Unmatched Power for
Text Processing and Scripting

4th Edition
Covers Version 5.14

Free Sampler

Programming

Perl

Tom Christiansen,
brian d foy & Larry Wall
with Jon Orwant

O'REILLY®

Programming Perl, Fourth Edition
by Tom Christiansen, brian d foy & Larry Wall, with Jon Orwant

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Holly Bauer
Proofreader: Marlowe Shaeffer
Indexer: Lucie Haskins

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

January 1991: First Edition.
September 1996: Second Edition.
July 2000: Third Edition.
February 2012: Fourth Edition.

Revision History for the Fourth Edition:
2011-02-13 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9780596004927* for release details.

ISBN: 978-0-596-00492-7

[M]

1329160875

Table of Contents

Part III. Perl as Technology

Part IV. Perl as Culture

CHAPTER 1

An Overview of Perl

Getting Started

We think that Perl is an easy language to learn and use, and we hope to convince you that we're right. One thing that's easy about Perl is that you don't have to say much before you say what you want to say. In many programming languages, you have to declare the types, variables, and subroutines you are going to use before you can write the first statement of executable code. And for complex problems demanding complex data structures, declarations are a good idea. But for many simple, everyday problems, you'd like a programming language in which you can simply say:

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

and expect the program to do just that.

Perl is such a language. In fact, this example is a complete program,[1] and if you feed it to the Perl interpreter, it will print "`Howdy, world!`" on your screen. (The `\n` in the example produces a newline at the end of the output.)

And that's that. You don't have to say much *after* you say what you want to say, either. Unlike many languages, Perl thinks that falling off the end of your program is just a normal way to exit the program. You certainly *may* call the `exit` function explicitly if you wish, just as you *may* declare some of your variables, or even *force* yourself to declare all your variables. But it's your choice. With Perl you're free to do The Right Thing, however you care to define it.

There are many other reasons why Perl is easy to use, but it would be pointless to list them all here, because that's what the rest of the book is for. The devil may be in the details, as they say, but Perl tries to help you out down there in the hot

1. Or script, or application, or executable, or doohickey. Whatever.

place, too. At every level, Perl is about helping you get from here to there with minimum fuss and maximum enjoyment. That's why so many Perl programmers go around with a silly grin on their face.

This chapter is an overview of Perl, so we're not trying to present Perl to the rational side of your brain. Nor are we trying to be complete, or logical. That's what the following chapters are for. Vulcans, androids, and like-minded humans should skip this overview and go straight to Chapter 2 for maximum information density. If, on the other hand, you're looking for a carefully paced tutorial, you should probably get *Learning Perl*. But don't throw this book out just yet.

This chapter presents Perl to the *other* side of your brain, whether you prefer to call it associative, artistic, passionate, or merely spongy. To that end, we'll be presenting various views of Perl that will give you as clear a picture of Perl as the blind men had of the elephant. Well, okay, maybe we can do better than that. We're dealing with a camel here (see the cover). Hopefully, at least one of these views of Perl will help get you over the hump.

Natural and Artificial Languages

Languages were first invented by humans, for the benefit of humans. In the annals of computer science, this fact has occasionally been forgotten.[2] Since Perl was designed (loosely speaking) by an occasional linguist, it was designed to work smoothly in the same ways that natural language works smoothly. Naturally, there are many aspects to this, since natural language works well at many levels simultaneously. We could enumerate many of these linguistic principles here, but the most important principle of language design is that easy things should be easy, and hard things should be possible. (Actually, that's two principles.) They may seem obvious to you, but many computer languages fail at one or the other.

Natural languages are good at both because people are continually trying to express both easy things and hard things, so the language evolves to handle both. Perl was designed first of all to evolve, and indeed it has evolved. Many people have contributed to the evolution of Perl over the years. We often joke that a camel is a horse designed by a committee, but if you think about it, the camel is pretty well adapted for life in the desert. The camel has evolved to be relatively self-sufficient. (On the other hand, the camel has not evolved to smell good. Neither has Perl.) This is one of the many strange reasons we picked the camel to be Perl's mascot, but it doesn't have much to do with linguistics.

2. More precisely, this fact has occasionally been remembered.

Now when someone utters the word "linguistics", many folks focus in on one of two things. Either they think of words, or they think of sentences. But words and sentences are just two handy ways to "chunk" speech. Either may be broken down into smaller units of meaning or combined into larger units of meaning. And the meaning of any unit depends heavily on the syntactic, semantic, and pragmatic context in which the unit is located. Natural language has words of various sorts: nouns and verbs and such. If someone says "dog" in isolation, you think of it as a noun, but you can also use the word in other ways. That is, a noun can function as a verb, an adjective, or an adverb when the context demands it. If you dog a dog during the dog days of summer, you'll be a dog tired dog-catcher.[3] Perl also evaluates words differently in various contexts. We will see how it does that later. Just remember that Perl is trying to understand what you're saying, like any good listener does. Perl works pretty hard to try to keep up its end of the bargain. Just say what you mean, and Perl will usually "get it". (Unless you're talking nonsense, of course—the Perl parser understands Perl a lot better than either English or Swahili.)

But back to nouns. A noun can name a particular object, or it can name a class of objects generically without specifying which one is currently being referred to. Most computer languages make this distinction, only we call the particular one a value and the generic one a variable. A value just exists somewhere, who knows where, but a variable gets associated with one or more values over its lifetime. So whoever is interpreting the variable has to keep track of that association. That interpreter may be in your brain or in your computer.

Variable Syntax

A variable is just a handy place to keep something, a place with a name, so you know where to find your special something when you come back looking for it later. As in real life, there are various kinds of places to store things, some of them rather private, and some of them out in public. Some places are temporary, and other places are more permanent. Computer scientists love to talk about the "scope" of variables, but that's all they mean by it. Perl has various handy ways of dealing with scoping issues, which you'll be happy to learn later when the time is right. Which is not yet. (Look up the adjectives `local`, `my`, `our`, and `state` in Chapter 27, when you get curious, or see "Scoped Declarations" on page 155 in Chapter 4.)

3. And you're probably dog tired of all this linguistics claptrap. But we'd like you to understand why Perl is different from the typical computer language, doggone it!

But a more immediately useful way of classifying variables is by what sort of data they can hold. As in English, Perl's primary type distinction is between singular and plural data. Strings and numbers are singular pieces of data, while lists of strings or numbers are plural. (And when we get to object-oriented programming, you'll find that the typical object looks singular from the outside but plural from the inside, like a class of students.) We call a singular variable a *scalar*, and a plural variable an *array*. Since a string can be stored in a scalar variable, we might write a slightly longer (and commented) version of our first example like this:

```
my $phrase = "Howdy, world!\n";       # Create a variable.
print $phrase;                        # Print the variable.
```

The `my` tells Perl that `$phrase` is a brand new variable, so it shouldn't go and look for an existing one. Note that we do not have to be very specific about what kind of variable `$phrase` is. The `$` character tells Perl that `phrase` is a scalar variable; that is, one containing a singular value. An array variable, by contrast, would start with an `@` character. (It may help you to remember that a `$` is a stylized "s" for "scalar", while `@` is a stylized "a" for "array".)[4]

Perl has some other variable types, with unlikely names like "hash", "handle", and "typeglob". Like scalars and arrays, these types of variables are also preceded by funny characters, commonly known as *sigils*. For completeness, Table 1-1 lists all the sigils you'll encounter.

Table 1-1. Variable types and their uses

Type	Sigil	Example	Is a Name For
Scalar	`$`	`$cents`	An individual value (number or string)
Array	`@`	`@large`	A list of values, keyed by number
Hash	`%`	`%interest`	A group of values, keyed by string
Subroutine	`&`	`&how`	A callable chunk of Perl code
Typeglob	`*`	`*struck`	Everything named `struck`

Some language purists point to these sigils as a reason to abhor Perl. This is superficial. Sigils have many benefits, not least of which is that variables can be interpolated into strings with no additional syntax. Perl scripts are also easy to read (for people who have bothered to learn Perl!) because the nouns stand out from verbs. And new verbs can be added to the language without breaking old scripts. (We told you Perl was designed to evolve.) And the noun analogy is not

4. This is a simplification of the real story of sigils, which we'll tell you more about in Chapter 2.

frivolous—there is ample precedent in English and other languages for requiring grammatical noun markers. It's how we think! (We think.)

Singularities

From our earlier example, you can see that scalars may be assigned a new value with the = operator, just as in many other computer languages. Scalar variables can be assigned any form of scalar value: integers, floating-point numbers, strings, and even esoteric things like references to other variables, or to objects. There are many ways of generating these values for assignment.

As in the Unix[5] shell, you can use different quoting mechanisms to make different kinds of values. Double quotation marks (double quotes) do *variable interpolation*[6] and *backslash interpolation* (such as turning \n into a newline), while single quotes suppress interpolation. And backquotes (the ones leaning to the left) will execute an external program and return the output of the program, so you can capture it as a single string containing all the lines of output.

```
my $answer = 42;                # an integer
my $pi = 3.14159265;            # a "real" number
my $avocados = 6.02e23;         # scientific notation
my $pet = "Camel";              # string
my $sign = "I love my $pet";    # string with interpolation
my $cost = 'It costs $100';     # string without interpolation
my $thence = $whence;           # another variable's value
my $salsa = $moles * $avocados; # a gastrochemical expression
my $exit = system("vi $file");  # numeric status of a command
my $cwd = `pwd`;                # string output from a command
```

And while we haven't covered fancy values yet, we should point out that scalars may also hold references to other data structures, including subroutines and objects.

```
my $ary = \@myarray;              # reference to a named array
my $hsh = \%myhash;               # reference to a named hash
my $sub = \&mysub;                # reference to a named subroutine

my $ary = [1,2,3,4,5];            # reference to an unnamed array
my $hsh = {Na => 19, Cl => 35};   # reference to an unnamed hash
my $sub = sub { print $state };   # reference to an unnamed subroutine
```

5. Here and elsewhere, when we say Unix we mean any operating system resembling Unix, including BSD, Mac OS X, Linux, Solaris, AIX, and, of course, Unix.

6. Sometimes called "substitution" by shell programmers, but we prefer to reserve that word for something else in Perl. So please call it interpolation. We're using the term in the textual sense ("this passage is a Gnostic interpolation") rather than in the mathematical sense ("this point on the graph is an interpolation between two other points").

```
my $fido = Camel->new("Amelia");  # reference to an object
```

When you create a new scalar variable, but before you assign it a value, it is automatically initialized with the value we call `undef`, which as you might guess means "undefined". Depending on context, this undefined value might be interpreted as a slightly more defined null value, such as `""` or `0`. More generally, depending on how you use them, variables will be interpreted automatically as strings, as numbers, or as "true" and "false" values (commonly called *Boolean* values). Remember how important context is in human languages. In Perl, various operators expect certain kinds of singular values as parameters, so we will speak of those operators as "providing" or "supplying" scalar context to those parameters. Sometimes we'll be more specific and say it supplies a numeric context, a string context, or a Boolean context to those parameters. (Later we'll also talk about list context, which is the opposite of scalar context.) Perl will automatically convert the data into the form required by the current context, within reason. For example, suppose you said this:

```
my $camels = "123";
print $camels + 1, "\n";
```

The first assigned value of `$camels` is a string, but it is converted to a number to add `1` to it, and then converted back to a string to be printed out as `124`. The newline, represented by `"\n"`, is also in string context, but since it's already a string, no conversion is necessary. But notice that we had to use double quotes there—using single quotes to say `'\n'` would result in a two-character string consisting of a backslash followed by an "`n`", which is not a newline by anybody's definition.

So, in a sense, double quotes and single quotes are yet another way of specifying context. The interpretation of the innards of a quoted string depends on which quotes you use. (Later, we'll see some other operators that work like quotes syntactically but use the string in some special way, such as for pattern matching or substitution. These all work like double-quoted strings, too. The *double-quote* context is the "interpolative" context of Perl, and it is supplied by many operators that don't happen to resemble double quotes.)

Similarly, a reference behaves as a reference when you give it a "dereference" context, but otherwise acts like a simple scalar value. For example, we might say:

```
my $fido = Camel->new("Amelia");
if (not $fido) { die "dead camel"; }
$fido->saddle();
```

Here we create a reference to a Camel object and put it into a new variable, `$fido`. On the next line, we test `$fido` as a scalar Boolean to see if it is "true", and

we throw an exception (that is, we complain) if it is not true, which in this case would mean that the `Camel->new` constructor failed to make a proper Camel object. But on the last line, we treat `$fido` as a reference by asking it to look up the `saddle` method for the object held in `$fido`, which happens to be a Camel, so Perl looks up the `saddle` method for Camel objects. More about that later. For now, just remember that context is important in Perl because that's how Perl knows what you want without your having to say it explicitly, as many other computer languages force you to do.

Pluralities

Some kinds of variables hold multiple values that are logically tied together. Perl has two types of multivalued variables: arrays and hashes. In many ways, these behave like scalars—new ones can be declared with `my`, for instance, and they are automatically initialized to an empty state. But they are different from scalars in that, when you assign to them, they supply *list* context to the right side of the assignment rather than scalar context.

Arrays and hashes also differ from each other. You'd use an array when you want to look something up by number. You'd use a hash when you want to look something up by name. The two concepts are complementary—you'll often see people using an array to translate month numbers into month names, and a corresponding hash to translate month names back into month numbers. (Though hashes aren't limited to holding only numbers. You could have a hash that translates month names to birthstone names, for instance.)

Arrays. An *array* is an ordered list of scalars, accessed[7] by the scalar's position in the list. The list may contain numbers, strings, or a mixture of both. (It might also contain references to subarrays or subhashes.) To assign a list value to an array, simply group the values together (with a set of parentheses):

```
my @home = ("couch", "chair", "table", "stove");
```

Conversely, if you use `@home` in list context, such as on the right side of a list assignment, you get back out the same list you put in. So you could create four scalar variables from the array like this:

```
my ($potato, $lift, $tennis, $pipe) = @home;
```

These are called list assignments. They logically happen in parallel, so you can swap two existing variables by saying:

```
($alpha,$omega) = ($omega,$alpha);
```

7. Or keyed, or indexed, or subscripted, or looked up. Take your pick.

As in C, arrays are zero-based, so while you would talk about the first through fourth elements of the array, you would get to them with subscripts 0 through 3.[8] Array subscripts are enclosed in square brackets [like this], so if you want to select an individual array element, you would refer to it as `$home[n]`, where *n* is the subscript (one less than the element number) you want. See the example that follows. Since the element you are dealing with is a scalar, you always precede it with a `$`.

If you want to assign to one array element at a time, you can; the elements of the array are automatically created as needed, so you could write the earlier assignment as:

```
my @home;
$home[0] = "couch";
$home[1] = "chair";
$home[2] = "table";
$home[3] = "stove";
```

Here we see that you can create a variable with `my` without giving it an initial value. (We don't need to use `my` on the individual elements because the array already exists and knows how to create elements on demand.)

Since arrays are ordered, you can do various useful operations on them, such as the stack operations `push` and `pop`. A stack is, after all, just an ordered list with a beginning and an end. Especially an end. Perl regards the end of your array as the top of a stack. (Although most Perl programmers think of an array as horizontal, with the top of the stack on the right.)

Hashes. A *hash* is an unordered set of scalars, accessed[9] by some string value that is associated with each scalar. For this reason hashes are often called *associative arrays*. But that's too long for lazy typists, and we talk about them so often that we decided to name them something short and snappy. The other reason we picked the name "hash" is to emphasize the fact that they're disordered. (They are, coincidentally, implemented internally using a hash-table lookup, which is why hashes are so fast and stay so fast no matter how many values you put into them.) You can't `push` or `pop` a hash, though, because it doesn't make sense. A hash has no beginning or end. Nevertheless, hashes are extremely powerful and useful. Until you start thinking in terms of hashes, you aren't really thinking in Perl. Figure 1-1 shows the ordered elements of an array and the unordered (but named) elements of a hash.

8. If this seems odd to you, just think of the subscript as an offset; that is, the count of how many array elements come before it. Obviously, the first element doesn't have any elements before it, and so it has an offset of 0. This is how computers think. (We think.)

9. Or keyed, or indexed, or subscripted, or looked up. Take your pick.

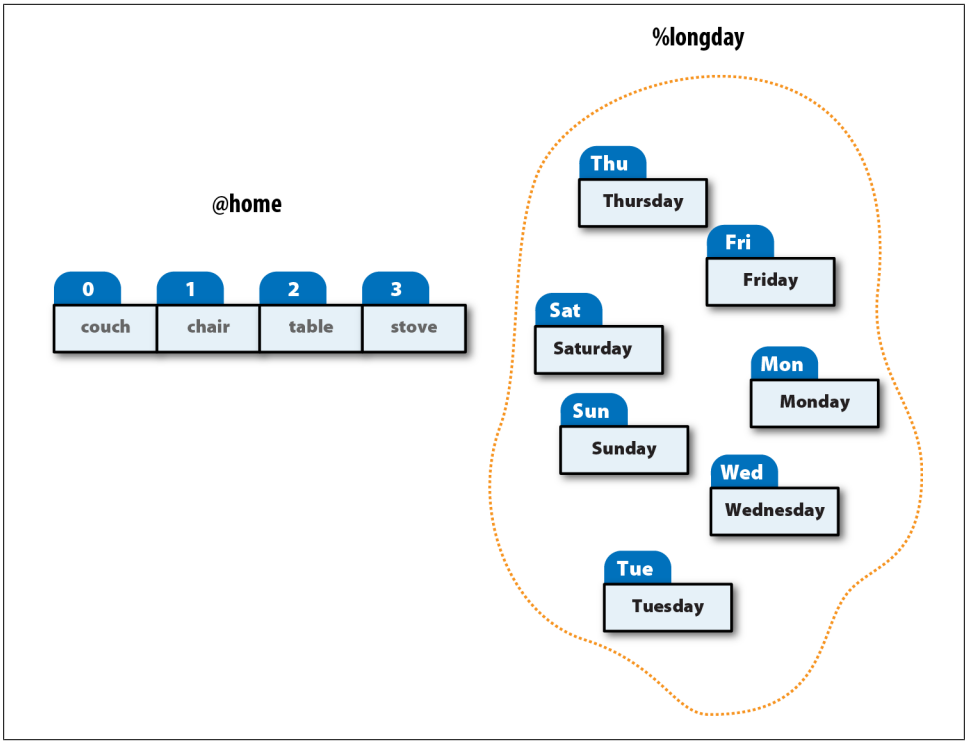

Figure 1-1. An array and a hash

Since the keys to a hash are not automatically implied by their position, you must supply the key as well as the value when populating a hash. You can still assign a list to it like an ordinary array, but each *pair* of items in the list will be interpreted as a key and a value. Since we're dealing with pairs of items, hashes use the `%` sigil to mark hash names. (If you look carefully at the `%` character, you can see the key and the value with a slash between them. It may help to squint.)

Suppose you wanted to translate abbreviated day names to the corresponding full names. You could write the following list assignment:

```
my %longday = ("Sun", "Sunday", "Mon", "Monday", "Tue", "Tuesday",
               "Wed", "Wednesday", "Thu", "Thursday", "Fri",
               "Friday", "Sat", "Saturday");
```

But that's rather difficult to read, so Perl provides the `=>` (equals sign, greater-than sign) sequence as an alternative separator to the comma. Using this syntactic sugar (and some creative formatting), it is much easier to see which strings are the keys and which strings are the associated values.

```
my %longday = (
    "Sun" => "Sunday",
    "Mon" => "Monday",
    "Tue" => "Tuesday",
    "Wed" => "Wednesday",
    "Thu" => "Thursday",
    "Fri" => "Friday",
    "Sat" => "Saturday",
);
```

Not only can you assign a list to a hash, as we did above, but if you mention a hash in list context, it'll convert the hash back to a list of key/value pairs, in a weird order. This is occasionally useful. More often people extract a list of just the keys, using the (aptly named) `keys` function. The key list is also unordered, but can easily be sorted if desired, using the (aptly named) `sort` function. Then you can use the ordered keys to pull out the corresponding values in the order you want.

Because hashes are a fancy kind of array, you select an individual hash element by enclosing the key in braces (those fancy brackets also known as "curlies"). So, for example, if you want to find out the value associated with `Wed` in the hash above, you would use `$longday{"Wed"}`. Note again that you are dealing with a scalar value, so you use `$` on the front, not `%`, which would indicate the entire hash.

Linguistically, the relationship encoded in a hash is genitive or possessive, like the word "of" in English, or like "'s". The wife *of* Adam is Eve, so we write:

```
my %wife;
$wife{"Adam"} = "Eve";
```

Complexities

Arrays and hashes are lovely, simple, flat data structures. Unfortunately, the world does not always cooperate with our attempts to oversimplify. Sometimes you need to build not-so-lovely, not-so-simple, not-so-flat data structures. Perl lets you do this by pretending that complicated values are really simple ones. To put it the other way around, Perl lets you manipulate simple scalar references that happen to refer to complicated arrays and hashes. We do this all the time in natural language when we use a simple singular noun like "government" to represent an entity that is completely convoluted and inscrutable. Among other things.

To extend our previous example, suppose we want to switch from talking about Adam's wife to Jacob's wife. Now, as it happens, Jacob had four wives. (Don't try this at home.) In trying to represent this in Perl, we find ourselves in the odd

situation where we'd like to pretend that Jacob's four wives were really one wife. (Don't try this at home, either.) You might think you could write it like this:

```
$wife{"Jacob"} = ("Leah", "Rachel", "Bilhah", "Zilpah");        # WRONG
```

But that wouldn't do what you want, because even parentheses and commas are not powerful enough to turn a list into a scalar in Perl. (Parentheses are used for syntactic grouping, and commas for syntactic separation.) Rather, you need to tell Perl explicitly that you want to pretend that a list is a scalar. It turns out that square brackets are powerful enough to do that:

```
$wife{"Jacob"} = ["Leah", "Rachel", "Bilhah", "Zilpah"];        # ok
```

That statement creates an unnamed array and puts a reference to it into the hash element `$wife{"Jacob"}`. So we have a named hash containing an unnamed array. This is how Perl deals with both multidimensional arrays and nested data structures. As with ordinary arrays and hashes, you can also assign individual elements, like this:

```
$wife{"Jacob"}[0] = "Leah";
$wife{"Jacob"}[1] = "Rachel";
$wife{"Jacob"}[2] = "Bilhah";
$wife{"Jacob"}[3] = "Zilpah";
```

You can see how that looks like a multidimensional array with one string subscript and one numeric subscript. To see something that looks more tree-structured, like a nested data structure, suppose we wanted to list not only Jacob's wives but all the sons of each of his wives. In this case we want to treat a hash as a scalar. We can use braces for that. (Inside each hash value we'll use square brackets to represent arrays, just as we did earlier. But now we have an array in a hash in a hash.)

```
my %kids_of_wife;
$kids_of_wife{"Jacob"} = {
    "Leah"   => ["Reuben", "Simeon", "Levi", "Judah", "Issachar", "Zebulun"],
    "Rachel" => ["Joseph", "Benjamin"],
    "Bilhah" => ["Dan", "Naphtali"],
    "Zilpah" => ["Gad", "Asher"],
};
```

That would be more or less equivalent to saying:

```
my %kids_of_wife;
$kids_of_wife{"Jacob"}{"Leah"}[0]   = "Reuben";
$kids_of_wife{"Jacob"}{"Leah"}[1]   = "Simeon";
$kids_of_wife{"Jacob"}{"Leah"}[2]   = "Levi";
$kids_of_wife{"Jacob"}{"Leah"}[3]   = "Judah";
$kids_of_wife{"Jacob"}{"Leah"}[4]   = "Issachar";
$kids_of_wife{"Jacob"}{"Leah"}[5]   = "Zebulun";
$kids_of_wife{"Jacob"}{"Rachel"}[0] = "Joseph";
```

```
$kids_of_wife{"Jacob"}{"Rachel"}[1] = "Benjamin";
$kids_of_wife{"Jacob"}{"Bilhah"}[0] = "Dan";
$kids_of_wife{"Jacob"}{"Bilhah"}[1] = "Naphtali";
$kids_of_wife{"Jacob"}{"Zilpah"}[0] = "Gad";
$kids_of_wife{"Jacob"}{"Zilpah"}[1] = "Asher";
```

You can see from this that adding a level to a nested data structure is like adding another dimension to a multidimensional array. Perl lets you think of it either way, but the internal representation is the same.

The important point here is that Perl lets you pretend that a complex data structure is a simple scalar. On this simple kind of encapsulation, Perl's entire object-oriented structure is built. When we earlier invoked the `Camel` constructor like this:

```
my $fido = Camel->new("Amelia");
```

we created a `Camel` object that is represented by the scalar `$fido`. But the inside of the `Camel` is more complicated. As well-behaved object-oriented programmers, we're not supposed to care about the insides of `Camel`s (unless we happen to be the people implementing the methods of the `Camel` class). But, generally, an object like a `Camel` would consist of a hash containing the particular `Camel`'s attributes, such as its name ("`Amelia`" in this case, not "`fido`"), and the number of humps (which we didn't specify, but probably defaults to 1; check the front cover).

Simplicities

If your head isn't spinning a bit from reading that last section, then you have an unusual head. People generally don't like to deal with complex data structures, whether governmental or genealogical. So, in our natural languages, we have many ways of sweeping complexity under the carpet. Many of these fall into the category of *topicalization*, which is just a fancy linguistics term for agreeing with someone about what you're going to talk about (and by exclusion, what you're probably not going to talk about). This happens on many levels in language. On a high level, we divide ourselves into various subcultures that are interested in various subtopics, and we establish sublanguages that talk primarily about those subtopics. The lingo of the doctor's office ("indissoluble asphyxiant") is different from the lingo of the chocolate factory ("everlasting gobstopper"). Most of us automatically switch contexts as we go from one lingo to another.

On a conversational level, the context switch has to be more explicit, so our language gives us many ways of saying what we're about to say. We put titles on our books and headers on our sections. On our sentences, we put quaint phrases like "In regard to your recent query" or "For all X". Usually, though, we just say

things like, "You know that dangly thingy that hangs down in the back of your throat?"

Perl also has several ways of topicalizing. One important topicalizer is the `package` declaration. Suppose you want to talk about `Camel`s in Perl. You'd likely start off your `Camel` module by saying:

```
package Camel;
```

This has several notable effects. One of them is that Perl will assume from this point on that any global verbs or nouns are about `Camel`s. It does this by automatically prefixing any global name[10] with the module name "`Camel::`". So if you say:

```
package Camel;
our $fido = &fetch();
```

then the real name of `$fido` is `$Camel::fido` (and the real name of `&fetch` is `&Camel::fetch`, but we're not talking about verbs yet). This means that if some other module says:

```
package Dog;
our $fido = &fetch();
```

Perl won't get confused, because the real name of this `$fido` is `$Dog::fido`, not `$Camel::fido`. A computer scientist would say that a package establishes a *namespace*. You can have as many namespaces as you like, but since you're only in one of them at a time, you can pretend that the other namespaces don't exist. That's how namespaces simplify reality for you. Simplification is based on pretending. (Of course, so is oversimplification, which is what we're doing in this chapter.)

Now it's important to keep your nouns straight, but it's just as important to keep your verbs straight. It's nice that `&Camel::fetch` is not confused with `&Dog::fetch` within the `Camel` and `Dog` namespaces, but the really nice thing about packages is that they classify your verbs so that *other* packages can use them. When we said:

```
my $fido = Camel->new("Amelia");
```

we were actually invoking the `&new` verb in the `Camel` package, which has the full name of `&Camel::new`. And when we said:

```
$fido->saddle();
```

10. You can declare global variables using `our`, which looks a lot like `my`, but tells people that it's a shared variable. A `my` variable is not shared and cannot be seen by anyone outside the current block. When in doubt, use `my` rather than `our` since unneeded globals just clutter up the world and confuse people.

we were invoking the `&Camel::saddle` routine, because `$fido` remembers that it is pointing to a `Camel`. This is how object-oriented programming works.

When you say `package Camel`, you're starting a new package. But sometimes you just want to borrow the nouns and verbs of an existing package. Perl lets you do that with a `use` declaration, which not only borrows verbs from another package, but also checks that the module you name is loaded in from disk. In fact, you *must* say something like:

```
use Camel;
```

before you say:

```
my $fido = Camel->new("Amelia");
```

because otherwise Perl wouldn't know what a `Camel` is.

The interesting thing is that you yourself don't really need to know what a `Camel` is, provided you can get someone else to write the `Camel` module for you. Even better would be if someone had *already* written the `Camel` module for you. It could be argued that the most powerful thing about Perl is not Perl itself, but CPAN (Comprehensive Perl Archive Network; see Chapter 19), which contains myriad modules that accomplish many different tasks that you don't have to know how to do. You just have to download whatever module you like and say:

```
use Some::Cool::Module;
```

Then you can use the verbs from that module in a manner appropriate to the topic under discussion.

So, like topicalization in a natural language, topicalization in Perl "warps" the language that you'll use from there to the end of the scope. In fact, some of the built-in modules don't actually introduce verbs at all, but simply warp the Perl language in various useful ways. We call these special modules *pragmas* (see Chapter 29). For instance, you'll often see people use the pragma `strict`, like this:

```
use strict;
```

What the `strict` module does is tighten up some of the rules so that you have to be more explicit about various things that Perl would otherwise guess about, such as how you want your variables to be scoped.[11] Making things explicit is helpful when you're working on large projects. By default, Perl is optimized for small projects, but with the `strict` pragma, Perl is also good for large projects that need to be more maintainable. Since you can add the `strict` pragma at any

11. More specifically, `use strict` requires you to use `my` , `state`, or `our` on variable declarations; otherwise, it just assumes undeclared variables are package variables, which can get you into trouble later. It also disallows various constructs that have proven to be error-prone over the years.

time, Perl is also good for evolving small projects into large ones, even when you didn't expect that to happen. Which is usually.

As Perl evolves, the Perl community also evolves, and one of the things that changes is how the community thinks Perl should behave by default. (This is in conflict with the desire for Perl to behave as it always did.) So, for instance, most Perl programmers now think that you should always put "`use strict`" at the front of your program. Over time we tend to accumulate such "culturally required" language-warping pragmas. So another built-in pragma is just the version number of Perl, which is a kind of "metapragma" that tells Perl it's okay to behave like a more modern language in all the ways it should:

```
use v5.14;
```

This particular declaration turns on several pragmas including "`use strict`";[12] it also enables new features like the `say` verb, which (unlike `print`) adds a newline for you. So we could have written our very first example above as:

```
use v5.14;
say "Howdy, world!";
```

The examples in this book all assume the v5.14 release of Perl; we will try to remember to include the `use v5.14` for you when we show you a complete program, but when we show you snippets, we will assume you've already put in that declaration yourself. (If you do not have the latest version of Perl, some of our examples may not work. In the case of `say`, you could change it back to a `print` with a newline—but it would be better to upgrade. You'll need to say at least `use v5.10` for `say` to work.)

Verbs

As is typical of your typical imperative computer language, many of the verbs in Perl are commands: they tell the Perl interpreter to do something. On the other hand, as is typical of a natural language, the meanings of Perl verbs tend to mush off in various directions depending on the context. A statement starting with a verb is generally purely imperative and evaluated entirely for its side effects. (We sometimes call these verbs *procedures*, especially when they're user-defined.) A frequently seen built-in command (in fact, you've seen it already) is the `say` command:

```
say "Adam's wife is $wife{'Adam'}.";
```

12. The implicit strictures feature was added in v5.12. Also see the `feature` pragma in Chapter 29.

This has the side effect of producing the desired output:

```
Adam's wife is Eve.
```

But there are other "moods" besides the imperative mood. Some verbs are for asking questions and are useful in conditionals such as `if` statements. Other verbs translate their input parameters into return values, just as a recipe tells you how to turn raw ingredients into something (hopefully) edible. We tend to call these verbs *functions*, in deference to generations of mathematicians who don't know what the word "functional" means in normal English.

An example of a built-in function would be the exponential function:

```
my $e = exp(1);   # 2.718281828459 or thereabouts
```

But Perl doesn't make a hard distinction between procedures and functions. You'll find the terms used interchangeably. Verbs are also sometimes called operators (when built-in), or subroutines (when user-defined).[13] But call them whatever you like—they all return a value, which may or may not be a meaningful value, which you may or may not choose to ignore.

As we go on, you'll see additional examples of how Perl behaves like a natural language. But there are other ways to look at Perl, too. We've already sneakily introduced some notions from mathematical language, such as subscripts, addition, and the exponential function. But Perl is also a control language, a glue language, a prototyping language, a text-processing language, a list-processing language, and an object-oriented language. Among other things.

But Perl is also just a plain old computer language. And that's how we'll look at it next.

An Average Example

Suppose you've been teaching a Perl class, and you're trying to figure out how to grade your students. You have a set of exam scores for each member of a class, in random order. You'd like a combined list of all the grades for each student, plus their average score. You have a text file (imaginatively named *grades*) that looks like this:

13. Historically, Perl required you to put an ampersand character (`&`) on any calls to user-defined subroutines (see `$fido = &fetch();` earlier). But with Perl v5, the ampersand became optional, so user-defined verbs can now be called with the same syntax as built-in verbs (`$fido = fetch();`). We still use the ampersand when talking about the *name* of the routine, such as when we take a reference to it (`$fetcher = \&fetch;`). Linguistically speaking, you can think of the ampersand form `&fetch` as an infinitive, "to fetch", or the similar form "do fetch". But we rarely say "do fetch" when we can just say "fetch". That's the real reason we dropped the mandatory ampersand in v5.

```
Noël 25
Ben 76
Clementine 49
Norm 66
Chris 92
Doug 42
Carol 25
Ben 12
Clementine 0
Norm 66
...
```

You can use the following script to gather all their scores together, determine each student's average, and print them all out in alphabetical order. This program assumes rather naïvely that you don't have two Carols in your class. That is, if there is a second entry for Carol, the program will assume it's just another score for the first Carol (not to be confused with the first Noël).

By the way, the line numbers are not part of the program, any other resemblances to BASIC notwithstanding.

```
 1  #!/usr/bin/perl
 2  use v5.14;
 3
 4  open(GRADES, "<:utf8", "grades")  || die "Can't open grades: $!\n";
 5  binmode(STDOUT, ':utf8');
 6
 7  my %grades;
 8  while (my $line = <GRADES>) {
 9      my ($student, $grade) = split(" ", $line);
10      $grades{$student} .= $grade . " ";
11  }
12
13  for my $student (sort keys %grades) {
14      my $scores = 0;
15      my $total = 0;
16      my @grades = split(" ", $grades{$student});
17      for my $grade (@grades) {
18          $total += $grade;
19          $scores++;
20      }
21      my $average = $total / $scores;
22      print "$student: $grades{$student}\tAverage: $average\n";
23  }
```

Now, before your eyes cross permanently, we'd better point out that this example demonstrates a lot of what we've covered so far, plus quite a bit more that we'll explain presently. But if you let your eyes go just a little out of focus, you may start to see some interesting patterns. Take some wild guesses now as to what's going on, and then later on we'll tell you if you're right.

We'd tell you to try running it, but you may not know how yet.

How to Do It

Gee, right about now you're probably wondering how to run a Perl program. The short answer is that you feed it to the Perl language interpreter program, which coincidentally happens to be named *perl*. The long answer starts out like this: There's More Than One Way To Do It.[14]

The first way to invoke *perl* (and the way most likely to work on any operating system) is to simply call *perl* explicitly from the command line.[15] If you are doing something fairly simple, you can use the *–e* switch (`%` in the following example represents a standard shell prompt, so don't type it). On Unix, you might type:

```
% perl -e 'print "Hello, world!\n";'
```

On other operating systems, you may have to fiddle with the quotes some. But the basic principle is the same: you're trying to cram everything Perl needs to know into 80 columns or so.[16]

For longer scripts, you can use your favorite text editor (or any other text editor) to put all your commands into a file and then, presuming you named the script *gradation* (not to be confused with graduation), you'd say:

```
% perl gradation
```

You're still invoking the Perl interpreter explicitly, but at least you don't have to put everything on the command line every time. And you no longer have to fiddle with quotes to keep the shell happy.

The most convenient way to invoke a script is just to name it directly (or click on it), and let the operating system find the interpreter for you. On some systems, there may be ways of associating various file extensions or directories with a particular application. On those systems, you should do whatever it is you do to associate the Perl script with the *perl* interpreter. On Unix systems that support the `#!` "shebang" notation (and most Unix systems do, nowadays), you can make the first line of your script be magical, so the operating system will know which program to run. Put a line resembling line 1 of our example into your program:

---

14. That's the Perl Slogan, and you'll get tired of hearing it, unless you're the Local Expert, in which case you'll get tired of saying it. Sometimes it's shortened to TMTOWTDI, pronounced "tim-toady". But you can pronounce it however you like. After all, TMTOWTDI.

15. Assuming that your operating system provides a command-line interface. If not, you should upgrade.

16. These types of scripts are often referred to as "one-liners". If you ever end up hanging out with other Perl programmers, you'll find that some of us are quite fond of creating intricate one-liners. Perl has occasionally been maligned as a write-only language because of these shenanigans.

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

(If *perl* v5.14 isn't in */usr/bin*, you'll have to change the `#!` line accordingly.[17]). Then all you have to say is:

```
% gradation
```

Of course, this didn't work because you forgot to make sure the script was executable (see the manpage for *chmod*(1)) and in your PATH. If it isn't in your PATH, you'll have to provide a complete filename so that the operating system knows how to find your script. Something like:

```
% /home/sharon/bin/gradation
```

Finally, if you are unfortunate enough to be on an ancient Unix system that doesn't support the magic `#!` line, or if the path to your interpreter is longer than 32 characters (a built-in limit on many systems), you may be able to work around it like this:

```
#!/bin/sh -- # perl, to stop looping
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if 0;
```

Some operating systems may require variants of this to deal with */bin/csh*, *DCL*, *COMMAND.COM*, or whatever happens to be your default command interpreter. Ask your Local Expert.

Throughout this book, we'll just use `#!/usr/bin/perl` to represent all these notions and notations, but you'll know what we really mean by it.

A random clue: when you write a test script, don't call your script *test*. Unix systems have a built-in *test* command, which will likely be executed instead of your script. Try *try* instead.

Now that you know how to run your own Perl program (not to be confused with the *perl* program), let's get back to our example.

Filehandles

Unless you're using artificial intelligence to model a solipsistic philosopher, your program needs some way to communicate with the outside world. In lines 4 and 8 of our Average Example you'll see the word `GRADES`, which exemplifies another of Perl's data types, the *filehandle*. A filehandle is just a name you give to a file, device, socket, or pipe to help you remember which one you're talking about,

17. If your */usr/bin/perl* is an old version, you can compile a new one and put it elsewhere, such as */usr/local/bin*, as long as you fix the `#!` line to point to it.

and to hide some of the complexities of buffering and such. (Internally, filehandles are similar to streams from a language like C++ or I/O channels from BASIC.)

Filehandles make it easier for you to get input from and send output to many different places. Part of what makes Perl a good glue language is that it can talk to many files and processes at once. Having nice symbolic names for various external objects is just part of being a good glue language.[18]

You create a filehandle and attach it to a file by using `open`. The `open` function takes at least two parameters: the filehandle and filename you want to associate it with. Perl also gives you some predefined (and preopened) filehandles. `STDIN` is your program's normal input channel, while `STDOUT` is your program's normal output channel. And `STDERR` is an additional output channel that allows your program to make snide remarks off to the side while it transforms (or attempts to transform) your input into your output.[19] In lines 4 and 5 of our program, we also tell our new `GRADES` filehandle and the existing `STDOUT` filehandle to assume that text is encoded in UTF-8, a common representation of Unicode text.

Since you can use the `open` function to create filehandles for various purposes (input, output, piping), you need to be able to specify which behavior you want. As you might do on the command line, you can simply add characters to the filename:

```
open(SESAME,    "filename")           # read from existing file
open(SESAME, "<  filename")           #   (same thing, explicitly)
open(SESAME, ">  filename")           # create file and write to it
open(SESAME, ">> filename")           # append to existing file
open(SESAME, "| output-pipe-command") # set up an output filter
open(SESAME, "input-pipe-command |")  # set up an input filter
```

However, the recommended three-argument form of `open` allows you to specify the `open` mode in an argument separate from the filename itself. This is useful when you're dealing with filenames that aren't literals and so might already contain characters that look like open modes or significant whitespace.

---

18. Some of the other things that make Perl a good glue language are: it handles non-ASCII data, it's embeddable, and you can embed other things in it via extension modules. It's concise, and it "networks" easily. It's environmentally conscious, so to speak. You can invoke it in many different ways (as we saw earlier). But most of all, the language itself is not so rigidly structured that you can't get it to "flow" around your problem. It comes back to that TMTOWTDI thing again.

19. These filehandles are typically attached to your terminal, so you can type to your program and see its output, but they may also be attached to files (and such). Perl can give you these predefined handles because your operating system already provides them, one way or another. Under Unix, processes inherit standard input, output, and error from their parent process, typically a shell. One of the duties of a shell is to set up these I/O streams so that the child process doesn't need to worry about them.

```
open(SESAME, "<",  $somefile)          # read from existing file
open(SESAME, ">",  $somefile)          # create file and write to it
open(SESAME, ">>", $somefile)          # append to existing file
open(SESAME, "|-", "output-pipe-command")  # set up an output filter
open(SESAME, "-|", "input-pipe-command")   # set up an input filter
```

As we did in our program, this form of `open` also lets you specify the character encoding of the file.

```
open(SESAME, "< :encoding(UTF-8)",     $somefile)
open(SESAME, "> :crlf",                $somefile)
open(SESAME, ">> :encoding(MacRoman)", $somefile)
```

As you can see, the name you pick for the filehandle is arbitrary. Once opened, the filehandle `SESAME` can be used to access the file or pipe until it is explicitly closed (with, you guessed it, `close(SESAME)`), or until the filehandle is attached to another file by a subsequent `open` on the same filehandle. Opening an already opened filehandle implicitly closes the first file, making it inaccessible to the filehandle, and opens a different file. You must be careful that this is what you really want to do. Sometimes it happens accidentally, like when you say `open($handle,$file)`, and `$handle` happens to contain a constant string. Be sure to set `$handle` to something unique, or you'll just open a new file on the same filehandle.

A much better idea is to leave `$handle` undefined, letting Perl fill it in for you. This is handy for when you get tired of choosing your own names for filehandles: if you pass `open` an undefined variable (such as `my` creates), Perl will pick the filehandle for you and fill it in automatically:

```
open(my $handle, "< :crlf :encoding(cp1252)", $somefile)
  || die "can't open $somefile: $!";
```

If the `open` succeeds, the `$handle` variable is now defined, and you can use it wherever a filehandle is expected.

Once you've opened a filehandle for input, you can read a line using the line reading operator, `<>`. This is also known as the angle operator because it's made of angle brackets. The angle operator encloses the filehandle (`<SESAME>` if a literal handle, and `<$handle>` for an indirect one) you want to read lines from. The empty angle operator, `<>`, will read lines from all the files specified on the command line, or `STDIN` if no arguments were specified. (This is standard behavior for many filter programs.) An example using the `STDIN` filehandle to read an answer supplied by the user would look something like this:

```
print STDOUT "Enter a number: ";      # ask for a number
$number = <STDIN>;                    # input the number
say STDOUT "The number is $number.";  # print the number
```

Did you see what we just slipped by you? What's that `STDOUT` doing there in those `print` and `say` statements? Well, that's just one of the ways you can use an output filehandle. A filehandle may be supplied between the command and its argument list, and if present, tells the output where to go. In this case, the filehandle is redundant because the output would have gone to `STDOUT` anyway. Much as `STDIN` is the default for input, `STDOUT` is the default for output. (In line 22 of our Average Example, we left it out to avoid confusing you until now.)

If you try the previous example, you may notice that you get an extra blank line. This happens because the line-reading operation does not automatically remove the newline from your input line (your input would be, for example, `"9\n"`). For those times when you do want to remove the newline, Perl provides the `chop` and `chomp` functions. `chop` will indiscriminately remove (and return) the last character of the string, while `chomp` will only remove the end of record marker (generally, `"\n"`) and return the number of characters so removed. You'll often see this idiom for inputting a single line:

```
chomp($number = <STDIN>);    # input a number, then remove its newline
```

which means the same thing as:

```
$number = <STDIN>;           # input a number
chomp($number);              # remove trailing newline
```

One last thing, just because we called our variable `$number` doesn't mean it was one. Any string will do. Perl only cares whether something is a number if you try to operate on that string as though it were a number—down which road lie operators, our next topic.

Operators

As we alluded to earlier, Perl is also a mathematical language. This is true at several levels, from low-level bitwise logical operations, up through number and set manipulation, on up to larger predicates and abstractions of various sorts. And as we all know from studying math in school, mathematicians love strange symbols. What's worse, computer scientists have come up with their own versions of these strange symbols. Perl has a number of these strange symbols, too—but take heart, as most are borrowed directly from C, FORTRAN, *sed*(1) or *awk*(1), so they'll at least be familiar to users of those languages.

The rest of you can take comfort in knowing that, by learning all these strange symbols in Perl, you've given yourself a head start on all those other strange languages.

Perl's built-in operators may be classified by number of operands into unary, binary, and trinary (or ternary) operators. They may be classified by whether they're prefix operators (which go in front of their operands) or infix operators (which go in between their operands). They may also be classified by the kinds of objects they work with, such as numbers, strings, or files. Later, we'll give you a table of all the operators, but first here are some handy ones to get you started.

Some Binary Arithmetic Operators

Arithmetic operators do what you would expect from learning them in school. They perform some sort of mathematical function on numbers; see Table 1-2.

Table 1-2. Mathematical operators

Example	Name	Result
`$a + $b`	Addition	Sum of `$a` and `$b`
`$a * $b`	Multiplication	Product of `$a` and `$b`
`$a % $b`	Modulus	Remainder of `$a` divided by `$b`
`$a ** $b`	Exponentiation	`$a` to the power of `$b`

Yes, we left out subtraction and division—we suspect you can figure out how they should work. Try them and see if you're right. (Or cheat and look in Chapter 3.) Arithmetic operators are evaluated in the order your math teacher taught you (exponentiation before multiplication; multiplication before addition). You can always use parentheses to make it come out differently.

String Operators

There is also an "addition" operator for strings that performs concatenation (that is, joining strings end to end). Unlike some languages that confuse this with numeric addition, Perl defines a separate operator (`.`) for string concatenation:

```
$a = 123;
$b = 456;
say $a + $b;     # prints 579
say $a . $b;     # prints 123456
```

There's also a "multiply" operator for strings, called the *repeat* operator. Again, it's a separate operator (`x`) to keep it distinct from numeric multiplication:

```
$a = 123;
$b = 3;
say $a * $b;     # prints 369
say $a x $b;     # prints 123123123
```

These string operators bind as tightly as their corresponding arithmetic operators. The repeat operator is a bit unusual in taking a string for its left argument but a number for its right argument. Note also how Perl is automatically converting from numbers to strings. You could have put all the literal numbers above in quotes, and it would still have produced the same output. Internally, though, it would have been converting in the opposite direction (that is, from strings to numbers).

A couple more things to think about. String concatenation is also implied by the interpolation that happens in double-quoted strings. And when you print out a list of values, you're also effectively concatenating strings. So the following three statements produce the same output:

```
say $a . " is equal to " . $b . ".";    # dot operator
say $a, " is equal to ", $b, ".";       # list
say "$a is equal to $b.";               # interpolation
```

Which of these you use in any particular situation is entirely up to you. (But in our opinion interpolation is often the most readable.)

The `x` operator may seem relatively worthless at first glance, but it is quite useful at times, especially for things like this:

```
say "-" x $scrwid;
```

which draws a line across your screen, presuming `$scrwid` contains your screen width, and not your screw identifier.

Assignment Operators

Although it's not exactly a mathematical operator, we've already made extensive use of the simple assignment operator, `=` . Try to remember that `=` means "gets set to" rather than "equals". (There is also a mathematical equality operator `==` that means "equals", and if you start out thinking about the difference between them now, you'll save yourself a lot of headache later. The `==` operator is like a function that returns a Boolean value, while `=` is more like a procedure that is evaluated for the side effect of modifying a variable.)

Like the operators described earlier, assignment operators are binary infix operators, which means they have an operand on either side of the operator. The right operand can be any expression you like, but the left operand must be a valid *lvalue* (which, when translated to English, means a valid storage location like a variable, or a location in an array). The most common assignment operator is simple assignment. It determines the value of the expression on its right side, and then sets the variable on the left side to that value:

```
$a = $b;
$a = $b + 5;
$a = $a * 3;
```

Notice the last assignment refers to the same variable twice; once for the computation, once for the assignment. There's nothing wrong with that, but it's a common enough operation that there's a shortcut for it (borrowed from C). If you say:

```
lvalue operator= expression
```

it is evaluated as if it were:

```
lvalue = lvalue operator expression
```

except that the lvalue is not computed twice. (This only makes a difference if evaluation of the lvalue has side effects. But when it *does* make a difference, it usually does what you want. So don't sweat it.)

So, for example, you could write the previous example as:

```
$a *= 3;
```

which reads "multiply `$a` by 3". You can do this with almost any binary operator in Perl, even some that you can't do it with in C:

```
$line .= "\n";  # Append newline to $line.
$fill x= 80;    # Make string $fill into 80 repeats of itself.
$val ||= "2";   # Set $val to 2 if it isn't already "true".
```

Line 10 of our Average Example[20] contains two string concatenations, one of which is an assignment operator. And line 18 contains a `+=`.

Regardless of which kind of assignment operator you use, the final value of the variable on the left is returned as the value of the assignment as a whole.[21] This will not surprise C programmers, who will already know how to use this idiom to zero out variables:

```
$a = $b = $c = 0;
```

You'll also frequently see assignment used as the condition of a `while` loop, as in line 8 of our Average Example.

What *will* surprise C programmers is that assignment in Perl returns the actual variable as an lvalue, so you can modify the same variable more than once in a statement. For instance, you could say:

```
($temp -= 32) *= 5/9;
```

20. Thought we'd forgotten it, didn't you?

21. This is unlike, say, Pascal, in which assignment is a statement and returns no value. We said earlier that assignment is like a procedure, but remember that in Perl, even procedures return values.

to do an in-place conversion from Fahrenheit to Celsius. This is also why earlier in this chapter we could say:

```
chop($number = <STDIN>);
```

and have it chop the final value of `$number`. Generally speaking, you can use this feature whenever you want to copy something and at the same time do something else with it.

Unary Arithmetic Operators

As if `$variable += 1` weren't short enough, Perl borrows from C an even shorter way to increment a variable. The autoincrement (and autodecrement) operators simply add (or subtract) one from the value of the variable. They can be placed on either side of the variable, depending on when you want them to be evaluated; see Table 1-3.

Table 1-3. Increment operators

Example	Name	Result
`++$a, $a++`	Autoincrement	Add 1 to `$a`
`--$a, $a--`	Autodecrement	Subtract 1 from `$a`

If you place one of these "auto" operators before the variable, it is known as a preincremented (predecremented) variable. Its value will be changed before it is referenced. If it is placed after the variable, it is known as a postincremented (postdecremented) variable, and its value is changed after it is used. For example:

```
$a = 5;        # $a is assigned 5
$b = ++$a;     # $b is assigned the incremented value of $a, 6
$c = $a--;     # $c is assigned 6, then $a is decremented to 5
```

Line 15 of our Average Example increments the number of scores by one so that we'll know how many scores we're averaging. It uses a postincrement operator (`$scores++`), but in this case it doesn't matter since the expression is in void context, which is just a funny way of saying that the expression is being evaluated only for the side effect of incrementing the variable. The value returned is being thrown away.[22]

22. The optimizer will notice this and optimize the postincrement into a preincrement, because that's a bit faster to execute. (You didn't need to know that, but we hoped it would cheer you up.)

Logical Operators

Logical operators, also known as "short-circuit" operators, allow the program to make decisions based on multiple criteria without using nested `if` statements. They are known as short-circuit operators because they skip (short circuit) the evaluation of their right argument if they decide the left argument has already supplied enough information to decide the overall value. This is not just for efficiency. You are explicitly allowed to depend on this short-circuiting behavior to avoid evaluating code in the right argument that you know would blow up if the left argument were not "guarding" it. You can say "California or bust!" in Perl without busting (presuming you do get to California).

Perl actually has two sets of logical operators: a traditional set borrowed from C and a newer (but even more traditional) set of ultralow-precedence operators borrowed from BASIC. Both sets contribute to readability when used appropriately. C's punctuational operators work well when you want your logical operators to bind more tightly than commas, while BASIC's word-based operators work well when you want your commas to bind more tightly than your logical operators. Often they work the same, and which set you use is a matter of personal preference. (For contrastive examples, see the section "Logical and, or, not, and xor" on page 127 in Chapter 3.) Although the two sets of operators are not interchangeable due to precedence, once they're parsed, the operators themselves behave identically; precedence merely governs the extent of their arguments. Table 1-4 lists logical operators.

Table 1-4. Logical operators

Example	Name	Result
`$a && $b`	And	`$a` if `$a` is false, `$b` otherwise
`$a || $b`	Or	`$a` if `$a` is true, `$b` otherwise
`! $a`	Not	True if `$a` is not true
`$a and $b`	And	`$a` if `$a` is false, `$b` otherwise
`$a or $b`	Or	`$a` if `$a` is true, `$b` otherwise
`not $a`	Not	True if `$a` is not true
`$a xor $b`	Xor	True if `$a` or `$b` is true, but not both

Since the logical operators "short circuit" the way they do, they're often used in Perl to conditionally execute code. The following line (line 4 from our Average Example) tries to open the file *grades*:

```
open(GRADES, "<:utf8", "grades")  || die "Can't open file grades: $!\n";
```

If it opens the file, it will jump to the next line of the program. If it can't open the file, it will provide us with an error message and then stop execution.

Literally, this line means "Open *grades* or bust!" Besides being another example of natural language, the short-circuit operators preserve the visual flow. Important actions are listed down the left side of the screen, and secondary actions are hidden off to the right. (The `$!` variable contains the error message returned by the operating system—see Chapter 25.) Of course, these logical operators can also be used within the more traditional kinds of conditional constructs, such as the `if` and `while` statements.

Some Numeric and String Comparison Operators

Comparison, or relational, operators tell us how two scalar values (numbers or strings) relate to each other. There are two sets of operators: one does numeric comparison and the other does string comparison. (In either case, the arguments will be "coerced" to have the appropriate type first.) Assuming left and right arguments of `$a` and `$b`, Table 1-5 shows us what we have.

Table 1-5. Comparison operators

Comparison	Numeric	String	Return Value
Equal	`==`	`eq`	True if `$a` is equal to `$b`
Not equal	`!=`	`ne`	True if `$a` is not equal to `$b`
Less than	`<`	`lt`	True if `$a` is less than `$b`
Greater than	`>`	`gt`	True if `$a` is greater than `$b`
Less than or equal	`<=`	`le`	True if `$a` not greater than `$b`
Greater than or equal	`>=`	`ge`	True if `$a` not less than `$b`
Comparison	`<=>`	`cmp`	0 if equal, 1 if `$a` greater, –1 if `$b` greater

The last pair of operators (`<=>` and `cmp`) are entirely redundant with the earlier operators. However, they're incredibly useful in `sort` subroutines (see Chapter 27).[23]

23. Some folks feel that such redundancy is evil because it keeps a language from being minimalistic, or orthogonal. But Perl isn't an orthogonal language; it's a diagonal language. By this we mean that Perl doesn't force you to always go at right angles. Sometimes you just want to follow the hypotenuse of the triangle to get where you're going. TMTOWTDI is about shortcuts. Shortcuts are about programmer efficiency.

Some File Test Operators

The file test operators allow you to test whether certain file attributes are set before you go and blindly muck about with the files. The most basic file attribute is, of course, whether the file exists. For example, it would be very nice to know whether your mail aliases file already exists before you go and open it as a new file, wiping out everything that was in there before. Table 1-6 gives a few of the file test operators.

Table 1-6. File test operators

Example	Name	Result
`-e $a`	Exists	True if file named in `$a` exists
`-r $a`	Readable	True if file named in `$a` is readable
`-w $a`	Writable	True if file named in `$a` is writable
`-d $a`	Directory	True if file named in `$a` is a directory
`-f $a`	File	True if file named in `$a` is a regular file
`-T $a`	Text File	True if file named in `$a` is a text file

You might use them like this:

```
-e "/usr/bin/perl" or warn "Perl is improperly installed.\n";
-f "/vmlinuz" and say "I see you are a friend of Linus.";
```

Note that a regular file is not the same thing as a text file. Binary files like */vmlinuz* are regular files, but they aren't text files. Text files are the opposite of binary files, while regular files are the opposite of "irregular" files like directories and devices.

There are a lot of file test operators, many of which we didn't list. Most of the file tests are unary Boolean operators, which is to say they take only one operand (a scalar that evaluates to a filename or a filehandle), and they return either a true or false value. A few of them return something fancier, like the file's size or age, but you can look those up when you need them in the section "Named Unary and File Test Operators" on page 106 in Chapter 3.

Control Structures

So far, except for our one large example, all of our examples have been completely linear; we executed each command in order. We've seen a few examples of using the short-circuit operators to cause a single command to be (or not to be) executed. While you can write some very useful linear programs (a lot of CGI scripts

fall into this category), you can write much more powerful programs if you have conditional expressions and looping mechanisms. Collectively, these are known as control structures. So you can also think of Perl as a control language.

But to have control, you have to be able to decide things, and to decide things, you have to know the difference between what's true and what's false.

What Is Truth?

We've bandied about the term truth,[24] and we've mentioned that certain operators return a true or a false value. Before we go any further, we really ought to explain exactly what we mean by that. Perl treats truth a little differently than most computer languages, but after you've worked with it a while, it will make a lot of sense. (Actually, we hope it'll make a lot of sense after you've read the following.)

Basically, Perl holds truths to be self-evident. That's a glib way of saying that you can evaluate almost anything for its truth value. Perl uses practical definitions of truth that depend on the type of thing you're evaluating. As it happens, there are many more kinds of truth than there are of nontruth.

Truth in Perl is always evaluated in scalar context. Other than that, no type coercion is done. So here are the rules for the various kinds of values a scalar can hold:

1. Any string is true except for `""` and `"0"`.
2. Any number is true except for 0.
3. Any reference is true.
4. Any undefined value is false.

Actually, the last two rules can be derived from the first two. Any reference (rule 3) would point to something with an address and would evaluate to a number or string containing that address, which is never 0 because it's always defined. And any undefined value (rule 4) would always evaluate to 0 or the null string.

And, in a way, you can derive rule 2 from rule 1 if you pretend that everything is a string. Again, no string coercion is actually done to evaluate truth, but if the string coercion *were* done, then any numeric value of 0 would simply turn into the string `"0"` and be false. Any other number would not turn into the string `"0"`, and so would be true. Let's look at some examples so we can understand this better:

24. Strictly speaking, this is not true.

```
0          # would become the string "0", so false.
1          # would become the string "1", so true.
10 - 10    # 10 minus 10 is 0, would convert to string "0", so false.
0.00       # equals 0, would convert to string "0", so false.
"0"        # is the string "0", so false.
""         # is a null string, so false.
"0.00"     # is the string "0.00", neither "" nor "0", so true!
"0.00" + 0 # would become the number 0 (coerced by the +), so false.
\$a        # is a reference to $a, so true, even if $a is false.
undef()    # is a function returning the undefined value, so false.
```

Since we mumbled something earlier about truth being evaluated in scalar context, you might be wondering what the truth value of a list is. Well, the simple fact is none of the operations in Perl will return a list in scalar context. They'll all notice they're in scalar context and return a scalar value instead, and then you apply the rules of truth to that scalar. So there's no problem, as long as you can figure out what any given operator will return in scalar context. As it happens, both arrays and hashes return scalar values that conveniently happen to be true if the array or hash contains any elements. More on that later.

The if and unless statements

We saw earlier how a logical operator could function as a conditional. A slightly more complex form of the logical operators is the `if` statement. The `if` statement evaluates a truth condition (that is, a Boolean expression) and executes a block if the condition is true:

```
if ($debug_level > 0) {
    # Something has gone wrong.  Tell the user.
    say "Debug: Danger, Will Robinson, danger!";
    say "Debug: Answer was '54', expected '42'.";
}
```

A block is one or more statements grouped together by a set of braces. Since the `if` statement executes a block, the braces are required by definition. If you know a language like C, you'll notice that this is different. Braces are optional in C if you have a single statement, but the braces are not optional in Perl.

Sometimes just executing a block when a condition is met isn't enough. You may also want to execute a different block if that condition *isn't* met. While you could certainly use two `if` statements, one the negation of the other, Perl provides a more elegant solution. After the block, `if` can take an optional second condition, called `else`, to be executed only if the truth condition is false. (Veteran computer programmers will not be surprised at this point.)

At times you may even have more than two possible choices. In this case, you'll want to add an `elsif` truth condition for the other possible choices. (Veteran computer programmers may well be surprised by the spelling of "`elsif`", for which nobody here is going to apologize. Sorry.)

```
if ($city eq "New York") {
    say "New York is northeast of Washington, D.C.";
}
elsif ($city eq "Chicago") {
    say "Chicago is northwest of Washington, D.C.";
}
elsif ($city eq "Miami") {
    say "Miami is south of Washington, D.C.  And much warmer!";
}
else {
    say "I don't know where $city is, sorry.";
}
```

The `if` and `elsif` clauses are each computed in turn, until one is found to be true or the `else` condition is reached. When one of the conditions is found to be true, its block is executed and all remaining branches are skipped. Sometimes, you don't want to do anything if the condition is true, only if it is false. Using an empty `if` with an `else` may be messy, and a negated `if` may be illegible; it sounds weird in English to say "if not this is true, do something". In these situations, you would use the `unless` statement:

```
unless ($destination eq $home) {
    say "I'm not going home.";
}
```

There is no `elsunless` though. This is generally construed as a feature.

The given and when Statements

To test a single value for a bunch of different alternatives, recent versions of Perl have what other languages sometimes call `switch` and `case`. Because we like to make Perl work like a natural language, however, we call these `given` and `when`. (Since you're already putting `use v5.14` at the top, you should have this functionality, which was introduced in 5.10.)

```
#!/usr/bin/perl
use v5.14;

print "What is your favorite color? ";
chomp(my $answer = <STDIN>);

given ($answer) {
    when ("purple")      { say "Me too." }
```

```
        when ("green")       { say "Go!" }
        when ("yellow")      { say "Slow!" }
        when ("red")         { say "Stop!" }

        when ("blue")        { say "You may proceed." }
        when (/\w+, no \w+/) { die "AAAUUUGHHHHH!" }

        when (42)            { say "Wrong answer." }

        when (['gray','orange','brown','black','white']) {
            say "I think $answer is pretty okay too.";
        }

        default {
            say "Are you sure $answer is a real color?";
        }
    }
```

First the `given` part takes the value of its expression and makes it the topic of conversation, so the `when` statements know which value to test. The cases are then evaluated by matching the argument of each `when` against the topic to find the first `when` statement that thinks the topic's value matches. The `when` statements try to match in order, and as soon as one matches, it doesn't try any of the subsequent statements, but drops out of the whole `given` construct.

The form of each `when` argument (`"red"` vs `42` vs `/\w+, no \w+/`) determines the type of match performed, so strings match as strings, numbers match as numbers, and patterns match as, well, patterns. Lists of values match if any of them match. The `when` statement uses an underlying operation called "smartmatching" that is designed to match the way you expect most of the time, except when it doesn't. See "Smartmatch Operator" on page 112 in Chapter 3 for more on that.

Looping Constructs

These statements allow a Perl program to repeatedly execute the same code, so they are often known as *iterative* constructs. There are several kinds, which differ primarily in how you know when you're done with the loop and can go on to other things.

Conditional loops

The `while` and `until` statements test an expression for truth just as the `if` and `unless` statements do, except that they'll execute the block repeatedly as long as the condition is satisfied each time through. The condition is always checked before each iteration. If the condition is met (that is, if it is true for a `while` or false for an `until`), the block of the statement is executed.

```
print "How many tickets have we sold so far? ";
my $before = <STDIN>;

my $sold = $before;
while ($sold < 10000) {
    my $available = 10000 - $sold;
    print "$available tickets are available.  How many would you like: ";
    my $purchase = <STDIN>;
    if ($purchase > $available) {
        say "Too many! Try again.";
        $purchase = 0;
    }
    $sold += $purchase;
}

say "This show is sold out, please come back later.";
```

Note that if the original condition is never met, the loop will never be entered at all. For example, if we've already sold 10,000 tickets, we will report the show to be sold out immediately.

In our Average Example earlier, line 8 reads:

```
while (my $line = <GRADES>) {
```

This assigns the next line to the variable `$line` and, as we explained earlier, returns the value of `$line` so that the condition of the `while` statement can evaluate `$line` for truth. You might wonder whether Perl will get a false negative on blank lines and exit the loop prematurely. The answer is that it won't. The reason is clear if you think about everything we've said. The line input operator leaves the newline on the end of the string, so a blank line has the value `"\n"`. And you know that `"\n"` is not one of the canonical false values. So the condition is true, and the loop continues even on blank lines.

On the other hand, when we finally do reach the end of the file, the line input operator returns the undefined value, which always evaluates to false. And the loop terminates, just when we wanted it to. There's no need for an explicit test of the `eof` function in Perl, because the input operators are designed to work smoothly in a conditional context.

In fact, almost everything is designed to work smoothly in a conditional (Boolean) context. If you mention an array in scalar context, the length of the array is returned. So you often see command-line arguments processed like this:

```
while (@ARGV) {
    process(shift @ARGV);
}
```

The `shift` operator removes one element from the argument list each time through the loop (and returns that element). The loop automatically exits when array `@ARGV` is exhausted; that is, when its length goes to 0. And 0 is already false in Perl. In a sense, the array itself has become "false".[25]

The three-part loop

Another iterative statement is the three-part loop, also known as a C-style `for` loop. The three-part loop runs exactly like the `while` loop above, but it looks a bit different because two of the statements get moved into the official definition of the loop. (C programmers will find it very familiar though.)

```
print "How many tickets have we sold so far? ";
my $before = <STDIN>;

for (my $sold = $before; $sold < 10000; $sold += my $purchase) {
    my $available = 10000 - $sold;
    print "$available tickets are available.  How many would you like: ";
    $purchase = <STDIN>;
    if ($purchase > $available) {
        say "Too many! Try again.";
        $purchase = 0;
    }
}

say "This show is sold out, please come back later.";
```

Within the loop's parentheses, the three-part loop takes three expressions (hence the name), separated by two semicolons. The first expression sets the initial state of the loop variable. The second is a condition to test the loop variable; this works just like the `while` statement's condition. The third expression modifies the state of the loop variable; this expression is effectively executed at the end of each iteration, just as we did explicitly in the previous `while` loop.

When the three-part loop starts, the initial state is set and the truth condition is checked. If the condition is true, the block is executed. When the block finishes, the modification expression is executed, the truth condition is again checked, and, if true, the block is rerun with the next value. As long as the truth condition remains true, the block and the modification expression will continue to be executed. (Note that only the middle expression is evaluated for its value. The first

25. This is how Perl programmers think. So there's no need to compare 0 to 0 to see if it's false. Despite the fact that other languages force you to, don't go out of your way to write explicit comparisons like `while (@ARGV != 0)`. That's just inefficient for both you and the computer. And anyone who has to maintain your code.

and third expressions are evaluated only for their side effects, and the resulting values are thrown away!)

Each of the three expressions may be omitted, but the two semicolons are always required. If you leave out the middle expression, it assumes you want to loop forever, so you can write an infinite loop like this:

```
for (;;) {
    say "Take out the trash!";
    sleep(5);
}
```

The foreach loop

The last of Perl's iterative statements is known as the *foreach* loop.[26] This loop executes the same code for each of a known list of scalars, such as you might get from an array:

```
for my $user (@users) {
    if (-f "$home{$user}/.nexrc") {
        say "$user is cool... they use a perl-aware vi!";
    }
}
```

Unlike the `if` and `while` statements, which provide scalar context to a conditional expression, the foreach statement provides list context to the expression in parentheses. So the expression is evaluated to produce a list, if possible (and, if not, a single scalar value will be considered a list of one element). Then each element of the list is aliased to the loop variable in turn, and the block of code is executed once for each list element. Note that the loop variable refers to the element itself, rather than a copy of the element. Hence, modifying the loop variable also modifies the original array.

You'll find many more of these loops in the typical Perl program than traditional three-part `for` loops, because it's very easy in Perl to generate the kinds of lists that a foreach wants to iterate over. (That's partly why we stole `for`'s keyword, since we're lazy and think commonly used words should be short.) One idiom you'll often see is a loop to iterate over the sorted keys of a hash:

```
for my $key (sort keys %hash) {
```

In fact, line 13 of our Average Example does precisely that, so we can print out the students in alphabetical order.

---

26. Historically, it was written with the `foreach` keyword, hence the name. These days we tend to use the `for` keyword instead, since it reads more like English when you include a `my` declaration (and because the syntax cannot be confused with the three-part loop). So many of us never write `foreach` anymore, though you can still do that if you like.

Breaking out: next and last

The `next` and `last` operators allow you to modify the flow of your loop. It is not at all uncommon to have a special case; you may want to skip it, or you may want to quit when you encounter it. For example, if you are dealing with Unix accounts, you may want to skip the system accounts (like *root* or *lp*). The `next` operator would allow you to skip to the end of your current loop iteration and start the next iteration. The `last` operator would allow you to skip to the end of your block, as if your loop's test condition had returned false. This might be useful if, for example, you are looking for a specific account and want to quit as soon as you find it.

```
for my $user (@users) {
    if ($user eq "root" || $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Found the special account.\n";
        # do some processing
        last;
    }
}
```

It's possible to break out of multilevel loops by labeling your loops and specifying which loop you want to break out of. Together with statement modifiers (another form of conditional which we'll talk about later), this can make for extremely readable loop exits (if you happen to think English is readable):

```
LINE: while (my $line = <EMAIL>) {
    next LINE if $line eq "\n";   # skip blank lines
    last LINE if $line =~ /^>/;   # stop on first quoted line
    # your ad here
}
```

You may be saying, "Wait a minute, what's that funny `^>` thing there inside the leaning toothpicks? That doesn't look much like English." And you're right. That's a pattern match containing a regular expression (albeit a rather simple one). And that's what the next section is about. Perl is just about the best text-processing language in the world, and regular expressions are at the heart of Perl's text processing.

Regular Expressions

Regular expressions (a.k.a. regexes, regexps, or REs) are used by many search programs such as *grep* and *findstr*, text-munging programs like *sed* and *awk*, and editors like *vi* and *emacs*. A regular expression is a way of describing a set of strings

without having to list all the strings in your set.[27] Many other computer languages incorporate regular expressions (some of them even advertise "Perl5 regular expressions"!), but none of these languages integrates regular expressions into the language the way Perl does. Regular expressions are used several ways in Perl. First and foremost, they're used in conditionals to determine whether a string matches a particular pattern, because in a Boolean context they return true and false. So when you see something that looks like `/foo/` in a conditional, you know you're looking at an ordinary *pattern-matching* operator:

```
if (/Windows 7/) { print "Time to upgrade?\n" }
```

Second, if you can locate patterns within a string, you can replace them with something else. So when you see something that looks like `s/foo/bar/`, you know it's asking Perl to substitute "bar" for "foo", if possible. We call that the *substitution* operator. It also happens to return true or false depending on whether it succeeded, but usually it's evaluated for its side effect:

```
s/IBM/lenovo/;
```

Finally, patterns can specify not only where something is, but also where it *isn't*. So the `split` operator uses a regular expression to specify where the data isn't. That is, the regular expression defines the *separators* that delimit the fields of data. Our Average Example has a couple of trivial examples of this. Lines 9 and 16 each split strings on whitespace in order to return a list of words. But you can split on any separator you can specify with a regular expression:

```
my ($good, $bad, $ugly) = split(/,/, "vi,emacs,teco");
```

(There are various modifiers you can use in each of these situations to do exotic things like ignore case when matching alphabetic characters, but these are the sorts of gory details that we'll cover in Part II when we get to the gory details.)

The simplest use of regular expressions is to match a literal expression. In the case of the `split` above, we matched on a single comma character. But if you match on several characters in a row, they all have to match sequentially. That is, the pattern looks for a substring, much as you'd expect. Let's say we want to show all the lines of an HTML file that contain HTTP links (as opposed to FTP links). Let's imagine we're working with HTML for the first time, and we're being a little naïve. We know that these links will always have "`http:`" in them somewhere. We could loop through our file with this:

27. A good source of information on regular expression concepts is Jeffrey Friedl's book, *Mastering Regular Expressions*.

```
while (my $line = <FILE>) {
    if ($line =~ /http:/) {
        print $line;
    }
}
```

Here, the `=~` (pattern binding) is telling Perl to look for a match of the regular expression "`http:`" in the variable `$line`. If it finds the expression, the operator returns a true value and the block (a `print` statement) is executed.[28]

By the way, if you don't use the `=~` binding operator, Perl will search a default string instead of `$line`. It's like when you say, "Eek! Help me find my contact lens!" People automatically know to look around near you without your actually having to tell them that. Likewise, Perl knows that there is a default place to search for things when you don't say where to search for them. This default string is actually a special scalar variable that goes by the odd name of `$_`. In fact, it's not the default just for pattern matching; many operators in Perl default to using the `$_` variable, so a veteran Perl programmer would likely write the last example as:

```
while (<FILE>) {
    print if /http:/;
}
```

(Hmm, another one of those statement modifiers seems to have snuck in there. Insidious little beasties.)

This stuff is pretty handy, but what if we wanted to find all of the link types, not just the HTTP links? We could give a list of link types, like "`http:`", "`ftp:`", "`mailto:`", and so on. But that list could get long, and what would we do when a new kind of link was added?

```
while (<FILE>) {
    print if /http:/;
    print if /ftp:/;
    print if /mailto:/;
    # What next?
}
```

Since regular expressions are descriptive of a set of strings, we can just describe what we are looking for: a number of alphabetic characters followed by a colon. In regular expression talk (Regexese?), that would be `/[a-zA-Z]+:/`, where the brackets define a *character class*. The `a-z` and `A-Z` represent all ASCII alphabetic characters (the dash means the range of all characters between the starting and ending character, inclusive). And the `+` is a special character that says "one or more of whatever was before me". It's what we call a *quantifier*, meaning a gizmo

28. This is very similar to what the Unix command `grep 'http:' file` would do.

that says how many times something is allowed to repeat. (The slashes aren't really part of the regular expression, but rather part of the pattern-match operator. The slashes are acting like quotes that just happen to contain a regular expression.)

Because certain classes like the alphabetics are so commonly used, Perl defines shortcuts for them, as listed in Table 1-7.

Table 1-7. Shortcuts for alphabetic characters

Name	ASCII Definition	Unicode Definition	Shortcut
Whitespace	`[ \t\n\r\f]`	`\p{Whitespace}`	`\s`
Word character	`[a-zA-Z_0-9]`	`[\p{Alphabetic}\p{Digit}\p{Mark}\p{Pc}]`	`\w`
Digit	`[0-9]`	`\p{Digit}`	`\d`

Note that these match *single* characters. A `\w` will match any single word character, not an entire word. (Remember that `+` quantifier? You can say `\w+` to match a word.) Perl also provides the negation of these classes by using the uppercased character, such as `\D` for a nondigit character.

We should note that `\w` is not always equivalent to `[a-zA-Z_0-9]` (and `\d` is not always `[0-9]`). Some locales define additional alphabetic characters outside the ASCII sequence, and `\w` respects them. Versions of Perl newer than 5.8.1 also know about Unicode letter and digit properties and treat Unicode characters with those properties accordingly. (Perl also considers ideographs and combining marks to be `\w` characters.)

There is one other very special character class, written with a "`.`", that will match any character whatsoever.[29] For example, `/a./` will match any string containing an "`a`" that is not the last character in the string. Thus, it will match "`at`" or "`am`" or even "`a!`", but not "`a`", since there's nothing after the "`a`" for the dot to match. Since it's searching for the pattern anywhere in the string, it'll match "`oasis`" and "`camel`", but not "`sheba`". It matches "`caravan`" on the first "`a`". It could match on the second "`a`", but it stops after it finds the first suitable match, searching from left to right.

29. Except that it won't normally match a newline. When you think about it, a "`.`" doesn't normally match a newline in *grep*(1) either.

Quantifiers

The characters and character classes we've talked about all match single characters. We mentioned that you could match multiple "word" characters with `\w+`. The `+` is one kind of quantifier, but there are others. All of them are placed after the item being quantified.

The most general form of quantifier specifies both the minimum and maximum number of times an item can match. You put the two numbers in braces, separated by a comma. For example, if you were trying to match North American phone numbers, the sequence `\d{7,11}` would match at least seven digits, but no more than eleven digits. If you put a single number in the braces, the number specifies both the minimum and the maximum; that is, the number specifies the exact number of times the item can match. (All unquantified items have an implicit `{1}` quantifier.)

If you put the minimum and the comma but omit the maximum, then the maximum is taken to be infinity. In other words, it will match at least the minimum number of times, plus as many as it can get after that. For example, `\d{7}` will match only the first seven digits (a local North American phone number, for instance, or the first seven digits of a longer number), while `\d{7,}` will match any phone number, even an international one (unless it happens to be shorter than seven digits). There is no special way of saying "at most" a certain number of times. Just say `.{0,5}`, for example, to find at most five arbitrary characters.

Certain combinations of minimum and maximum occur frequently, so Perl defines special quantifiers for them. We've already seen `+`, which is the same as `{1,}`, or "at least one of the preceding item". There is also `*`, which is the same as `{0,}`, or "zero or more of the preceding item", and `?`, which is the same as `{0,1}`, or "zero or one of the preceding item" (that is, the preceding item is optional).

You need to be careful of a couple things about quantification. First of all, Perl quantifiers are by default *greedy*. This means that they will attempt to match as much as they can as long as the whole pattern still matches. For example, if you are matching `/\d+/` against "`1234567890`", it will match the entire string. This is something to watch out for especially when you are using "`.`", any character. Often, someone will have a string like:

```
larry:JYHtPh0./NJTU:100:10:Larry Wall:/home/larry:/bin/bash
```

and will try to match "`larry:`" with `/.+:/`. However, since the `+` quantifier is greedy, this pattern will match everything up to and including "`/home/larry:`", because it matches as much as possible before the last colon, including all the other colons. Sometimes you can avoid this by using a negated character class;

that is, by saying `/[^:]+:/`, which says to match one or more noncolon characters (as many as possible), up to the first colon. It's that little caret in there that negates the Boolean sense of the character class.[30] The other point to be careful about is that regular expressions will try to match as *early* as possible. This even takes precedence over being greedy. Since scanning happens left to right, the pattern will match as far left as possible, even if there is some other place where it could match longer. (Regular expressions may be greedy, but they aren't into delayed gratification.) For example, suppose you're using the substitution command (`s///`) on the default string (variable `$_`, that is), and you want to remove a string of x's from the middle of the string. If you say:

```
$_ = "fred xxxxxxx barney";
s/x*//;
```

it will have absolutely no effect! This is because the `x*` (meaning zero or more "`x`" characters) will be able to match the "nothing" at the beginning of the string, since the null string happens to be zero characters wide and there's a null string just sitting there plain as day before the "`f`" of "`fred`".[31] There's one other thing you need to know. By default, quantifiers apply to a single preceding character, so `/bam{2}/` will match "`bamm`" but not "`bambam`". To apply a quantifier to more than one character, use parentheses. So to match "`bambam`", use the pattern `/(bam){2}/`.

Minimal Matching

If you were using a prehistoric version of Perl and you didn't want greedy matching, you had to use a negated character class. (And, really, you were still getting greedy matching of a constrained variety.)

In modern versions of Perl, you can force nongreedy, minimal matching by placing a question mark after any quantifier. Our same username match would now be `/.*?:/`. That `.*?` will now try to match as few characters as possible, rather than as many as possible, so it stops at the first colon rather than at the last.

Nailing Things Down

Whenever you try to match a pattern, it's going to try to match in every location until it finds a match. An *anchor* allows you to restrict where the pattern can match. Essentially, an anchor is something that matches a "nothing", but a

30. Sorry, we didn't pick that notation, so don't blame us. That's just how negated character classes are customarily written in Unix culture.

31. Don't feel bad. Even the authors get caught by this from time to time.

special kind of nothing that depends on its surroundings. You could also call it a rule, a constraint, or an assertion. Whatever you care to call it, it tries to match something of zero width and either succeeds or fails. (Failure merely means that the pattern can't match that particular way. The pattern will go on trying to match some other way, if there are any other ways left to try.)

The special symbol `\b` matches at a word boundary, which is defined as the "nothing" between a word character (`\w`) and a nonword character (`\W`), in either order. (The characters that don't exist off the beginning and end of your string are considered to be nonword characters.) For example:

```
/\bFred\b/
```

would match "`Fred`" in both "`The Great Fred`" and "`Fred the Great`", but not in "`Frederick the Great`" because the "`d`" in "`Frederick`" is not followed by a nonword character.

In a similar vein, there are also anchors for the beginning and the end of the string. If it is the first character of a pattern, the caret (`^`) matches the "nothing" at the beginning of the string. Therefore, the pattern `/^Fred/` would match "`Fred`" in "Frederick the Great" but not in "The Great Fred", whereas `/Fred^/` wouldn't match either. (In fact, it doesn't even make much sense.) The dollar sign (`$`) works like the caret, except that it matches the "nothing" at the end of the string instead of the beginning.[32] So now you can probably figure out that when we said:

```
next LINE if $line =~ /^#/;
```

we meant "Go to the next iteration of `LINE` loop if this line happens to begin with a `#` character."

Earlier we said that the sequence `\d{7,11}` would match a number from seven to eleven digits long. While strictly true, the statement is misleading: when you use that sequence within a real pattern-match operator such as `/\d{7,11}/`, it does not preclude there being extra unmatched digits after the 11 matched digits! You often need to anchor quantified patterns on either or both ends to get what you expect.

Backreferences

We mentioned earlier that you can use parentheses to group things for quantifiers, but you can also use parentheses to remember bits and pieces of what you

32. This is a bit oversimplified, since we're assuming here that your string contains no newlines; `^` and `$` are actually anchors for the beginnings and endings of lines rather than strings. We'll try to straighten this all out in Chapter 5 (to the extent that it can be straightened out).

matched. A pair of parentheses around a part of a regular expression causes whatever was matched by that part to be remembered for later use. It doesn't change what the part matches, so `/\d+/` and `/(\d+)/` will still match as many digits as possible, but in the latter case they will be remembered in a special variable to be backreferenced later.

How you refer back to the remembered part of the string depends on where you want to do it from. Within the same regular expression, you use a backslash followed by an integer. The integer corresponding to a given pair of parentheses is determined by counting left parentheses from the beginning of the pattern, starting with one. So, for example, to match something similar to an HTML tag like "`<B>Bold</B>`", you might use `/<(.*?)>.*?<\/\1>/`. This forces the two parts of the pattern to match the exact same string, such as the "`B`" in this example.

Outside the regular expression itself, such as in the replacement part of a substitution, you use a `$` followed by an integer; that is, a normal scalar variable named by the integer. So if you wanted to swap the first two words of a string, for example, you could use:

```
s/(\S+)\s+(\S+)/$2 $1/
```

The right side of the substitution (between the second and third slashes) is mostly just a funny kind of double-quoted string, which is why you can interpolate variables there, including backreference variables. This is a powerful concept: interpolation (under controlled circumstances) is one of the reasons Perl is a good text-processing language. The other reason is the pattern matching, of course. Regular expressions are good for picking things apart, and interpolation is good for putting things back together again. Perhaps there's hope for Humpty Dumpty after all.

If you get tired of numbered backreferences, v5.10 or later also supports named backreferences. This is the same substitution as just given but this time using named groups:

```
s/(?<alpha>\S+)\s+(?<beta>\S+)/$+{beta} $+{alpha}/
```

Table 1-8. Regular expression backreferences

Where	Numbered Group	Named Group
Declare	`( ... )`	`(?<`*NAME*`> ... )`
Inside same regex	`\1`	`\k<`*NAME*`>`
In regular Perl code	`$1`	`$+{`*NAME*`}`

It may take longer to type in the code that way, but once your patterns grow in size and complexity, you'll be glad you can name your groups with meaningful words instead of just numbers.

List Processing

Much earlier in this chapter, we mentioned that Perl has two main contexts: scalar context (for dealing with singular things) and list context (for dealing with plural things). Many of the traditional operators we've described so far have been strictly scalar in their operation. They always take singular arguments (or pairs of singular arguments for binary operators) and always produce a singular result, even in list context. So if you write this:

```
@array = (1 + 2, 3 - 4, 5 * 6, 7 / 8);
```

you know that the list on the right side contains exactly four values, because the ordinary math operators always produce scalar values, even in the list context provided by the assignment to an array.

However, other Perl operators can produce either a scalar or a list value, depending on their context. They just "know" whether a scalar or a list is expected of them. But how will you know that? It turns out to be pretty easy to figure out, once you get your mind around a few key concepts.

First, list context has to be provided by something in the "surroundings". In the previous example, the list assignment provides it. Earlier we saw that the list of a `foreach` loop provides it. The `print` operator also provides it. But you don't have to learn these one by one.

If you look at the various syntax summaries scattered throughout the rest of the book, you'll see various operators that are defined to take a *`LIST`* as an argument. Those are the operators that *provide* list context. Throughout this book, *`LIST`* is used as a specific technical term to mean "a syntactic construct that provides list context". For example, if you look up `sort`, you'll find the syntax summary:

```
sort LIST
```

That means that `sort` provides list context to its arguments.

Second, at compile time (that is, while Perl is parsing your program and translating to internal opcodes), any operator that takes a *`LIST`* provides list context to each syntactic element of that *`LIST`*. So every top-level operator or entity in the *`LIST`* knows at compile time that it's supposed to produce the best list it knows how to produce. This means that if you say:

```
sort @dudes, @chicks, other();
```

then each of `@dudes`, `@chicks`, and `other()` knows at compile time that it's supposed to produce a list value rather than a scalar value. So the compiler generates internal opcodes that reflect this.

Later, at runtime (when the internal opcodes are actually interpreted), each of those *`LIST`* elements produces its list in turn, and then (this is important) all the separate lists are joined together, end to end, into a single list. And that squashed-flat, one-dimensional list is what is finally handed off to the function that wanted the *`LIST`* in the first place. So if `@dudes` contains `(Fred,Barney)`, `@chicks` contains `(Wilma,Betty)`, and the `other` function returns the single-element list `(Dino)`, then the *`LIST`* that `sort` sees is:

```
(Fred,Barney,Wilma,Betty,Dino)
```

and the *`LIST`* that `sort` returns is:

```
(Barney,Betty,Dino,Fred,Wilma)
```

Some operators produce lists (like `keys`), while some consume them (like `print`), and others transform lists into other lists (like `sort`). Operators in the last category can be considered filters, except that, unlike in the shell, the flow of data is from right to left, since list operators operate on arguments passed in from the right. You can stack up several list operators in a row:

```
print reverse sort map {lc} keys %hash;
```

That takes the keys of `%hash` and returns them to the `map` function, which lowercases all the keys by applying the `lc` operator to each of them, and passes them to the `sort` function, which sorts them, and passes them to the `reverse` function, which reverses the order of the list elements, and passes them to the `print` function, which prints them.

As you can see, that's much easier to describe in Perl than in English.

There are many other ways in which list processing produces more natural code. We can't enumerate all the ways here, but for an example, let's go back to regular expressions for a moment. We talked about using a pattern in scalar context to see whether it matched, but if instead you use a pattern in list context, it does something else: it pulls out all the backreferences as a list. Suppose you're searching through a log file or a mailbox, and you want to parse a string containing a time of the form "12:59:59 am". You might say this:

```
my ($hour, $min, $sec, $ampm) = /(\d+):(\d+):(\d+) *(\w+)/;
```

That's a convenient way to set several variables simultaneously. But you could just as easily say:

```
my @hmsa = /(\d+):(\d+):(\d+) *(\w+)/;
```

and put all four values into one array. Oddly, by decoupling the power of regular expressions from the power of Perl expressions, list context increases the power of the language. We don't often admit it, but Perl is actually an orthogonal language in addition to being a diagonal language. Have your cake and eat it, too.

What You Don't Know Won't Hurt You (Much)

Finally, allow us to return once more to the concept of Perl as a natural language. Speakers of a natural language are allowed to have differing skill levels, to speak different subsets of the language, to learn as they go, and, generally, to put the language to good use before they know the whole language. You don't know all of Perl yet, just as you don't know all of English. But that's Officially Okay in Perl culture. You can work with Perl usefully, even though we haven't even told you how to write your own subroutines yet. We've scarcely begun to explain how to view Perl as a system management language, or a rapid prototyping language, or a networking language, or an object-oriented language. We could write entire chapters about some of these things. (Come to think of it, we already did.)

But, in the end, you must create your own view of Perl. It's your privilege as an artist to inflict the pain of creativity on yourself. We can teach you how *we* paint, but we can't teach you how *you* paint. There's More Than One Way To Do It.

Have the appropriate amount of fun.