Chapter 1. Introduction

Welcome to the Llama book, our affectionate name for our book covering Perl 5.

This is the eighth edition of a book that millions of readers have enjoyed since we released the first one in 1993. We hope they've enjoyed it. It's a sure thing that we enjoyed writing it. At least, that's how we remember it after we've turned in the book and waited the months it took to see it show up on shelves. And by "shelves," we mean online.

This is the second edition of our popular Perl 5 book after the release of Perl 6, a language that started its life as something based on Perl but has now taken on a life of its own with the new name "Raku." Unfortunately, that bit of history means both languages have "Perl" in the name even though they are only lightly related. It's likely that you want Perl 5 and this book unless you know that you don't. And from this point, "Perl" means Perl 5, the same Perl that's been getting work done for a couple of decades.

Questions and Answers

You probably have some questions about Perl, and maybe even some about this book, especially if you've already flipped through it to see what's coming. So, we'll use this chapter to answer them, including how to find answers that we *don't* provide.

Is This the Right Book for You?

This is not a reference book. It's a tutorial on the very basics of Perl, which is just enough for you to create simple programs mostly for your own use. We don't cover every detail of every topic, and we spread out some of the topics over several chapters so that you pick up concepts as you need them.

Our intended readers are people who know at least a little bit about programming and just need to learn Perl. We assume you have at least some background in using a terminal, editing files, and running programs—just not Perl programs. You already know about variables and subroutines and the like, but you just need to see how Perl does it.

This doesn't mean that the absolute beginner, having never touched a terminal program or written a single line of code, will be completely lost. You might not catch everything we say the first time you go through the book, but many beginners have used the book with only minor frustrations. The trick is to not worry about everything you might be missing and to focus on just the core concepts we present. You might take a little longer than an experienced programmer, but you have to start somewhere.

We're assuming you know a little about Unicode, so we don't bog down in the details, but we've explained a bit more in Appendix C. You can peruse it before you start the book and refer to it as needed.

We've included an appendix on experimental features (Appendix D). Some exciting new things await you, but we're not going to force you to use them. Whenever possible, we'll try to show you how to do the same amazing thing the old boring way.

And this shouldn't be the only Perl book you ever read. It's just a tutorial. It's not comprehensive. We get you started in the right direction so you can go on to our other books, *Intermediate Perl* and *Mastering Perl*, when you are ready. The definitive reference for Perl is *Programming Perl*, also known as the "Camel book."

We should also note that even though this book covers up to Perl 5.34, it's still useful even if you have an earlier version. You might miss out on some of the cool new features, but you'll still learn how to use basic Perl. The least recent version that we'll think about, however, is Perl 5.8, even though that was released almost 20 years ago.

What About the Exercises and Their Answers?

We've included exercises at the end of each chapter because, between the three of us, we've presented this same course material to several thousand students. We know that making mistakes by working through the exercises is the best way to learn. We have carefully crafted these exercises to give you the chance to make mistakes as well.

It's not that we *want* you to make mistakes, but you need to have the *chance*. You're going to make most of these mistakes during your Perl programming career, and it may as well be now. Any mistake that you make while reading this book you won't make again when you're writing a program on a deadline. And we're always here to help you out if something goes wrong, in the form of Appendix A, which has our answers for each exercise and a little text to go with it. Check out the answers when you're done with the exercises.

Try not to peek at the answer until you've given the problem a good try, though. You'll learn better if you figure it out rather than read about it. Don't knock your head repeatedly against the wall if you don't figure out a solution: move on to the next chapter and don't worry too much about it.

Even if you never make any mistakes, you should look at the answers when you're done; the accompanying text will point out some details of the program that might not be obvious at first.

When you look at our answer, keep in mind that you could have done it in a different manner and still be right. You don't have to do the same thing we did. In some cases, we'll have multiple solutions. Not only that, but after you make it through this book, you'll probably do the same task differently since we've limited our answers to only the concepts we've shown so far. Other features you encounter later might make short work of the problem.

Each exercise has a number in square brackets in front of the exercise text, looking something like this:

1. [37] What does the number 37 inside square brackets mean, when it appears at the start of an exercise's text?

That number is our (very rough) estimate of how many minutes you can expect to spend on that particular exercise. It's rough, so don't be too surprised if you're all done (with writing, testing, and debugging) in half that time, or not done in twice that long. On the other hand, if you're really stuck, we won't tell anyone that you peeked at Appendix A to see our answer.

If you want additional exercises, check out the *Learning Perl Exercises*, which adds several exercises for each chapter.

What If I'm a Perl Course Instructor?

If you're a Perl instructor who has decided to use this as your textbook (as many have over the years), you should know that we've tried to make each set of exercises short enough that most students could do the whole set in 45 minutes to an hour, with a little time left over for a break. Some chapters' exercises should be quicker, and some may take longer. That's because, once we had written all of those little numbers in square brackets, we discovered we don't know how to add (luckily we know how to make computers do it for us).

We also have a companion book, the *Learning Perl Exercises*, which has additional exercises for each chapter. If you get the version of the workbook for previous editions, you might have to adjust the chapter order.

What Does "Perl" Stand For?

Perl is sometimes called the "Practical Extraction and Report Language," although it has also been called a "Pathologically Eclectic Rubbish Lister," among other expansions. It's actually a backronym, not an acronym—Larry Wall, Perl's creator, came up with the name first and the expansion later. That's why "Perl" isn't in all caps. There's no point in arguing which expansion is correct: Larry endorses both.

You may also see "perl" with a lowercase *p* in some writing. In general, "Perl" with a capital *P* refers to the language and "perl" with a lowercase

p refers to the actual interpreter that compiles and runs your programs.

Why Did Larry Create Perl?

Larry created Perl in the mid-1980s when he was trying to produce some reports from a Usenet-news-like hierarchy of files for a bug-reporting system, and *awk* ran out of steam. Larry, being the lazy programmer that he is, decided to overkill the problem with a general-purpose tool that he could use in at least one other place. The result was Perl version zero.

We're not insulting Larry by saying he's lazy; Laziness is a virtue. So are Impatience and Hubris, as Larry wrote in the first edition of *Programming perl*. The wheelbarrow was invented by someone who was too lazy to carry things; writing was invented by someone who was too lazy to memorize; Perl was invented by someone who was too lazy to get the job done without inventing a whole new computer language.

Why Didn't Larry Just Use Some Other Language?

There's no shortage of computer languages, is there? But at the time, Larry didn't see anything that really met his needs. If one of the other languages of today had been available back then, perhaps Larry would have used one of those. He needed something with the quickness of coding available in shell or *awk* programming, and with some of the power of more advanced tools like *grep*, *cut*, *sort*, and *sed*, without having to resort to a language like C.

Perl tries to fill the gap between low-level programming (such as in C or C++ or assembly) and high-level programming (such as shell programming). Low-level programming is usually hard to write and ugly, but fast and unlimited; it's hard to beat the speed of a well-written low-level program on a given machine. And there's not much you can't do there. High-level programming, at the other extreme, tends to be slow, hard, ugly, and limited; there are many things you can't do at all with the shell or batch programming if there's no command on your system that provides the needed functionality. Perl is easy, nearly unlimited, mostly fast, and kind of ugly.

Let's take another look at those four claims we just made about Perl.

First, *Perl is easy*. As you'll see, though, this means it's easy to *use*. It's not especially easy to *learn*. If you drive a car, you spent many weeks or months learning how, and now it's easy to drive. When you've been programming Perl for about as many hours as it took you to learn to drive, Perl will be easy for you.

Perl is nearly unlimited. There are very few things you can't do with Perl. You wouldn't want to write an interrupt-microkernel-level device driver in Perl (even though that's been done), but most things that ordinary folks need most of the time are good tasks for Perl, from quick little one-off programs to major industrial-strength applications.

Perl is mostly fast. That's because nobody is developing Perl who doesn't also use it—so we all want it to be fast. If someone wants to add a feature that would be really cool but would slow down other programs, the Perl developers are almost certain to refuse the new feature until we find a way to make it quick enough.

Perl is kind of ugly. This is true. The symbol of Perl has become the camel, from the cover of the venerable Camel book (also known as *Programming Perl*), a cousin of this book's Llama (and her sister, the Alpaca). Camels are kind of ugly, too. But they work hard, even in tough conditions. Camels are there to get the job done despite all difficulties, even when they look bad and smell worse and sometimes spit at you. Perl is a little like that.

Is Perl Easy or Hard?

Perl is easy to use, but sometimes hard to learn. This is a generalization, of course. In designing Perl, Larry made many trade-offs. When he's had the chance to make something easier for the programmer at the expense of being more difficult for the student, he's decided in the programmer's favor nearly every time. That's because you'll learn Perl only once, but you'll use it again and again.

If you're going to use a programming language for only a few minutes each week or month, you'd prefer one that is easier to learn since you'll have forgotten nearly all of it from one use to the next. Perl is for people who are programmers for at least 20 minutes per day, and probably most of that in Perl.

Perl has any number of conveniences that let the programmer save time. For example, most functions will have a default; frequently, the default is the way that you'll want to use the function. So you'll see lines of Perl code like these:

```
while (<>) {
  chomp;
  print join("\t", (split /:/)[0, 2, 1, 5] ), "\n";
}
```

Don't worry that you don't know what any of that means yet. Written out in full, without using Perl's defaults and shortcuts, that snippet would be roughly 10 or 12 times longer, so it would take much longer to read and write. It would be harder to maintain and debug, too, with more variables. If you already know some Perl and you don't see the variables in that code, that's part of the point. They're all being used by default. But to have this ease at the programmer's tasks means paying the price when you're learning; you have to learn those defaults and shortcuts.

A good analogy is the proper and frequent use of contractions in English. Sure, "will not" means the same as "won't." But most people say "won't" rather than "will not" because it saves time, and because everybody knows it and it makes sense. Similarly, Perl's "contractions" abbreviate common "phrases" so that they can be "spoken" quicker and understood by the maintainer as a single idiom rather than a series of unrelated steps.

Once you become familiar with Perl, you may find yourself spending less time trying to get shell quoting (or C declarations) right, and more time surfing the web because Perl is a great tool for leverage. Perl's concise constructs allow you to create (with minimal fuss) some very cool one-up

solutions or general tools. Also, you can drag those tools along to your next job because Perl is highly portable and readily available, so you'll have even more time to surf.

Perl is a very high-level language. That means the code is quite dense; a Perl program may be around a quarter to three-quarters as long as the corresponding program in C. This makes Perl faster to write, faster to read, faster to debug, and faster to maintain. It doesn't take much programming before you realize that, when the entire subroutine is small enough to fit onscreen all at once, you don't have to keep scrolling back and forth to see what's going on. Also, since the number of bugs in a program is roughly proportional to the length of the source code (rather than being proportional to the program's functionality), the shorter source in Perl will mean fewer bugs on average.

Like any language, Perl can be "write-only"—it's possible to write programs that are impossible to read. But with proper care, you can avoid this common accusation. Yes, sometimes Perl looks like line noise to the uninitiated, but to the seasoned Perl programmer, it looks like the notes of a grand symphony. If you follow the guidelines of this book, your programs should be easy to read and easy to maintain, and they probably won't win an Obfuscated Perl Contest.

How Did Perl Get to Be So Popular?

After playing with Perl a bit, adding stuff here and there, Larry released it to the community of Usenet readers, commonly known as "the Net." The users on this ragtag fugitive fleet of systems around the world (tens of thousands of them) gave him feedback, asking for ways to do this, that, or the other thing, many of which Larry had never envisioned his little Perl handling.

But as a result, Perl grew, and grew, and grew. It grew in features. It grew in portability. What was once a little language available on only a couple of Unix systems now has thousands of pages of free online documentation, dozens of books, several mainstream Usenet newsgroups (and a dozen newsgroups and mailing lists outside the mainstream) with an un-

countable number of readers, and implementations on nearly every system in use today—and don't forget this Llama book as well.

What's Happening with Perl Now?

Perl 5 development underwent an amazing revitalization while most people were waiting for its successor, Perl 6. Those are actually different languages now, but Perl 5, which is still doing a lot of good work in the world, keeps chugging along. They shared a name for awhile, but Perl 6 is now on its own as "Raku" (although brian's book on that language is still titled *Learning Perl 6*).

Starting with v5.10, Perl developed a way to include new features in the language without disturbing old programs. We'll show you how to get those new features when you want them, as well as how to enable experimental features that you might want to play with. You can peek at Appendix D for more details.

The Perl 5 Porters also adopted an official support policy. After 20 years of playing it fast and loose, they decided they would support the last two stable releases. By the time we're finished with this book, we expect those to be v5.32 and v5.34. The odd numbers after the dot are reserved for development releases.

In 2019, Perl development moved to GitHub. That means you can now easily file issues, send pull requests, and check out the "bleading" edge sources. This frees up a lot of the effort that formerly went into maintaining the aging infrastructure.

There's also talk of a new major version of Perl, Perl 7, which will be mostly v5.34 with different defaults. Learning the current version of Perl means you should be ready for Perl 7. brian covers some things to expect in *Preparing for Perl 7*, and we'll add notes in this book when we can.

What's Perl Really Good For?

Perl is good for quick-and-dirty programs that you whip up in three minutes. Perl is also good for long and extensive programs that will take a dozen programmers three years to finish. Of course, you'll probably find yourself writing many programs that take you less than an hour to complete, from the initial plan to the fully tested code.

Perl is optimized for problems that are about 90% working with text and about 10% everything else. That description seems to fit most programming tasks that pop up these days. In a perfect world, every programmer would know every language; you'd always be able to choose the best language for each project. Most of the time, we hope you'd choose Perl.

What Is Perl Not Good For?

So, if it's good for so many things, what is Perl *not* good for? Well, you shouldn't choose Perl if you're trying to make an *opaque binary*. That's a program that you could give away or sell to someone who then can't see your secret algorithms in the source, and thus can't help you maintain or debug your code either. When you give someone your Perl program, you'll normally be giving them the source, not an opaque binary.

If you're wishing for an opaque binary, though, we have to tell you that they don't exist. If someone can install and run your program, they can turn it back into source code. Granted, this won't necessarily be the same source you started with, but it will be some kind of source code. The real way to keep your secret algorithm a secret is, alas, to apply the proper number of attorneys; they can write a license that says, "You can do *this* with the code, but you can't do *that*. And if you break our rules, we've got the proper number of attorneys to ensure that you'll regret it."

How Can I Get Perl?

You probably already have it. At least, we find Perl wherever *we* go. It ships with many systems, and system administrators often install it on every machine at their site. But if you can't find it already on your system, you can still get it for free. It comes pre-installed with most Linux or *BSD

systems, macOS, and some others. Companies such as ActiveState provide pre-built and enhanced distributions for several platforms, including Windows. You can also get Strawberry Perl for Windows, which comes with all the same stuff as regular Perl plus extra tools to compile and install third-party modules.

Perl is distributed under two different licenses. For most people, since you'll merely be *using* it, either license is as good as the other. If you'll be modifying Perl, however, you'll want to read the licenses more closely, because they put some small restrictions on distributing the modified code. For people who won't modify Perl, the licenses essentially say, "It's free—have fun with it."

In fact, it's not only free, but it runs rather nicely on nearly everything that calls itself Unix and has a C compiler. You download it, type a command or two, and it starts configuring and building itself. Or, better yet, you get your package manager to do it for you. Besides Unix and Unix-like systems, people addicted to Perl have ported it to other systems, such as VMS, OS/2, even MS-DOS, and every modern species of Windows—and probably even more by the time you read this. Many of these ports of Perl come with an installation program that's even easier to use than the process for installing Perl in Unix. Check for links in the "ports" section on CPAN.

It's nearly always better to compile Perl from the source in Unix systems. Other systems may not have a C compiler and other tools needed for compilation, so CPAN has binaries for these. When you use your local package manager, you're changing the *perl* that your system may use to do its own tasks. You might make a big mess. We suggest you install a *perl* just for your use, but that's not necessary for this book.

What Is CPAN?

CPAN is the Comprehensive Perl Archive Network, your one-stop shop for Perl. It has the source code for Perl itself, ready-to-install ports of Perl to all sorts of non-Unix systems, examples, documentation, extensions to

Perl, and archives of messages about Perl. In short, CPAN is comprehensive.

CPAN is replicated on hundreds of mirror machines around the world; start at metacpan to browse or search the archive.

Is There Any Kind of Support?

Sure. One of our favorites is the Perl Mongers. This is a worldwide association of Perl users groups. There's probably a group near you with an expert or someone who knows an expert. If there's no group, you can easily start one.

Of course, for the first line of support, you shouldn't neglect the documentation. Besides the included documentation, you can also read the documentation on CPAN, MetaCPAN, as well as other sites that have the Perl documentation, and check out the latest version of the perlfaq.

Another authoritative source is the book *Programming Perl*, commonly called "the Camel book" because of its cover animal (just as this book is known as "the Llama book"). The Camel book contains the complete reference information, some tutorial stuff, and a bunch of miscellaneous information about Perl. There's also a separate pocket-sized *Perl 5 Pocket Reference* by Johan Vromans (O'Reilly) that's handy to keep at hand (or in your pocket).

If you need to ask a question, there are any number of mailing lists—many listed at *http://lists.perl.org*. There's also The Perl Monastery and Stack Overflow. At any hour of the day or night, there's a Perl expert awake in some time zone answering questions somewhere—the sun never sets on the Perl empire. This means that if you ask a question, you'll often get an answer within minutes. And if you didn't check the documentation and FAQ first, you'll get flamed within minutes.

You can also check out *http://learn.perl.org* and its associated mailing list, beginners@perl.org. Many well-known Perl programmers also have blogs

that regularly feature Perl-related posts, most of which you can read through /r/perl on Reddit.

If you find yourself needing a support contract for Perl, there are a number of firms that are willing to charge as much as you'd like. In most cases, these other support avenues will take care of you for free.

What If I Find a Bug in Perl?

As a new Perl programmer, you're likely to create a situation where you think something is wrong with Perl. You're using a big language that you don't know yet, so you don't know who to blame for the unexpected behavior. It happens.

The first thing to do when you find a bug is to check the documentation again. Maybe even two or three times. Many times, we've gone into the documentation looking to explain a particular unexpected behavior and found some new little nuance that ends up on a slide or in a magazine article. Perl has so many special features and exceptions to rules that you may have discovered a feature, not a bug. Also, check that you don't have an older version of Perl; maybe you found something that's been fixed in a more recent version.

Once you're 99% certain that you've found a real bug, ask around. Ask someone at work, at your local Perl Mongers meeting, or at a Perl conference. Chances are, it's *still* a feature, not a bug.

When you're 100% certain that you've found a real bug, cook up a test case. (What, you haven't done so already?) The ideal test case is a tiny self-contained program that any Perl user could run to see the same (mis-)behavior as you've found. Once you've got a test case that clearly shows the bug, create an issue on GitHub at *https://github.com/Perl/perl5/issues*.

Once you've sent off your bug report, if you've done everything right, it's not unusual to get a response within minutes. Typically, you can apply a simple patch and get right back to work. Of course, you may (at worst) get

no response at all; the Perl developers are under no obligation to read your bug reports. But all of us love Perl, so nobody likes to let a bug escape our notice.

How Do I Make a Perl Program?

It's about time you asked (even if you didn't). Perl programs are text files; you can create and edit them with your favorite text editor. You don't need any special development environment, although there are some commercial ones available from various vendors. We've never used any of these enough to recommend them (but long enough to stop using them). Besides, your environment is a personal choice. Ask three programmers what you should use and you'll get eight answers.

You should generally use a programmer's text editor rather than an ordinary editor. What's the difference? Well, a programmer's text editor will let you do things that programmers need, like indenting or un-indenting a block of code, or finding the matching closing curly brace for a given opening curly brace.

On Unix systems, the two most popular programmer editors are *emacs* and *vi* (and their variants and clones). BBEdit, TextMate, and Sublime Text are good editors for macOS, and a lot of people have said nice things about UltraEdit, SciTE, Komodo Edit, and PFE (Programmer's File Editor) in Windows. The perlfaq3 documentation lists several other editors too. Ask your local expert about text editors on your system.

For the simple programs you'll write for the exercises in this book, none of which should be more than about 20 or 30 lines of code, any text editor will be fine.

Some beginners try to use a word processor instead of a text editor. We recommend against this—it's inconvenient at best and impossible at worst. But we won't try to stop you. Be sure to tell the word processor to save your file as "text only"; the word processor's own format will almost certainly be unusable. Most word processors will probably also tell you

that your Perl program is spelled incorrectly and you should use fewer semicolons.

In some cases, you may need to compose the program on one machine, then transfer it to another to run it. If you do this, be sure that the transfer uses "text" or "ASCII" mode, and not "binary" mode. This step is needed because of the different text formats on different machines. Without it, you may get inconsistent results—some versions of Perl actually abort when they detect a mismatch in the line endings.

A Simple Program

According to the oldest rule in the book, any book about a computer language that has Unix-like roots has to start with showing the "Hello, world" program. So, here it is in Perl:

```
#!/usr/bin/perl
print "Hello, world!\n";
```

Let's imagine that you've typed that into your text editor. (Don't worry yet about what the parts mean and how they work. You'll see about those in a moment.) You can generally save that program under any name you wish. Perl doesn't require any special kind of filename or extension, and it's better not to use an extension at all. But some systems may require an extension like *.plx* (meaning PerL eXecutable).

You may also need to do something so that your system knows it's an executable program (that is, a command). What you'll do depends on your system; maybe you won't have to do anything more than save the program in a certain place. (Your current directory will generally be fine.) On Unix systems, you mark a program as being executable using the *chmod* command:

```
$ chmod a+x my_program
```

The dollar sign (and space) at the start of the line represents the shell prompt, which will probably look different on your system. If you're used

to using *chmod* with a number like `755` instead of a symbolic parameter like `a+x`, that's fine too, of course. Either way, it tells the system that this file is now a program.

Now you're ready to run it:

```
$ ./my_program
```

The dot and slash at the start of this command mean to find the program in the current working directory instead of looking through PATH to find a program. That's not needed in all cases, but you should use it at the start of each command invocation until you fully understand what it's doing.

You can also run this by explicitly stating *perl*. If you're in Windows, you must specify *perl* on the command line because Windows won't guess the program you want to run:

```
C:\> perl my_program
```

If everything worked, it's a miracle. More often, you'll find that your program has a bug. Edit and try again—but you don't need to use *chmod* each time, as that should "stick" to the file. (Of course, if the bug is that you didn't use *chmod* correctly, you'll probably get a "permission denied" message from your shell.)

There's another way to write this simple program in v5.10 or later, and we might as well get that out of the way right now. Instead of `print`, we use `say`, which does almost the same thing but with less typing. It adds the newline for us, meaning that we can save some time forgetting to add it ourselves. Since it's a new feature and you might not be using Perl 5.10 yet, we include a `use v5.10` statement that tells Perl we used new features:

```
#!/usr/bin/perl
use v5.10;

say "Hello World!";
```

This program only runs under v5.10 or later. When we introduce features from v5.10 or later in this book, we'll explicitly say they are new features in the text and include that `use v5.10` statement to remind you.

Typically, we only require the earliest version of Perl for the features that we need. This book covers up to v5.34, and in many of the new features we preface the examples to remind you the minimum version of Perl that you need for a feature:

```
use v5.34;
```

We could also write this version requirement without the `v`, but we have to remember that the minor number is actually three digits:

```
use 5.034;
```

Instead of that, we'll use the `v` form throughout this book.

What's Inside That Program?

Like other "free-form" languages, Perl generally lets you use insignificant whitespace (like spaces, tabs, and newlines) at will to make your program easier to read. Most Perl programs use a fairly standard format, though, much like most of what we show here. There is some general advice (not rules!) in the perlstyle documentation. We strongly encourage you to properly indent your programs, as that makes your programs easier to read; a good text editor will do most of the work for you. Good comments also make a program easier to read. In Perl, comments run from a pound sign (`#`) to the end of the line.

NOTE

There are no "block comments" in Perl, but there are a number of ways to fake them. See the perlfaq portions of the documentation.

We don't use many comments in the programs in this book because the surrounding text explains their workings, but you should use comments as needed in your own programs.

So, another way (a very strange way, it must be said) to write that same "Hello, world" program might be like this:

```
#!/usr/bin/perl
    print    # This is a comment
"Hello, world!\n"
;    # Don't write your Perl code like this!
```

That first line is actually a very special comment. On Unix systems, if the very first two characters on the first line of a text file are `#!` (pronounced "sh-bang," or SHəˈbaNG for you dictionary nerds), then what follows is the name of the program that actually executes the rest of the file. In this case, the program is stored in the file */usr/bin/perl*.

This `#!` line is actually the least portable part of a Perl program because you'll need to find out what goes there for each machine. Fortunately, it's almost always either */usr/bin/perl* or */usr/local/bin/perl*. If that's not it, you'll have to find where your system is hiding *perl*, then use that path. On some Unix systems, you might use a shebang line that finds *perl* for you:

```
#!/usr/bin/env perl
```

Beware though, that finds the first *perl*, which might not be the one that you wanted. If *perl* is not in any of the directories in your search path, you might have to ask your local system administrator or somebody using the same system as you.

On non-Unix systems, it's traditional (and even useful) to make the first line say `#!perl` . If nothing else, it tells your maintenance programmer as soon as they get ready to fix it that it's a Perl program.

If that `#!` line is wrong, you'll generally get an error from your shell. This may be something unexpected, like "file not found" or "bad interpreter." It's not your program that's not found, though; it's that */usr/bin/perl* wasn't where it should have been. We'd make the message clearer if we could, but it's not coming from Perl; it's the shell that's complaining.

Another problem you could have is that your system doesn't support the `#!` line at all. In that case, your shell (or whatever your system uses) will probably try to run your program all by itself, with results that may disappoint or astonish you. If you can't figure out what some strange error message is telling you, search for it in the perldiag documentation.

The "main" program consists of all the ordinary Perl statements (not including anything in subroutines, which you'll see later). There's no "main" routine, as there is in languages like C or Java. In fact, many programs don't even have routines (in the form of subroutines).

There's also no required variable declaration section, as there is in some other languages. If you've always had to declare your variables, you may be startled or unsettled by this at first. But it allows us to write quick-and-dirty Perl programs. If your program is only two lines long, you don't want to have to use one of those lines just to declare your variables. If you really want to declare your variables, that's a good thing; you'll see how to do that in Chapter 4.

Most statements are an expression followed by a semicolon. Here's the one you've seen a few times so far:

```
print "Hello, world!\n";
```

You really only need semicolons to separate statements, not terminate them. You could leave the semicolon off if there's no statement following it (or it's the last statement in a scope):

```
print "Hello, world!\n"
```

As you may have guessed by now, this line prints the message `Hello, world!`. At the end of that message is the shortcut `\n`, which is probably familiar to you if you've used another language like C, C++, or Java; it means a newline character. When that's printed after the message, the print position drops down to the start of the next line, allowing the following shell prompt to appear on a line of its own, rather than being attached to the message. Every line of output should end with a newline character. We'll see more about the newline shortcut and other so-called backslash escapes in the next chapter.

How Do I Compile My Perl Program?

Just run your Perl program. The *perl* interpreter compiles and runs your program in one user step:

```
$ perl my_program
```

When you run your program, Perl's internal compiler first runs through your entire source, turning it into internal *bytecodes*, which is an internal data structure representing the program. Perl's bytecode engine takes over and actually runs the bytecode. If there's a syntax error on line 200, you'll get that error message before you start running line 2. If you have a loop that runs 5,000 times, it's compiled just once; the actual loop can then run at top speed. And there's no runtime penalty for using as many comments and as much whitespace as you need to make your program easy to understand. You can even use calculations involving only constants, and the result is a constant computed once as the program is beginning—not each time through a loop.

To be sure, this compilation does take time—it's inefficient to have a voluminous Perl program that does one small quick task (out of many potential tasks, say) and then exits because the runtime for the program will be dwarfed by the compile time. But the compiler is very fast; normally the compilation will be a tiny percentage of the runtime.

What if you could save the compiled bytecodes to avoid the overhead of compilation? Or, even better, what if you could turn the bytecodes into

another language, like C, and then compile that? Well, both of these things are possible in some cases, but they probably won't make most programs any easier to use, maintain, debug, or install, and they may even make your program slower.

A Whirlwind Tour of Perl

So, you want to see a real Perl program with some meat? (If you don't, just play along for now.) Here you are:

```
#!/usr/bin/perl
@lines = `perldoc -u -f atan2`;
foreach (@lines) {
  s/\w<(.+?)>/\U$1/g;
  print;
}
```

NOTE

If *perldoc* is not available, that probably means your system doesn't have a command-line interface or your particular system includes it in a different package.

Now, the first time you see Perl code like this, it can seem pretty strange. (In fact, every time you see Perl code like this, it can seem pretty strange.) But let's take it line by line and see what this example does. These explanations are very brief; this is a whirlwind tour, after all. We'll see all of this program's features in more detail in upcoming chapters. You're not really supposed to understand the whole thing until later.

The first line is the `#!` line, as you saw before. You might need to change that line for your system, as we showed you earlier.

The second line runs an external command, named within backquotes (``). (The backquote key is often found next to the number 1 on full-size American keyboards. Be sure not to confuse the backquote with the single quote, `'`.) The command we used is *perldoc -u -f atan2*; try typing that in

your command line to see what its output looks like. The *perldoc* command is used on most systems to read and display the documentation for Perl and its associated extensions and utilities, so it should normally be available. This command tells you something about the trigonometric function `atan2`; we're using it here just as an example of an external command whose output we wish to process.

The output of that command in the backquotes is saved in an array variable called `@lines`. The next line of code starts a loop that will process each one of those lines. Inside the loop, the statements are indented. Although Perl doesn't require this, good programmers do.

The first line inside the loop body is the scariest one; it says `s/\w<(.+?)>/\U$1/g;`. Without going into too much detail, we'll just say that this can change any line that has a special marker made with angle brackets (`< >`), and there should be at least one of those in the output of the *perldoc* command.

The next line, in a surprise move, prints out each (possibly modified) line. The resulting output should be similar to what *perldoc -u -f atan2* would do on its own, but there will be a change where any of those markers appear.

Thus, in the span of a few lines, you've run another program, saved its output in memory, updated the memory items, and printed them out. This kind of program is a fairly common use of Perl, where one type of data is converted to another.

Exercises

Normally, each chapter will end with some exercises, with the answers in Appendix A. But you don't need to write the programs needed to complete this section—those are supplied within the chapter text.

If you can't get these exercises to work on your machine, double-check your work and then consult your local expert. Remember that you may need to tweak each program a little, as described in the text:

1. [7] Type in the “Hello, world” program and get it to work! You may name it anything you wish, but a good name might be *ex1-1*, for simplicity, as it’s Exercise 1 in Chapter 1. This is a program that even an experienced programmer would write, mostly to test the setup of a system. If you can run this program, your *perl* is working.
2. [5] Type the command *perldoc -u -f atan2* at a command prompt and note its output. If you can’t get that to work, find out from a local administrator or the documentation for your version of Perl about how to invoke *perldoc* or its equivalent. You’ll need this for the next exercise anyway.
3. [6] Type in the second example program (from the previous section) and see what it prints. Hint: be careful to type those punctuation marks exactly as shown! Do you see how it changed the output of the command?