Exercises

These exercises are intended to help you consolidate what you have read, and to get you writing Perl scripts. They are arranged by chapter, so that an exercise given under Chapter x should only need information provided in Chapters 1 to x, unless the exercise requires you to do some extra research, in which case the necessary pointers are provided. Questions marked with a dagger (†) are simple drill exercises and should not take you more than a few minutes; those marked with an asterisk (*) are substantial programming projects, which you might prefer to omit if you have projects of your own to work on. The unmarked exercises lie in between: they all require some thought and, usually, the writing of a Perl script. They vary in difficulty but, in general, as the material covered in the book gets more complex towards the end, so too do the exercises.

There are no best answers to these exercises; if you are happy with something, and it works, that's fine. If you can, you should experiment with different solutions, to get a feel for how you like to use Perl. Be ambitious, do more than you think you can. Be daring, push the language to its limits if you need to. But try to have some sort of fun.

Chapter 2.

†2.1. For each of the strings (a)–(e), say which of the patterns (i)–(xii) it matches. Where there is a match, what would be the values of `$MATCH`, `$1`, `$2`, etc.?

(a) `the quick brown fox jumped over the lazy dog`

(b) `The Sea! The Sea!`

(c) `(.+)\s*\1`

(d) `9780471975632`

(e) `C:\DOS\PATH\NAME`

(i) `/[a-z]/`
(ii) `/(\W+)/`
(iii) `/\W*/`
(iv) `/^\w+$/`
(v) `/[^\w+$]/`
(vi) `/\d/`
(vii) `/(.+)\s*\1/`
(viii) `/((.+)\s*\1)/`
(ix) `/(.+)\s*((\1))/`
(x) `/\DOS/`
(xi) `/\\DOS/`
(xii) `/\\\DOS/`

†2.2. Write down regular expressions to be used to search a file of song titles that correspond to the following informal queries:

(a) The researcher: 'I want everything that mentions Jelly Roll. Sometimes it's spelt as one word, sometimes as two.'
(b) The expert: 'It's really called *Blind Boy Blues*, but you'll often see it referred to as *Blind Willie's Blues*, or even just as *Blind Willie*.'
(c) The amnesiac: 'Errrm...now, let me see...it's called something wild something, or wild something something I forget...'

Think about the assumptions you need to make, both about the format of the data and about the correct interpretation of the natural language queries.

†2.3. Write down regular expressions that are matched by:

(a) a sentence (something that begins with a capital letter and ends with a full stop);
(b) any number that is a multiple of 5;
(c) any string whose length is a multiple of 5;
(d) any four digit number that reads the same backwards as forwards (like 4114);
(e) a number in Roman numerals (can you ensure that it is correctly formed?);
(f) any 'word character' except `Q` (try to make your regular expression as short as possible).

(You should be able to think of several sensible regular expressions for each.)

2.4. I mention on page 19 that most songs with the word 'blue' in their titles are not blues songs at all. Write a regular expression that is matched by such titles and excludes the blues songs matched by the pattern given on that page.

2.5. Write a regular expression that is matched by any string that contains both of the words `blue` and `green` somewhere.

†2.6. The CCITT changed its name to the ITU. Write a Perl script to update any text file that refers to CCITT documents. Would you use this script to update files without checking the output by hand? (How about the file containing the text of these exercises?)

2.7. Until recently all UK phone numbers began with a 0 (zero) followed by the area dialling code, then the number. A couple of years ago an extra digit was added to accommodate growing demand for numbers; all existing numbers had a 1 inserted after the 0, since, by then no numbers started with 01. Write a Perl script that scans a file and updates any old-style numbers, but leaves new-style ones alone. (Don't worry about things like 0800 numbers, unless you want to and know all the details.)

Chapter 3.

†3.1. What does the following script produce as its output? (Answer *before* you run it.)

```
$s1 = '100';
$s2 = '$100';

print "$s1\n";
print "$s2\n";
print "\$s1\n";
print "\Q$s2\E\n";
print "\Q\Q$s2\E\E\n";
```

†3.2. Write a Perl script to print any lines containing both of the words `blue` and `green` somewhere, using an operation introduced in this chapter to produce a much simpler solution than the regular expression in exercise 2.5.

3.3. Write a Perl script to find and print the longest word in a text file.

3.4. The excessive use of quoted text in mail messages and news postings is rife. Write a Perl script that prints the percentage of lines in its input that begin with a > character.

3.5. Many programmers use variable names made up of several words. One convention is to write names entirely in lower-case letters, with underline characters separating words, as in `a_long_variable_name`. An alternative, which appears to be growing in popularity, uses mixed case, with each word beginning with an upper-case letter, except that the whole variable name always begins with a lower-case one (usually to distinguish variables from type names) as in `aLongVariableName`. Write a Perl script to convert variable names from the first form to the second. (Or, if you prefer, *vice versa*.)

3.6. Write a Perl script that computes the average word length (in characters) and the average sentence length (in words) of a text file.

3.7. Write a Perl script that writes accidental concrete poetry by extracting every fifth word from a text file and arranging them in lines of roughly equal length, or some other visually appealing pattern.

Chapter 4.

4.1. Write a Perl script whose input consists of a set of URLs, one per line, where the first is a complete URL and the remainder are partial, and whose output consists of the full URLs obtained by resolving the partial ones relative to the first. (See *The HTML Sourcebook* if you are unsure about partial URLs and how they are resolved.)

†4.2. Consider the fashionable subject of JavaBeans.[6] Their only feature of interest for the present exercise is that a rigid convention is used to classify the names that programmers define when they construct a JavaBean. If *PropName* is what they call a *property name*, then there will be some *methods* associated with it. In particular, there will be a method whose name is of the form `get`*PropName*, which is called a *get method* for *PropName*. Similarly, `set`*PropName* is a *set method* for *PropName*. If *PropName* is the name of a Boolean property, though, its get method is called `is`*PropName*, and is a test method. If *EvName* is something called an event name, then `add`*EvName*`Listener` is a *listener adding method*, while `remove`*EvName*`Listener` is a *listener removing method*. Property and event names always begin with an upper-case letter. Write a Perl script that takes a file of method names and classifies them, identifying the property and event names. Your script should produce output like the following:

```
addTimerListener :  listener adder for Timer event
setFileName :  set method for FileName property
isFileOpen : test method for Boolean property FileOpen
```

[6]It doesn't actually matter if you have never heard of JavaBeans.

4.3. Re-do the previous exercise by writing individual subroutines for each category of method name, which return true of false depending on whether `$ARG` belongs to the corresponding category.

4.4. The syntax diagram compiler in this chapter just gives up and dies when it finds an error in its input. Real compilers usually try to recover from errors and parse the remaining input so that they find as many errors as possible on a single run. Modify the compiler so that it attempts some error recovery. (One way to do this is by using `eval` and exceptions, but it isn't the only way.)

4.5. Perl does not have a switch or case statement, unlike most contemporary languages. Investigate different ways of simulating such a statement using the various control flow structures introduced in this chapter.

Chapter 5.

5.1. The `rot13` encryption method described in Chapter 2 is a special case of a more general encryption method, sometimes called a Caesarean cypher, in which letters are coded by moving them an arbitrary number n through the alphabet; for `rot13`, $n = 13$. The key to the cypher is the value of n. The trouble is that messages encrypted this way are very easy to crack, provided you know what the most frequent letter in them is. For most English texts, it is safe to assume that the most frequently occurring letter is `e`. Write a Perl script that reads an encrypted text, finds the key, and prints the original message.

5.2. Write a Perl script to turn text into Morse code (representing dots and dashes with suitable characters). You can find a table of Morse codes at `http://www.cris.com/~Gsraven/codes.html`.

5.3. I mention on page 104 that `push`, `pop`, `shift`, and `unshift` can be defined in terms of `splice`. Do it.

5.4. Write a subroutine that returns a string which is the representation of its first numerical argument to the base of its second. For example, if you give it the arguments 47 and 12 it will return the string 3B. Assume for simplicity that the base is at most 16. Check the validity of the arguments.

5.5. Word order in Latin is not very important (or so they told me once). Write a Perl script to generate all the different permutations of the words in the sentence: `mens sana in corpore sano`.

5.6. Devise arguments to the `equal_arrays` subroutine that will fool the tests on pages 107–108. Can you make the checks foolproof?

†5.7. Write a subroutine to sort a list of strings in increasing order of their length.

5.8. Write a Perl script to sort the video clips database used in this chapter in increasing order of duration. You can assume it is safe to read the whole database into memory at once.

5.9. A Perl script written to the specification in exercise 3.5 will sometimes produce undesired results: for example, a flag called `is_dos_compatible` will be turned into `isDosCompatible`, instead of `isDOSCompatible`, which is more likely to be what would be required. Adapt your solution to exercise 3.5 so that sub-words belonging to a set which you specify are translated specially.

5.10. If you want to be sure that your files can be read on any computer system, one of the things you have to ensure is that their names are legal under MS-DOS: names must be made up entirely of upper-case letters, digits and underlines and be in '8.3' form, that is, they must consist of a root with at most eight characters, optionally followed by a dot and an extension of at most three characters. Write a Perl script that takes a file of arbitrary file names, up to 255 characters in total length, mixed case, including non-alphanumeric characters, and generates unique MS-DOS legal names from them, preserving as much as possible of the original name (for example, you might truncate names and extensions or throw away vowels, but don't just call the files `AAAAAAAA.AAA` etc.). Note the word 'unique' in the previous sentence. Your script should print a table showing the correspondence between the old names and the new ones.

5.11. Extend your program for exercise 4.2 to identify and classify all the property and event names in the data according to the following rules: a property is read/write if it has a get and a set method; read/write Boolean if it has a get and a test method; read-only if it has only get or test methods; a listener if it has a listener adder; probably malformed if it only has a set method or a listener remover, or if it has get, set or test and a listener adder or remover. *Make no assumptions* about the order in which the method names appear in the data file.

Chapter 6.

†6.1. Write a Perl script that throws away any values for the `DATA` filehandle included at the end of another Perl script. (Use a line range.)

6.2. Write a Perl script that prints the longest paragraph (measured in words) in a document.

6.3. Although you will often see it said that NTSC video has a frame rate of 30fps, this is not actually true: for obsolete technical reasons, NTSC video is broadcast at a rate of 29.97 frames per second. Since there is not an exact number of frames in a second, SMPTE timecode, which must use exactly 30, drifts with respect to the elapsed time. The expedient adopted to work round this is called *drop frame timecode*, in which frames 00:00 and 00:01 are omitted at the start of every minute except the tenth. (It's a bit like a leap year.) So your count jumps from, say, 00:59:29 to 1:00:02. Adapt the script on pages 133–134 to add extra columns for drop frame timecodes, changing the layout appropriately to arrange all the information neatly on the page.

*6.4. As you probably know, the GIF image file format can accommodate multiple images in a single file; this is the basis of GIF animation, which is currently enjoying a vogue on the World-Wide Web. Write a Perl script that examines a GIF file and finds out how many frames it contains. To do this, you will need to read the GIF format specification. It is available at `ftp://ftp.ncsa.uiuc.edu/misc/file.formats/graphics.formats/gif89a.doc`, and is also described in *The Encyclopedia of Graphics File Formats*, by James D. Murray and William vanRyper (O'Reilly and Associates, 2nd ed., 1996).

6.5. Adapt your answer to exercise 5.10 to produce a script that scans a directory and changes the names of all the files in it to conform to MS-DOS restrictions. Write a log file recording the changes in a suitable format for subsequently querying for the original name or reversing the renaming process.

6.6. Consider the sequence of numbered pictures produced as the output of the script on page 144. Suppose that it turns out that you want to make a movie out of these pictures in reverse order (stranger things have happened). It is possible to reverse movies, but sometimes it is quicker just to reverse the sequence numbering of the individual pictures.

(a) Assuming that the pictures have consecutive three digit extensions, starting at `.001`, write a Perl script that reverses their order; i.e. if there are 48 pictures, they will be called something like `frame.001` to `frame.048`; your script should rename the original `frame.001` as `frame.048`, `frame.002` as `frame.047`, and so on, until the original `frame.048` becomes the new `frame.001`. (Take care not to...but I don't have to tell you that, do I?)

(b) Instead of assuming that the pictures have consecutive three digit extensions, starting at `.001`, add code to find out how long the extensions are and which is the starting number, and to verify that they are indeed consecutive. (What will you do if they are not?) Modify the reversal code so that it will correctly rename the files consistently with their original numbering scheme. (For example, if the original extensions ran from `.07` to `.45`, so should the new ones, but with the pictures in reverse order.)

Chapter 7.

Most of the exercises for this chapter share a common theme of data structures. You may find them difficult if you are not comfortable with structures such as trees and directed graphs. A good book, for example, Mitchell L. Model's *Data Structures, Data Abstraction* (for C++ programmers), or Robert L. Kruse et al's *Data Structures and Program Design in C* (for C programmers) may help. Many other books are available on this subject, if neither C nor C++ is your preferred programming language. (Don't forget Donald Knuth's *Art of Computer Programming.*)

†7.1. Suppose the r-value of `$hr` is a reference to a hash and that of `$ar` is a reference to an array. Write an expression that evaluates to the value in `%$hr` associated with the key that is equal to the fourth element of the list `@$ar`.

†7.2. Suppose `$a` contains a reference to an array. Consider the following (contrived) sequence of assignments:

```
$words = [ qw(richard of york gave battle in vain) ];
$$a[0] = \$words;
$b = \$a;
```

Draw a diagram showing the l- and r-values of the variables `$words`, `$a` and `$b` and the relationship between them. Which, if any, of the following is equal to `richard`? (Try to answer without running a Perl script.)

(a) `$$a[0]->[0]`

(b) `$${$a[0]}[0]`

(c) `${$$a[0]}->[0]`

(d) `$$$a[0]->[0]`

(e) `$${$$a[0]}[0]`

(f) `$${$$$b[0]}[0]`

7.3. Suppose a data file contains rural bus timetable information in the following format: the first line contains the names of each village on the route, separated by colons; subsequent lines contain the times, using a 24 hour clock, at which the bus should arrive at each village, also separated by colons.

(a) Write a Perl script that takes a pair of village names as command line arguments, and, using such a data file, produces a table with one column for each of the two, showing the times of buses travelling between them.

(b) Write another script that reads such a file and produces a timetable with one row for each village showing the times each bus calls there.

Your scripts should probably validate their input.

*7.4. Consider an input file that records information about the links in a Web site. It has the following structure:

$\langle \textit{base URL}_1 \rangle$ `->`
$\langle \textit{URL}_{1,1} \rangle$
$\langle \textit{URL}_{1,2} \rangle$
$\langle \textit{URL}_{1,3} \rangle$
.
.
.
$\langle \textit{base URL}_2 \rangle$ `->`
$\langle \textit{URL}_{2,1} \rangle$
$\langle \textit{URL}_{2,2} \rangle$
$\langle \textit{URL}_{2,3} \rangle$
.
.
.

and so on. Each $\langle \textit{base URL}_i \rangle$ identifies a page on the site; the $\langle \textit{URL}_{i,j} \rangle$s are URLs found in links on that page. (If you want to make this exercise even more interesting, you can suppose that some of the $\langle \textit{URL}_{i,j} \rangle$s are partial URLs.)

Write a Perl script that prompts the user for pairs of URLs and determines, using the data in such a file, whether or not it is possible to reach the second URL from the first, via zero or more clicks on links.

*7.5. A file maintained by an international conservation organization contains data concerning the captive breeding of an endangered species, let us say the three-toed Nandi bear. Each line of the file consists of four items, separated by colons, the first being the name of a bear born in captivity, the next its sex, the last two the names of its parents. You can assume that the zoos taking part in the captive breeding programme ensure that each bear's name is unique.

Write a Perl script that reads the data in such a file, and then, for each bear that has not yet had any offspring, produces a list of potential mates. To avoid in-breeding, an animal may only be paired with another if they are less closely related than first cousins (i.e. they must not have the same grandparents).

*7.6. As you may be starting to observe, one of the problems in doing exercises like these is finding enough data to test your solutions. Write a data generator in Perl. It should take as its input a description of the desired data format, using a data description language of your devising (regular expressions might provide a good starting point for the language design), and produce a set of random data conforming to that format. (I have included this exercise here because one potential approach is to parse the data description and build a data structure that you can use to direct the generation. You may prefer to use a lower-tech approach, though.)

7.7. (a) Write a Perl script that takes a directory name as its argument, creates a new sub-directory called `executables` within it, then scans the directory and all its sub-directories looking for executables and moves them into the new sub-directory. (Look up `mkdir` in the Perl on-line documentation to see how to create a sub-directory.) Be prepared for identically named executables.

(b) Write a Perl script that takes a directory name as its argument, creates a new sub-directory called `graphics` within it, then scans the directory and all its sub-directories looking for graphic files (those with extensions belonging to some set you should specify that is appropriate for your environment) and moves them into the new sub-directory.

(c) If you have not already done so, write a higher-order function that captures the common behaviour of the scripts you have just written, and re-implement parts (a) and (b) using it. If you can, design your higher-order function so that it can also be used to implement the directory traversals on pages 169–170 as well.

Chapter 8.

Object-oriented programming is as much about design as it is about language features. If you are not already familiar with object-oriented methods, you will find these exercises more difficult than if you are. In either case, you may prefer to return to them after reading Chapter 9.

8.1. The file test operations described in pages 141–143 are very useful but their syntax is unlike anything else in Perl. Define a class `FileInfo`, with a constructor that takes the name of a file and associates it with the object, and methods corresponding to each of the file test operators, which return the value that operator would return for the file associated with the object used to call the method.

8.2. A collection of numbered pictures, such as we considered in the previous chapter, might profitably be considered as an object belonging to a class

`NumberedPictureSet`, say. Design and implement such a class, with a constructor that takes as its argument the pathname of a directory in which the collection resides, and methods to perform the renumbering operation described in the text and in exercise 6.6.

8.3. Re-work your solution to one of the first three exercises from the previous chapter in an object-oriented style.

8.4. For the purpose of taxation, the government of Freedonia classifies motor vehicles into three broad categories: public service vehicles, goods vehicles, and private cars. Emergency vehicles are considered a special subcategory of public service vehicles; goods vehicles are divided into light (vans) and heavy (lorries or trucks). Every vehicle has a unique registration number, and the government records the name of all vehicle owners; for goods vehicles, a goods operator's licence number is also recorded, whereas for public service vehicles, the government department responsible for them is recorded.

Tax is assessed as follows: public service vehicles play a flat rate (this is required for inter-departmental accounting purposes), except for emergency vehicles, which are exempt. All goods vehicles pay a rate proportional to their weight, but light goods vehicles pay a supplement if they have more than two windows. Private cars all pay the same amount of tax, but cars with engines smaller than 1100cc receive a rebate if they are more than three years old.

Design a collection of classes that is suitable for modelling this scenario, and implement them in Perl. Allow for the possibility that the rates of tax might be changed at budget time. Write a driver program to simulate the recording of data about registered vehicles and the collection of taxes.

8.5. Consider the following scenario: every person (let us suppose) has a name and an address; some people are students, and also have a university or college to which they belong; some people are musicians, and can play one or more instruments. If, for some reason, we wished to model this situation in Perl, we could define classes `Person`, `Student`, and `Musician`, with the latter two setting `@ISA = ('Person')`. The constructors could be written in an obvious way, so that the name and address were passed to `Person->new`, which stored them in a hash and returned a blessed reference to it. The constructor `Student->new` would take the student's name, address and college, call `SUPER->new` with the name and address, then add the college, in the manner we have seen; `Musician` would deal with name, address and instrument in a similar way.

Now consider music students. A music student is a student, and also a musician (to the extent of playing an instrument). The natural way to add music students to your model is by using multiple inheritance: define a class `MusicStudent`, which sets `@ISA = qw(Student Musician)`. Write a definition of the constructor for this new class. (Do not modify the constructors of the other classes.) What can you say about the maintainability and re-usability of the classes in this hierarchy?

Can you re-design the hierarchy so that it only uses single inheritance, but still accurately reflects the semantics of the scenario?

Chapter 9.

†9.1. Use `Benchmark` to compare the relative efficiencies of the built-in list manipulation functions, and those you defined in exercise 5.3.

9.2. Use `Benchmark` to determine whether there is any significant speed difference between the scripts for computing a table of squares given at the bottom of page 88 (using `for`) and the top of page 89 (using `foreach`). Devise and carry out further experiments to investigate the relative efficiency of these two forms of loops for a variety of tasks and number of loop iterations.

9.3. The `-M` file test operator tells you how many days ago a file was last modified. Using this operator and `Date::Manip`, write a Perl script that tells you the date on which a file was last modified.

9.4. Use a tied hash to store the data on vehicle registrations for your solution to exercise 8.4, in the following way:

(a) Generate a unique identifier every time an object is created, and store a flattened version of its data in the tied hash. Add `store` and `retrieve` methods and use them as necessary to write data to and read it from disk.

(b) Make the reading and writing transparent by modifying your other methods as necessary so that data is automatically read from disk when it is needed, and written whenever it changes.

9.5. Implement a simple system to be used by a dentist to generate reminders when patients are due for a check-up. You will need a database to record patients' names and addresses together with the date of their last visit, a simple data entry program to insert and update information, and a script to be run once a week to generate reminders for patients who have not been to the dentist in the preceding six months. Perhaps you should generate more urgent reminders for patients who have not been for over a year.

9.6. In a complex computer set-up, with many hard disks and removable media of varying size and speed, it may be necessary to move files around between disks, with the not inconceivable result that their origins get forgotten — especially if more than one person has access to the system. Design and implement in Perl a system for managing files on multiple permanent and removable disks, that maintains a history of every file known to it. The system should provide a means of moving and renaming files (a simple command line interface, such as that described in Chapter 4 will be adequate), and use a database to record every location and the date of any changes, so that it can answer questions such as 'Where is the file that I put on the cartridge `Scratch Zip II` on 13th April?' (You don't need to provide a natural language interface unless you *really* want to.) Your system need only keep track of changes made through its interface.

9.7. Take some time to browse among the modules in CPAN: the full list can be found at `http://www.perl.com/CPAN/modules/00modlist.long.html`. Choose a module that is related to one of your interests, examine its documentation to discover how it is used, and evaluate its usefulness. (In other words, play with it.)

*9.8. Add a graphical user interface to the data entry program you wrote for exercise 9.5. This will be easiest if you are using the X windows system, since you can use the Perl/Tk modules (although you could try the `X11:*` modules if you prefer to work directly with X). For Windows or MacOS systems, you should be able to find system-dependent modules for constructing interfaces with Perl on those platforms.

Chapter 10.

10.1. A characteristic problem of being a freelance Web designer, Perl coder, or graphic artist is finding clients; potential clients, on the other hand, sometimes have trouble finding competent freelancers. Implement a World-Wide Web-based system to help. You will need to maintain a database of freelance workers, recording at least their contact details, areas of expertise, and consultancy rates. You will also need a Web page, with a form to elicit these details from people who want to sign up with your service, and another with a form to be filled in by potential clients with their requirements and the rate they are willing to pay. Finally, you will need a database search routine, to find freelancers with the required skills willing to work for the offered rate of pay. All these components must be tied together via CGI.

10.2. The speed of Internet connections varies widely; pictures that somebody with an ISDN connection might download casually can take a very long time for somebody with a 14.4kbps modem. One response is to provide different versions of your picture to suit different connections.

(a) Write a CGI script that asks a user what speed their connection is, and sends them one of several versions of a file, depending on the answer.

(b) What you really want to do is find out the connection speed when the user arrives at your site, and then choose appropriate versions of all graphics, sound and video files to send them. Unfortunately, since HTTP creates a new connection for every request, it does not give you any way of remembering information about a client between requests. This is why 'cookies' were invented. A cookie is a small amount of data, with its own name, that you can associate with a Web page or site, which can be stored by a Web browser and retrieved by a CGI script. A full specification of cookies and their interaction with HTTP can be found at `http://home.netscape.com/newsref/std/cookie_spec.html`; the documentation accompanying Lincoln Stein's `CGI` module describes how cookies can be manipulated by CGI scripts using that module.[7] Modify your solution to part (a) so that the information about line speed need only be elicited once, and will be remembered for the duration of a session.

*10.3. The widely used POP3 mail protocol requires mail clients to download messages from a server to be read, even if they are the most egregious junk mail ('MAK $$$$ NOW!!!!!'). The newer IMAP protocol makes it possible to examine mail on the server and delete it without downloading, but not all ISPs have yet implemented IMAP. An interim solution is to provide a Web-based interface to the POP3 server, allowing you to connect to a Web page, give your mail address and POP3 password, and then have the headers of any waiting mail messages displayed to you. You can then choose to read or delete selected messages from within your Web browser. Using the `CGI` and appropriate `Mail::*` modules, design and implement such an interface. (Do not concern yourself with the security and privacy aspects that attend real implementations of such interfaces.)

Chapter 11.

11.1. Re-do all the above exercises in C++ or Java.

Epilogue

Write a Perl script that examines the dialogue of each of the characters and, using stylometric measures of your own devising, compares it with a large collection of writings on programming methodology in order to discover the true identities of the participants. (The waiter may prove especially tricky.)

[7] Although the documentation refers exclusively to Netscape cookies, MS Internet Explorer 3.0 and higher can also store and retrieve cookies.