place such as a library, you must be very careful to also log out. It is very easy to inadvertently click on a large window that will hide the login window. You might then forget about the hidden login window, and a later user of the public computer may have access to your private account.

Connecting to a friends Unix machine. The remote connection capabilities of Unix may allow you to be connected to a machine of your friend at a different location provided both of you are connected to the Internet. For example, assume your friend using a machine running Red Hat Linux has an account with the Internet service provider Mindspring. As a first step, she connects to her Internet service provider Mindspring, using the protocol ppp.

As the second step, she emails you her local IP address. In the discussion above we mentioned that this is written in the file

`/var/log/messages`

and the simplest way for her to send it to you is by emailing the whole file to you, provided she trusts you with the information in this file (otherwise she needs to find her local IP address in this file and only email that to you).

Since the file `/var/log/messages` is not readable (and cannot be emailed) by a regular user, she first must become a superuser to do this. Having done so, she can email the file `/var/log/messages` to you by typing

`# mail -v` *your_email_address* `< /var/log/messages`

Here `#` indicates the superuser prompt, as mentioned above. Upon learning the IP address of your friend, you can log into her computer by typing

`$ rsh -l` *login_name IP_address*

Here *login_name* is your login name on her machine (unless you are a user on her machine or she lets you know her login name and her password, you cannot log in), and *IP_address* is the IP address of her machine.

If your Internet connection ties up your only phone line, you can still communicate in writing with your friend by using Unix's talk command. You can do this by typing

`$ talk -l` *yourfriends_login_name*`@`*IP_address*

where *yourfriends_login_name* is the login name of your friend on her machine and *IP_address* is the same IP address that you used to log into her machine. You would normally type this command in a window representing your own machine, and not in the window you used to log into her machine (though the command should also work in the latter window).

28. Solution to Problem 1 in Section 16.

a)
$$\begin{aligned} 25 &= 14\cdot 0 + 25\cdot 1\\ 14 &= 14\cdot 1 + 25\cdot 0\\ 11 &= 14\cdot(-1) + 25\cdot 1\\ 3 &= 14\cdot 2 + 25\cdot(-1)\\ 2 &= 14\cdot(-7) + 25\cdot 4\\ 1 &= 14\cdot 9 + 25\cdot(-5) \end{aligned}$$

b)
$$\begin{aligned} 488 &= 384\cdot 0 + 488\cdot 1\\ 384 &= 384\cdot 1 + 488\cdot 0\\ 104 &= 384\cdot(-1) + 488\cdot 1\\ 72 &= 384\cdot 4 + 488\cdot(-3)\\ 32 &= 384\cdot(-5) + 488\cdot 4\\ 8 &= 384\cdot 14 + 488\cdot(-11) \end{aligned}$$

c)
$$\begin{aligned} 645 &= 242\cdot 0 + 645\cdot 1\\ 242 &= 242\cdot 1 + 645\cdot 0\\ 161 &= 242\cdot(-2) + 645\cdot 1\\ 81 &= 242\cdot 3 + 645\cdot(-1)\\ 80 &= 242\cdot(-5) + 645\cdot 2\\ 1 &= 242\cdot 8 + 645\cdot(-3) \end{aligned}$$

d)
$$\begin{aligned} 122 &= 48\cdot 0 + 122\cdot 1\\ 48 &= 48\cdot 1 + 122\cdot 0\\ 26 &= 48\cdot(-2) + 122\cdot 1\\ 22 &= 48\cdot 3 + 122\cdot(-1)\\ 4 &= 48\cdot(-5) + 122\cdot 2\\ 2 &= 48\cdot 28 + 122\cdot(-11) \end{aligned}$$

e)
$$\begin{aligned} 735 &= 141\cdot 0 + 735\cdot 1\\ 141 &= 141\cdot 1 + 735\cdot 0\\ 30 &= 141\cdot(-5) + 735\cdot 1\\ 21 &= 141\cdot 21 + 735\cdot(-4)\\ 9 &= 141\cdot(-26) + 735\cdot 5\\ 3 &= 141\cdot 73 + 735\cdot(-14) \end{aligned}$$

f)
$$\begin{aligned} 984 &= 452\cdot 0 + 984\cdot 1\\ 452 &= 452\cdot 1 + 984\cdot 0\\ 80 &= 452\cdot(-2) + 984\cdot 1\\ 52 &= 452\cdot 11 + 984\cdot(-5)\\ 28 &= 452\cdot(-13) + 984\cdot 6\\ 24 &= 452\cdot 24 + 984\cdot(-11)\\ 4 &= 452\cdot(-37) + 984\cdot 17 \end{aligned}$$

INDEX

`&`, *see* ampersand, `&`
∧, 71
`:=`, 40, 51
`*`, asterisk
    in TEX, *see* TEX, asterisk in
    wildcard in filenames, *see* file name, wildcard (`*`) in
\, *see* character, backslash (\)
□, 80
□ (cursor), 43
`<=` (less than or equal to in Pascal), 44
¬, 71
∨, 71
`%` sign in TEX, *see* TEX, percent sign in
`#`, *see* pound sign, `#`
␣ (space character), 14, 43
∑, 47
`~`, tilde
    escape in email, 89
    user's home directory, 10

`a.out`, 37
Adobe Corporation, 79
Aho, Alfred V., 107
AIX, 6
alias
    mail, 88
    shell, 121
America Online, 127
American Mathematical Society, 83
American Standard Code for Information Interchange, *see* ASCII
ampersand, `&`, 20, 124
AMS, *see* American Mathematical Society
$\mathcal{AMS}$-TEX, *see* TEX, $\mathcal{AMS}$-TEX
`amsppt`, *see* TEX, AMS Preprint Style
`and` in Pascal, *see* Pascal, key word `and`
arithmetic
    Fundamental Theorem of, 67–69
arithmetic expression in Pascal, *see* Pascal, arithmetic expression in
arithmetic operators in Pascal, *see* Pascal, operators in, arithmetic
`array` in Pascal, *see* Pascal, key word `array`
ASCII, 8
ASCII file, *see* file, ASCII
assignment in Pascal, *see* Pascal, assignment in
asterisk in TEX, *see* TEX, asterisk in
Atrium, 8
    Computer Laboratory at Brooklyn College, 8

Web site of, 8
attack
symlink, 113
awk, 107

background
running jobs in the, 20
background, program running in, *see* program, running in background
backslash character, *see* character, backslash (\)
Backus, John, 36
bash, *see* shell, Bourne-again
.bash_profile file, 97
.bashrc file, 121
begin, *see* Pascal, key word begin
Bell Laboratories, 6
Berkeley Software Distribution, *see* BSD Unix
Berners-Lee, Tim, 20
\bf, *see* TEX, \bf
bg, 125
binary arithmetic, 40
binary file, *see* file, binary
bold face in TEX, *see* TEX, bold face in
bookmarks in Netscape, *see* Netscape, bookmarks
Boole, George, 69
boolean type in Pascal, *see* Pascal, boolean type in
Bourne shell, *see* shell, Bourne
Bourne-again shell, *see* shell, Bourne-again
breaking up Unix commands, *see* Unix commands, breaking up
Brooklyn College
Web site of, 20
BSD Unix, 6
buffers in vi, 16
named, 16
\bye, *see* TEX, \bye

C shell, *see* shell, C
C, programming language, 36
calendar
Gregorian, 5, 76
Julian, 76
Capital Lock key
dangers in *vi*, 16
caret, ^, 14, 84
carriage return character, 13
case, *see* Pascal, case statement in
cat, 9, 92
cd, 10
cd -, 124
centering lines in TEX, *see* TEX, centering lines in
\centerline, *see* TEX, \centerline
char type in Pascal, *see* Pascal, char type in
character
backslash (\), 78
end-of-line, 34, 42
character in Pascal, *see* Pascal, char type in
chmod, 18
CLI, *see* interface, command line
Command Line Interface, *see* interface, command line
command substitution, 110
comment character in shell, *see* shell, comment character
comment in Pascal, *see* Pascal, comment in
comment in TEX, *see* TEX, comment in
common divisor, 52
greatest, *see* greatest common divisor
compiler, 37

compound statement in Pascal, *see* Pascal, statement in, compound
compression of data, 31
    lossless, 32
    lossy, 32
computer
    —s, information on the Web about using, 20
    language
        compiled, 91
        interpreted, 91
conditional
    command in Bourne shell, *see* Bourne shell, conditional command in
    statement in Pascal, *see* Pascal, statement in, conditional
conjunction, 70
`const`, *see* Pascal, key word `const`
constant in Pascal, *see* Pascal, constant in
control key, *see* key, control
control sequence in TEX, *see* TEX, control sequence in
control variable in Pascal, *see* Pascal, `for` statement, control variable in
Control `[`, for *vi* escape, 12
Control-`c`, 80, 125
Control-`z`, 125
`cp`, 12
    `-p` option of, 32, 99
    `-r` option of, 12
⟨*CR*⟩, *see* carriage return character
creating directories, *see* directory, creating
`.cshrc` file, 121
`Ctl-[`, *see* Control-`[`, for *vi* escape
`Ctl-c`, *see* Control-`c`
`Ctl-z`, *see* Control-`z`
cursor, 43, 80
    movement in `vi`, *see* `vi`, cursor movement in

data compression, *see* compression of data
data type in Pascal, *see* Pascal, type in
DEC, 6
declaration in Pascal, *see* Pascal, declaration in
default, 26, 41, 80
    filename for compiled Pascal programs, *see* `a.out`
delimiters in TEX, *see* TEX, delimiters in
Digital Equipment Corporation, *see* DEC
Diophantus, 82
directory, *8*
    creating, 11
    print working, 10
    root, 10
    sub-, 10
disjunction, 71
displayed mathematical formulas in TEX, *see* TEX, mathematical formulas in, displayed
`div`, *see* Pascal, key word `div`
documenting Pascal programs, *see* Pascal, program, documenting of
domain name, 126
DOS, 37
dvi file, *see* TEX, dvi file
`dvips`, 79

e-mail, *see* email
`echo`, 97, 98
`echo`, 110
`ed`, 95

editing Unix commands, *see* Unix, commands, editing with `fc`
editor, *see* Unix, editor
    text, 12
electronic mail, *see* email
Electronic Numerical Integrator And Calculator, *see* ENIAC
email, 86–96
    reading, 86–88
    sending, 88–96
    sentmark
        Pascal program for, *see* Pascal program for email sentmark
    shell escape in, 87
    shell script for writing, *see* script, shell, for writing email
    tilde escape in, 89
    vs. e-mail
        Knuth's essay on Web, 86
empty statement in Pascal, *see* Pascal, empty statement in
`end`, *see* Pascal, key word `end`
end-of-line character, *see* character, end-of-line
`\endhead`, *see* TEX, `\endhead`
ENIAC, 40
enter key, *see* key, enter
errors, programming, *see* programming errors
Euclid (of Alexandria), 53
Euclid's lemma, 68
Euclidean algorithm, 5, 52–54
    extended, 59–60
        implementing in Pascal of the, 61–66
    implementing in Pascal of the, 54–56
`ex`, 95
executable files
    locations for, 96
`exit`, 125
exponent in TEX, *see* TEX, exponent in
`export`, shell command, 97, 124, 128
extended Euclidean algorithm, *see* Euclidean algorithm, extended

`fc`, 16
`FCEDIT`, *see* shell variable `FCEDIT`
Fermat's Last Theorem, 82
    Web site to read about, 83
Fermat, Pierre, 82
    Last Theorem of, *see* Fermat's Last Theorem
`fg`, 125
file, *8*
    ASCII, 8
    binary, 8
    compression, *see* compression of data
    executable, 18
    GIF, *26*, 32, 33
    handling in Pascal, 57–59
    hidden, 21, 88, 97
    JPEG, 32, *33*
    name
        extension, 37
        unique in shell scripts, 103
        wild card (`*`) in, 119
        wildcard (`*`) in, *12*
    owner of, 17
    readable, 18
    text, 8
    type text in Pascal, *see* Pascal, key word `text`
    writable, 18
`file`, Unix command, 8

`find`, 120
`for`, *see* Pascal, `for` statement in
foreground
    running jobs in the, 20
FORTRAN, 36
Free Software Foundation, 7, 38
    Web site of, 38
FreeBSD, 6
    Website of, 6
Fundamental Theorem of Arithmetic, *see* arithmetic, Fundamental Theorem of

Gauss, Karl Friedrich, 67
GCD, *see* greatest common divisor
geekspeak, 29
ghostview, PostScript viewer, 30
GIF file, *see* file, GIF
GNU, 38
GNU Pascal
    differences in behavior from SUN Pascal, 70
Graphical User Interface, *see* interface, graphical user
graphical user interface, 19
greatest common divisor, 52
Gregorian calendar, *see* calendar, Gregorian
Gregory XIII, Pope, 76
`grep`, 105
    `-v` option, 112
GUI, *see* graphical user interface, *see* interface, graphical user
`gunzip`, 32
`gzip`, 32

`\head`, *see* TEX, `\head`
header page, *see* `lpr`, header page, suppressing
here document, *see* script, shell, here document
Hewlett-Packard, 6
hidden file, *see* file, hidden
`HOME`, *see* shell variable `HOME`
host name, 126
HP-UX, 6
HTML, 5, 19, 23–29
    4.0 specifications on the Web, 29
    accented letters in, 26
    anchor in, 26
    attribute, 26
    attribute values
        `<LEFT>` of `<ALIGN>`, 26
        `<LEFT>` of `<CLEAR>`, 28
    attributes
        `<ALIGN>`, 26
        `<ALT>`, 26
        `<BORDER>`, 26
        `<CLEAR>`, 28
        `<HSPACE>`, 26
        `<SRC>`, 26
    centering phrases in, 26
    comment in, 26
    diacritical marks in, *see* HTML, accented letters in
    element, 26
    heading in, 26
    horizontal rule in, 26
    hyperlink in, 27
    including images in, 26
    legal issues, 29
    line break in, 26
    paragraph in, 28
    tag, 25
        paired in, 25
        solitary in, 26
    tags
        `<BODY>`, 25
        `<BR>`, 26
        `<CENTER>`, 26
        `<H1>`–`<H6>`, 26
        `<HEAD>`, 25

<HR>, 26
<HTML>, 25
<IMG>, 26
<TITLE>, 25
target too long in, 28
title in, 25
Uniform Resource Identifier in, 28
Uniform Resource Locator in, 28
URI in, *see* HTML, Uniform Resource Identifier in
URL in, *see* HTML, Uniform Resource Locator in
hung window, *see* window, locked up
Hypertext Markup Language, *see* HTML

IBM, 6
identifier
seePascal, identifier, 38
if
command in in Bourne shell, *see* shell, Bourne, if command in
statement in Pascal, *see* Pascal, statement in, if
inclusive *or*, 68
increment, 47
induction
mathematical, 69
infinite descent
method of, 69
Information Theory, 31
Infoworld Electric, Web site of, 5
input
standard, 57
input in Pascal, *see* Pascal, input in
integer overflow in Pascal, *see* Pascal, integer overflow in
integer type in Pascal, *see* Pascal, integer type in
interface
command line, 100
graphical user, 100
International Business Machines, *see* IBM
International Standards Organisation (ISO)
Pascal Report, 39, 70
Internet, 19
IP address, 126
dynamic allocation of, 126
static allocation of, 126
ISO, *see* International Standards Organisation (ISO)
\it, *see* TEX, \it
italics in TEX, *see* TEX, italics in

job control, 124–126
jobs, 125
JPEG file, *see* file, JPEG
Julian calendar, *see* calendar, Julian
Julius Caesar, 76
justified
right, 46

Kernighan, Brian W., 107
key
capitals lock, 13
control, 12, 80
enter, 9, 13
escape, 12
return, 9, 13
shift, 80
kill, 125
killing programs, *see* program, terminating
Knuth, Donald Ervin, 78, 86
Korn shell, *see* shell, Korn
.kshrc file, 121

languages, programming, *see* programming languages
Latex, *see* TEX, Latex
LCM, *see* least common multiple
leap year, 76
least common multiple, 52
⟨*LF*⟩, *see* line feed character
line feed character, 13
linear combination, 59
Linux, 5, *6*
    information on the Web about, *20*
locations for executable files, *see* executable files, locations for
`lock` file of Netscape, *see* Netscape, `lock` file of, *see* Netscape, `lock` file of
locked-up window, *see* window, locked up
log file in TEX, *see* TEX, log file
logic, propositional, 69
logical operators in Pascal, *see* Pascal, operators in, logical
`.login` file, 97
login name, 7
loop
    nested, 71
loop in Pascal, *see* Pascal, loop in
`lpr`, 80
    header page
        suppressing, 80
`ls`, 9
    `-F` option of, 9
    `-a` option of, 21
    `-d` option of, 12
    `-l` option of, 17
Lucent Technologies, 6
lynx, 26
machine language, 36
magnification in TEX, *see* TEX, magnification in
`\magstep`, *see* TEX, `\magstep`
mail alias, *see* alias, mail
`.mailrc` file, 88
maintenance of programs, *see* program, maintenance of
`man`, 9
manual
    on-line, 9
mathematical formulas in TEX, *see* TEX, mathematical formulas in
mathematical induction, *see* induction, mathematical
mathematical italics in TEX, *see* TEX, italics, mathematical, in
mathematical typesetting, *see* typesetting, mathematical
McGough, Nancy, 5
metacharacter in *vi*, 14
    escaping, 14
method of infinite descent, *see* infinite descent, method of
Microsoft Corporation, 7
    finding of facts in the antitrust case, 7
Mindspring, Internet service provider, 127, 129
mistake in TEX files, *see* TEX, mistake in files
`mkdir`, 11
`mod`, *see* Pascal, key word `mod`
`more`, 105
mouse, 19–20
Mozilla, 35
Mozzochi, Charles Jeffrey, 24, 27
`mv`, 11

negation, 70

nested loops, *see* loop, nested
NetBSD, 6
Netscape, 19–22
    bookmarks, 20
    cache of, 21
    command line control of, in Unix, 34
    empowering, 29–35
    link, 20
    `lock` file of, 21, 108
    quitting, 20
    shift-click in, 31
Neumann, John von, *see* von Neumann, John
New York Public Library
    remotely connecting to computers from, 129
`not` in Pascal, *see* Pascal, key word `not`
number
    binary, 18
    octal, 18

object program, 39, 91
Open Source Software, 6
OpenBSD, 6
OpenWindows, 19
option, in Unix commands, 11
or
    inclusive, 68
`or` in Pascal, *see* Pascal, key word `or`
ordinal type in Pascal, *see* Pascal, ordinal type in
output
    standard, 57
output in Pascal, *see* Pascal, output in
output stream, 107

packed character in Pascal, *see* Pascal, packed character in
`\par`, *see* TEX, `\par`
paragraph in TEX, new, *see* TEX, paragraph in, new
Pascal, 5, 36–51
    arithmetic expression in, 42
        priority rules for, 42
    assignment in, 40, 42
    body of program, 38
    boolean type in, 69
    `case` statement
        selector in, 74
    `case` statement in, 74
    char type in, 73
    character in, *see* Pascal, char type in
    comment in, 51
    compiling programs, 37
    constant in, 51
    declaration in, 40
    empty statement in, 41
    file-handling in, 57–59
    `for` statement
        control variable in, 73
    `for` statement in, 73
    identifier, 38
    input in, 43
    integer overflow in, 48
    integer type in, 40
    key word, *see* Pascal, word symbol
    key word `and`, 71
    key word `array`, 73
    key word `begin`, 38, 44
    key word `const`, 51
    key word `div`, 42, 63
    key word `end`, 38, 44
    key word `mod`, 42, 54
    key word `not`, 70
    key word `or`, 71
    key word `program`, 38
    key word `readln`, 43
    key word `reset`, 58
    key word `text`, 93

key word `var`, 40, 42
key word `write`, 42, 93
key word `writeln`, 38, 40, 42
loops in, 44–47
operators in
arithmetic, 42
logical, 70, 71
ordinal type in, 73
output in, 38
packed character in, 73
program
documenting of, 51
for email sentmark, 93
tracing of, 45
truth tables, for, 69–72
program heading, 38
reading from a file, 58
real type in, 42
`repeat` statement in, 70
reserved word, *see* Pascal, word symbol
running programs, 38
semicolon in, 41
separator in, 38
statement in, 38
compound, 44
conditional, 48–51
`if`, 49
type in, 40
variable in, 40
versions of, *see* GNU Pascal *and* Sun Pascal
`while ... do` statement in, 44
word symbol, 38
`writeln`
widths of fields, 46
Pascal, Blaise, 36
password, 7
changing, 8
`PATH` shell variable, *see* shell variable `PATH`
Perl, 91, 119
permission codes, 17–18
Personal Identification Number, *see* PIN number
PID, *see* process-id
`pidof`, to find out process-id, 107
PIN number
in file names, 29
pipe in Unix, 88, 105
pixel, 32
Plain TEX, *see* TEX, Plain
Pope Gregory XIII, *see* Gregory XIII, Pope
portability of programs, *see* program, portability of
porting programs, 38
PostScript, 30, 79
viewer, *see* ghostview
pound sign, `#`, 122
ppp, 127
previewing TEX, *see* TEX, previewing
prime factorization, 67
uniqueness of, 67
prime number, 67
print working directory, *see* directory, print working
priority rules for arithmetic expression in Pascal, *see* Pascal, arithmetic expression in, priority rules for
process in Unix, *see* Unix, process
process-id, 103
`.profile` file, 97
program
maintenance of, 50
portability of, 50
running, 125
running in background, 79, 124
running in foreground, 124
stopped, 125

terminating, 80, 125
`program`, *see* Pascal, key word `program`
programming errors
logic, 39
syntax, 39
programming languages, 36
prompt
shell, 7
primary, 34
secondary, 34
propositional logic, *see* logic, propositional
propositional variable, *see* variable, propositional
`pwd`, 10

quitting Netscape, *see* Netscape, quitting
quotes in TEX, *see* TEX, quotes in

`read`
to read from a file, *see* Pascal, reading from a file
`readln`, *see* Pascal, key word `readln`
real type in Pascal, *see* Pascal, real type in
Red Hat, Linux distributor, 127, 129
Register, The, Web site of, 5
remote access, 126–129
remote computer connections from the New York Pulic Library, *see* New York Public Library, remotely connecting to computers from
`repeat`, *see* Pascal, `repeat` statement in
repetitive statement, *see* Pascal, loop in
`reset`, *see* Pascal, key word `reset`
return key, *see* key, return
right justified, *see* justified, right
`rlogin`, 128
`rm`, 11
`-f` option of, 22
`-i` option of, 22
`-r` option of, 11
`-rf` options of, 22
`rmdir`, 11
root directory, *see* directory, root
Rowe, James, 5
`rsh`, 126, 129

script
shell, 90–98
for writing email, 93
here document, 95
source, 121–124
search path, 97
secret Web pages, *see* Web pages, secret
selector, in `case` statement in Pascal, *see* Pascal, `case` statement, selector in
semantics, 44
semicolon in Pascal programs, *see* Pascal, semicolon in
separator in Pascal, *see* Pascal, separator in
Shannon, Claude E., 31
shell, 7
alias, *see* alias, shell
bash, *see* shell, Bourne-again
Bourne, 91
conditional command in, 111
`if` command in, 111
Bourne-again, 91
C, 91
comment character, 91, *106*

escape character !, 15, 87
Korn, 91
script, *see* script, shell
variable, 92
`FCEDIT`, 17
`HOME`, 92
`PATH`, 97
shift-key, *see* key, shift
`\sl`, *see* TEX, `\sl`
slanted in TEX, *see* TEX, slanted in
slash, 9
small capitals, 83
`\smc`, *see* TEX, `\smc`
Solaris, 6
`source`, 122
source program, 39, 91
source script, *see* script, source
standard input, 57, 102
standard output, 57, 102
statement in Pascal, *see* Pascal, statement in
stored program computer, 40
stuck window, *see* window, locked up
subdirectory, *see* directory, sub-
subscript in TEX, *see* TEX, subscript in
substitute command
of `ed`, 108
of `vi`, 14
Sun Microsystems, 6
SUN Pascal
differences in behavior from GNU Pascal, 70
SunOS, 6
superscript in TEX, *see* TEX, superscript in
superuser, 127
symlink attack, *see* attack, symlink
syntax, 44
<*tab*>, tab character, 110
`tail`, Unix command, 128
`-f` option of, 128
technical jargon, online dictionary of, 29
`telnet`, 128, 129
temporary files, directory for, *see* `tmp`, directory for temporary files
terminal type, 128
vt100, *see* vt100
terminating programs, *see* program, terminating
`test`, Unix command, 112
`-z` option of, 112
TEX, 5, 78–85
`*` in, *see* TEX, asterisk in
`\$` in, 85
`%` sign in, *see* TEX, percent sign in
`\%` in, 85
AMS Preprint Style, 83
$\mathcal{AMS}$-TEX, 83
asterisk in, 81
`\bf`, 84
bold face in, 84
`\bye`, 78, 85
centering lines in, 85
`\centerline`, 85
comment in, 85
control sequence in, 78
control sequence `\TeX` in, 78
delimiters in, 83
dvi file, 79
`\endhead`, 83
exponent in, 84
`\head`, 83
`\it`, 84
italics
mathematical, in, 84
italics in, 84
Latex, 83
log file, 79

magnification in, 83
\magstep, 83
mathematical formulas in, 84
mathematical formulas in, displayed, 84
mistake in files, 80
\par, 84
paragraph in, new, 84
percent sign in, 85
Plain, 83
previewing, 79
quotes in, 83
\sl, 84
slanted in, 84
\smc, 83
subscript in, 84
superscript in, 84
\tt, 85
typesetting files in, 78
typewriter font in, 85
Unix directory tree, and, 85
Users Group, 85
Web site of, 85
Web sites about, 85
xdvi, previewer, 79
\TeX, *see* TEX, control sequence \TeX in
text file, *see* file, text
text file type in Pascal, *see* Pascal, key word text
The Register, *see* Register, The
tilde escape in email, *see* email, tilde escape in
tmp, directory for temporary files, 112
touch, 23
tracing Pascal programs, *see* Pascal, program, tracing of
truth tables in Pascal, *see* Pascal, program, truth tables, for
\tt, *see* TEX, \tt
Turing, Alan, 36
type
text in Pascal, 58
type in Pascal, *see* Pascal, type in
typesetting
mathematical, 78–85
typesetting TEX files, *see* TEX, typesetting files in
typewriter font in TEX, *see* TEX, typewriter font in

Ultrix, 6
uncomment, 65
underscore, _, 9, 84
Unices, plural of Unix, 111
Uniform Resource Identifier, *see* HTML, Uniform Resource Identifier in
Uniform Resource Locator, *see* HTML, Uniform Resource Locator in
unique file names, *see* file name, unique in shell scripts
universal machine, 36
Unix, 5–18
commands
breaking up, 34
editing with fc, 16
compared to Windows NT, 7
directory tree and TEX, *see* TEX, Unix directory tree, and
editor
ed, *see* ed
ex, *see* ex
vi, *see* vi
process, 103
quote character backslash (\), 14, 34
URI, *see* HTML, Uniform Resource Identifier in

URL, *see* HTML, Uniform Resource Locator in
user groups, 17

v. Neumann, *see* von Neumann, John
`var`, *see* Pascal, key word `var`
variable
    of type `text` in Pascal, 58
    propositional, 69
variable in Pascal, *see* Pascal, variable in
variable in shell, *see* shell variable
`vi`, 6, 12–15
    command mode, 12
    command-line mode, 14
    cursor movement in, 12
    insert mode, 12, 13
    line numbers in, 14
    quitting, 15
`view`, 127
von Neumann architecture, 40
von Neumann, John, 40
vt100, 128

Web, *see* World Wide Web
    World Wide, *see* World Wide Web
Web page
    how to build a, 22–29
    permission codes for, 23
Web pages
    secret, 29
Web site, 20
Web site of the World Wide Web Consortium, *see* World Wide Web Consortium, Web site of
Weinberger, Peter J., 107
`whereis`, Unix command, 31
`which`, Unix command, 31
`while ... do`, *see* Pascal, `while ... do` statement in
`whoami`, 11
width in `writeln` statements, *see* Pascal, `writeln`, width of fields
Wiles, Andrew J., 83
window
    locked up, 125
Windows, 7
    95, 7
    98, 7
    2000, 7
    NT, 7
        compared to Unix, *see* Unix, compared to Windows NT
Wirth, Niklaus, 36
World Wide Web, 5, 19
World Wide Web Consortium, 28
    Web site of, 29
`write`, *see* Pascal, key word `write`
`writeln`, *see* Pascal, key word `writeln`, *see* Pascal, key word `writeln`
writing to a file in Pascal, *see* Pascal, writing to a file

X windows, 19
`xdvi`, TEX previewer, *see* TEX, `xdvi`, previewer
`xhost`, 126
`xterm`, 124

YMMV, 29