TALKING WITH COMPUTERS

Explorations in the Science and Technology of Computing

THOMAS DEAN
Brown University, Providence, Rhode Island

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2004

Printed in the United States of America

Typefaces Stone Serif 9.5/13.5 pt., Optima, and Lucida Typewriter *System* LaTeX 2_ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Dean, Thomas L., 1950–
Talking with computers : explorations in the science and technology of computing / Thomas Dean.
p. cm.
Includes bibliographical references and index.
ISBN 0-521-83425-2 – ISBN 0-521-54204-9 (pb.)
1. Computer Science. 2. Computer programs. I. Title
QA76.D3333328 2004
946–dc22 2003065531

ISBN 0 521 83425 2 hardback
ISBN 0 521 54204 9 paperback

Contents

CHAPTER ONE

Talking with Computers

Hardly a day goes by that I don't write at least one short computer program: a few lines of code to explore an idea or help organize my thoughts. I think of it as simply talking with my computer, and more and more often there is a computer available to talk with, often several of them joining in the conversation simultaneously. Each time you click on a link in a browser, you cause a sequence of computations involving dozens if not hundreds of computers scattered all over the world.

Making a computation happen is not, however, the same thing as programming. There are lots of powerful programs written by talented programmers that you can call up with a click of a mouse or few keystrokes. These programs animate computers, breathing life and spirit into lumps of metal and plastic. Even if you know what's going on inside computers and computer programs, it's easy to imagine that programs are spells and the programmers who create them are sorcerers. When you click on the icon for a program, you invoke these spells and the spells conjure up spirits in the machine. But this book isn't about invoking the spells of others; it's about creating your own spells and conjuring spirits of your own design.

This is not to say I won't encourage you to use code written by other programmers. Quite the contrary: an important part of the power of computing is that good spells can be reused as often as needed. Programming is about weaving together the spells of others, conjuring your own spirits, and animating the computer to dance to your bidding. This book is about practical conjuring, about revealing what's behind some of the magic associated with computing while at the same time learning to create your own magic. And the best way to begin a book about computing and computer programming is to sit down in front of a computer and start programming.

1.1 COMPUTERS EVERYWHERE

The laptop computer on our breakfast table is connected to a wide-area network (usually called the World Wide Web or the Internet) through a local-area wireless network (our house) and a broadband connection supported by a local cable television company. I leave the laptop on the table so I can read the headlines from the online news services or check the weather while I eat breakfast. I also use it to write and run small programs.

I use a program on my laptop (it's called `ssh` for "secure shell") to tunnel through the firewall protecting the computers in the computer science department at Brown and open a shell (a special program that lets me interact more or less directly with the operating system – a variant of Unix in this case) on the machine sitting in my office (its name, by the way, is "klee" for the artist Paul Klee – see Figure 1.1 for the inspiration for this naming – and its symbolic address on the Internet is "klee.cs.brown.edu").

When I say "open a shell," I mean that I make a window appear on my laptop screen into which I can type commands to be interpreted by the shell program. When I say that a program "interprets" a command, I mean that the program reads the command I've typed and converts it into instructions that the computer can carry out, thereby executing the command. The results of executing the command, usually one or more lines of text, are then displayed in the same window as the command was typed. The shell lets me write and run programs to do all sorts of routine tasks from checking football statistics to keeping track of all of my email messages, digital photos and music files.

The program `ssh` allows me to work remotely on computers that "trust me" in such a way that the information sent back and forth between my laptop and klee can't be deciphered by someone with access to the wires on which the information is transmitted and doesn't allow a malicious hacker to break into either my laptop or klee. I could open a shell on any of several hundred machines residing within the firewall, but I generally choose to do it on my own machine rather than slow down or "steal cycles" from a machine being used by someone else.

The time will soon come however when it won't make much sense to talk about "my machine" – computation will become as pervasive as indoor plumbing. The Internet has blurred the distinction among individual computers. I'm almost always connected to the Internet, but most of the time I don't think about what computer I'm talking with. When I'm in the department at Brown but not in my office, I walk around with my laptop connected to the department wireless network, which connects to a wide-area network and then to the Internet. Right this minute I'm working at my laptop, typing into a shell that's running on the

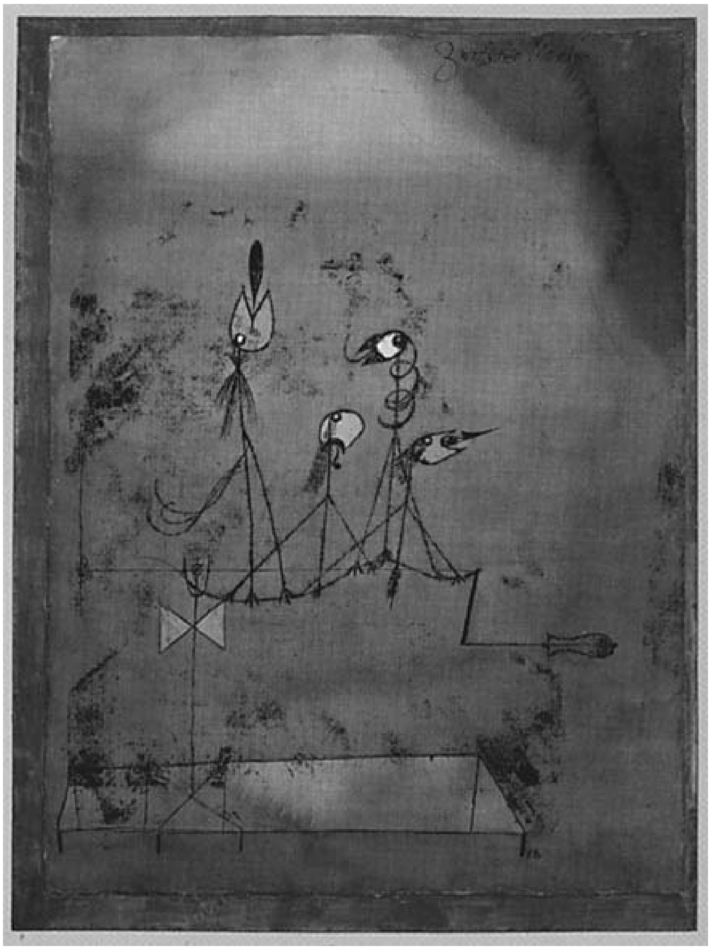

Figure 1.1: Paul Klee's "Twittering Machine" (1922) ©2003 Artist Rights Society (ARS) New York, VG Bild-Kunst, Bonn Digital Image ©The Museum A Modern Art/Licensed by SCALA/Art Resource, NY

computer in my home office a few feet away, but in another window running on my laptop I'm connected to klee. For all I know, the data that's flowing between these computers may be circling the globe, zipping through cables under the ocean and bouncing off satellites along the way. Indeed, I could pretty easily force the data to go through Zurich, Seattle or Tokyo.

Given the current state of the art, though, I do have to think a bit about where I am, or rather where the program is that's currently interpreting my keystrokes. The reason I have to know which computer I'm working on is that different machines have different software, offer different services and have access to different sources of data. I'm pretty confident that I won't lose data that's stored on the machines in the department because I trust the folks who maintain those machines and perform the backups on the file system there. But I have to do the backups on my laptop and the machine in my home office myself, and I know I'm not very careful about doing them.

Eventually, with the exception of very specialized programs and services, I won't have to worry about what computers are running the programs I need. This is already true to a certain extent if you restrict your computing to what you can do within a web browser, and yes, you can do a lot of useful computing within a web browser. Some people don't even distinguish between their web browser and their computer; they do everything – email, news, shopping, entertainment and education – from within their web browser.

For the last twenty years, I've been using programs to work on computers thousands of miles away. In the early '90s, it seemed miraculous to be sitting in a Paris hotel room running programs on the computer in my office in Providence or telling a computer at Stanford to transfer files to the portable computer on my bed in the hotel room. Today, most "netizens" take this amazing connectivity for granted and, though they may not know the magic incantations that animate these processes, they routinely run programs on remote computers and fetch files with the click of a mouse.

1.2 EVERYDAY MAGIC

I want to give you some examples of everyday programming, not fancy programming, just examples of talking with computers and getting them to do interesting things. I'll use the phrase "invoking a program" to mean making a program run, usually by typing its name and then zero or more expressions or "arguments" that provide additional direction or information. Invoking programs with specific arguments is one of the simplest ways to talk with a computer.

In summer 2002, I kept a journal to record ideas for this book. I put the journal entries in a collection of files and directories on klee. Here I'm invoking a program called `wc` (for "word count") by typing into a shell running on klee in order to see how much I wrote in my journal during August:

```
/u/tld/email/book % wc -l ./journal/02/08/*/*.txt
     465 ./journal/02/08/01/day.txt
     323 ./journal/02/08/02/day.txt
     207 ./journal/02/08/04/day.txt
     445 ./journal/02/08/08/day.txt
     215 ./journal/02/08/12/day.txt
     299 ./journal/02/08/16/day.txt
     700 ./journal/02/08/24/day.txt
     335 ./journal/02/08/30/day.txt
     857 ./journal/02/08/31/day.txt
    3846 total
```

The `/u/tld/email/book %` part was printed by the shell. It's called the "prompt" and when I'm in the shell window (the portion of my computer screen dedicated to the shell) the cursor is positioned at the end of the prompt waiting for me to type something. I've modified the shell – the shell is itself programmable – so that the prompt always displays the default directory in which the shell looks for files.

When I'm finished typing I signal the shell, usually by hitting the "return" (or "enter") key on my keyboard, to interpret what I just typed. The directory `/u/tld/email/` is where I generally store files related to my daily activities. `/u/tld/email/book/` is the temporary directory I created for files related to working on this book.

I typed `wc -l ./journal/02/08/*/*.txt` and then hit the return key as part of my conversation with the shell and so indirectly with the operating system running on klee. More often than not when you invoke one program, that program invokes another program, and that program another, and so on, with some programs possibly invoking several other programs at once. A computer operating system is just another program, really a collection of many programs written (and rewritten) by many different people. You can think of the operating system as the accumulated wisdom of a host of very clever programmers who packed it with everything they felt was fundamentally useful for building other programs.

Other programs, applications such as web browsers and word processors, are run "on top of" or "under the control of" the operating system. The operating system sees all and controls all; it's only through the operating system that your programs can get information from the outside world (through a local network or the World Wide Web) or send files to printers or grab data stored on disks or CDs. If this seems mysterious, don't worry; it really is complicated. The good news is that for the most part you don't have to understand the details, since the operating system hides a lot of the computer's complexities from the programmer. This ability to hide complexity is essential in developing large complicated programs and makes learning to program much easier.

The specific command I typed told the shell to run the program `wc` to count lines (the `-l` argument) in the files specified by the pattern `./journal/`

02/08/*/*.txt, where * is a “wildcard” that matches any string of characters. With no perceptible pause, the shell printed out the next ten lines, which you can think of as the answer to my question or the result of the computation. The specified pattern matched nine files. Each of the first nine lines contains the name of a file that matched the pattern preceded by the number of lines of text in that file. For the first file listed, the first * matched 01 and the second * matched day. The last line is the total number of lines of text in all of the files.

Let me say a few words about file systems and the strange strings of characters containing slashes (/). A slash with no preceding text indicates the “root” directory; as far we’re concerned, everything is stored under the root of the file system. The u in /u/ is a symbolic link to the /home/ directory on the Brown file system where the directories and files of computer users like me are stored. For the most part, symbolic links are invisible to users but allow system managers to handle large file systems more efficiently and transparently. The /u/tld/ designates my home directory, where all my files are stored; my login name is tld for the initials of my name, Thomas Linus Dean.

Most computer file systems are organized hierarchically. So, for instance, my email directory /u/tld/email/ is one of many files and directories stored in my home directory, and the directory /u/tld/email/book/ is one of many files and directories stored in my email directory.

Files can be named *absolutely* with respect to the root directory or *relatively* with respect to some other starting directory. When you’re in the directory /u/tld/email/book/, ./journal/02/08/30/day.txt is a shorthand reference (or *relative path name*) for /u/tld/email/book/journal/02/08/30/day.txt, which is the full name (or *absolute path name*) for the file. I keep all files for journal entries written in 2002 in ./journal/02/, all files for August 2002 in ./journal/02/08/, and all files for 30 August 2002 in ./journal/02/08/30/. If I had typed wc -l ./journal/02/*/01/*.txt, the shell would have reported on all journal entries written on the first day of some month in 2002.

Absolute and relative path names can be confusing until you’ve played with them a bit, and even then you can easily get lost in a file system consisting of thousands of directories, in the same way that you can get lost navigating in a collection of web pages. For the most part, however, the nested, hierarchical directory structure makes it relatively easy to keep track of where you are and is a useful way to organize all sorts of data (including web pages). Consider these files from my journal directory:

```
/u/tld/email/book/journal/02/year.txt
/u/tld/email/book/journal/02/year.htm
/u/tld/email/book/journal/02/08/month.txt
```

```
/u/tld/email/book/journal/02/08/month.htm
/u/tld/email/book/journal/02/08/30/day.txt
/u/tld/email/book/journal/02/08/30/day.htm
/u/tld/email/book/journal/02/08/31/day.txt
/u/tld/email/book/journal/02/08/31/day.htm
/u/tld/email/book/journal/02/09/month.txt
/u/tld/email/book/journal/02/09/month.htm
/u/tld/email/book/journal/02/09/01/day.txt
/u/tld/email/book/journal/02/09/01/day.htm
/u/tld/email/book/journal/02/09/02/day.txt
/u/tld/email/book/journal/02/09/02/day.htm
```

When listed this way, it's hard to discern the organizational structure inherent in calendars, though it's there in the absolute path names if you look hard enough. The underlying structure is similar to a tree, with files corresponding to leaves and directories corresponding to branches. Figure 1.2 shows these files as a tree (or, rather, the branch of the tree called `/u/tld/email/book/journal/02/`). You can think of Figure 1.2 as grafted onto the tree rooted at `/` at the branch `/u/tld/email/journal/`. The same basic tree-like structure that underlies hierarchical file systems in most modern operating systems appears again and again in computer science.

The incantation `wc -l ./journal/02/08/*/*.txt` really is a program of sorts, albeit a short and rather cryptic one. That this short program called another program `wc` is not at all unusual: most programming languages provide access to all sorts of specialized programs. Even `+` in a language that allows `1 + 2` is a program (and not a simple one if you understand how computers handle arithmetic).

Shells and other means of interacting with operating systems offer a wide range of powerful programs that can be orchestrated to perform tasks. For example, the next program (called a *shell script*) renames all files with the extension `html` to have the extension `htm`. The HTML ("hypertext markup language") files that comprise web sites are conventionally identified using either the three-letter extension `htm` or the four-letter extension `html`. Both conventions are common in practice. Unfortunately, some programs require one or the other exclusively and if your files are in the wrong format, you have to convert them. I used the program `ls` (for "list directory contents") to list all of the files in the current directory prior to executing the program to rename the files. After executing the program, I used `ls` again to show that the shell script worked as advertised. In the remainder of this chapter, I've simplified the prompt to just `%`; each appearance of `%` signals the beginning of another typed command.

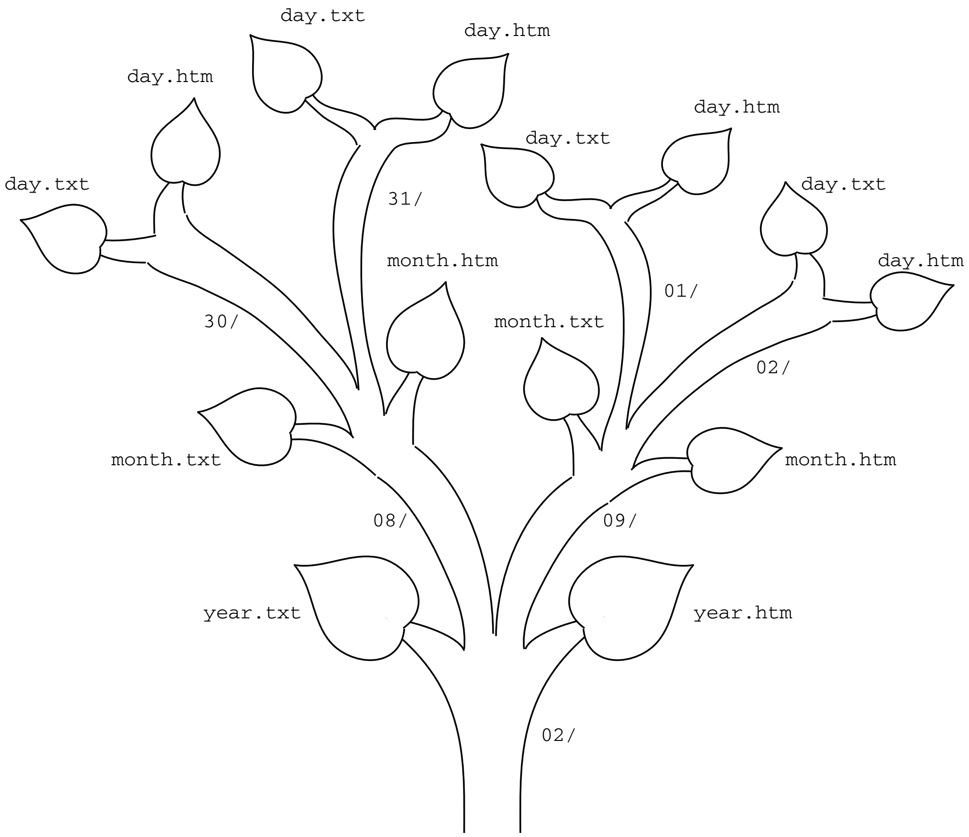

Figure 1.2: Files and directories organized hierarchically in a tree-like structure

```
% ls
home.html     syllabus.html
% ls | sed 's/html//' | awk '{print "mv " $1 "html " $1 "htm"}' | sh
% ls
home.htm      syllabus.htm
```

The program starts by listing the set of files with the extension `html`.[1] Unix programmers call the vertical bars (|) "pipes": they convert the output of one program, `ls` in this case, into the input to another program. The `sed 's/html//'`

[1] In this example, all the files in the current directory have the extension `html`. If they didn't, we could modify the shell script by telling `ls` to list only files with the extension `html`. For example, substituting `ls *.html` for `ls` would do the trick here. We'll learn more about shells and shell scripts in Chapter 2.

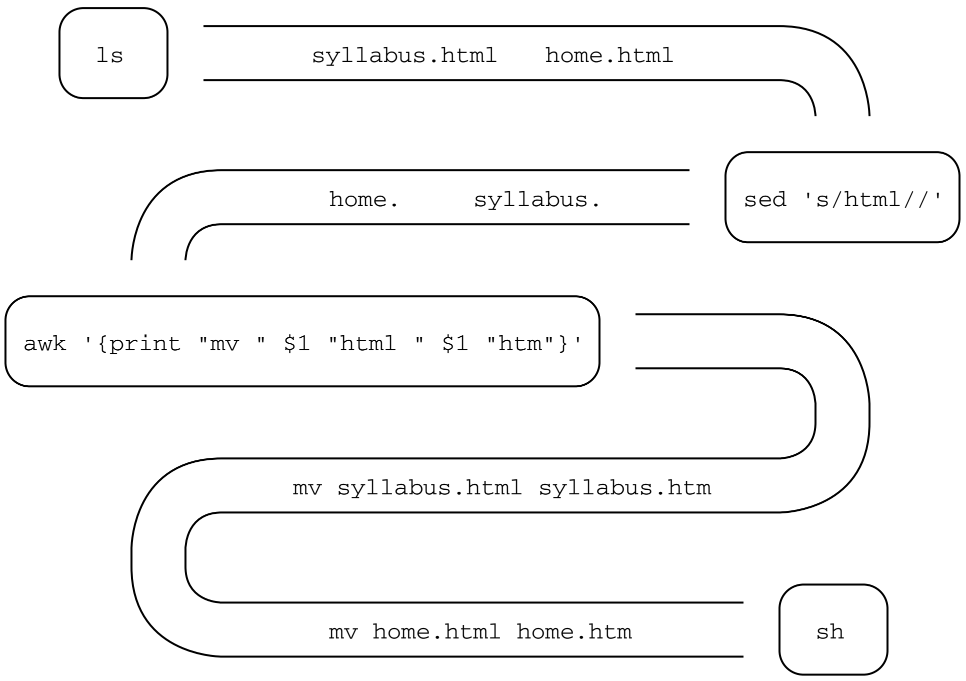

Figure 1.3: Intermediate results flowing through pipes connecting one command to the next in a shell script

part of the program takes each file name in turn and rips off the `html` part; the output of `sed 's/html//'` is two truncated file names `home.` and `syllabus.`. The next `|` causes these file names to be piped into the program fragment `awk '{print "mv " $1 "html " $1 "htm"}'` that essentially writes two little programs that are themselves shell scripts and look like `mv home.html home.htm` and `mv syllabus.html syllabus.htm` (`mv` is the "move" or "rename" command and requires you to specify both the original and the new names of the file you're renaming). The output of `awk '{print "mv " $1 "html " $1 "htm"}'` is fed into the program `sh` (yet another shell – remember we're already typing to one shell) via the last `|`. Figure 1.3 illustrates how the intermediate results from the different steps in this computation are piped from one step to the next.

If you think about it, this little program is pretty interesting despite its simple task. The program actually wrote a couple of littler programs, started up a shell and submitted those programs to the new shell to run, producing the desired outcome. Programs that write and run other programs and even replicate or improve upon themselves are relatively common, for example, computer viruses.

With a few simple modifications, this program could change the names of thousands of files stored in any number of directories and on any number of computers. With just a little more work, you could write a program that would go *inside* each of these files and change any reference in the text to a file with extension `html` to have the extension `htm`. If you were maintaining a web site with thousands of web pages spread across hundreds of directories, you might end up writing and running similar programs frequently.

By the way, there's always more than one way to solve any given programming problem, and I certainly don't claim that this shell script is the most elegant or efficient way of handling renaming. We could, for example, eliminate `awk` from our script and manage everything using `sed`, as in `ls | sed 's/\(.*\)html/mv \1html \1htm/' | sh`. However, the most compact program is not always the best: I find the original script clearer and more appealing, though I admit this is an aesthetic judgment.

Let's try something a little more complicated. Many web masters use a language called Perl both to maintain web pages and to do computations for visitors to their web pages. Suppose a bunch of your text files refer to dates in the format `mm/dd/yy` and you want to change them so as to name months explicitly, perhaps to avoid confusion with the alternative format `dd/mm/yy`. You might also want to try to clear up the ambiguity of the century implied when only two digits are used for the year. I just created a short text file of the sort I'd like to modify and I'll get the shell to print it out using `cat`, a Unix command for creating, displaying and stringing together (or *concatenating*) files:

```
% cat dates.txt
The date 1/1/00 should be changed,
as should 12/31/1999 and 1/1/2002,
but not the file /usr/local/bin/.
```

I've written a little Perl program in a file called `program.pl` to perform the conversion, and again I get the shell to print it out:

```
% cat program.pl
@month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
while ($_ = <ARGV>) {
  s"\b(\d\d?)/(\d\d?)/(?:19)?(\d\d)\b"$month[$1-1] $2, 19$3";
  s"\b(\d\d?)/(\d\d?)/(?:20)(\d\d)\b"$month[$1-1] $2, 20$3";
  print "$_";
}
```

Finally, I invoke my program indirectly by calling a program called `perl` with the name of the file in which I've stored my program as an argument. The `perl` program knows how to interpret programs written in the Perl language and `program.pl` has become another command that I can execute from the shell. Now I tell `perl` and the shell – in some cases several programs are reading and processing what I type – to take as input the contents of the file `dates.txt` and to store the output of the program, the result of executing the print statement in `program.pl` multiple times, in a file called `dates.out`:

```
% perl program.pl dates.txt > dates.out
```

I'll tell the shell to print `dates.out` so we can see what the program did:

```
% cat dates.out
The date Jan 1, 1900 should be changed,
as should Dec 31, 1999 and Jan 1, 2002,
but not the file /usr/local/bin/.
```

Not too exciting, and I'll bet that `program.pl` will get the year wrong as often as it gets it right, but you get the general idea. It's not important that you understand the exact syntax of the Perl program except to note that it uses a loop (the part of the program that begins with `while`) to read each line of the file, makes substitutions where appropriate and then prints out the line with the substitutions. The syntax for specifying substitutions is particularly terse in large part because Perl programmers use it so often. Programming languages, like natural languages, typically have short ways of saying things you want to say often.

My Perl program was long enough that I couldn't type it out on a single line, or at least not conveniently, so I put it in a file and submitted the file to the Perl interpreter, `perl`. The Perl interpreter is a program that converts programs written in Perl into commands that are carried out by other programs (such as those comprising the operating system) and ultimately into instructions that run directly on the hardware of a particular machine. You may have heard that computer programs have to be "compiled" into some other form before they can be "executed" but, while this is true, it's largely beside the point. The exact manner in which a piece of syntax such as `while ($_ = <ARGV>) { ... }` is converted into a form that can be handled by the primitive hardware of a particular machine is very complicated, and you don't need to know it in order to be an effective programmer.

Some people distinguish between programs called compilers and programs called interpreters. However, they both do basically the same thing: convert programs in one format into a different format that is more readily available for performing computations. Usually a programmer works in a *programming environment* made up of tools for editing, debugging, testing, using, and packaging programs. I prefer programming environments that let me interact easily with my programs while I'm developing them. Most modern programming environments allow some sort of interaction, and often there is little distinction between writing, running and using programs. Writing programs and using the tools that are part of a programming environment are just how you get your work done, whether you're working in biotechnology, analyzing data from an archaeological dig, or building a web site for your startup.

Different programming languages and programming environments support different styles of programming, different modes of interaction and different ways of thinking about computation. Here are some other languages and environments I use frequently.

1.3 HACKING IN MATHEMATICS

I use the Mathematica programming environment for all sorts of programming that involves mathematics. Most of the time I use the fancy graphical front end that lets me create two- and three-dimensional graphics. It's great for visualizing mathematical functions and analyzing data. But I also occasionally invoke Mathematica in a shell and use it as a fancy calculator. Mathematica can solve algebraic equations much better than I can with paper and pencil, and it can do symbolic differentiation and integration in a snap. It doesn't replace a good mathematics education so much as it augments it.

When I invoke the Mathematica program, called `mathematica` on my computer, it takes over from the shell and interprets what I type. Mathematica displays the prompt, `In[`*n*`]:=`, after interpreting the first $n - 1$ commands, thereby inviting me to type my *n*th command, and it then reads each subsequent line I type until it encounters a "shift return" (hold down the shift key and simultaneously hit the return key). At this point Mathematica attempts to make sense of my command, prints out the prefix `Out[`*n*`]=`, and displays the result of interpreting the *n*th command:

```
% mathematica
In[1]:= Solve[ x^2 - 4 == 0, x ]
Out[1]= {{x -> -2}, {x -> 2}}
In[2]:= Solve[ x^2 + 2 x - 7 == 0, x]
Out[2]= {{x -> -1 - 2 Sqrt[2]}, {x -> -1 + 2 Sqrt[2]}}
```

```
In[3]:= D[ x^3, x ]
Out[3]= 3 x^2
In[4]:= Integrate[ Cos[x], x]
Out[4]= Sin[x]
```

This exchange doesn't begin to show off what Mathematica can do, and the standard graphical user interface that most people use to interact with Mathematica is amazingly powerful; other products such as Maple and Matlab provide similar functionality. The neat thing about all these programs is that some very smart programmers have developed powerful tools that let us exploit their knowledge of mathematics.

You may have heard some pontificating math instructor berate students with "I've forgotten more mathematics than you'll *ever* learn," but at times I feel I've forgotten more mathematics than I ever learned. I barely remember that the indefinite integral $\int \cos\theta \, d\theta$ is equal to $\sin\theta$ and I have to think for a minute to calculate the roots of equations like $x^2 + 2x - 7 = 0$, but programs like Mathematica allow me to handle these problems and all sorts of mathematical gymnastics without retaining tomes of esoteric mathematical knowledge. Using Mathematica doesn't make me as good as a practicing mathematician, but I'm a lot better off than I would be with just a library of math textbooks.

Programs like Mathematica act as intelligence amplifiers, combining the functionality of supercalculators with that of electronic encyclopedias that place huge amounts of knowledge at your fingertips.[2] Just as a diesel engine amplifies physical strength and a telescope amplifies visual acuity, so carefully crafted programs can amplify intellectual power. We'll see several examples in the following chapters of how programs like Mathematica can help in writing programs that rely on mathematical concepts.

1.4 PROGRAMMING IN LOGIC

Prolog is a programming language in which programs are specified using statements in logic. Prolog is great for writing programs that depend on complex rules

[2] In 1945, Vannevar Bush, Director of the United States Office of Scientific Research and Development, wrote an *Atlantic Monthly* article called "As We May Think" that, anticipating the Internet, outlined how computers could be used to supplement our memories, provide ready access to all human knowledge, and amplify our mental abilities. J. C. R. Licklider (1960) espoused a similar view with further embellishment in his very readable "Man-Computer Symbiosis." Both these articles are now available in several history-of-computing archives on the World Wide Web, as anticipated by Bush and Licklider.

and relationships. If I were writing a program that computed employees' medical benefits or depended on the tax laws, I'd definitely write it in Prolog. Even if you never use Prolog to write a practical program, learning about it teaches you strategies for solving problems such as those involving complex rules. Indeed, programmers working on large programs written in Java or C++ often write a mini version of Prolog as part of the large program just to handle the parts of the problem involving complex rules. It's also not uncommon for a program written in one language to provide input to and accept output from a program written in another language; in this way a programmer can use the most appropriate language for the problem at hand.

Here we'll create a Prolog *database* by first asserting three simple facts: Fred is one of Anne's parents, Anne is one of Lucy's parents and Lucy is one of Bill's parents. Next we specify two general rules indicating that your parents are your ancestors, as are their parents. Then we query the database to see if Prolog gets it right.

Terms that begin with lowercase letters, like `fred`, are constants and those that begin with uppercase letters, like `X`, are variables. If you don't know the difference between constants and variables, think back to when you learned algebra: constants were typically numbers and variables were often denoted by letters like *x* or *y*. Variables in Prolog don't behave exactly like the variables in algebraic formulas (nor, for that matter, like the variables in other programming languages), but the analogy to algebra will work for our present purposes. Instead of *x* and *y*, in Prolog we'll use `X` and `Y`.

In the next interaction, the part corresponding to the assertion of the three facts and two rules should be pretty clear – I typed the strings beginning with `assert` and ending with a period. The period tells Prolog that I'm finished typing and that it should go ahead and try to interpret whatever preceded the period. Statements of the form `assert(`*`expression`*`)` correspond to assertions that end up in the Prolog database, and all other statements are interpreted as queries or requests for Prolog to answer questions about the information contained in its database.

Prolog responds to each individual assertion with "yes." The rule `ancestor(X, Y) :- parent(X, Y))` can be read as "`X` is the ancestor of `Y` if `X` is the parent of `Y`". The `:-` corresponds to "if." Similarly, `ancestor(X, Y) :- parent(X, Z), parent(Z, Y)` can be read as "`X` is the ancestor of `Y` if `X` is the parent of `Z` and `Z` is the parent of `Y`". The comma separating `parent(X, Z)` and `parent(Z, Y)` means "and."

The line `ancestor(fred, X)` is a query requesting Prolog to find assignments to the variable `X`, for example, assigning `X` the constant `anne`, so that, if you substitute the assigned constants for the indicated variables, the resulting expression, say `ancestor(fred, anne)`, would be true. We use this query to ask Prolog to list all Fred's descendants.[3] In response to this line, Prolog prints out a variable assignment, say `X = anne`, followed by a `?` asking if this assignment is the one I was looking for. In response to each `?`, I typed "no" and thereby made Prolog look for another assignment.

```
% prolog
| ?- assert( parent(fred, anne) ).
yes
| ?- assert( parent(anne, lucy) ).
yes
| ?- assert( parent(lucy, bill) ).
yes
| ?- assert( (ancestor(X, Y) :- parent(X, Y)) ).
yes
| ?- assert( (ancestor(X, Y) :- parent(X, Z), parent(Z, Y)) ).
yes
| ?- ancestor(fred, X).
X = anne ? no
X = lucy ? no
no
```

By repeatedly typing "no" to each assignment, I forced Prolog to print out all the answers to my query that it could find as facts in the database or derive from the rules. The final "no" printed by Prolog indicates that it couldn't find any more assignments. Did it get them all? Almost, but our second rule wasn't as general as it could have been. We expect Fred to be one of Bill's ancestors, but our rule doesn't cover this case. A query containing no variables, for example, `ancestor(fred, bill)`, just asks Prolog to verify if the statement corresponding to the query is true. So if we ask Prolog about Fred and Bill, it simply returns "no":

```
| ?- ancestor(fred, bill).
no
```

[3] The relations "ancestor" and "descendant" are said to be *inverses* of one another. We could have written `descendant(anne, fred)` instead of `ancestor(fred, anne)` but it's not necessary to assert both facts: The query `ancestor(X, anne)` finds all of Anne's ancestors and the query `ancestor(anne, X)` finds all of Anne's descendants. If you really want to use both the ancestor and descendant relation explicitly, you can define the descendant relation in terms of the ancestor relation by adding the rule `descendant(X, Y) :- ancestor(Y, X)`. This is a good example of how Prolog makes it easy to think about logical relationships.