SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF BIO AND CHEMICAL ENGINEERING

DEPARTMENT OF BIOINFORMATICS

UNIT – I - Perl for Bioinformatics – SBIA1304

UNIT I INTRODUCTION

Biology and Computer Science - Getting Started with Perl - Perl's Benefits - Installing Perl - Running Perl Programs on various platforms - The Art of Programming - Programming Strategies - The Programming Process - Sequences and Strings - Representing Sequence Data - A Program to Store a DNA Sequence - Control Flow - Comments Revisited - Command Interpretation – Statements – Variables – Strings

Biology and Computer Science

One of the most exciting things about being involved in computer programming and biology is that both fields are rich in new techniques and results.

Of course, biology is an old science, but many of the most interesting directions in biological research are based on recent techniques and ideas. The modern science of genetics, which has earned a prominent place in modern biology, is just about 100 years old, dating from the widespread acknowledgement of Mendel's work. The elucidation of the structure of deoxyribonucleic acid (DNA) and the first protein structure are about 50 years old, and the polymerase chain reaction (PCR) technique of cloning DNA is almost 20 years old. The last decade saw the launching and completion of the Human Genome Project that revealed the totality of human genes and much more. Today, we're in a golden age of biological research—a point in human history of great medical, scientific, and philosophical importance.

Computer science is relatively new. Algorithms have been around since ancient times (Euclid), and the interest in computing machinery is also antique (Pascal's mechanical calculator, for instance, or Babbage's steam-driven inventions of the 19th century). But programming was really born about 50 years ago, at the same time as construction of the first large, programmable, digital/electronic (the ENIAC) computers. Programming has grown very rapidly to the present day. The Internet is about 20 years old, as are personal computers; the Web is about 10 years old. Today, our communications, transportation, agricultural, financial, government, business, artistic, and of course, scientific endeavors are closely tied to computers and their programming.

This rapid and recent growth gives the field of computer programming a certain excitement and requires that its professional practitioners keep on their toes. In a way, programming represents procedural knowledge—the knowledge of how to do things— and one way to look at the importance of computers in our society and our history is to see the enormous growth in procedural knowledge that the use of computers has occasioned. We're also seeing the concepts of computation and algorithm being adopted widely, for instance, in the arts and in the law, and of course in the sciences. The computer has become the ruling metaphor for explaining things in general. Certainly, it's tempting to think of a cell's molecular biology in terms of a special kind of computing machinery.

Similarly, the remarkable discoveries in biology have found an echo in computer science. There are evolutionary programs, neural networks, simulated annealing, and more. The exchange of ideas and metaphors between the fields of biology and computer science is, in itself, a spur to discovery (although the dangers of using an improper metaphor are also real).

Getting Started with Perl

Perl is a popular programming language that's extensively used in areas such as bioinformatics and web programming. Perl has become popular with biologists because it's so well-suited to several bioinformatics tasks.

Perl is also an application, just like any other application you might install on your computer. It is available (at no cost) and runs on all the operating systems found in the average biology lab (Unix and Linux, Macintosh, Windows, VMS, and more).The Perl application on your computer takes a Perl language program (such as one of the programs you will write in this book), translates it into instructions the computer can understand, and runs (or "executes") it.

An operating system manages the running of programs and other basic services that a computer provides, such as how files are stored.

So, the word Perl refers both to the language in which you will write programs and to the application on your computer that runs those programs. You can always tell from context which meaning is being used.

Every computer language such as Perl needs to have a translator application (called an interpreter or compiler) that can turn programs into instructions the computer can actually run. So the Perl application is often referred to as the Perl interpreter, and it includes a Perl compiler as well. You will often see Perl programs referred to as Perl scripts or Perl code. The terms program, application, script, and executable are somewhat interchangeable. I refer to them as "programs" in this book.

Perl's Benefits

The following sections illustrate some of Perl's strong points.

Ease of Programming

Computer languages differ in which things they make easy. By "easy" I mean easy for a programmer to program. Perl has certain features that simplifies several common bioinformatics tasks. It can deal with information in ASCII text files or flat files, which are exactly the kinds of files in which much important biological data appears. Perl makes it easy to process and manipulate long sequences such as DNA and proteins. Perl makes it convenient to write a program that controls one or more other programs. As a final example, Perl is used to put biology research labs, and their results, on their own dynamic web sites. Perl does all this and more.

Although Perl is a language that's remarkably suited to bioinformatics, it isn't the only choice nor is it always the best choice. Other programming languages such as C and Java are also used in bioinformatics. The choice of language depends on the problem to be programmed, the skills of the programmers, and the available system.

Rapid Prototyping

Another important benefit of using Perl for biological research is the speed with which a programmer can write a typical Perl program (referred to as rapid prototyping). Many problems can be solved in far fewer lines of Perl code than in C or Java. This has been important to its success in research. In a research environment there are frequent needs for programs that do something new, that are needed only once or occasionally, or that need to be frequently modified. In Perl, you can often toss such a program off in a few minutes or a few hours work, and the research can proceed. This rapid prototyping ability is often a key consideration when choosing Perl for a job. It is common to find programmers familiar with both Perl and C who claim that Perl is five to ten times faster to program in than C. The difference can be critical in the typical understaffed research lab.

Portability, Speed, and Program Maintenance

Portability means how many types of computer systems the language can run on. Perl has no problems there, as it's available for virtually all modern computers found in biology labs. If you write a DNA analyzer in Perl on your Mac, then move it to a Windows computer, you'll find it usually runs as is or with only minor retrofitting. *Speed* means the speed with which the program runs. Here Perl is pretty good but not the best. For speed of execution, the usual language of choice is C. A program written in C typically runs two or more times faster than the comparable Perl program. (There are ways of speeding up Perl with compilers and such, but still... .)

In many organizations, programs are first written in Perl, and then only the programs that absolutely need to have maximum speed are rewritten in C. The fact is, maximum speed is only occasionally an important consideration.

Programming is relatively expensive to do: it takes time, and skilled personnel. It's labor-intensive. On the other hand, computers and computer time (often called CPU time after the central processing unit) are relatively inexpensive. Most desktop computers sit idle for a large part of the day, anyway. So it's usually best to let the computer do the work, and save the programmer's time. Unless your program absolutely must run in say, four seconds instead of ten seconds, you're okay with Perl.

Program maintenance is the general activity of keeping everything working: such activities as adding features to a program, extending it to handle more types of input, porting it to run on other computer systems, fixing bugs, and so forth. Programs take a certain amount of time, effort and cost to write, but successful programs end up costing more to maintain than they did to write in the first place. It's important to write in a language, and in a style, that makes maintenance relatively easy, and Perl allows you to do so. (You can write obscure, hard-to-maintain code in Perl, as in other languages, but I'll give you pointers on how to make your code easy for other programmers to read.)

Installing Perl on Your Computer

The following sections provide pointers for installing Perl on the most common types of computer systems.

Perl May Already Be Installed!

Many computers—especially Unix and Linux computers—come with Perl already installed. (Note that Unix and Linux are essentially the same kind of operating system; Linux is a clone, or functional copy, of a Unix system.) So first check to see if Perl is already there. On Unix and Linux, type the following at a command prompt:

```
$ perl -v
```

If Perl is already installed, you'll see a message like the one I get on my Linux machine:

```
This is perl, v5.6.1 built for i686-linux
Copyright 1987-2001, Larry Wall
Perl may be copied only under the terms of either the
Artistic License or the
GNU General Public License, which may be found in the Perl
5 source kit.
Complete documentation for Perl, including FAQ lists,
should be found on
this system using 'man perl' or 'perldoc perl'.  If you
have access to the
Internet, point your browser at http://www.perl.com/, the
Perl Home Page.
```

If Perl isn't installed, you'll get a message like this:

```
perl: command not found
```

If you get this message, and you're on a shared Unix system at a university or business, be sure to check with the system administrator, because Perl may indeed be installed, but your environment may not be set to find it. (Or, the system administrator may say, "You need Perl? Okay, I'll install it for you.")

On Windows or Macintosh, look at the program menus, or use the *find* program to search for `perl`. You can also try typing `perl -v`, at an MS-DOS command window or at a shell window on the MacOS X. (Note that the MacOS X is a Unix system!)

Installation

The next sections provide specific installation instructions for specific platforms.

Unix and Linux

If Perl isn't installed on your Unix or Linux machine, first try to find a binary to install. At the Downloads page of http://www.perl.com, you'll see the subheading Binary Distributions. Select Unix or Linux, and then see if your particular flavor of operating

system has a binary available. Several versions are available, and the web-site instructions should be enough to get Perl installed once you've downloaded the binary. Most versions of Linux maintain up-to-date Perl binaries on their web sites. For instance, if you have a Red Hat Linux system, you need to identify which version of the system you have (by typing `uname -a`) and then get the appropriate *rpm* file to download and install. Red Hat has an *rpm* for Perl that Red Hat Linux users can install by typing:

```
rpm -Uvh perl.rpm
```

(the actual name of the perl.rpm file varies).

If no binary version of Perl is available for your flavor of Unix or Linux, you must compile Perl from its source code. In this case, starting from the Perl web page, click on the Downloads button and then select Source Code Distribution. The source code has an INSTALL file with instructions that guide you through the process of downloading the source code, installing it on your system, compiling the source code into a binary, and finally installing the binary.

As mentioned previously, compiling from source code is a considerably longer process than installing an already made binary, and requires a bit more reading of instructions, but it usually works quite well. You will need a C compiler on your computer to install from source code. Nowadays, some Unix systems ship without a complete C compiler. Linux will always have the free C compiler called *gcc* installed, and you can also install *gcc* on any Unix (or Windows, or Mac) system that lacks a C compiler.

Macintosh

The MacPerl installation steps are clearly explained on the MacPerl web page, http://www.macperl.com/ (which you can also get to from the Perl web page and its Downloads button). Here's a very brief overview.

From the MacPerl page, click on Get MacPerl, and follow the directions to download the application. It will appear on your desktop. Double-click it to unstuff it. If you don't have Aladdin Stuffit Expander (most Macs already do), this won't work, and you'll have to go to http://www.aladdinsys.com to download and install Stuffit.

MacPerl can be installed as a standalone application under the MacOS Finder or as a tool under the Macintosh Programmer's Workbench; you will probably want the standalone application. Perl Version 5 is available for MacOS 7.0 and later. Details about which Perl version is available for your particular hardware and MacOS version are available at the MacPerl web page.

Windows

Several binaries for different Windows versions are available. Since Windows is closely coupled with Intel 32-bit chips, these binaries are often called Wintel or Win32 binaries. The current standard Perl distribution is ActivePerl from ActiveState, at http://www.activestate.com/ActivePerl/, where you can find complete installation directions. You can also get to ActivePerl via the Downloads button from the Perl web site. Under the subheading Binary Distributions, go to Perl for Win32, and then click on the ActivePerl site. From the ActiveState web site's ActivePerl page, click the Downloads button. You can then download the Windows-Intel binary. Note that installing it requires a program called Windows Installer, which is available at ActivePerl if it's not already on your computer.

The Art of Programming

This chapter provides an overview of how programmers accomplish their jobs.

Just as visitors to a biology lab tend to have a clueless awe of "all those test tubes," so the newcomer to programming may regard the world of the programmer as a kind of arcane black box full of weird terminology and abstruse skills. So, to make the whole enterprise a little more congenial, let's take a short tour of some important realities that affect all programmers. Two of the most important are practical strategies that good programmers use and where to go to find answers to questions that arise while you are programming. Using a couple of brief narrative case studies, we'll look at how programmers find solutions to problems.

Individual Approaches to Programming

What's the best way to learn programming? The answer depends on what you hope to accomplish. There are several ways to get started. You can:

Take classes of many different kinds
Read a tutorial book like this one
Get the programming manuals and plunge in
Be tutored by a programmer
Identify a program you need
Try any and all of the above until you've managed to write the program

The answer also depends on how you choose to learn. Some people prefer classes, because the information is often presented in a well-organized way, and questions can be answered by the teacher. Others learn best with self-paced study.

Some things about learning to program are common to all these approaches. If you've never programmed at all, the information in the following sections is a "heads-up" about what's ahead.

Edit—Run—Revise (and Save)

The most important thing about programming is that it's a hands-on learning activity such as dancing, playing music, cooking, or some other family-oriented activity. You can read about it, but you can't actually do it until you actually do it.

While learning to program in Perl, you need to read about how Perl works, as you will in the chapters that follow. You also need to look at plenty of examples of programs. But you especially need to attempt to write your own programs, as you are asked to do in the exercises at the end of the later chapters. Only this kind of direct experience will make you a programmer.

So I want to give you an overview of the most important tasks involved in writing programs, to help you approach your first programs with a clearer idea of what's really involved.

What exactly will you be doing at the computer? The bulk of a programmer's work involves the steps of writing or revising a program in an editor, then running the program and watching how it behaves, and on the basis of that behavior going back and revising the program again. A typical programmer spends more than half of his or her time editing the program.

Saves and Backups

Once you have even a few lines of code written, it's important to save it. In fact, you should always remember to save a version of your program at regular intervals during editing, so if you make a bunch of edits and the computer crashes, you don't lose hours of work. Also, make sure you back up your work on another disk. Hard disks fail, and when yours does, the information on it will be lost. Therefore it's essential to make regular (daily) backups of your work onto some other medium—tape, floppy disk, Zip disk, another hard disk, writable CD—whatever, just so you won't lose all your work if a disk failure occurs.

In addition to backups of your disks, it's also a good idea to save a dated version of your program at regular intervals. This will allow you to go back to an earlier version of your program should that prove necessary.

It's also a good idea to make sure the backups you're making actually work. So, for instance, if you're backing up to a tape drive, try restoring the files from your tape drive every once in a while, just to make sure that the software and the tapes themselves are all working. You may also want to print out ("make a hardcopy") of your programs at regular intervals for extra insurance against system failures. Finally, it's good policy to keep the backups somewhere away from the computer, so in case of fire or other disaster, the backups will be safe.

Error Messages

Fixing errors is an essential step in writing programs. After you've written and edited a program, the next step is to run it to see if it works. Very often, you'll find that you've made some typographical error, like forgetting to put in a semicolon. As a result, your program isn't valid, and you'll get various error messages from the system. You then have to read the error messages and reedit your program to repair the offending code.

These error messages are sometimes rather cryptic. In the event of an error, the Perl interpreter may have some trouble knowing exactly where you went wrong. It may only recognize that there is something wrong. So it guesses where the problem is, and in the process, it may give you some extraneous information.

The most important thing about using error messages is to look at the first one or two error messages and ignore the rest; fix the top problems, and try running the program again. Error messages are often verbose and can run on for several pages. Just ignore everything but the first errors reported. Another important point is that the line numbers reported in those first error messages are usually right. Sometimes they're off by a line, and they're rarely way off. Later on, we'll practice generating and reading error messages.

Debugging

Perhaps your edits created a valid program, and the Perl interpreter reads in your program and runs it. You find, however, that the program isn't doing what you want it to do. Now you have to go back, look at the program, and try to figure out what's wrong.

Perhaps you made a simple mistake, such as adding instead of subtracting. You may have misread the documentation, and you're using the language the wrong way (reread the documentation). You may simply have an inadequate plan for accomplishing your goal (rethink your strategy and reprogram that part of the code). Sometimes you can't see what's wrong, and you have to look elsewhere (try searching newsgroup archives or FAQs or asking colleagues for help).

For errors that are difficult to find, there are programs called debuggers that allow you to run the program step by step, looking at everything that's happening in the program.

There are other tools and techniques you can use. For instance, you can examine your program by adding `print` statements that print out intermediate values or results. There are also special helper programs that can observe your program while it's running and then report about it, telling you, for instance, about where the program is spending most of its time. These tools, and others like them, are essential to programming, and you need to learn how to use them.

An Environment of Programs

Programming is an exercise in problem solving. It's an iterative, gradual process. Although it can be done by one person alone, it's often a social activity (this surprises many newcomers). It requires developing specific problem-solving skills and learning a few tools. Programming is sometimes tricky and can be frustrating. On the other hand, for those with an aptitude, there's a great sense of satisfaction that comes from building a working program.

Computer programs can be many things, from barely useful, to aesthetically and intellectually stimulating, to important generators of new knowledge. They can be

beautiful. (They can also be destructive, stupid, silly, or vicious; they are human creations, after all.) Because writing a program is an iterative, building, gradual process, there can be real satisfaction in seeing the work unfold from simple beginnings to complete structures. For the beginning student, this gradual unfolding of a new program mirrors the gradual mastery of the language.

As our culture began writing and accumulating programs in the middle of the 20th century, a programming environment began to develop. Gradually, we've been accumulating a substantial body of procedural knowledge. Programs often reflect the fact that they swim in waters populated by many other programs, and beginning programmers can expect to learn a lot from this environment.

Open Source Programs

As programming has become important in the world, it has also become economically valuable. As a result, the source code for many programs is kept hidden to protect commercial assets and stymie the competition.

However, the source code for many of the best and most used programs are freely available for anyone to examine. Freely available source code is called open source. (There are various kinds of copyrights that may attach to open source program code, but they all allow anyone to examine the source code.) The open source movement treats program source code in a similar manner to the way scientists publish their results: publicly and open to unfettered examination and discussion.

The source code for these programs can be a wonderful place for the beginning programmer to learn how professional programmers write. The programs available in open source include the Perl interpreter and a large amount of Perl code, the Linux operating system, the Apache web server, the Netscape web browser, the *sendmail* mail transfer agent, and much more.

Programming Strategies

In order to give you, the beginning programmer, an idea of how programming is done, let's see how an experienced programmer goes about solving problems by giving a couple of instructive case studies.

Imagine that you want to count all the regulatory elements[1] in a large chunk of DNA that you just got from the sequencing lab. You're a professional bioinformatics programmer. What do you do? There are two possible solutions: find a program or write one yourself.

[1] A regulatory element is a stretch of DNA used by the cell in the control of a coding region, helping to determine if and when it's used to create a protein.

It's likely there is already a perfectly good, working, and maybe even free program that does exactly what you need. Very often, you can find exactly what you need on the Web and avoid the cost and expense of reinventing the wheel. This is programming at its best—minimal work for maximal effect. It's the classic case of the experimentalist's adage: a day in the library can save you six months in the lab.

An important part of the art of programming is to keep aware of collections of programs that are available. Then you can simply use the code if it does exactly what you need, or you can take an existing program and alter it to suit your own needs. Of course, copyright laws must be observed, but much is available at no cost, especially to educational and nonprofit organizations. Most Perl module code has a copyright, but you are allowed to use it and modify it given certain restrictions. Details are available at the Perl web site and with the particular modules.

How do you find this wonderful, free, and already existing program? The Perl community has an organized collection of such programming code at the Comprehensive Perl Archive Network (CPAN) web site, http://www.CPAN.org. Try exploring: you'll find it's organized by topic, so it's possible to quickly find, for example, web, statistics, or graphics programs. In our case, you will find the Bioperl module, which includes several useful bioinformatics functions. A module is a collection of Perl code that can be easily loaded and used by your Perl programs.

The most useful kinds of code are convenient libraries or modules that package a suite of functions. These packages offer a great deal of flexibility in creating new programs. Although you still have to program, the job may be only a small fraction of the work of writing the whole program from scratch. For instance, to continue our example of looking for regulatory elements, your search may turn up a convenient module that lists the regulatory elements plus code that takes a list of elements and searches for them in a DNA library. Then all you have to do is combine the existing code, provide the DNA library, and with a little bit of programming, you're done.

There are lots of other places to look for already existing code. You can search the Internet with your favorite search engines. You can browse collections of links for bioinformatics, looking for programs. You can also search the other sources we've already covered, such as newsgroups, relevant experts, etc.

If you haven't hit paydirt yet, and you know that the program will take a significant amount of time to write yourself, you may want to search the literature in the library, and perhaps enlist the aid of a librarian. You can search Medline for articles about regulatory elements, since often an article will advertise code (an actual program in a language like Perl) that the authors will forward. You can consult conference proceedings, books, and journals. Conferences and trade shows are also great places to look around, meet people, and ask questions.

In many cases you succeed, and despite the effort involved, you saved yourself and your laboratory days, weeks, or months of effort.

However, one big warning about modifying existing code: depending on how much alteration is required, it can sometimes be more difficult to modify existing code than to write a whole program from scratch. Why? Well, depending on who wrote the program, it may be difficult just to see what the different parts of the code do. You can't make modifications if you can't understand what methods the program uses in the first place. (We'll talk more about writing readable code, and the importance of comments in code, later.) This factor alone accounts for a large part of the expense of programming; many programs can't be easily read, or understood, so they can't be maintained. Also, testing the program may be difficult for various reasons, and it may take a lot of time and effort to assure yourself that your modifications are working correctly.

Okay, let's say that you spent three days looking for an existing program, and there really wasn't anything available. (Well, there was one program, but it cost $30,000 which is way outside your budget, and your local programming expert was too busy to write one for you.) So you absolutely have to write the program yourself.

How do you start from scratch and come up with a program that counts the regulatory elements in some DNA? Read on.

The Programming Process

You've been assigned to write a program that counts the regulatory elements in DNA. If you've never programmed you probably have no idea of how to start. Let's talk about what you need to know to write the program.

Here's a summary of the steps we'll cover:
Identify the required inputs, such as data or information given by the user.

Make an overall design for the program, including the general method—the algorithm— by which the program computes the output.

Decide how the outputs will print; for example, to files or displayed graphically.

Refine the overall design by specifying more detail.

Write the Perl program code.

These steps may be different for shorter or longer programs, but this is the general approach you will take for most of your programming.

The Design Phase

First, you need to conceive a plan for how the program is going to work. This is the overall design of the program and an important step that's usually done before the actual writing of the program begins. Programs are often compared to kitchen recipes, in that they are specific instructions on how to accomplish some task. For instance, you need an idea of what inputs and outputs the program will have. In our example, the input would be the new DNA. You then need a strategy for how the program will do the necessary computing to calculate the desired output from the input.

Algorithms

An algorithm is the design, or plan, for the computation done by a computer program. (It's actually a tricky term to define, outside of a formal mathematical system, but this is a reasonable definition.) An algorithm is implemented by coding it in a specific computer language, but the algorithm is the idea of the computation. It's often well represented in pseudocode, which gives the idea of a program without actually being a real computer program.

Sequences and Strings

In this chapter you will begin to write Perl programs that manipulate biological sequence data, that is, DNA and proteins. Once you have the sequences in the computer, you'll start writing programs that do the following with the sequence data:

Transcribe DNA to RNA
Concatenate sequences
Make the reverse complement of sequences Read sequence data from files

You'll also write programs that give information about your sequences. How GC-rich is your DNA? How hydrophobic is your protein? You'll see programming techniques you can use to answer these and similar questions.

Representing Sequence Data

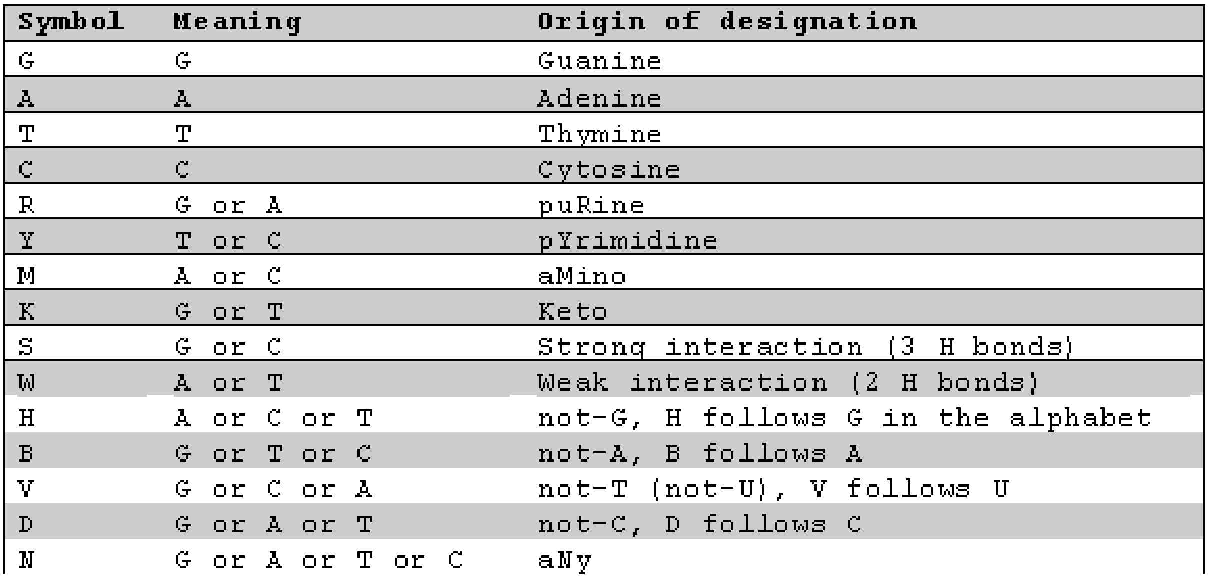
The majority of this book deals with manipulating symbols that represent the biological sequences of DNA and proteins. The symbols used in bioinformatics to represent these sequences are the same symbols biologists have been using in the literature for this same purpose.

As stated earlier, DNA is composed of four building blocks: the nucleic acids, also called nucleotides or bases. Proteins are composed of 20 building blocks, the amino acids, also called residues. Fragments of proteins are called peptides. Both DNA and proteins are essentially polymers, made from their building blocks attached end to end. So it's possible to summarize the structure of a DNA molecule or protein by simply giving the sequence of bases or amino acids.

These are brief definitions; I'm assuming you are either already familiar with them or are willing to consult an introductory textbook on molecular biology for more specific details. Table 1 shows bases; add a sugar and you get the nucleotides adenosine, guanosine, cytidine, thymidine, and uridine. You can further add a phosphate and get the nucleotides adenylic acid, guanylic acid, cytidylic acid, thymidylic acid, and uridylic acid. A nucleic acid is a chemically linked sequence of nucleotides. A peptide is a small number of joined amino acids; a longer chain is a polypeptide. A protein is a biologically functional unit made of one or more polypeptides. A residue is an amino acid in a polypeptide chain.

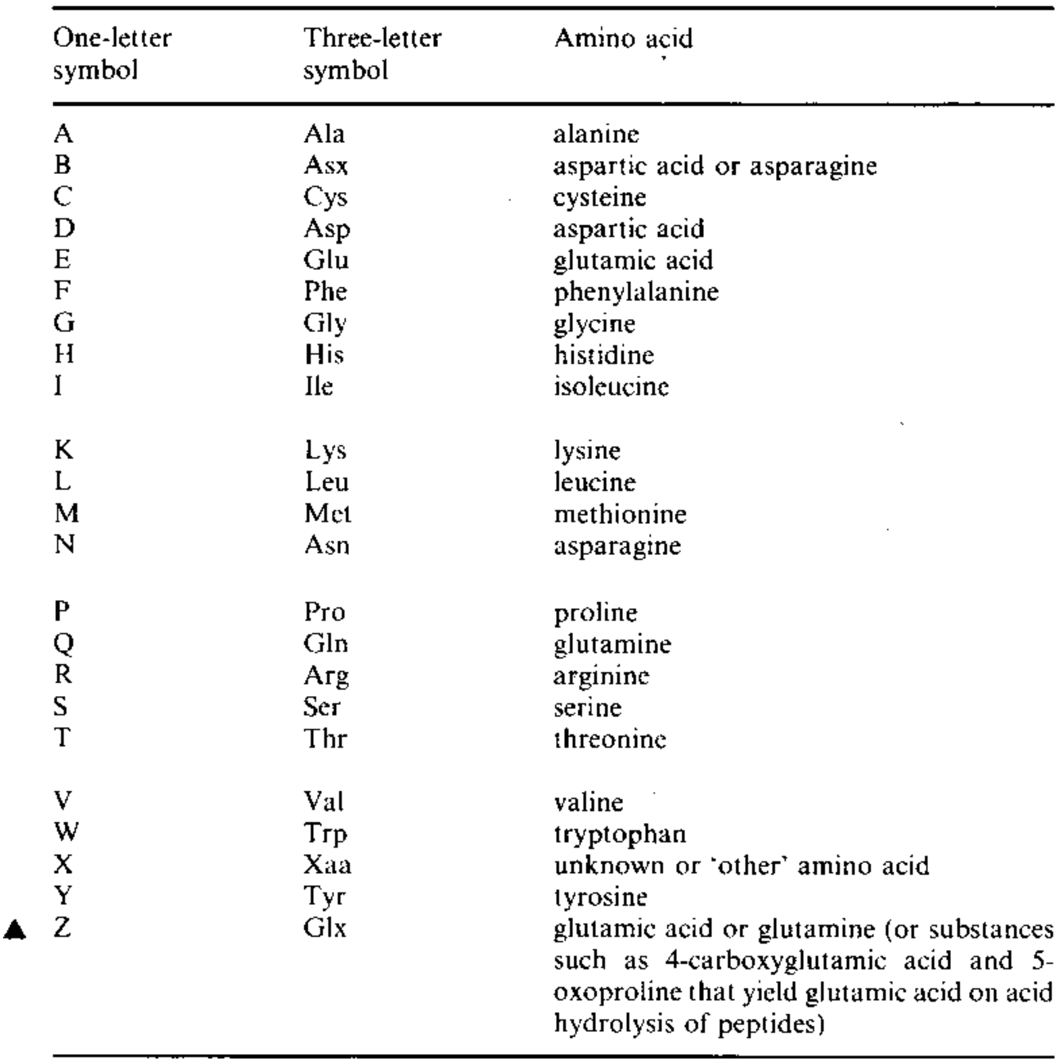
For expediency, the names of the nucleic acids and the amino acids are often represented as one- or three-letter codes, as shown in Table 1 and Table 2. (This book mostly uses the one-letter codes for amino acids.)

Table 1



Symbol	Meaning	Origin of designation
G	G	Guanine
A	A	Adenine
T	T	Thymine
C	C	Cytosine
R	G or A	puRine
Y	T or C	pYrimidine
M	A or C	aMino
K	G or T	Keto
S	G or C	Strong interaction (3 H bonds)
W	A or T	Weak interaction (2 H bonds)
H	A or C or T	not-G, H follows G in the alphabet
B	G or T or C	not-A, B follows A
V	G or C or A	not-T (not-U), V follows U
D	G or A or T	not-C, D follows C
N	G or A or T or C	aNy

Table 2



One-letter symbol	Three-letter symbol	Amino acid
A	Ala	alanine
B	Asx	aspartic acid or asparagine
C	Cys	cysteine
D	Asp	aspartic acid
E	Glu	glutamic acid
F	Phe	phenylalanine
G	Gly	glycine
H	His	histidine
I	Ile	isoleucine
K	Lys	lysine
L	Leu	leucine
M	Met	methionine
N	Asn	asparagine
P	Pro	proline
Q	Gln	glutamine
R	Arg	arginine
S	Ser	serine
T	Thr	threonine
V	Val	valine
W	Trp	tryptophan
X	Xaa	unknown or 'other' amino acid
Y	Tyr	tyrosine
▲ Z	Glx	glutamic acid or glutamine (or substances such as 4-carboxyglutamic acid and 5-oxoproline that yield glutamic acid on acid hydrolysis of peptides)

the lowercase versions of these single-letter codes is also used on occasion, frequently for DNA, rarely for protein.

The computer-science terminology is a little different from the biology terminology for the codes in Table 4-1 and Table 4-2. In computer-science parlance, these tables define two alphabets, finite sets of symbols that can make strings. A sequence of symbols is called a string. For instance, this sentence is a string. A language is a (finite or infinite) set of strings. In this book, the languages are mainly DNA and protein sequence data. You often hear bioinformaticians referring to an actual sequence of DNA or protein as a "string," as opposed to its representation as sequence data. This is an example of the terminologies of the two disciplines crossing over into one another.

As you've seen in the tables, we'll be representing data as simple letters, just as written on a page. But computers actually use additional codes to represent simple letters. You won't have to worry much about this; just remember that when using your text editor to save as ASCII, or plain text.

ASCII is a way for computers to store textual (and control) data in their memory. Then when a program such as a text editor reads the data, and it knows it's reading ASCII, it can actually draw the letters on the screen in a recognizable fashion because it's programmed to know that particular code. So the bottom line is: ASCII is a code to represent text on a computer.[1]

A Program to Store a DNA Sequence

Let's write a small program that stores some DNA in a variable and prints it to the screen. The DNA is written in the usual fashion, as a string made of the letters A, C, G, and T, and we'll call the variable `$DNA`. In other words, `$DNA` is the name of the DNA sequence data used in the program. Note that in Perl, a variable is really the name for some data you wish to use. The name gives you full access to the data. The following shows the entire program.

Putting DNA into the computer

```
#!/usr/bin/perl -w
# Storing DNA in a variable, and printing it out
# First we store the DNA in a variable called $DNA
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
# Next, we print the DNA onto the screen
print $DNA;
# Finally, we'll specifically tell the program to exit.
exit;
```

Using what you've already learned about text editors and running Perl programs in Chapter 2, enter the code (or copy it from the book's web site) and save it to a file. Remember to save the program as ASCII or text-only format, or Perl may have trouble reading the resulting file.

The second step is to run the program. The details of how to run a program depend on the type of computer you have (see Chapter 2). Let's say the program is on your computer in a file called example4-1. As you recall from Chapter 2, if you are running this program on Unix or Linux, you type the following in a shell window:

```
perl example4-1
```

On a Mac, open the file with the MacPerl application and save it as a droplet, then just double-click on the droplet. On Windows, type the following in an MS-DOS command window:

```
perl example4 -1
```

If you've successfully run the program, you'll see the output printed on your computer screen.

Control Flow

[1] A new character encoding called Unicode, which can handle all the symbols in all the world's languages, is becoming widely accepted and is supported by Perl as well.

Example 4-1 illustrates many of the ideas all our Perl programs will rely on. One of these ideas is control flow , or the order in which the statements in the program are executed by the computer.

Every program starts at the first line and executes the statements one after the other until it reaches the end, unless it is explicitly told to do otherwise. Example 4-1 simply proceeds from top to bottom, with no detours.

In later chapters, you'll learn how programs can control the flow of execution.

Comments Revisited

Now let's take a look at the parts of Example 4-1. You'll notice lots of blank lines. They're there to make the program easy for a human to read. Next, notice the comments that begin with the # sign. Remember from Chapter 3 that when Perl runs, it throws these away along with the blank lines. In fact, to Perl, the following is exactly the same program as Example 4-1:

```
#!/usr/bin/perl -w
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'; print $DNA; exit;
```

In Example 4-1, I've made liberal use of comments. Comments at the beginning of code can make it clear what the program is for, who wrote it, and present other information that can be helpful when someone needs to understand the code. Comments also explain what each section of the code is for and sometimes give explanations on how the code achieves its goals.

It's tempting to belabor the point about the importance of comments. Suffice it to say that in most university-level, computer-science class assignments, the program without comments typically gets a low or failing grade; also, the programmer on the job who doesn't comment code is liable to have a short and unsuccessful career.

Command Interpretation

Because it starts with a # sign, the first line of the program looks like a comment, but it doesn't seem like a very informative comment:

```
#!/usr/bin/perl -w
```

This is a special line called command interpretation that tells the computer running Unix and Linux that this is a Perl program. It may look slightly different on different computers. On some machines, it's also unnecessary because the computer recognizes Perl from other information. A Windows machine is usually configured to assume that any program ending in .pl is a Perl program. In Unix or Linux, a Windows command window, or a MacOS X shell, you can type `perl my_program`, and your Perl program `my_program` won't need the special line. However, it's commonly used, so we'll have it at start all our programs.

Notice that the first line of code uses a flag `-w`. The "w" stands for warnings, and it causes Perl to print messages in case of an error. Very often the error message suggests the line number where it thinks the error began. Sometimes the line number is wrong, but the error is usually on or just before the line the message suggests. Later in the book, you'll also see the statement use warnings as an alternative to -w.

Statements

The next line of Example 4-1 stores the DNA in a variable: $DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';

This is a very common, very important thing to do in a computer language, so let's take a leisurely look at it. You'll see some basic features about Perl and about programming languages in general, so this is a good place to stop skimming and actually read.

This line of code is called a statement. In Perl, statements end in a semicolon (;). The use of the semicolon is similar to the use of the period in the English language.

To be more accurate, this line of code is an assignment statement. Its purpose in this program is to store some DNA into a variable called $DNA. There are several fundamental things happening here as you will see in the next sections.

Variables

First, let's look at the variable $DNA. Its name is somewhat arbitrary. You can pick another name for it, and the program behaves the same way. For instance, if you replace the two lines:

```
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';

print $DNA;
```

with these:

```
$A_poem_by_Seamus_Heaney =
'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
print $A_poem_by_Seamus_Heaney;
```

the program behaves in exactly the same way, printing out the DNA to the computer screen. The point is that the names of variables in a computer program are your choice. (Within certain restrictions: in Perl, a variable name must be composed from upper- or lowercase letters, digits, and the underscore _ character. Also the first character must not be a digit.)

This is another important point along the same lines as the remarks I've already made about using blank lines and comments to make your code more easily read by humans. The computer attaches no meaning to the use of the variable name $DNA instead of $A_poem_by_Seamus_Heaney, but whoever reads the program certainly will. One name makes perfect sense, clearly indicates what the variable is for in the program, and eases the chore of understanding the program. The other name makes it unclear what the program is doing or what the variable is for. Using well-chosen variable names is part of what's called self-documenting code. You'll still need comments, but perhaps not as many, if you pick your variable names well.

You've noticed that the variable name `$DNA` starts with dollar sign. In Perl this kind of variable is called a scalar variable, which is a variable that holds a single item of data. Scalar variables are used for such data as strings or various kinds of numbers (e.g., the string `hello` or numbers such as 25, 6.234, 3.5E10, -0.8373). A scalar variable holds just one item of data at a time.

Strings

In Example 4-1, the scalar variable `$DNA` is holding some DNA, represented in the usual way by the letters A, C, G, and T. As stated earlier, in computer science a sequence of letters is called a string. In Perl you designate a string by putting it in quotes. You can use single quotes, as in Example 4-1, or double quotes. (You'll learn the difference later.) The DNA is thus represented by: `'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'`

Assignment

In Perl, to set a variable to a certain value, you use the = sign. The = sign is called the assignment operator . In Example 4-1, the value:
`'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'`
is assigned to the variable `$DNA`. After the assignment, you can use the name of the variable to get the value, as in the `print` statement in Example 4-1.

The order of the parts is important in an assignment statement. The value assigned to something appears to the right of the assignment operator. The variable that is assigned a value is always to the left of the assignment operator. In programming manuals, you sometimes come across the terms lvalue and rvalue to refer to the left and right sides of the assignment operator.

This use of the = sign has a long history in programming languages. However, it can be a source of confusion: for instance, in most mathematics, using = means that the two things on either side of the sign are equal. So it's important to note that in Perl, the = sign doesn't mean equality. It assigns a value to a variable. (Later, we'll see how to represent equality.)

So, to summarize what we've learned so far about this statement:

```
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
```

It's an assignment statement that sets the value of the scalar variable `$DNA` to a string representing some DNA.

Print

The statement:

```
print $DNA;
```

prints `ACGGGAGGACGGGAAAATTACTACGGCATTAGC` out to the computer screen. Notice that the `print` statement deals with scalar variables by printing out their values—in this case, the string that the variable `$DNA` contains. You'll see more about printing later.

Exit

Finally, the statement `exit;` tells the computer to exit the program. Perl doesn't require an `exit` statement at the end of a program; once you get to the end, the program exits automatically. But it doesn't hurt to put one in, and it clearly indicates the program is over. You'll see other programs that exit if something goes wrong before the program normally finishes, so the `exit` statement is definitely useful.

UNIT III GLYCOMICS AND LIPIDOMICS Glycomics – Challenges, Importance, Tools used to analyse glycans, softwares and databases. Lipidomics – Structural diversity of lipids, extraction, separation, detection and imaging. Challenges and applications.

UNIT IV TRANSCRIPTOMICS Gene Expression Profiling – DNA microarrays. Transcriptomics: Data Collection- Isolation of RNA, ESTs, SAGE analysis, Microarrays. RNA-seq: Principle and advances. Image processing

UNIT V OTHER OMICS IN BIOLOGY Secretomics, Metablolomics, fluxomics, nutrigenomics, Metagenomics, Organomics, Pharmacogenomics, Phytochemomics, Microbiomics

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF BIO AND CHEMICAL ENGINEERING

DEPARTMENT OF BIOINFORMATICS

UNIT – II - Perl for Bioinformatics – SBIA1304

UNIT I INTRODUCTION

Biology and Computer Science - Getting Started with Perl - Perl's Benefits - Installing Perl - Running Perl Programs on various platforms - The Art of Programming - Programming Strategies - The Programming Process - Sequences and Strings - Representing Sequence Data - A Program to Store a DNA Sequence - Control Flow - Comments Revisited - Command Interpretation – Statements – Variables – Strings

Concatenating DNA Fragments

Now we'll make a simple modification of Example 4-1 to show how to concatenate two DNA fragments. Concatenation is attaching something to the end of something else. A biologist is well aware that joining DNA sequences is a common task in the biology lab, for instance when a clone is inserted into a cell vector or when splicing exons together during the expression of a gene. Many bioinformatics software packages have to deal with such operations; hence its choice as an example.

Example 4-2 demonstrates a few more things to do with strings, variables, and print statements.

Example 4-2. Concatenating DNA

```
#!/usr/bin/perl -w
# Concatenating DNA
# Store two DNA fragments into two variables called $DNA1
and $DNA2
$DNA1 = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
$DNA2 = 'ATAGTGCCGTGAGAGTGATGTAGTA';
# Print the DNA onto the screen
print "Here are the original two DNA fragments:\n\n";
print $DNA1, "\n";
print $DNA2, "\n\n";
# Concatenate the DNA fragments into a third variable and
print them
# Using "string interpolation"
$DNA3 = "$DNA1$DNA2";
print "Here is the concatenation of the first two fragments
(version 1):\n\n";
print "$DNA3\n\n";
# An alternative way using the "dot operator":
# Concatenate the DNA fragments into a third variable and
print them
$DNA3 = $DNA1 . $DNA2;
print "Here is the concatenation of the first two fragments
(version 2):\n\n";
print "$DNA3\n\n";
# Print the same thing without using the variable $DNA3
print "Here is the concatenation of the first two fragments
(version 3):\n\n";
```

```
print $DNA1, $DNA2, "\n";

exit;
```

As you can see, there are three variables here, $DNA1, $DNA2, and $DNA3. I've added print statements for a running commentary, so that the output of the program that appears on the computer screen makes more sense and isn't simply some DNA fragments one after the other.

Here's what the output of Example 4-2 looks like:

```
Here are the original two DNA fragments:

ACGGGAGGACGGGAAAATTACTACGGCATTAGC
ATAGTGCCGTGAGAGTGATGTAGTA
Here is the concatenation of the first two fragments
(version 1):
ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGT
AGTA
Here is the concatenation of the first two fragments
(version 2):
ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGT
AGTA
Here is the concatenation of the first two fragments
(version 3):
ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGT
AGTA
```

Example 4-2 has many similarities to Example 4-1. Let's look at the differences. To start with, the print statements have some extra, unintuitive parts:

```
print $DNA1, "\n";

print $DNA2, "\n\n";
```

The print statements have variables containing the DNA, as before, but now they also have a comma and then "\n" or "\n\n". These are instructions to print newlines. A newline is invisible on the page or screen, but it tells the computer to go on to the beginning of the next line for subsequent printing. One newline, "\n", simply positions you at the beginning of the next line. Two new lines, "\n\n", moves to the next line and then positions you at the beginning of the line after that, leaving a blank line in between.

Look at the code for Example 4-2 and to make sure you see what these newline

directives do to the output. A blank line is a line with nothing printed on it. Depending on your operating system, it may be just a newline character or a combination formfeed and carriage return (in which cases, it may also be called an empty line), or it may include nonprinting whitespace characters such as spaces and tabs. Notice that the newlines are enclosed in double quotes, which means they are parts of strings. (Here's one difference between single and double quotes, as mentioned earlier: "\n" prints a newline; '\n' prints \n as written.)

Notice the comma in the print statement. A comma separates items in a list. The print statement prints all the items that are listed. Simple as that.

Now let's look at the statement that concatenates the two DNA fragments $DNA1 and $DNA2 into the variable $DNA3:

```
$DNA3 = "$DNA1$DNA2";
```

The assignment to $DNA3 is just a typical assignment as you saw in Example 4-1, a

variable name followed by the = sign, followed by a value to be assigned.

The value to the right of the assignment statement is a string enclosed in double quotes. The double quotes allow the variables in the string to be replaced with their values. This is called string interpolation .[2] So, in effect, the string here is just the DNA of variable $DNA1, followed directly by the DNA of variable $DNA2. That concatenation of the two DNA fragments is then assigned to variable $DNA3.

[2] There are occasions when you might add curly braces during string interpolation. The extra curly braces make sure the variable names aren't confused with anything else in the double- quoted string. For example, if you had variable $prefix and tried to interpolate it into the string I am $prefixinterested, Perl might not recognize the variable, confusing it with a nonexistent variable $prefixinterested. But the string I am ${prefix}interested is unambiguous to Perl.

After assigning the concatenated DNA to variable $DNA3, you print it out, followed by a blank line:

```
print "$DNA3\n\n";
```

One of the Perl catch phrases is, "There's more than one way to do it." So, the next part of the program shows another way to concatenate two strings, using the dot operator. The dot operator, when placed between two strings, creates a single string that concatenates the two original strings. So the line:

```
$DNA3 = $DNA1 . $DNA2;
```

illustrates the use of this operator.

An operator in a computer language takes some arguments—in this case, the strings $DNA1 and $DNA2—and does something to them, returning a value—in this case, the concatenated string placed in the variable $DNA3. The most familiar operators from arithmetic—plus, minus, multiply, and divide—are all operators that take two numbers as arguments and return a number as a value.

Finally, just to exercise the different parts of the language, let's accomplish the same concatenation using only the print statement:

```
print $DNA1, $DNA2, "\n";
```

Here the print statement has three parts, separated by commas: the two DNA fragments in the two variables and a newline. You can achieve the same result with the following print statement:

```
print "$DNA1$DNA2\n";
```

Maybe the Perl slogan should be, "There are more than two ways to do it."

Before leaving this section, let's look ahead to other uses of Perl variables. You've seen the use of variables to hold strings of DNA sequence data. There are other types of data, and programming languages need variables for them, too. In Perl, a scalar variable such as $DNA can hold a string, an integer, a floating-point number (with a decimal point), a boolean (true or false) value, and more. When it's required, Perl figures out what kind of data is in the variable. For now, try adding the following lines to Example 4-1 or Example 4-2, storing a number in a scalar variable and printing it out:

```
$number = 17;
print $number,"\n";
```

Transcription: DNA to RNA

A large part of what you, the Perl bioinformatics programmer, will spend your time doing amounts to variations on the same theme as Examples 4-1 and 4-2. You'll get some data, be it DNA, proteins, GenBank entries, or what have you; you'll manipulate the data; and you'll print out some results.

Example 4-3 is another program that manipulates DNA; it transcribes DNA to RNA. In the cell, this transcription of DNA to RNA is the outcome of the workings of a delicate, complex, and error-correcting molecular machinery.[3] Here it's a simple substitution. When DNA is transcribed to RNA, all the T's are changed to U's, and that's all that our program needs to know.[4]

Briefly, the coding DNA strand is the reverse complement of the other strand, which is used as a template to synthesize its reverse complement as RNA, with T's replaced as U's. With the two reverse complements, this is the same as the coding strand with the T[4] We're ignoring the mechanism of the splicing out of introns, obviously. The T stands for thymine; the U stands for uracil.

Example 4-3. Transcribing DNA into RNA

```
#!/usr/bin/perl -w
# Transcribing DNA into RNA
# The DNA
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
# Print the DNA onto the screen
print "Here is the starting DNA:\n\n";
print "$DNA\n\n";
# Transcribe the DNA to RNA by substituting all T's with
U's.
$RNA = $DNA;
$RNA =~ s/T/U/g;
# Print the RNA onto the screen
print "Here is the result of transcribing the DNA to
RNA:\n\n";
print "$RNA\n";
# Exit the program.

exit;
```

Here's the output of Example 4-3: Here is the starting DNA:

```
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
Here is the result of transcribing the DNA to RNA:
ACGGGAGGACGGGAAAAUUACUACGGCAUUAGC
```

This short program introduces an important part of Perl: the ability to easily manipulate text data such as a string of DNA. The manipulations can be of many different sorts: translation, reversal, substitution, deletions, reordering, and so on. This facility of Perl is one of the main reasons for its success in bioinformatics and among programmers in general.

First, the program makes a copy of the DNA, placing it in a variable called $RNA:

```
$RNA = $DNA;
```

Note that after this statement is executed, there's a variable called $RNA that actually contains DNA.[5] Remember this is perfectly legal—you can call variables anything you like—but it is potentially confusing to have inaccurate variable names. Now in this case,

the copy is preceded with informative comments and followed immediately with a statement that indeed causes the variable $RNA to contain RNA, so it's all right. Here's a way to prevent $RNA from containing anything except RNA:

```
($RNA = $DNA) =~ s/T/U/g;
```

In Example 4-3, the transcription happens in this statement: $RNA =~ s/T/U/g;

There are two new items in this statement: the binding operator (=~) and the substitute command s/T/U/g.

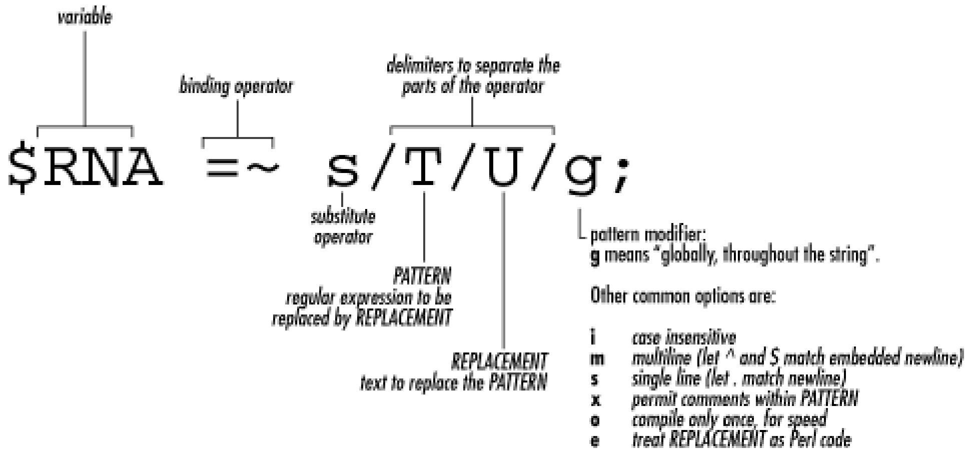
The binding operator =~ is used, obviously enough, on variables containing strings; here the variable $RNA contains DNA sequence data. The binding operator means "apply the operation on the right to the string in the variable on the left."

The substitution operator , shown in Figure 4-1, requires a little more explanation. The different parts of the command are separated (or delimited) by the forward slash. First, the s indicates this is a substitution. After the first / comes a T, which represents the element in the string that will be substituted. After the second / comes a U, which represents the element that's going to replace the T. Finally, after the third / comes g. This g stands for "global" and is one of several possible modifiers that can appear in this part of the statement. Global means "make this substitution throughout the entire string," that is to say, everywhere possible in the string.

Figure 4-1. The substitution operator

Thus, the meaning of the statement is: "substitute all T's for U's in the string data stored in

[5] Recall the discussion in Section 4.2.4.3 about the importance of the order of the parts in an assignment statement. Here, the value of $DNA, that is, the DNA sequence data that has been stored in the $DNA variable, is being assigned to the variable $RNA. If you had written $DNA = $RNA;, the value of the $RNA variable (which is empty) would have been assigned to the $DNA variable, in effect wiping out the DNA sequence data in that variable and leaving two empty variables.


variable
binding operator
delimiters to separate the parts of the operator
$RNA =~ s/T/U/g;
substitute operator
PATTERN regular expression to be replaced by REPLACEMENT
REPLACEMENT text to replace the PATTERN
pattern modifier:
g means "globally, throughout the string".
Other common options are:
i case insensitive
m multiline (let ^ and $ match embedded newline)
s single line (let . match newline)
x permit comments within PATTERN
o compile only once, for speed
e treat REPLACEMENT as Perl code

the variable $RNA."

The substitution operator is an example of the use of regular expressions. Regular expressions are the key to text manipulation, one of the most powerful features of Perl as you'll see in later chapters.

Using the Perl Documentation

A Perl programmer's most important resource is the Perl documentation. It should be installed on your computer, and it may also be found on the Internet at the Perl site. The Perl documentation may come in slightly different forms on your computer system, but the web version is the same for everybody. That's the version I refer to in this book. See the references in Appendix A for more discussion about different sources of Perl documentation.

Just to try it out, let's look up the print operator. First, open your web browser, and go to http://www.perl.com. Then click on the Documentation link. Select "Perl's Builtin Functions" and then "Alphabetical Listing of Perl's Functions". You'll see a rather lengthy alphabetical listing of Perl's functions. Once you've found this page, you may want to bookmark it in your browser, as you may find yourself turning to it frequently. Now click on Print to view the print operator.

Check out the examples they give to see how the language feature is actually used. This is usually the quickest way to extract what you need to know.

Once you've got the documentation on your screen, you may find that reading it answers some questions but raises others. The documentation tends to give the entire story in a concise form, and this can be daunting for beginners. For instance, the documentation for the *print* function starts out: "Prints a string or a comma-separated list of strings. Returns TRUE if successful." But then comes a bunch of gibberish (or so it seems at this point in your learning curve!) Filehandles? Output streams? List context?

All this information is necessary in documentation; after all, you need to get the whole story somewhere! Usually you can ignore what doesn't make sense.

The Perl documentation also includes several tutorials that can be a great help in learning Perl. They occasionally assume more than a beginner's knowledge about programming languages, but you may find them very useful. Exploring the documentation is a great way to get up to speed on the Perl language.

Calculating the Reverse Complement in Perl

As you recall from Chapter 1, a DNA polymer is composed of nucleotides. Given the close relationship between the two strands of DNA in a double helix, it turns out that it's pretty straightforward to write a program that, given one strand, prints out the other. Such a calculation is an important part of many bioinformatics applications. For instance, when searching a database with some query DNA, it is common to automatically search for the reverse complement of the query as well, since you may have in hand the opposite strand of some known gene.

Without further ado, here's Example 4-4, which uses a few new Perl features. As you'll see, it first tries one method, which fails, and then tries another method, which succeeds.

Example 4-4. Calculating the reverse complement of a strand of DNA

```
#!/usr/bin/perl -w
# Calculating the reverse complement of a strand of DNA
# The DNA
$DNA = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
# Print the DNA onto the screen
print "Here is the starting DNA:\n\n";
print "$DNA\n\n";
# Calculate the reverse complement
#  Warning: this attempt will fail!
#
# First, copy the DNA into new variable $revcom
# (short for REVerse COMplement)
# Notice that variable names can use lowercase letters like
# "revcom" as well as uppercase like "DNA".  In fact,
# lowercase is more common.
#
# It doesn't matter if we first reverse the string and then
# do the complementation; or if we first do the
complementation
# and then reverse the string.  Same result each time.
# So when we make the copy we'll do the reverse in the same
statement.
#
$revcom = reverse $DNA;
#
# Next substitute all bases by their complements,
# A->T, T->A, G->C, C->G
#
$revcom =~ s/A/T/g;
$revcom =~ s/T/A/g;
$revcom =~ s/G/C/g;
```

```
$revcom =~ s/C/G/g;
# Print the reverse complement DNA onto the screen
print "Here is the reverse complement DNA:\n\n";
print "$revcom\n";

#
# Oh-oh, that didn't work right!
# Our reverse complement should have all the bases in it, since the
# original DNA had all the bases--but ours only has A and G! #
# Do you see why?
#
# The problem is that the first two substitute commands above change
# all the A's to T's (so there are no A's) and then all the # T's to A's (so all the original A's and
T's are all now A's).
# Same thing happens to the G's and C's all turning into G's.
#

print "\nThat was a bad algorithm, and the reverse
complement was wrong!\n";
print "Try again ... \n\n";
# Make a new copy of the DNA (see why we saved the
original?)
$revcom = reverse $DNA;
# See the text for a discussion of tr///
$revcom =~ tr/ACGTacgt/TGCAtgca/;
# Print the reverse complement DNA onto the screen
print "Here is the reverse complement DNA:\n\n";
print "$revcom\n";
print "\nThis time it worked!\n\n";

exit;
```

Here's what the output of Example 4-4 should look like on your screen:

```
Here is the starting
DNA:

ACGGGAGGACGGGAAAATTACTACGGCATTAGC
Here is the reverse complement DNA:
GGAAAAGGGGAAGAAAAAAAGGGGAGGAGGGGA
That was a bad algorithm, and the reverse complement was
wrong!
Try again ...
Here is the reverse complement DNA:
GCTAATGCCGTAGTAATTTTCCCGTCCTCCCGT
This time it worked!
```

You can check if two strands of DNA are reverse complements of each other by reading one left to right, and the other right to left, that is, by starting at different ends. Then compare each pair of bases as you read the two strands: they should always be paired C to G and A to T.

Just by reading in a few characters from the starting DNA and the reverse complement DNA from the first attempt, you'll see the that first attempt at calculating the reverse complement failed. It was a bad algorithm.

This is a taste of what you'll sometimes experience as you program. You'll write a program to accomplish a job and then find it didn't work as you expected. In this case, we used parts of the language we already knew and tried to stretch them to handle a new problem. Only they weren't quite up to the job. What went wrong?

You'll find that this kind of experience becomes familiar: you write some code, and it doesn't work! So you either fix the syntax (that's usually the easy part and can be done from the clues the error messages provide), or you think about the problem some more, find why the program failed, and then try to devise a new and successful way. Often this requires browsing the language documentation, looking for the details of how the language works and hoping to find a feature that fixes the problem. If it can be solved on a computer, you can solve it using Perl. The trick is, how exactly?

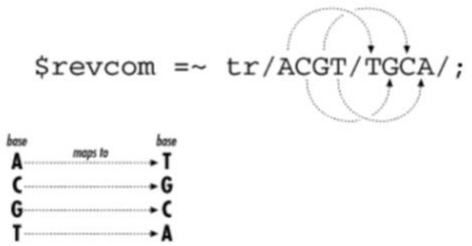
In Example 4-4, the first attempt to calculate the reverse complement failed. Each base in the string was translated as a whole, using four substitutions in a global fashion. Another way is needed. You could march though the DNA left to right, look at each base one at a time, make the change to the complement, and then look at the next base in the DNA, marching on to the end of the string. Then just reverse the string, and you're done. In fact, this is a perfectly good method, and it's not hard to do in Perl, although it requires some parts of the language not found until Chapter 5.

However, in this case, the tr operator—which stands for transliterate or translation—is exactly suited for this task. It looks like the substitute command, with the three forward slashes separating the different parts.

tr does exactly what's needed; it translates a set of characters into new characters, all at once. Figure 4-2 shows how it works: the set of characters to be translated are between the first two forward slashes. The set of characters that replaces the originals are between the second and third forward slashes. Each character in the first set is translated into the character at the same position in the second set. For instance, in Example 4-4, C is the second character in the first set, so it's translated into the second character of the second set, namely, G. Finally, since DNA sequence data can use upper- or lowercase letters (even though in this program the DNA is in uppercase only), both cases are included in the *tr* statement in Example 4-4.

Figure 4-2. The tr statement

Figure 4-2. The tr statement



The reverse function also does exactly what's needed, with a minimum of fuss. It's designed to reverse the order of elements, including strings as seen in Example 4-4.

Proteins, Files, and Arrays

So far we've been writing programs with DNA sequence data. Now we'll also include the equally important protein sequence data. Here's an overview of what is covered in the following sections:

How to use protein sequence data in a Perl program
How to read protein sequence data in from a file
Arrays in the Perl language
For the rest of the chapter, both protein and DNA sequence data are used.

Reading Proteins in Files

Programs interact with files on a computer disk. These files can be on hard disk, CD, floppy disk, Zip drive, magnetic tape—any kind of permanent storage.

Let's take a look at how to read protein sequence data from a file. First, create a file on your computer (use your text editor) and put some protein sequence data into it. Call the file *NM_021964fragment.pep* (you can download it from this book's web site). You will be using the following data (part of the human zinc finger protein NM_021964):
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERL
PQ GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR

You can use any name, except one that's already in use in the same folder.

Just as well-chosen variable names can be critical to understanding a program, well- chosen file and folder names can also be critical. If you have a project that generates lots of computer files, you need to carefully consider how to name and organize the files and folders. This is as true for individual researchers as for large, multi-national teams. It's important to put some effort into assigning informative names to files.

The filename NM_021964fragment.pep is taken from the GenBank ID of the record where this protein is found. It also indicates the fragmentary nature of the data and contains the filename extension .pep to remind you that the file contains peptide or protein sequence data. Of course, some other scheme might work better for you; the point is to get some idea of what's in the file without having to look into it.

Now that you've created or downloaded a file with protein sequence data in it, let's develop a program that reads the protein sequence data from the file and stores it into a variable. Example 4-5 shows a first attempt, which will be added to as we progress.

Example 4-5. Reading protein sequence data from a file

```
#!/usr/bin/perl -w
# Reading protein sequence data from a file
# The filename of the file containing the protein sequence
data
$proteinfilename = 'NM_021964fragment.pep';
# First we have to "open" the file, and associate
# a "filehandle" with it.  We choose the filehandle
# PROTEINFILE for readability.
open(PROTEINFILE, $proteinfilename);
# Now we do the actual reading of the protein sequence data
from the file,
# by using the angle brackets < and > to get the input from
the

# filehandle. We store the data into our variable $protein. $protein = <PROTEINFILE>;

# Now that we've got our data, we can close the file.
close PROTEINFILE;
# Print the protein onto the screen
print "Here is the protein:\n\n";
print $protein;
exit;
```

Here's the output of Example 4-5: Here is the protein:

```
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
```

Notice that only the first line of the file prints out. I'll show why in a moment.

Let's look at Example 4-5 in more detail. After putting a filename into the variable $proteinfilename, the file is opened with the following statement: open(PROTEINFILE, $proteinfilename);

After opening the file, you can do various things with it, such as reading, writing, searching, going to a specific location in the file, erasing everything in the file, and so on. Notice that the program assumes the file named in the variable $proteinfilename exists and can be opened. You'll see in a little bit how to check for that, but here's something to try: change the name of the filename in $proteinfilename so that there's no file of that name on your computer, and then run the program. You'll get some error messages if the file doesn't exist.

If you look at the documentation for the open function, you'll see many options. Mostly, they enable you to specify exactly what the file will be used for after it's opened.

Let's examine the term PROTEINFILE, which is called a filehandle. With filehandles, it's not important to understand what they really are. They're just things you use when you're dealing with files. They don't have to have capital letters, but it's a widely followed convention. After the *open* statement assigns a filehandle, all the interaction with a file is done by naming the filehandle.

The data is actually read in to the program with the statement:

```
$protein = <PROTEINFILE>;
```

Why is the filehandle PROTEINFILE enclosed within angle brackets? These angle brackets are called input operators; a filehandle within angle brackets is how you bring in data from some source outside the program. Here, we're reading the file called *NM_021964fragment.pep* whose name is stored in variable $proteinfilename, and which has a filehandle associated with it by the open statement. The data is being stored in the variable $protein and then printed out.

However, as you've already noticed, only the first line of this multiline file is printed out. Why? Because there are a few more things to learn about reading in files.

There are several ways to read in a whole file. Example 4-6 shows one way.

Example 4-6. Reading protein sequence data from a file, take 2

```
#!/usr/bin/perl -w
# Reading protein sequence data from a file, take 2
# The filename of the file containing the protein sequence
data
$proteinfilename = 'NM_021964fragment.pep';
# First we have to "open" the file, and associate
# a "filehandle" with it.  We choose the filehandle
# PROTEINFILE for readability.
open(PROTEINFILE, $proteinfilename);
# Now we do the actual reading of the protein sequence data
from the file,
# by using the angle brackets < and > to get the input from
the

# filehandle. We store the data into our variable $protein. #
# Since the file has three lines, and since the read only
is

# returning one line, we'll read a line and print it, three
times.
# First line
$protein = <PROTEINFILE>;
# Print the protein onto the screen
print "\nHere is the first line of the protein file:\n\n";
print $protein;
# Second line
$protein = <PROTEINFILE>;
```

```
# Print the protein onto the screen
print "\nHere is the second line of the protein file:\n\n";
print $protein;
# Third line
$protein = <PROTEINFILE>;
# Print the protein onto the screen
print "\nHere is the third line of the protein file:\n\n";
print $protein;
# Now that we've got our data, we can close the file.
close PROTEINFILE;

exit;
```

Here's the output of Example 4-6:

```
Here is the first line of the protein file:

MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
Here is the second line of the protein file:
SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERL
PQ
Here is the third line of the protein file:
GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

The interesting thing about this program is that it shows how reading from a file works. Every time you read into a scalar variable such as $protein, the next line of the file is read. Something is remembering where the previous read was and is picking it up from there.

On the other hand, the drawbacks of this program are obvious. Having to write a few lines of code for each line of an input file isn't convenient. However, there are two Perl features that can handle this nicely: arrays (in the next section) and loops (in Chapter 5).

Arrays

In computer languages an array is a variable that stores multiple scalar values. The values can be numbers, strings, or, in this case, lines of an input file of protein sequence data. Let's examine how they can be used. Example 4-7 shows how to use an array to read all the lines of an input file.

Example 4-7. Reading protein sequence data from a file, take 3

```
#!/usr/bin/perl -w
# Reading protein sequence data from a file, take 3
# The filename of the file containing the protein sequence
data
$proteinfilename = 'NM_021964fragment.pep';
# First we have to "open" the file
open(PROTEINFILE, $proteinfilename);
# Read the protein sequence data from the file, and store
it
# into the array variable @protein
```

```
@protein = <PROTEINFILE>;
# Print the protein onto the screen
print @protein;
# Close the file.
close PROTEINFILE;

exit;
```

Here's the output of Example 4-7:

```
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERL
PQ GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

which, as you can see, is exactly the data that's in the file. Success!

The convenience of this is clear—just one line to read all the data into the program.

Notice that the array variable starts with an at sign (@) rather than the dollar sign ($) scalar variables begin with. Also notice that the print function can handle arrays as well as scalar variables. Arrays are used a lot in Perl, so you will see plenty of array examples as the book continues.

An array is a variable that can hold many scalar values. Each item or element is a scalar value that can be referenced by giving its position in the array (its subscript or offset). Let's look at some examples of arrays and their most common operations. We'll define an array @bases that holds the four bases A, C, G, and T. Then we'll apply some of the most common array operators.

Here's a piece of code that demonstrates how to initialize an array and how to use subscripts to access the individual elements of an array:

```
# Here's one way to declare an array, initialized with a
list of four scalar values.
@bases = ('A', 'C', 'G', 'T');
# Now we'll print each element of the array
print "Here are the array elements:";
print "\nFirst element: ";
print $bases[0];
print "\nSecond element: ";
print $bases[1];
print "\nThird element: ";
print $bases[2];
print "\nFourth element: ";
print $bases[3];
```

This code snippet prints out:

```
First element: A
Second element: C
Third element: G
```

```
Fourth element: T
```

You can print the elements one a after another like this:

```
@bases = ('A', 'C', 'G', 'T');
print "\n\nHere are the array elements: ";
print @bases;
```

which produces the output:

```
Here are the array elements: ACGT
```

You can also print the elements separated by spaces (notice the double quotes in the print statement):

```
@bases = ('A', 'C', 'G', 'T');
print "\n\nHere are the array elements: ";

print "@bases";
```

which produces the output:

Here are the array elements: A C G T You can take an element off the end of an array with pop:

```
@bases = ('A', 'C', 'G', 'T');
$base1 = pop @bases;
print "Here's the element removed from the end: ";
print $base1, "\n\n";
print "Here's the remaining array of bases: ";
print "@bases";
```

which produces the output:

```
Here's the element removed from the end: T
Here's the remaining array of bases: A C G
```

You can take a base off of the beginning of the array with shift:

```
@bases = ('A', 'C', 'G', 'T');
$base2 = shift @bases;
print "Here's an element removed from the beginning: "; print $base2, "\n\n";

print "Here's the remaining array of bases: ";
print "@bases";
```

which produces the output:

```
Here's an element removed from the beginning: A
```

```
Here's the remaining array of bases: C G T
```

You can put an element at the beginning of the array with unshift:

```
@bases = ('A', 'C', 'G', 'T');
$base1 = pop @bases;
unshift (@bases, $base1);
print "Here's the element from the end put on the beginning: ";

print "@bases\n\n";
```

which produces the output:

```
Here's the element from the end put on the beginning: T A C G
```

You can put an element on the end of the array with push:

```
@bases = ('A', 'C', 'G', 'T');

$base2 = shift @bases;
push (@bases, $base2);
print "Here's the element from the beginning put on the end: ";
print "@bases\n\n";
```

which produces the output:

```
Here's the element from the beginning put on the end: C G T
A
```

You can reverse the array:

```
@bases = ('A', 'C', 'G', 'T');
@reverse = reverse @bases;
print "Here's the array in reverse: ";
print "@reverse\n\n";
```

which produces the output:

```
Here's the array in reverse: T G C A
```

You can get the length of an array:

```
@bases = ('A', 'C', 'G', 'T');
print "Here's the length of the array: ";
print scalar @bases, "\n";
```

which produces the output:

```
Here's the length of the array: 4
```

Here's how to insert an element at an arbitrary place in an array using the Perl splice

function:

```
@bases = ('A', 'C', 'G', 'T');
splice ( @bases, 2, 0, 'X');
print "Here's the array with an element inserted after the
2nd element: ";
print "@bases\n";
```

which produces the output:

```
Here's the array with an element inserted after the 2nd
element: A C X G T
```

Scalar and List Context

Many Perl operations behave differently depending on the context in which they are used. Perl has scalar context and list context; both are listed in Example 4-8.

Example 4-8. Scalar context and list context

```
#!/usr/bin/perl -w
# Demonstration of "scalar context" and "list context"
@bases = ('A', 'C', 'G', 'T');
print "@bases\n";
$a = @bases;
print $a, "\n";
($a) = @bases;
print $a, "\n";

exit;
```

Here's the output of Example 4-8: ACGT

```
4
A
```

First, Example 4-8 declares an array of the four bases. Then the assignment statement tries to assign an array (which is a kind of list) to a scalar variable $a:

```
$a = @bases;
```

In this kind of scalar context , an array evaluates to the size of the array, that is, the number of elements in the array. The scalar context is supplied by the scalar variable on the left side of the statement.

Next, Example 4-8 tries to assign an array (to repeat, a kind of list) to another list, in this case, having just one variable, $a:

```
($a) = @bases;
```

In this kind of list context , an array evaluates to a list of its elements. The list context is supplied by the list in parentheses on the left side of the statement. If there aren't enough variables on the left side to assign to, only part of the array gets assigned to variables. This behavior of Perl pops up in many situations; by design, many features of Perl behave differently depending on whether they are in scalar or list context. See Appendix B for more about scalar and list content.

Now you've seen the use of strings and arrays to hold sequence and file data, and learned the basic syntax of Perl, including variables, assignment, printing, and reading files. You've transcribed DNA to RNA and calculated the reverse complement of a strand of DNA. By the end of Chapter 5, you'll have covered the essentials of Perl programming.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF BIO AND CHEMICAL ENGINEERING

DEPARTMENT OF BIOINFORMATICS

UNIT – III - Perl for Bioinformatics – SBIA1304

UNIT III WRITING MORE CODE
Finding Motifs - Regular Expressions - Counting Nucleotides - Exploding Strings into Arrays - Operating on Strings - Writing to Files - Subroutines and Bugs - Subroutines - Scoping and Subroutines - Command-Line Arguments and Arrays - Passing Data to Subroutines - Modules and Libraries of Subroutines - Fixing Bugs in Your Code - The Perl Debugger

Motifs and Loops

This chapter continues demonstrating the basics of the Perl language begun in Chapter 4. By the end of the chapter, you will know how to:

Search for motifs in DNA or protein Interact with users at the keyboard Write data to files Use loops

Use basic regular expressions
Take different actions depending on the outcome of conditional tests Examine sequence data in detail by operating on strings and arrays

These topics, in addition to what you learned in Chapter 4, will give you the skills necessary to begin to write useful bioinformatics programs; in this chapter, you will learn to write a program that looks for motifs in sequence data.

Flow Control

Flow control is the order in which the statements of a program are executed. A program executes from the first statement at the top of the program to the last statement at the bottom, in order, unless told to do otherwise. There are two ways to tell a program to do otherwise: conditional statements and loops. A conditional statement executes a group of statements only if the conditional test succeeds; otherwise, it just skips the group of statements. A loop repeats a group of statements until an associated test fails.

Conditional Statements

Let's take another look at the open statement. Recall that if you try to open a nonexistent file, you get error messages. You can test for the existence of a file explicitly, before trying to open it. In fact, such tests are among the most powerful features of computer languages. The if , if-else, and unless conditional statements are three such testing mechanisms in Perl.

The main feature of these kinds of constructs is the testing for a conditional. A conditional evaluates to a `true` or `false` value. If the conditional is `true`, the statements following are executed; if the conditional is `false`, they are skipped (or vice versa).

However, "What is truth?" It's a question that programming languages may answer in

slightly different ways.

This section contains a few examples that demonstrate some of Perl's conditionals. The true-false condition in each example is equality between two numbers. Notice that equality of numbers is represented by two equal signs `==`, because the single equal sign `=` is already used for assignment to a variable.

Confusion between = for assignment and == for numeric equality is a frequent programming bug, so watch for it!

The following examples demonstrate whether the conditional will evaluate to `true` or `false`. You don't ordinarily have much use for such simple tests. Usually you test the values that have been read into variables or the return value of function calls—things you don't necessarily know beforehand.

The `if` statement with a `true` conditional: `if( 1 == 1) {`

```
    print "1 equals 1\n\n";
}
```

produces the output:

```
1 equals 1
```

The test is `1 == 1`, or, in English, "Does 1 equal 1?" Since it does, the conditional evaluates to `true`, the statement associated with the `if` statement is executed, and a message is printed out.

You can also just say:

```
if( 1) {
    print "1 evaluates to true\n\n";

}
```

which produces the output:

```
1 evaluates to true
```

The `if` statement with a `false` conditional: `if( 1 == 0) {`

```
    print "1 equals 0\n\n";
}
```

produces no output! The test is `1 == 0` or, in English, "Does 1 equal 0?" Since it doesn't, the conditional evaluates to `false`, the statements associated with the `if` statement aren't executed, and no message is printed out.

You can also just say:

```
if( 0 ) {

    print "0 evaluates to true\n\n";
}
```

which produces no output, since 0 evaluates to `false`, so the statements associated with the `if` statement are skipped entirely.

There's another way to write short `if` statements that mirrors how the English language works. In English, you can say, equivalently, "If you build it, they will come" or "They will come if you build it." Not to be outdone, Perl also allows you to put the `if` after the action:

```
print "1 equals 1\n\n" if (1 == 1);
```

which does the same thing as the first example in this section and prints out:

```
1 equals 1
```

Now, let's look at an `if-else` statement with a `true` conditional: `if( 1 == 1) {`

```
    print "1 equals 1\n\n";
} else {
    print "1 does not equal 1\n\n";
}
```

which produces the output:

```
1 equals 1
```

The `if-else` does one thing if the test evaluates to `true` and another if the test evaluates to `false`. Here is `if-else` with a `false` conditional: `if( 1 == 0) {`

```
    print "1 equals 0\n\n";
} else {
    print "1 does not equal 0\n\n";
}
```

which produces the output:

```
1 does not equal 0
```

The final example is `unless`—the opposite of `if`. It works like the English word "unless": e.g., "Unless you study Russian literature, you are ignorant of Chekov." If the conditional evaluates to `true`, no action is taken; if it evaluates to `false`, the associated statements are executed. If "you study Russian literature" is `false`, "you are ignorant of Chekov."

```
unless( 1 == 0) {

    print "1 does not equal 0\n\n";

}
```

produces the output:

```
1 does not equal 0
```

Conditional tests and matching braces

Two more comments are in order about these statements and their conditional tests.

First, there are several tests that can be used in the conditional part of these statements. In addition to numeric equality `==` as in the previous example, you can also test for inequality `!=`, greater than `>`, less than `<`, and more.

Similarly, you can test for string equality using the eq operator: if two strings are the same, it's `true`. There are also file test operators that allow you to test if a file exists, is empty, if permissions are set a certain way, and so on (see Appendix B). One common test is just a variable name: if the variable contains zero, it's considered `false`; any other number evaluates to `true`. If the variable contains a nonempty string, it evaluates to `true`; the empty string, designated by `""` or `''`, is `false`.

Second, notice that the statements that follow the conditional are enclosed within a matching pair of curly braces. These statements

within curly braces are called a block and arise frequently in Perl.[1] Matching pairs of parentheses, brackets, or braces, i.e., ( ), [ ], < >, and { }, are common programming features. Having the same number of left and right braces in the right places is essential for a Perl program to run correctly.

[1] As something of an oddity, the last statement within a block doesn't need a semicolon after it.

Matching braces are easy to lose track of, so don't be surprised if you miss some and get error messages when you try to run the program. This is a common syntax error; you have to go back and find the missing brace. As code gets more complex, it can be a challenge to figure out where the matching braces are wrong and how to fix them. Even if the braces are in the right place, it can be hard to figure out what statements are grouped together when you're reading code. You can avoid this problem by writing code that doesn't try to do too much on any one line and uses indentation to further highlight the blocks of code (see Section 5.2).[2]

[2] Some text editors help you find a matching brace (for instance, in the *vi* editor, hitting the percent key % over a parenthesis bounces you to the matching parenthesis).

Back to the conditional statements. The *if-else* also has an *if-elsif-else* form, as in Example 5-1. The conditionals, first the *if* and then the *elsif*s, are evaluated in turn, and as soon as one evaluates to `true`, its block is executed, and the rest of the conditionals are ignored. If none of the conditionals evaluates to `true`, the *else* block is

executed if there is one—it's optional.

Example 5-1. if-elsif-else

```
#!/usr/bin/perl -w
# if-elsif-else
$word = 'MNIDDKL';
# if-elsif-else conditionals
```

```
if($word eq 'QSTVSGE') {
    print "QSTVSGE\n";
} elsif($word eq 'MRQQDMISHDEL') {
    print "MRQQDMISHDEL\n";
} elsif ( $word eq 'MNIDDKL' ) {
    print "MNIDDKL--the magic word!\n";
} else {
    print "Is \"$word\" a peptide? This program is not
sure.\n";

}

exit;
```

Notice the `\"` in the *else* block's `print` statement; it lets you print a double-quote sign (`"`) within a double-quoted string. The backslash character tells Perl to treat the following `"` as the sign itself and not interpret it as the marker for the end of the string. Also note the use of *eq* to check for equality between strings.

Example 5-1 gives the output:

```
MNIDDKL--the magic word!
```

5.1.2 Loops

A loop allows you to repeatedly execute a block of statements enclosed within matching curly braces. There are several ways to loop in Perl: *while* loops, *for* loops, *foreach* loops, and more. Example 5-2 (from Chapter 4) displays the *while* loop and how it's used while reading protein sequence data in from a file.

Example 5-2. Reading protein sequence data from a file, take 4

```
#!/usr/bin/perl -w
# Reading protein sequence data from a file, take 4
# The filename of the file containing the protein sequence
data
$proteinfilename = 'NM_021964fragment.pep';
# First we have to "open" the file, and in case the
# open fails, print an error message and exit the program.
unless ( open(PROTEINFILE, $proteinfilename) ) {
    print "Could not open file $proteinfilename!\n";

exit; }

# Read the protein sequence data from the file in a "while"
loop,
# printing each line as it is read.
while( $protein = <PROTEINFILE> ) {
    print "  ######  Here is the next line of the file:\n";
    print $protein;
}
# Close the file.
close PROTEINFILE;
```

```
exit;
```

Here's the output of Example 5-2:

```
###### Here is the next line of the file:

MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
  ######  Here is the next line of the file:
SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
  ######  Here is the next line of the file:

GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

In the *while* loop, notice how the variable `$protein` is assigned each time through the loop to the next line of the file. In Perl, an assignment returns the value of the assignment. Here, the test is whether the assignment succeeds in reading another line. If there is another line to read in, the assignment occurs, the conditional is `true`, the new line is stored in the variable `$protein`, and the block with the two `print` statements is executed. If there are no more lines, the assignment is undefined, the conditional is `false`, and the program skips the block with the two `print` statements, quits the *while* loop, and continues to the following parts of the program (in this case, the *close* and *exit* functions).

open and unless

The *open* call is a system call, because to open a file, Perl must ask for the file from the operating system. The operating system may be a version of Unix or Linux, a Microsoft Windows versions, one of the Apple Macintosh operating systems, and so on. Files are managed by the operating system and can be accessed only by it.

It's a good habit to check for the success or failure of system calls, especially when opening files. If a system call fails, and you're not checking for it, your program will continue, perhaps attempting to read or write to a file you couldn't open in the first place. You should always check for failure and let the user of the program know right away when a file can't be opened. Often you may want to exit the program on failure or try to open a different file.

In Example 5-2, the *open* system call is part of the test of the *unless* conditional. *unless* is the opposite of *if*. Just as in English you can say "do the statements in the block if the condition is true"; you can also say the opposite, "do the statements in the block unless the condition is true." The *open* system call gives you a true value if it successfully opens the file; so here, in the conditional test of the *unless* statement, if the *open* call fails, the statements in the block are performed, the program prints an error message, and then exits.

To sum up, conditionals and loops are simple ideas and not difficult to learn in Perl. They are among the most powerful features of programming languages. Conditionals allow you to tailor a program to several alternatives, and in that way, make decisions based on the type of input it gets. They are responsible for a large part of whatever artificial intelligence there is in a computer program. Loops harness the speed of the computer so that in a few lines of code, you can handle large amounts of input or continually iterate and refine a computation.

Code Layout

Once you start using loops and conditional statements, you need to think seriously about formatting. You have many options when formatting Perl code on the page. Compare these variant ways of formatting an *if* statement inside a `while` *loop*:

Format A

```
     while ( $alive ) {
         if ( $needs_nutrients ) {
             print "Cell needs nutrients\n";
         }

}
```

Format B

```
     while ( $alive )
     {
         if ( $needs_nutrients )
print "Cell needs nutrients\n";
         }

}
```

Format C

```
while ( $alive )
  {
     if ( $needs_nutrients )
     {
         print "Cell needs nutrients\n";

} }
```

Format D

```
     while($alive){if($needs_nutrients){print "Cell needs
     nutrients\n";}}
```

These code fragments are equivalent as far as the Perl interpreter is concerned. That's because Perl doesn't rely on how the statements are laid out on the lines; Perl cares only about the correct order of the syntactical elements. Some elements need some whitespace (such as spaces, tabs, or newlines) between them to make them distinct, but in general, Perl doesn't restrict how you use whitespace to lay out your code.

Formats A and B are common ways to lay out code. They both make the program structure clear to the human reading it. Notice how the statements that have a block associated with them—the while and if statements—line up the curly braces and indent the statements within the blocks. These layouts make clear the extent of the block associated with the statements. (This can be critical for long, complicated blocks.) The statements inside the blocks are indented, for which you normally use the Tab key or groups of four or eight spaces. (Many text editors allow you to insert spaces when you hit the Tab key, or you can instruct them to set the tab stops at four, eight, or whatever number of spaces.) The overall structure of the program becomes clearer this way; you can easily see which statements are grouped in a block and associated with a given loop or conditional. Personally, I prefer the layout in Format A, although I'm also perfectly happy with Format B.

Format C is an example of badly formatted code. The flow control of the code isn't clear; for instance, it's hard to see if the print statement is in the block of the while statement.

Format D demonstrates how hard it is to read code with essentially no formatting, even a simple fragment like this.

The Perl style guide, available from the main Perl manual page or from the command line by typing:

```
perldoc perlstyle
```

has some recommendations and some suggestions for ways to write readable code.

However, they are not rules, and you may use your own judgment as to the formatting practices that work best for you.

Finding Motifs

One of the most common things we do in bioinformatics is to look for motifs, short segments of DNA or protein that are of particular interest. They may be regulatory elements of DNA or short stretches of protein that are known to be conserved across many species. (The PROSITE web site at http://www.expasy.ch/prosite/ has extensive information about protein motifs.)

The motifs you look for in biological sequences are usually not one specific sequence. They may have several variants—for example, positions in which it doesn't matter which base or residue is present. They may have variant lengths as well. They can often be represented as regular expressions, which you'll see more of in the discussion following Example 5-3, in Chapter 9, and elsewhere in the book.

Perl has a handy set of features for finding things in strings. This, as much as anything, has made it a popular language for bioinformatics. Example 5-3 introduces this string-searching capability; it does something genuinely useful, and similar programs are used all the time in biology research. It does the following:

Reads in protein sequence data from a file

Puts all the sequence data into one string for easy searching

Looks for motifs the user types in at the keyboard

Example 5-3. Searching for motifs

```
#!/usr/bin/perl -w
```

```
# Searching for motifs
# Ask the user for the filename of the file containing
# the protein sequence data, and collect it from the
keyboard
print "Please type the filename of the protein sequence
data: ";
$proteinfilename = <STDIN>;
# Remove the newline from the protein filename
chomp $proteinfilename;
# open the file, or exit
unless ( open(PROTEINFILE, $proteinfilename) ) {
print "Cannot open file \"$proteinfilename\"\n\n";

exit; }

# Read the protein sequence data from the file, and store
it
# into the array variable @protein
@protein = <PROTEINFILE>;
# Close the file - we've read all the data into @protein
now.
close PROTEINFILE;
# Put the protein sequence data into a single string, as
it's easier
# to search for a motif in a string than in an array of
# lines (what if the motif occurs over a line break?)
$protein = join( '', @protein);
# Remove whitespace
$protein =~ s/\s//g;

# In a loop, ask the user for a motif, search for the motif, #
and report if it was found.
# Exit if no motif is entered.
do {

    print "Enter a motif to search for: ";
    $motif = <STDIN>;
    # Remove the newline at the end of $motif
    chomp $motif;
    # Look for the motif
    if ( $protein =~ /$motif/ ) {
        print "I found it!\n\n";
    } else {
        print "I couldn\'t find it.\n\n";
    }
# exit on an empty user input
} until ( $motif =~ /^\s*$/ );
# exit the program

exit;
```

Here's some typical output from Example 5-3:

```
Please type the filename of the protein sequence data:
NM_021964fragment.pep
Enter a motif to search for: SVLQ
I found it!

Enter a motif to search for: jkl
I couldn't find it.
Enter a motif to search for: QDSV
I found it!
Enter a motif to search for: HERLPQGLQ
I found it!
Enter a motif to search for:
I couldn't find it.
```

As you see from the output, this program finds motifs that the user types in at the keyboard. With such a program, you no longer have to search manually through potentially huge amounts of data. The computer does the work and does it much faster and more accurately than a human.

It'd be nice if this program not only reported it found the motif but at what position. You'll see how this can be accomplished in Chapter 9. An exercise in that chapter challenges you to modify this program so that it reports the positions of the motifs.

The following sections examine and discuss the parts of Example 5-3 that are new:
Getting user input from the keyboard
Joining lines of a file into a single scalar variable
Regular expressions and character classes

`do-until` loops Pattern matching

Getting User Input from the Keyboard

You first saw filehandles in Example 4-5. In Example 5-3 (as was true in Example 4-3), a filehandle and the angle bracket input operator are used to read in data from an opened file into an array, like so:

```
@protein = <PROTEINFILE>;
```

Perl uses the same syntax to get input that is typed by the user at the keyboard. In Example 5-3, a special filehandle called STDIN (short for standard input), is used for

this purpose, as in this line that collects a filename from the user:

```
$proteinfilename = <STDIN>;
```

So, a filehandle can be associated with a file; it can also be associated with the keyboard where the user types responses to questions the program asks.

If the variable you're using to save the input is a scalar variable starts with a dollar sign $), as in this fragment, only one line is read, which is almost always what you want in this case.

In Example 5-3, the user is requested to enter the filename of a file containing protein sequence data. After getting a filename in this fashion, there's one more step before you can open the file. When the user types in a filename and sends a newline by hitting the Enter key (also known as the Return key), the filename also gets a newline character at the end as it is stored in the variable. This newline is not part of the filename and has to be removed before the open system call will work. The Perl function chomp removes newlines (or its cousins linefeeds and carriage returns) from the end of a string. (The older function chop removes the last character, no matter what it is; this caused trouble, so chomp was introduced and is almost always preferred.)

So this part of Perl requires a little bit extra: removing the newline from the input collected from the user at the keyboard. Try commenting out the `chomp` function, and you'll see that the `open` fails, because no filename has a newline at the end. (Operating systems have rules as to which characters are allowed in filenames.)

Turning Arrays into Scalars with join

It's common to find protein sequence data broken into short segments of 80 or so characters each. The reason is simple: when data is printed out on paper or displayed on the screen, it needs to be broken up into lines that fit into the space. Having your data broken into segments, however, is inconvenient for your Perl program. What if you're searching for a motif that's split by a newline character? Your program won't find it. In fact, some of the motifs searched for in Example 5-3 are split by line breaks. In Perl you deal with this sort of segmented data with the Perl function *join*. In Example 5-3 *join* collapses an array `@protein` by combining all the lines of data into a single string stored in a new scalar variable `$protein`:

```
$protein = join( '', @protein);
```

You specify a string to be placed between the elements of the array as they're joined. In this case, you specify the empty string to be placed between the lines of the input file. The empty string is represented with the pair of single quotes `''` (double quotes `""` also serve).

Recall that in Example 4-2, I introduced several equivalent ways to concatenate two fragments of DNA. The use of the join function is very similar. It takes the scalar values that are the elements of the array and concatenates them into a single scalar value. Recall the following statement from Example 4-2, which is one of the equivalent ways to concatenate two strings:

```
$DNA3 = $DNA1 . $DNA2;
```

Another way to accomplish the same concatenation uses the *join* function: `$DNA3 = join( "", ($DNA1, $DNA2) );`

In this version, instead of giving an array name, I specify a list of scalar elements:

```
($DNA1, $DNA2)
```

do-until Loops

There's a new kind of loop in Example 5-3, the `do-until` loop, which first executes a block and then does a conditional test. Sometimes this is more convenient than the usual order in which you test first, then do the block if the test succeeds. Here, you want to prompt the user, get the user's input, search for the motif, and report the results. Before doing it again, you check the conditional test to see if the user has input an empty line. This means that the user has no more motifs to look for, so you exit the loop.

Regular Expressions

Regular expressions let you easily manipulate strings of all sorts, such as DNA and protein sequence data. What's great about regular expressions is that if there's something you want to do with a string, you usually can do it with Perl regular expressions.

Some regular expressions are very simple. For instance, you can just use the exact text of what you're searching for as a regular expression: if I was looking for the word "bioinformatics" in the text of this book, I could use the regular expression:

```
/bioinformatics/
```

Some regular expressions can be more complex, however. In this section, I'll explain their use in Example 5-3.

Regular expressions and character classes

Regular expressions are ways of matching one or more strings using special wildcard-like operators. Regular expressions can be as simple as a word, which matches the word itself, or they can be complex and made to match a large set of different words (or even every word!).

After you join the protein sequence data into the scalar variable `$protein` in Example 5-3, you also need to remove newlines and anything else that's not sequence data. This can include numbers on the lines, comments, informational or "header" lines, and so on.

In this case, you want to remove newlines and any spaces or tabs that might be invisibly present. The following line of code in Example 5-3 removes this whitespace: `$protein =~ s/\s//g;`

The sequence data in the scalar variable `$protein` is altered by this statement. You first saw the binding operator =~ and the substitute function s/// back in Example 4-3,

where they were used to change one character into another. Here, they're used a little differently. You substitute any one of a set of whitespace characters, represented by `\s` with nothing and by the lack of anything between the second and third forward slashes. In other words, you delete any of a set of whitespace characters, which is done globally throughout the string by virtue of the `g` at the end of the statement.

The \s is one of several metasymbols. You've already seen the metasymbol \n. The \s metasymbol matches any space, tab, newline, carriage return, or formfeed. \s can also be written as:

```
[ \t\n\f\r]
```

This expression is an example of a character class and is enclosed in square brackets. A character class matches one character, any one of the characters named within the square brackets. A space is just typed as a space; other whitespace characters have their own metasymbols: \t for tab, \n for newline, \f for formfeed, and \r for carriage return. A carriage return causes the next character to be written at the beginning of the line, and a formfeed advances to the next line. The two of them together amount to the same thing as a newline character.

Each s/// command I've detailed has some kind of regular expression between the first two forward slashes /. You've seen single letters as the C in s/C/G/g in that position. The C is an example of a valid regular expression.

There's another use of regular expressions in Example 5-3. The line of code:

```
if ( $motif =~ /^\s*$/ ) {
```

is, in English, testing for a blank line in the variable $motif. If the user input is nothing except for perhaps some whitespace, represented as \s*, the match succeeds, and the program exits. The whole regular expression is:

```
/^\s*$/
```

which translates as: match a string that, from the beginning (indicated by the ^), is zero or more (indicated by the *) whitespace characters (indicated by the \s) until the end of the string (indicated by the $).

If this seems somewhat cryptic, just hang in there and you'll soon get familiar with the terminology. Regular expressions are a great way to manipulate sequence and other text-based data, and Perl is particularly good at making regular expressions relatively easy to use, powerful, and flexible. Many of the references in Appendix A contain material on regular expressions, and there's a concise summary in Appendix B.

Pattern matching with =~ and regular expressions

The actual search for the motif happens in this line from Example 5-3: `if ( $protein =~ /$motif/ ) {`

Here, the binding operator =~ searches for the regular expression stored as the value of the variable $motif in the protein $protein. Using this feature, you can interpolate the value of a variable into a string match. (Interpolation in Perl strings means inserting the value of a variable into a string, as you first saw in Example 4-2 when you were concatenating strings). The actual motif, that is, the value of the string variable $motif, is your regular expression. The simplest regular expressions are just strings of characters, such as the motif AQQK, for example.

You can use Example 5-3 to play with some more features of regular expressions. You can type in any regular expression to search for in the protein. Try starting up the program, referring to the documentation on regular expressions, and play! Here are some examples of typing in regular expressions:

Search for an A followed by a D or S, followed by a V:

- `Enter a motif to search for: A[DS]V I couldn't find it.`
- Search for K, N, zero or more D's, and two or more E's (note that `{2,}` means "two or more"):
- `Enter a motif to search for: KND*E{2,} I found it!`

  Search for two E's, followed by anything, followed by another two E's:

- `Enter a motif to search for: EE.*EE I found it!`

In that last search, notice that a period stands for any character except a newline, and ".*" stands for zero or more such characters. (If you want to actually match a period, you have to escape it with a backslash.)

Counting Nucleotides

There are many things you might want to know about a piece of DNA. Is it coding or noncoding?[3] Does it contain a regulatory element? Is it related to some other known DNA, and if so, how? How many of each of the four nucleotides does the DNA contain? In fact, in some species the coding regions have a specific nucleotide bias, so this last question can be important in finding the genes. Also, different species have different

patterns of nucleotide usage. So counting nucleotides can be interesting and useful.

[3] Coding DNA is DNA that codes for a protein, that is, it is part of a gene. In many organisms, including humans, a large part of the DNA is noncoding—not part of genes and doesn't code for proteins. In humans, about 98-99% of DNA is noncoding.

In the following sections are two programs, Examples 5-4 and 5-6, that make a count of each type of nucleotide in some DNA. They introduce a few new parts of Perl:

"Exploding" a string
Looking at specific locations in strings Iterating over an array
Iterating over the length of a string

To get the count of each type of nucleotide in some DNA, you have to look at each base, see what it is, and then keep four counts, one for each nucleotide. We'll do this in two ways:

Explode the DNA into an array of single bases, and iterate over the array (that is, deal with the elements of the array one by one)

Use the substr Perl function to iterate over the positions in the string of DNA while counting

First, let's start with some pseudocode of the task. Afterwards, we'll make more detailed pseudocode, and finally write the Perl program for both approaches.

The following pseudocode describes generally what is needed:

```
for each base in the DNA
    if base is A
        count_of_A = count_of_A + 1
    if base is C
        count_of_C = count_of_C + 1
    if base is G
        count_of_G = count_of_G + 1
    if base is T
        count_of_T = count_of_T + 1
done

print count_of_A, count_of_C, count_of_G, count_of_T
```

As you can see, this is a pretty simple idea, mirroring what you'd do by hand if you had to. (If you want to count the relative frequencies of the bases in all human genes, you can't do it by hand—there are too many of them—and you have to use such a program. Thus bioinformatics.) Now let's see how it can be coded in Perl.

Exploding Strings into Arrays

Let's say you decide to explode the string of DNA into an array. By *explode* I mean separating out each letter in the string—sort of like blowing the string into bits. In other words, the letters representing the bases of the DNA in the string are separated, and each letter becomes its own scalar value in an array. Then you can look at the array elements (each of which is a single character) one by one, making the count as you go along. This is the inverse of the `join` function in Section 5.3.2, which takes an array of strings and makes a single scalar value out of them. (After exploding a string into an array, you could then join the array back into an identical string using `join`, if you so desire.)

I'm also adding to this version of the pseudocode the instructions to get the DNA from a file and manipulate that file data until it's a single string of DNA sequence. So first, you join the data from the array of lines of the original file data, clean it up by removing whitespace until only sequence is left, and then explode it back into an array. But, of course, the point is that the last array has exactly what is needed, the data in a convenient form to use in the counting loop. Instead of an array of lines, with newlines and possibly other unwanted characters, there's an exact array of the individual bases.

```
read in the DNA from a file
join the lines of the file into a single string $DNA
# make an array out of the bases of $DNA
@DNA = explode $DNA
# initialize the counts
count_of_A = 0
count_of_C = 0
count_of_G = 0
```

```
count_of_T = 0
for each base in @DNA
    if base is A
        count_of_A = count_of_A + 1
    if base is C
        count_of_C = count_of_C + 1
    if base is G
        count_of_G = count_of_G + 1
    if base is T
        count_of_T = count_of_T + 1

done
print count_of_A, count_of_C, count_of_G, count_of_T
```

As promised, this version of the pseudocode is a bit more detailed. It suggests a method

to look at each of the bases by exploding the string of DNA into an array of single characters. It also initializes the counts to zero to ensure they start off right. It's easier to see what's happening if you spell out the initialization in the program, and it can prevent certain kinds of errors from creeping into your code. (It's not a rule, however; sometimes, you may prefer to leave the values of variables undefined until they are used.) Perl assumes that an uninitialized variable has the value 0 if you try to use it as a number, for instance by adding another number to it. But you'll most likely get a warning if that is the case.

We now have a design for the program, let's turn it into Perl code. Example 5-4 is a workable program; you'll see other ways to accomplish the same task more quickly as you proceed in this chapter, but speed is not the main concern at this point.

Example 5-4. Determining frequency of nucleotides

```
#!/usr/bin/perl -w
# Determining frequency of nucleotides

# Get the name of the file with the DNA sequence data
print "Please type the filename of the DNA sequence data: ";

$dna_filename = <STDIN>;
# Remove the newline from the DNA filename
chomp $dna_filename;
# open the file, or exit
unless ( open(DNAFILE, $dna_filename) ) {
    print "Cannot open file \"$dna_filename\"\n\n";

exit; }

# Read the DNA sequence data from the file, and store it
# into the array variable @DNA
@DNA = <DNAFILE>;
# Close the file
close DNAFILE;
# From the lines of the DNA file,
# put the DNA sequence data into a single string.
$DNA = join( '', @DNA);
# Remove whitespace
$DNA =~ s/\s//g;
# Now explode the DNA into an array where each letter of
the
# original string is now an element in the array.
# This will make it easy to look at each position.
# Notice that we're reusing the variable @DNA for this
purpose.
@DNA = split( '', $DNA );

# Initialize the counts.
# Notice that we can use scalar variables to hold numbers.
$count_of_A = 0;
$count_of_C = 0;
$count_of_G = 0;
$count_of_T = 0;
$errors =0;

# In a loop, look at each base in turn, determine which of
the
# four types of nucleotides it is, and increment the
# appropriate count.
foreach $base (@DNA) {

if ($baseeq'A'){ ++$count_of_A;

  } elsif ( $base eq 'C' ) {
    ++$count_of_C;
  } elsif ( $base eq 'G' ) {
    ++$count_of_G;
  } elsif ( $base eq 'T' ) {
    ++$count_of_T;
  } else {
    print "!!!!!!!! Error - I don\'t recognize this
base: $base\n";

++$errors; }

}

# print the results
print "A = $count_of_A\n";
print "C = $count_of_C\n";
print "G = $count_of_G\n";
print "T = $count_of_T\n";
print "errors = $errors\n";
```

```
# exit the program
exit;
```

To demonstrate Example 5-4, I have created the following small file of DNA and

called it *small.dna*: `AAAAAAAAAAAAAAGGGGGGGTTTTCCCCCCCC`

```
CCCCCGTCGTAGTAAAGTATGCAGTAGCVG
CCCCCCCCCCGGGGGGGGAAAAAAAAAAAAAAATTTTTTAT AAACG
```

The file small.dna can be typed into your computer using your favorite text editor, or you can download it from this book's web site.

Notice that there is a V in the file, an error.[4] Here is the output of Example 5-4:

[4] Files of DNA sequence data sometimes include such characters as N, meaning "some undetermined base," or other special characters. You sometimes have to look at the documentation for the source, say an ABI sequencer or a GenBank file or whatever, to discover which characters are used and what they mean.

```
Please type the filename of the DNA sequence data:
small.dna
!!!!!!!! Error - I don't recognize this base: V
A = 40
C = 27
G = 24
T = 17
```

Now let's look at the new stuff in this program. Opening and reading the sequence data is the same as previous programs. The first new thing is at this line:

```
@DNA = split( '', $DNA);
```

which the comments say will explode the string `$DNA` into an array of single characters `@DNA`.

split is the companion to *join*, and it's a good idea to take a little while to look over the documentation for these two commands. Calling *split* with an empty string as the first argument causes the string to explode into individual characters; that's just what we want.[5]

[5] As you'll see in the documentation for the *split* function, the first argument can be any regular expression, such as `/\s+/` (one or more adjacent whitespace characters.)

Next, there are five scalar variables initialized to `0`, the variables `$count_of_A` and so forth. *I* nitializing means assigning an initial value, in this case, the value `0`.

Example 5-4 illustrates the concepts of type and initialization. The type of a variable determines what kind of data it can hold, for instance, strings or numbers. Up to now we've been using scalar variables such as `$DNA` to store strings of letters such as A, C, G, and T. Example 5-4 shows that you can also use scalar variables to store numbers. For example, the variable `$count_of_A` keeps a running count of the character A.

Scalar variables can store integers (0, 1, -1, 2, -2, ...), decimal or floating-point numbers such as 6.544, and numbers in scientific notation such as 6.544E6, which translates as 6.544 x 106, or 6,544000. (See Appendix B for more details on types of numbers.)

In Example 5-4, the variables `$count_of_A` through `$count_of_T` are initialized to 0. Initializing a variable means giving it a value after it's declared. If you don't initialize your variables, they assume the value of `'undef'`. In Perl, an undefined variable is 0 if it is asked for in numerical context; it's an empty string if used in a string operation. Although Perl programmers often choose not to initialize variables, it's a critical step in many other languages. In C for instance, uninitialized variables have unpredictable values. This can wreak havoc with your output. You should get in the habit of initializing variables; it makes the program easier to read and maintain, and that's important.

To declare a variable means to specify its name and other attributes such as an initial value and a scope (for scoping, see Chapter 6 and the discussion of `my` variables).

Many languages require you to declare all variables before using them. For this book, up to now, declarations have been an unnecessary complication. The next chapter begins to require declarations. In Perl, you may declare a variable's scope (see Chapter 6 and the discussion of `my` variables) in addition to an initial value. Many languages also require you to declare the type of a variable, for example "integer," or "string," but Perl does not.

Perl is written to be smart about what's in a scalar variable. For instance, you can assign the number `1234` (without quotes) to a variable, or you can assign the string `'1234'` (with quotes). Perl treats the variable as a string for printing, and as a number for using in arithmetic operations, without your having to worry about it. Example 5-5 demonstrates this ability. In other words, Perl isn't strict about specifying the type of data a variable is used for.

Example 5-5. Demonstration of Perl's built-in knowledge about numbers and strings

```
#!/usr/bin/perl -w
# Demonstration of Perl's built-in knowledge about numbers
and strings
$num = 1234;
$str = '1234';
# print the variables
print $num, " ", $str, "\n";
# add the variables as numbers
$num_or_str = $num + $str;
print $num_or_str, "\n";
# concatenate the variables as strings
$num_or_str = $num . $str;
print $num_or_str, "\n";
exit;
```

Example 5-5 produces the output: `1234 1234`

```
2468
12341234
```

Example 5-5 illustrates the smart way Perl determines the datatype of a scalar variable, whether it's a string or a number, and whether you're trying to add or subtract it like a number or concatenate it like a string. Perl behaves accordingly, which makes your job as a programmer a little bit easier; Perl "does the right thing" for you.

Next is a new kind of loop, the `foreach` loop. This loop works over the elements of an array. The line:

```
foreach $base (@DNA) {
```

loops over the elements of the array `@DNA`, and each time through the loop, the scalar

variable `$base` (or whatever name you choose) is set to the next element of the array.

The body of the loop checks for each base and increments the count for that base if found. There are four ways to add `1` to a number in Perl. Here, you put a `++` in front of the variable, like this:

```
++$count;
```

You can also put the `++` after the variable: `$count++;`

You can spell it out like this, a combination of adding and assignment:

```
$count = $count + 1;
```

or, as a shorthand of that, you can say:

```
$count += 1;
```

Almost an embarrassment of riches. The plus-plus (`++`) notation is convenient for incrementing counts, as we're doing here. The plus-equals (`+=`) notation saves some typing and is very popular for adding other numbers besides 1.

The `foreach` loop in Example 5-5 could have been written like this:

```
foreach (@DNA) {

if (/A/){ ++$count_of_A;

    } elsif ( /C/ ) {
        ++$count_of_C;
    } elsif ( /G/ ) {
        ++$count_of_G;
    } elsif ( /T/ ) {
        ++$count_of_T;
    } else {
        print "!!!!!!!! Error - I don\'t recognize this
base: ";
        print;

print "\n";
++$errors; }

}
```

This version of the `foreach` loop: `foreach(@DNA) {.`

doesn't have a scalar value. In a `foreach` loop, if you don't specify a scalar variable to hold the scalars that are being read from the array (`$base` served that function in the version of this loop in Example 5-5), Perl uses the special variable `$_` .

Furthermore, many Perl built-in functions operate on this special variable if no argument is provided to them. Here, the conditional tests are simply patterns; Perl assumes you're doing a pattern match on the `$_` variable, so it behaves as if you had said `$_ =~ /A/`, for instance. Finally, in the error message, the statement `print;` prints the value of the `$_` variable.

This special variable `$_` that doesn't have to be named appears in many Perl programs, although I don't use it extensively in this book.

Operating on Strings

It's not necessary to explode a string into an array in order to look at each character. In fact, sometimes you'd want to avoid that. A large string takes up a large amount of memory in your computer. So does a large array. When you explode a string into an array, the original string is still there, and you also have to make a copy of each character for the elements of the new array you're creating. If you have a large string, that already uses a good portion of available memory, creating an additional array can cause you to run out of memory. When you run out of memory, your computer performs poorly; it can slow to a crawl, crash, or freeze ("hang"). These haven't been worrisome considerations up to now, but if you use large data sets (such as the human genome), you have to take these things into account.

So let's say you'd like to avoid making a copy of the DNA sequence data into another variable. Is there a way to just look at the string `$DNA` and count the bases from it

directly? Yes. Here's some pseudocode, followed by a Perl program:

```
read in the DNA from a file
join the lines of the file into a single string of $DNA
# initialize the counts
count_of_A = 0
count_of_C = 0
count_of_G = 0
count_of_T = 0
for each base at each position in $DNA
    if base is A
        count_of_A = count_of_A + 1
    if base is C
        count_of_C = count_of_C + 1
    if base is G
        count_of_G = count_of_G + 1
    if base is T
```

```
        count_of_T = count_of_T + 1
done
print count_of_A, count_of_C, count_of_G, count_of_T
```

Example 5-6 shows a program that examines each base in a string of DNA.

Determining frequency of nucleotides, take 2

```
#!/usr/bin/perl -w
# Determining frequency of nucleotides, take 2

# Get the DNA sequence data
print "Please type the filename of the DNA sequence data: ";

$dna_filename = <STDIN>;
chomp $dna_filename;
# Does the file exist?
unless ( -e $dna_filename) {
    print "File \"$dna_filename\" doesn\'t seem to
exist!!\n";

exit; }

# Can we open the file?
unless ( open(DNAFILE, $dna_filename) ) {
    print "Cannot open file \"$dna_filename\"\n\n";

exit; }

@DNA = <DNAFILE>;
close DNAFILE;
$DNA = join( '', @DNA);
# Remove whitespace
$DNA =~ s/\s//g;

# Initialize the counts.
# Notice that we can use scalar variables to hold numbers.
$count_of_A = 0;
$count_of_C = 0;
$count_of_G = 0;
$count_of_T = 0;
$errors =0;

# In a loop, look at each base in turn, determine which of
the
# four types of nucleotides it is, and increment the
# appropriate count.
for ( $position = 0 ; $position < length $DNA ;
++$position ) {
    $base = substr($DNA, $position, 1);
if ($baseeq'A'){ ++$count_of_A;

    } elsif ( $base eq 'C' ) {
        ++$count_of_C;
    } elsif ( $base eq 'G' ) {
        ++$count_of_G;
    } elsif ( $base eq 'T' ) {
        ++$count_of_T;
    } else {
        print "!!!!!!!! Error - I don\'t recognize this
base: $base\n";
        ++$errors;

}

}

# print the results
print "A = $count_of_A\n";
print "C = $count_of_C\n";
print "G = $count_of_G\n";
print "T = $count_of_T\n";
print "errors = $errors\n";
# exit the program

exit;
```

Here's the output of Example 5-6:

```
Please type the filename of the DNA sequence data:
small.dna
!!!!!!!! Error - I don't recognize this vase: V
A = 40
C = 27
G = 24
T = 17
errors = 1
```

In Example 5-6, I added a line of code to see if the file exists:

```
unless ( -e $dna_filename) {
```

There are file test operators for several conditions; see Appendix B or Perl documentation under *-X*. Note that files have several attributes, such as size, permission, location in the filesystem, and type of file, and that many of these things can be tested for easily with the file test operators.

Notice, also, that I have kept the detailed comments about the regular expression, because regular expressions can be hard to read, and a little commenting here helps a reader to skim the code.

Everything else is familiar, until you hit the `for` loop; it requires a little explanation: `for ( $position = 0 ; $position < length $DNA ; ++$position ) {`

```
    # the statements in the block
}
```

This *for* loop is the equivalent of this *while* loop: $position = 0;

```
while( $position < length $DNA ) {
    # the same statements in the block, plus ...
    ++$position;
}
```

Take a moment and compare these two loops. You'll see the same statements but in different locations.

As you can see, the for loop brings the initialization and increment of a counter ($position) into the loop statement, whereas in the while loop, they are separate statements. In the for loop, both the initialization and the increment statement are placed between parentheses, whereas you find only the conditional test in the while loop. In the for loop, you can put initializations before the first semicolon and increment statements after the second semicolon. The initialization statement is done just once before starting the loop, and the increment statement is done at the end of each iteration through the block before going back to the conditional test. It's really just a shorthand for the equivalent while loop as just shown.

The conditional test checks to see if the position reached in the string is less than the length of the string. It uses the length Perl function. Obviously, you don't want to check characters beyond the length of the string. But a word is in order here about the numbering of positions in strings and arrays.

By default, Perl assumes that a string begins at position 0 and its last character is at a position that's numbered one less than the length of the string. Why do it this way instead of numbering the positions from 1 up to and including the length of the string? There are reasons, but they're somewhat abstruse; see the documentation for enlightenment. If it's any comfort, many other programming languages make the same choice. (However, many do it the intuitive way, starting at 1. Ah well.)

This way of numbering is important to biologists because they are used to numbering sequences beginning with 1, not with 0 the way Perl does it. You sometimes have to add 1 to a position before printing out results so they'll make sense to nonprogrammers. It's mildly annoying, but you'll get used to it.

The same holds true for numbering the elements of an array. The first element of an array is element 0; the last is element $length-1.

Anyway, you see that the conditional test evaluates to true while the value of $position is length-1 or less and fails when $position reaches the same value as the length of the string. For example, say you have a string that contains the text "seeing." This has a length of six characters. The "s" is at position 0, and the "g" is at position 5, which is one less than the string length 6.

Back in the block, you call the *substr* function to look into the string:

```
$base = substr($DNA, $position, 1);
```

This is a fairly general-purpose function for working with strings; you can also insert and delete things. Here, you look at just one character, so you call *substr* on the string `$DNA`, ask it to look in position `$position` for one character, and save the result in scalar variable `$base`. Then you proceed to accumulate the count as in the preceding version of the program, Example 5-4.

Writing to Files

Example 5-7 shows one more way to count nucleotides in a string of DNA. It uses a Perl trick that was designed with exactly this kind of job in mind. It puts a global regular expression search in the test for a *while* loop, and as you'll see, it's a compact way of counting characters in a string.

One of the nice things about Perl is that if you need to do something fairly regularly, the language has probably got a relatively succinct way to do it. (The downside of this is that Perl has a lot of things about it to learn.)

The results of Example 5-7, besides being printed to the screen, will also be written to a file. The code that accomplishes this writing to a file is as follows:

```
# Also write the results to a file called "countbase"

$outputfile = "countbase";
(
unless ( open(COUNTBASE, ">$outputfile") ) {
    print "Cannot open file \"$outputfile\" to write
to!!\n\n";

exit; }

print COUNTBASE "A=$a C=$c G=$g T=$t errors=$e\n";
close(COUNTBASE);
```

As you see, to write to a file, you do an *open* call, just as when reading from a file, but with a difference: you prepend a greater-than sign > to the filename. The filehandle becomes a first argument to a `print` statement (but without a comma following it). This makes the `print` statement direct its output into the file.[6]

[6] In this case, if the file already exists, it's emptied out first. It's possible to specify several other behaviors. As mentioned earlier, the Perl documentation has all the details of the *open* function, which sets the options for reading from, and writing to, files as well as other actions.

Example 5-7 is the third version of the Perl program that examines each base in a string of DNA.

Example 5-7. Determining frequency of nucleotides, take 3

```
#!/usr/bin/perl -w
# Determining frequency of nucleotides, take 3

# Get the DNA sequence data
print "Please type the filename of the DNA sequence data: ";

$dna_filename = <STDIN>;
chomp $dna_filename;
# Does the file exist?
unless ( -e $dna_filename) {
    print "File \"$dna_filename\" doesn\'t seem to
exist!!\n";

exit; }

# Can we open the file?
unless ( open(DNAFILE, $dna_filename) ) {
    print "Cannot open file \"$dna_filename\"\n\n";

exit; }

@DNA = <DNAFILE>;
close DNAFILE;
$DNA = join( '', @DNA);
# Remove whitespace
$DNA =~ s/\s//g;
# Initialize the counts.
# Notice that we can use scalar variables to hold numbers.
$a = 0; $c = 0; $g = 0; $t = 0; $e = 0;
# Use a regular expression "trick", and five while loops,
#  to find the counts of the four bases plus errors
while($DNA =~ /a/ig){$a++}
while($DNA =~ /c/ig){$c++}
while($DNA =~ /g/ig){$g++}
while($DNA =~ /t/ig){$t++}
while($DNA =~ /[^acgt]/ig){$e++}
print "A=$a C=$c G=$g T=$t errors=$e\n";
# Also write the results to a file called "countbase"
$outputfile = "countbase";
unless ( open(COUNTBASE, ">$outputfile") ) {
print "Cannot open file \"$outputfile\" to write
to!!\n\n";

exit; }

print COUNTBASE "A=$a C=$c G=$g T=$t errors=$e\n";
close(COUNTBASE);
# exit the program

exit;
```

Example 5-7 looks like this when you run it:

```
Please type the filename of the DNA sequence data: small.dna
A=40 C=27 G=24 T=17 errors=1
```

The output file *countbase* has the following contents after you run Example 5-7: `A=40 C=27 G=24 T=17 errors=1`

The *while* loop:

```
while($dna =~ /a/ig){$a++}
```

has as its conditional test, within the parentheses, a string-matching expression:

```
$dna =~ /a/ig
```

This expression is looking for the regular expression `/a/`, that is, the letter `a`. Since it has the `i` modifier, it's a case-insensitive match, which means it matches `a` or `A`. It also has the global modifier, which means match all the `a`'s in the string. (Without the global modifier, it just keeps returning `true` every time through the loop, if there is an "a" in `$dna`.)

Now, this string-matching expression, in the context of a while loop, causes the while loop to execute its block on every match of the regular expression. So, append the one- statement block:

```
{$a++}
```

to increment the counter at each match of the regular expression; in other words, you're counting all the `a`'s.

One other point should be made about this third version of the program. You'll notice some of the statements have been changed and shortened this time around. Some variables have shorter names, some statements are lined up on one line, and the print statement at the end is more concise. These are just alternative ways of writing. As you program, you'll find yourself experimenting with different approaches: try some on for size.

The way to count bases in this third version is flexible; for instance, it

allows you to count non-ACGT characters without specifying them individually. In later chapters, you'll use those `while` loops to good effect. However, there's an even faster way to count bases. You can use the *tr* transliteration function from Chapter 4; it's faster, which is helpful if you have a lot of DNA to count:

```
$a = ($dna =~ tr/Aa//);
$c = ($dna =~ tr/Cc//);
$g = ($dna =~ tr/Gg//);
$t = ($dna =~ tr/Tt//);
```

The *tr* function returns the count of the specified characters it finds in the string, and if the set of replacement characters is empty, it doesn't actually change the string. So it makes a good character counter. Notice that with *tr*, you have to spell out the upper- and lowercase letters. Also, because *tr* doesn't accept character classes, there's no direct way to count nonbases. You could, however, say:

```
$basecount = ($dna = ~ tr/ACGTacgt//);
$ nonbase = (length $dna) - $basecount)
```

The program however, runs faster using tr than using the while loops of Example 5-7.

You may find it a bit much to have three (really, four) versions of this base-counting program, especially since much of the code in each version is identical. The only part of the program that really changed was the part that did the counting of the bases. Wouldn't it have been convenient to have a way to just alter the part that counts the bases? In Chapter 6, you'll see how subroutines allow you to partition your programs in just such a way.

Subroutines and Bugs

In this chapter you'll extend your basic knowledge in two directions: Subroutines
Using the Perl debugger

Subroutines are an important way to structure programs. You'll use them in Chapter 7, where you'll learn how to use randomization to simulate the mutation of DNA. The Perl debugger examines a program's behavior in "slow motion" and helps you find those pesky bugs.

Subroutines

Subroutines are an important way to organize a program and are used in all major programming languages.

A subroutine wraps up a bit of code, gives the code a name, and provides a way to pass in some values for its calculations and then report back the results. The rest of the program can then use the subroutine's code just by calling its name, giving the needed values to pass in to the subroutine code and then collecting the results. This use or "invocation" of a subroutine is commonly referred to as *calling* the subroutine. You can think of a subroutine as a program within a program; just as you run programs to get results, so your programs call subroutines to get results. Once you have a subroutine, you can use it in a program simply by knowing which values to pass in and what kind of values to expect it to pass out.

opportunities for something to go wrong.

Faster to write, since you may, for example, have already written some subroutines that handle basic statistics and can just call the one that calculates the mean without having to write it again. Or better yet, you found a good statistics library someone else wrote, and you never had to write it at all.

There is another subtle, yet powerful idea at work here. Subroutines can themselves call other subroutines, that is, a subroutine can use another subroutine for help in its calculations.[1] By writing a set of subroutines, each of which does one or a few things well, you can combine them in various ways to make new subroutines. You can then combine the new subroutines, and so on, and the end result can be large and flexible programming systems. Decomposing problems into sets of subroutines that can be conveniently combined allows you to create environments that can grow and adapt to changing conditions with a minimum of effort.

[1] Subroutines can even call themselves, and this so-called recursion can be an elegant way to compute (see Chapter 11).

The trick of all this is in how you partition the code into subroutines. You want subroutines that encapsulate something that will be generally useful, and not just called once (although that sometimes can be useful too). There are various rules of thumb: a subroutine should do one thing well, and it should be no more than a page or two of code. These are not real rules, and exceptions are frequent, but they can help you divide your code into manageable chunks, suitable for subroutines.

Writing Subroutines

Let's look at how subroutines are used and then at how they're defined.

To use a subroutine, you pass data into the subroutine as arguments, and then you collect the return value(s) of the subroutine. For example, say you want a subroutine that, given some DNA, appends "ACGT" to the end of the DNA and returns the new, longer DNA. Let's call the subroutine *addACGT*. In Perl, you usually call a subroutine by typing its name, followed by a parenthesized list of arguments (if any). For example, here's a call to addACGT with the one argument $dna:

```
addACGT($dna);
```

When calling a subroutine, older versions of Perl required starting the name of a subroutine with the & (ampersand) character. It's still okay to do so (e.g., : *&addACGT)*, but these days the ampersand is usually omitted.[2]

[2] There are times, even in the newer versions of Perl, when an ampersand is required; you'll see one such case in Chapter 11, in Section 11.2.3, which describes the *File::Find* module. (See also the *defined* and *undef* functions in the documentation or the *perlref* manpage).

Example 6-1 demonstrates a subroutine that shows in detail how this works. **Example 6-1. A subroutine to append ACGT to DNA**

```
#!/usr/bin/perl -w
# A program with a subroutine to append ACGT to DNA
# The original DNA
$dna = 'CGACGTCTTCTCAGGCGA';
# The call to the subroutine "addACGT".
# The argument being passed in is $dna; the result is saved
in $longer_dna
$longer_dna = addACGT($dna);
print "I added ACGT to $dna and got $longer_dna\n\n";
exit;
###########################################################
#####################
# Subroutines for Example 6-1
###########################################################
#####################
# Here is the definition for subroutine "addACGT"
sub addACGT {
    my($dna) = @_;
    $dna .= 'ACGT';
    return $dna;
}
```

Example 6-1 produces the following output:

```
I added ACGT to CGACGTCTTCTCAGGCGA and got
CGACGTCTTCTCAGGCGAACGT
```

We'll now look at this code to see how subroutines are defined and used in a Perl program.

The first thing to notice, taking the large view, is that the program now has two sections. The first section starts from the beginning of the program and ends with the `exit` command. Following that (and announced by a blizzard of comments for easy reading) is a section for subroutine definitions, in this case, only the one definition for subroutine addACGT. It is common to place all subroutine definitions together at the end of a program, for ease in reading. Usually they're listed alphabetically or in some other convenient way.

Actually, it is legal to put the subroutine definitions almost anywhere in a program. This is because Perl first scans through the code and does things like check the syntax and learn subroutine definitions, before it starts to run the program. In particular, subroutine definitions can come after the point in the code where you use them (not necessarily before, which many people assume is the rule), and they don't have to be grouped together but can be scattered throughout the code. But our method of collecting them together at the end can make reading a program much easier. The possible exception is when a small subroutine is used in one section of code, as sometimes happens with the sort function, for instance. In this case having the definition right there can save the reader paging back and forth between the subroutine definition and its use. Usually, it's more convenient to read the program without the subroutine definitions, to get the overall flow of the program first, and then go back and look into the subroutines, if necessary.

As you see, Example 6-1 is very simple. It first stores some DNA into the variable `$dna` and then passes that variable as an argument to the subroutine call, which looks like this: `addACGT($dna)`. The subroutine is called by its name, followed by parentheses containing the arguments to the subroutine. There may be no arguments, or if more than one, they are separated by commas. The value returned by the subroutine can be saved; in this program the value is saved in a variable called `$longer_dna`, which is then printed, and the program exits.

The part of the program from the beginning to the exit statement is called variously the main program or the main body of the program. By looking over this section of the code, you can see what happens from the beginning to the end of the program without looking into the details of the subroutines.

Now that you've looked over the main program of Example 6-1, it's time to look at the subroutine definition and how it uses the principal of scoping.

Scoping and Subroutines

A subroutine is defined by the *reserved word* [3] for subroutine definitions, `sub`; the subroutine's name, in this case, *addACGT*; and a *block*, enclosed in a pair of matching curly braces. This is the same kind of block seen earlier in loops and conditional statements that groups statements together.

[3] A reserved word is a fundamental, defined word in the Perl language, such as `if`, `while`, `foreach`, or `sub`.

In Example 6-1, the name of the subroutine is `addACGT`, and the block is everything after the name. Here is the subroutine definition again:

```
sub addACGT {

    my($dna) = @_;
    $dna .= 'ACGT';
    return $dna;
}
```

Now let's look into the block of the subroutine.

A subroutine is like a separate helper program for the main program, and it needs to have its own variables. You will use two types of variables in your subroutines in this book:[4]

[4] In the subroutines in this book, we won't use global variables, which can be seen by both the main program and the subroutines; nor will we use variables declared with `local`, which provides a different kind of scoping restriction than `my`.

Arguments passed in to the subroutine

Other variables declared with `my` and restricted to the scope of the subroutine

Arguments are the values given to a subroutine when it is used, or called. The values of the arguments are passed into the subroutine by means of the special variable `@_`, as you'll see in the next section.

Other variables a subroutine might use must be protected from interacting with variables in other parts of the program, so they have effect only within the subroutine's own scope. This is accomplished by declaring them as `my` variables, as will be explained shortly.

Finally, most subroutines return their results via the *return* function. This can return a single scalar as in `return $dna;` in our subroutine *addACGT*, in a list of scalars as in `return ($dna1, $dna2);`, in an array as in `return @lines;`, and more.

Arguments

To *call* a subroutine means to type its name and give it appropriate arguments and, usually, collect its results. *Arguments* , sometimes called *parameters*, usually contain the data that the subroutine computes on. In Example 6-1, this is the call of the subroutine *addACGT* with the argument `$dna`:

```
$longer_dna = addACGT($dna);
```

The essential point is that whenever you, the programmer, want to use a subroutine, you can call it with whatever argument(s) it is designed to accept and with which you need to compute (in this case, whatever DNA that needs `ACGT` appended to it) and the value of each argument appears in the subroutine in the `@_` array.

When you call a subroutine with certain arguments, the names of the arguments you provide in the call are not important inside the subroutine. Only the values of those arguments that are actually passed inside the subroutine are important. The subroutine typically collects the values from the `@_` array and assigns them to new variables that may or may not have the same names as the variables with which you called the subroutine. The only thing preserved is the order of the values, not the names of the variables containing the values.

Here's how it works. The first line in the subroutine's block is:

```
my($dna) = @_;
```

The values of the arguments from the call of the subroutine are passed into the subroutine in the special array variable `@_`. You know it's an array because it starts with the `@` character. It has the brief name "_", and it's a special array variable that comes predefined in Perl programs. (It's not a name you should pick for your own arrays.) The array `@_` contains all the scalar values passed into the subroutine. These scalar values are the values of the arguments to the subroutine. In this case, there is one scalar value: the string of DNA that's the value of the variable `$dna` passed in as an argument.

If the subroutine has more arguments—for instance one argument for DNA, one for the associated protein, and one for the name of the gene—they are all passed in and assigned to `my` variables inside the subroutine:

```
my($dna,$protein,$name_of_gene) = @_;
```

If there are no arguments, just omit that statement in the subroutine.

After the statement:

```
my($dna) = @_;
```

executes in the subroutine, the passed-in value is assigned to the subroutine's variable `$dna`. The next section explains why this is a new variable specific to the subroutine. The subroutine's variable can be called anything; it certainly doesn't have to be the same name as the argument, as it happens to be in this example. What's cool about scoping is that it doesn't matter if it is or not.

Beware the common mistake of forgetting the `@_` array when naming your arguments in a subroutine, that is, using the statement:

```
my($dna);
```

instead of:

```
my($dna) = @_;
```

If you make this mistake, the values of the arguments won't appear in your subroutine, even though their names are declared.

Scoping

By keeping all variables a subroutine uses active only within the subroutine, you can make it safe to call the subroutines from anywhere. You make the variables specific only to the subroutine by declaring them as `my`variables. `my` is a keyword defined in Perl that limits variables to the block in which they are used (in this case, the block is the subroutine).[5]

[5] There are different models of scoping; `my` implements a type called *lexical scoping*, also known as *static scoping*. Another method is available in Perl via the *local* construct, but you almost always want to use `my`.

Hiding variables and making them local to only a restricted part of a program, is called scoping. In Perl, using `my` variables is known as lexical scoping, and it's an essential part of modularizing your programs.

You declare that a variable is a `my`variable like this: `my($x);`
or:

```
my $x ;
```

or, combining the declaration with an initialization to a value:

```
my($x) = '49';
```

or, if you're collecting an argument within a subroutine:

```
my($x) = @_;
```

Once a variable is declared in this fashion, it exists only until the end of the block it was declared in. So in a subroutine, if you declare all your variables like this (both the arguments and any other variables), they are active only in the subroutine. If any variable has the same name as another variable elsewhere in the program, you don't have to worry, because the `my` declaration actually creates a new variable, active only in the enclosing block, and any other variable of the same name used elsewhere outside the block is kept separate.

The example that showed collecting an argument in a subroutine uses parentheses around the variable. Because `@_` is an array, the parentheses around the new variables put them in array context and ensure that they are initialized correctly.

Always declare all your variables in your subroutines—even those variables that don't come in as arguments—such as the `my` construct.

Why use scoping? Example 6-2 shows the trouble that can happen when you don't. Recall that one of the advantages of subroutines is writing a useful bit of code once and then using it whenever you need it. Example 6-2 is a program that has a variable in the main program with the same name as a variable in a subroutine it calls. This can easily happen if you write the subroutine at a time other than the main program (say six months later) or if you call a subroutine someone else wrote.

The pitfalls of not using my variables

```
#!/usr/bin/perl -w
# Illustrating the pitfalls of not using my variables
$dna = 'AAAAA';
$result = A_to_T($dna);
print "I changed all the A's in $dna to T's and got
$result\n\n";

exit;

############################################################
#####################
# Subroutines
############################################################
#####################
sub A_to_T {
    my($input) = @_;
    $dna = $input;
    $dna =~ s/A/T/g;
    return $dna;
}
```

Example 6-2 gives the following output:

```
I changed all the A's in TTTTT to T's and got TTTTT
```

What was expected was this output:

```
I changed all the A's in AAAAA to T's and got TTTTT
```

You can get by this expected output by changing the definition of subroutine *A_to_T* to the following, in which the variable `$dna` in the subroutine is declared as a `my`variable: `sub A_to_T {`

```
    my($input) = @_;
    my($dna) = $input;
    $dna =~ s/A/T/g;
    return $dna;

}
```

Where exactly did Example 6-2 go wrong? When the program entered the subroutine, and used the variable `$dna` to calculate the string with A's changed to T's, the Perl language saw that there was already a variable `$dna` being used in the main part of the

program and just kept using it. When the program returned from the subroutine and got to the `print` statement, it was still using the same (the one and only) variable `$dna`. So, when it printed the results, the variable `$dna`, instead of having the original DNA in it, had the altered DNA that had been computed in the subroutine.

Now this sort of thing can happen a lot. Programmers tend to use certain names for variables a great deal: the usual suspects are names such as `$tmp`, `$temp`, `$x`, `$a`, `$number`, `$variable`, `$var`, `$array`, `$input`, `$output`, `$result`, `$data`, `$file`, `$filename`, and so on. Bioinformaticians are quite fond of `$dna`, `$protein`, `$motif`, `$sequence`, and the like. As you start using libraries of subroutines from other people and as your programs get larger, it's much easier—and a whole lot safer—to let the Perl language worry about avoiding the problem of name collisions.

In fact, from now on we're going to stop using undeclared variables. From this point forward, all our variables, even those in the main program, will be declared with `my`. You can enforce this discipline by adding the following directive to your programs:

```
use strict;
```

which has the effect of insisting that your programs have all their variables declared as `my` variables.

Lest you rail at this seemingly unnecessary complication to your coding, compared to the simpler and happier days of Chapter 4 and Chapter 5, you should know that many languages require declarations for all their variables. The fact that in Perl you don't have to enforce strict scoping is handy when you're writing short programs, for example, or when you're trying to teach programming without hitting the students with a thousand details at the beginning.

Another benefit you get from strict scoping happens if you accidently misspell a variable name while writing a program. If the variables aren't being declared, Perl creates a new variable with the (misspelled) name. The program may not work correctly, and it may be hard to find where the problem is. By strictly scoping the program, any misspelled variables are also undeclared, and Perl complains about it, saving you hours or days of hair-pulling and bad language.

Finally, let's recap how scoping, arguments, and subroutines work by taking another look at Example 6-1. The subroutine is called by writing its name *addACGT*, passing it

the argument `$dna`, and collecting results (if any) by assignment to `$longer_dna`:

```
$longer_dna = addACGT($dna);
```

The first line in the subroutine gets the value of the argument from the special variable `@_`, and stores it in its own variable called `$dna`, which can't be seen outside the subroutine because it uses `my`. Even though the original variable outside the subroutine is also called `$dna`, the variable called `$dna` within the subroutine is an entirely new variable (with the same name) that belongs only to the subroutine due to the use of `my`.

This new variable is in effect only during the time the program is in the subroutine. Notice in the output from the `print` statement at the end of Example 6-2 that even though a variable called $dna is lengthened inside the subroutine, the original variable, $dna, outside the subroutine isn't changed.

Command-Line Arguments and Arrays

Example 6-3 is another program that uses subroutines. You use the command line to give the program information it needs (such as filenames, or strings of DNA) without having to interactively answer the program's prompts. This is useful if you're scheduling a program to run at a time when you won't be there, for instance.

Example 6-3 also shows a little more about using arrays. You'll see how to use subscripts to access a specific element of an array.

For command-line programs, you type the name of the program, followed by the arguments to the program, if any, and then hit the Enter (or Return) key to start the program running. In Example 6-3, when the user types the program name, she follows that with the argument, which, in this case, is just the string of DNA in which she'll count the G's. So the program is called and returns an answer like so:

```
AAGGGGTTTCCC
The DNA AAGGGGTTTCCC has 4 G's in it!
```

Of course, many programs come with a graphical user interface (GUI). This gives the program some or all of the computer screen and usually includes such things as menus, buttons, and places to type in values to set parameters from the keyboard.

However, many programs are run from a command line. Even the newer MacOS X, which is built on top of Unix, now provides a command line. (Although most Windows users don't use the MS-DOS command window much, it's still useful, e.g., for running Perl programs.) As already mentioned, running a program noninteractively, passing parameters in as command-line arguments, allows you to run the program automatically, say in the middle of the night when no one is actually sitting at the computer.

Example 6-3 counts the number of G's in a string of DNA.

Example 6-3. Counting the G's in some DNA on the command line

```
#!/usr/bin/perl -w
# Counting the number of G's in some DNA on the command
line
use strict;
# Collect the DNA from the arguments on the command line
#   when the user calls the program.
# If no arguments are given, print a USAGE statement and
exit.

# $0 is a special variable that has the name of the program.
my($USAGE) = "$0 DNA\n\n";

# @ARGV is an array containing all command-line arguments.
```

```
#
# If it is empty, the test will fail and the print USAGE
and exit
#   statements will be called.
unless(@ARGV) {
    print $USAGE;

exit; }

# Read in the DNA from the argument on the command line.
my($dna) = $ARGV[0];
# Call the subroutine that does the real work, and collect
the result.
my($num_of_Gs) = countG ( $dna );
# Report the result and exit.
print "\nThe DNA $dna has $num_of_Gs G\'s in it!\n\n";

exit;

###########################################################
#####################
# Subroutines for Example 6-3
###########################################################
#####################
sub countG {
    # return a count of the number of G's in the argument

$dna

    # initialize arguments and variables
    my($dna) = @_;
    my($count) = 0;
    # Use the fourth method of counting nucleotides in DNA,
as shown in
    # Chapter Four, "Motifs and Loops"
$count = ( $dna =~ tr/Gg//);
    return $count;
}
```

Now let's look at how this program works, while examining and explaining the new features. For starters, notice the new line:

```
use strict;
```

which I will use from now on to ensure all variables are declared with `my`, thus enforcing lexical scoping.

Perl has some special variables it sets so you can easily use the arguments from the command line. Every Perl program has an array variable `@ARGV` that contains any command-line arguments. Also, there's a special variable called $0 (a zero) that has the name of the program as it was called from the command line.

Notice in Example 6-3 that an informative message is defined in the variable $USAGE and that it begins with the value of the variable $0, followed an indication of the arguments the program needs. This is a common practice; if the user doesn't give the program what it needs, which is determined by some kind of test, the program prints information about how to properly use it and exits.

In fact, this program does check to see if any arguments were typed on the command line. It checks if @ARGV has anything in it, in which case it evaluates to true; or if it is completely empty, in which case it evaluates to false. If you want the program to require an argument be given, you can use the unless conditional, and if @ARGV is empty, to print out the $USAGE statement and exit the program:

```
unless(@ARGV) {
    print $USAGE;

exit; }
```

The next bit of code shows something new about arrays, namely, how to extract one element from an array, as referenced by a subscript. In other words, it shows how to get at the first, fourth, or whichever element. The code in Example 6-3 shows how to extract the first element, which as you've seen, is numbered 0:

```
my($dna) = $ARGV[0];
```

Now you already know there is a first element, since you've just tested to make sure the array isn't empty. You get the first element of array @ARGV by changing the @ to a $ and appending square brackets containing the desired subscript; 0 for the first element, 1 for the second element, and so on. This syntax indicates that since you're now looking at just one element of the array, and it's a scalar variable, you use the dollar sign, as you would any other scalar variables.

In Example 6-3, you copy this first (and only) element of the command-line array @ARGV into the variable $dna.

Finally comes the call to the subroutine, which contains nothing new but fulfills a dream from the final paragraph of Chapter 5:

```
my($num_of_Gs) = countG ( $dna );
```

Passing Data to Subroutines

When you start parsing GenBank, PDB, and BLAST files in later chapters, you'll need more complicated arguments to your subroutines to hold the several fields of data you'll parse out of the records. These next sections explain the way it's done in Perl. You can skim this section and return for a closer read when you get to Chapter 10.

6.4.1 Subroutines: Pass by Value

So far, all our subroutines have had fairly simple arguments. The values of these arguments are copied and passed to the subroutines, and whatever happens to those values in the subroutine doesn't affect the values of the arguments in the main program. This is called *pass by value* or *call by value*. For example:

```
#!/usr/bin/perl -w
# Example of pass-by-value (a.k.a. call-by-value)

use strict;

my $i = 2;

simple_sub($i);
print "In main program, after the subroutine call, \$i
equals $i\n\n";

exit;

############################################################
#####################
# Subroutines
############################################################
#####################
sub simple_sub {
    my($i) = @_;

$i += 100;

print "In subroutine simple_sub, \$i equals $i\n\n";
}
```

This gives the following output:

```
In subroutine simple_sub, $i equals 102
In main program, after the subroutine call, $i equals 2
```

Subroutines: Pass by Reference

If you have more complicated arguments, say a mixture of scalars, arrays, and hashes, Perl often cannot distinguish between them. Perl passes all arguments into the subroutine as a single array, the special `@_` array. If there are arrays or hashes as arguments, their elements get "flattened" out into this single `@_` array in the subroutine. Here's an example:

```
#!/usr/bin/perl -w

# Example of problem of pass-by-value with two arrays

use strict;
```

```
my @i = ('1', '2', '3');
my @j = ('a', 'b', 'c');
print "In main program before calling subroutine: i = " .
"@i\n";
print "In main program before calling subroutine: j = " .
"@j\n";
reference_sub(@i, @j);
print "In main program after calling subroutine: i = " .
"@i\n";
print "In main program after calling subroutine: j = " .
"@j\n";

exit;

###########################################################
#####################
# Subroutines
###########################################################
#####################
sub reference_sub {
    my(@i, @j) = @_;
print "In subroutine : i = " . "@i\n";
    print "In subroutine : j = " . "@j\n";
    push(@i, '4');

shift(@j); }
```

The following output illustrates the problem of this approach:

```
In main program before calling subroutine: i = 1 2 3
In main program before calling subroutine: j = a b c
In subroutine : i = 1 2 3 a b c
In subroutine : j =

In main program after calling subroutine: i = 1 2 3
In main program after calling subroutine: j = a b c
```

As you see, in the subroutine all the elements of `@i` and `@j` were grouped into one `@_` array. All distinction between the two arrays you started with was lost in the subroutine. When you try to get the two arrays back in the statement:

```
my(@i, @j) = @_;
```

Perl assigns everything to the first array, `@i`. This behavior makes passing multiple arrays into subroutines somewhat dicey.

Also, as usual, the original arrays in the main program were not affected by the subroutine, since you used lexical scoping (`my` variables).

To get around this problem, you can pass arguments into subroutines in a style called *pass by reference* or *call by reference*. Using pass by reference, you can pass a subroutine any collection of scalars, arrays, hashes, and more, and the subroutine can distinguish between them. There is a price to pay: the resulting code looks a little more complex. But the payoff is often well worth it.

There is one big difference in the behavior of arguments that are passed by reference. When argument variables are passed in this fashion, anything you do to the values of the argument variables in the subroutine also affects the values of the arguments in the main program.

To call a subroutine that has its arguments passed by reference, you call it the same way as before, with one difference: you must preface the argument names with a backslash. In the example of pass by reference in this section, the subroutine call is accomplished like so:

```
reference_sub(\@i, \@j);
```

As you see here, the arguments are two arrays, and, to preserve the distinction between them as they are passed into the reference_sub subroutine, they are passed by reference by prepending their names with a backslash.

Within the subroutine, there are a few changes. First, the arguments are collected from the `@_` array, and saved as scalar variables. This is because a reference is a special kind of data that is stored in a scalar variable, no matter whether it's a reference to a scalar, an array, a hash, or other. The example collects its arguments as follows:

```
my($i, $j) = @_;
```

reading them from the `@_` array as scalars.

The subroutine has to do one more thing with these referenced arguments. When it uses them, it has to dereference them. To dereference a referenced argument, you have to prepend the reference with the symbol that shows what kind of variable it is: a `$` for a scalar, `@` for an array, `%` for a hash. So these variables have two symbols before their name—reading left to right, their usual symbol and then a `$` that indicates the variable is a reference. The lines:

```
push(@$i, '4');
shift(@$j);
```

in the following subroutine are the ones that manipulate the arguments. The *push* adds an element '4' to the end of the `@i` array, and the *shift* removes the first element from the `@j` array. Because these arrays have been passed by reference, their names in the subroutine are `@$i` and `@$j`. (If you want to look at the third element of the `@j` array, which normally is `$j[2]`, you'd say `$$j[2]`.)

Whatever changes you make to the arguments in the subroutine also take effect in the main program. This is because the references are references to the actual arguments; they are not copies of their values as in pass by value. So, as you see in the example, after calling the subroutine, the arrays in the main program have been altered accordingly:

```
#!/usr/bin/perl
# Example of pass-by-reference (a.k.a. call-by-reference)
use strict;
use warnings;
```

```
my @i = ('1', '2', '3');
my @j = ('a', 'b', 'c');
print "In main program before calling subroutine: i = " .
"@i\n";
print "In main program before calling subroutine: j = " .
"@j\n";
reference_sub(\@i, \@j);
print "In main program after calling subroutine: i = " .
"@i\n";
print "In main program after calling subroutine: j = " .
"@j\n";

exit;

############################################################
#####################
# Subroutines
############################################################
#####################
sub reference_sub {
    my($i, $j) = @_;
    print "In subroutine : i = " . "@$i\n";
    print "In subroutine : j = " . "@$j\n";
    push(@$i, '4');
    shift(@$j);
}
```

This gives the following output:

```
In main program before calling subroutine: i = 1 2 3
In main program before calling subroutine: j = a b c
In subroutine : i = 1 2 3
In subroutine : j = a b c
In main program after calling subroutine: i = 1 2 3 4
In main program after calling subroutine: j = b c
```

The subroutine can now distinguish between the two arrays passed on as arguments.The changes that were made inside the subroutine to the variables remain in effect after the subroutine has ended, and you've returned to the main program. This is the essential characteristic of pass by reference.

Modules and Libraries of Subroutines

As you start to build a collection of subroutines, you'll find that you're copying them a lot from existing programs and pasting them into new programs. The subroutines then appear in multiple program. This makes the listings of your program code a bit verbose and repetitive. It also makes modifying a subroutine more complicated because you have to modify all the copies.

In short, subroutines are great, but if you have to keep copying them into each new program you write, it gets tiresome. So it's time to start collecting subroutines into the handy files called modules or libraries.

Here's how it works. You put all your reusable subroutines into a separate file. (Or, as you keep writing more and more code, and things get complicated, you may want to organize them into several files.) Then you just name the file in your program and presto: the subroutine's definitions all get read in, just as if they were in your program. To do this, you use the Perl built-in function use, which reads in the subroutine library file.

Let's call this module BeginPerlBioinfo.pm. You can put all your subroutine definitions into it, just as they appear in the program code. Then you can create the module by typing in the subroutine definitions as you read the book; or, more easily, it can be downloaded from the book's web site. But there is one thing to remember when creating or adding to a module: the last line in a module must be `1;` or it won't work. This `1;` should be the last line of the .pm file, not part of the last subroutine. If you forget this, you'll get an error message something like:

```
BeginPerlBioinfo.pm did not return a true value at jkl line
14.
BEGIN failed--compilation aborted at jkl line 14.
```

Now, to use any of the subroutines in *BeginPerlBioinfo.pm*, you just have to put the following statement in your code, near the top (near the `use strict` statement):

```
use BeginPerlBioinfo;
```

Note that .pm is left off the name on purpose: that's how Perl handles the names of modules.

There's one last thing to know about using modules to load in subroutines: the Perl program needs to know where to find the module. If you're doing all your work in one folder, everything should work okay. If Perl complains about not being able to find BeginPerlBioinfo.pm, give full pathname information to the module. If the full pathname is /home/tisdall/book/BeginPerlBioinfo.pm, then use this in your program:

```
use lib '/home/tisdall/book';
use BeginPerlBioinfo;
```

There are other ways to tell Perl where to look for modules; consult the Perl documentation for use.

Beginning in Chapter 8, I'll define subroutines and show the code, but you'll be putting them into your module and typing:

```
use BeginPerlBioinfo;
```

This module is also available for download at this book's web site.

Fixing Bugs in Your Code

Now let's talk about what to do when your program is having trouble.

A program can go wrong in any number of ways. Maybe it won't run at all. A look at the error messages, especially the first line or two of the error messages, usually leads you to the problem, which will be somewhere in the syntax, and its solution, which will be to use the correct syntax (e.g., matching braces or ending each statement with a semicolon).

Your program may run but not behave as you planned. Then you have some problem with the logic of the program. Perhaps at some point, you've zigged when you should have zagged, like adding instead of subtracting or using the assignment operator `=` when you meant to test for equality between two numbers with `==`. Or, the problem could be that you just have a poor design to accomplish your task, and it's only when you actually try it out that the flaw becomes evident.

However, sometimes the problem is not obvious, and you have to resort to the heavy artillery.

Fortunately, Perl has several ways to help you find and fix bugs in your programs. The use of the statements `use strict;` and `use warnings;` should become a habit, as you can catch many errors with them. The Perl debugger gives you complete freedom to examine a program in detail as it runs.

use warnings; and use strict;

In general, it's not too hard to tell when the syntax of a program is wrong because the Perl interpreter will produce error messages that usually lead you right to the problem. It's much harder to tell when the program is doing something you didn't really want. Many such problems can be caught if you turn on the warnings and enforce the strict use of declarations.

You have probably noticed that all the programs in this book up until now start with the command interpreter line:

```
#!/usr/bin/perl -w
```

That `-w` turns on Perl's warnings and attempts to find potential problems in your code and then to warn you about them. It finds common problems such as variables that are declared more than once, and so on, things that are not syntax errors but that can lead to bugs.

Another way to turn on warnings is to add the following statement near the top of the program:

```
use warnings;
```

The statement `use warnings;` may not be available on your version of Perl, if it's an

old one. So if your Perl complains about it, take it out and use the *-w* command instead, either on the command interpreter line, or from the command line:

```
$ perl -w my_program
```

However, use warnings; is a bit more portable between different operating systems. So, from now on, that's the way I'll turn on warnings in my code. Another important helper you should use is the following statement placed near the top of your program (next to use warnings;):

```
use strict;
```

As mentioned previously, this forces you to declare your variables. (It has some options, that are beyond the scope of this book.) It finds misspelled variables, undeclared variables that may be interfering with other parts of the program, and so on.

It's best to always use both use strict; and use warnings; when writing your Perl code.

Fixing Bugs with Comments and Print Statements

Sometimes you can identify misbehaving code by selectively commenting out sections of the program until you find the part that seems to cause the problem. You can also add print statements at suspicious parts of a misbehaving program to check what certain variables are doing. Both of these are time-honored programming techniques, and they work well in almost any programming language.

Commenting out sections of code can be particularly helpful when the error messages that you get from Perl don't point you directly at the offending line. This happens occasionally. When it does happen you may, by trial and error, discover that commenting out a small section of code causes the error messages to go away; then you know where the error is occurring.

Adding print statements can also be a quick way to pinpoint a problem, especially if you already have some idea of where the problem is. As a novice programmer, however, you may find that using the Perl debugger is easier than adding print statements. In the debugger, you can easily set print statements at any line. For instance, the following debugger command says to print the values of $i and $k before line 48:

```
  a 48 print "$i $k\n"
```

Once you learn how to do it, this method is generally faster and easier than editing the Perl program and adding print statements by hand. Using this method is partly a matter of taste, since some extremely good Perl programmers prefer to do it the old-fashioned way, by adding print statements.

The Perl Debugger

My favorite way to deal with nonobvious bugs in my programs is to use the Perl debugger. The problem with bugs in code is that once a program starts running, all you can see is the output; you can't see the steps a program is taking. The Perl debugger lets you examine your program in detail, step by step, and almost always can lead you quickly to the problem. You'll also find that it's easy to use with a little practice.

There are situations the Perl debugger can't handle well: interacting processes that depend on timing considerations, for instance. The debugger can examine only one program at a time, and while examining, it stops the program, so timing considerations with other processes go right out the window.

For most purposes, the Perl debugger is a great, essential, programming tool. This section introduces its most important features.

A program with bugs

Example 6-4 has some bugs we can examine. It's supposed to take a sequence and two bases, and output everything from those two bases to the end of the sequence (if it can find them in the sequence). The two bases can be given as an argument, or if no argument is given, the program uses the bases TA by default.

There is one new thing in Example 6-4. The next statement affects the control flow in a loop. It immediately returns the control flow to the next iteration of the loop, skipping whatever else would have followed. Also, you may want to recall `$_` , which we discussed back in Example 5-5 in the context of a `foreach` loop.

Example 6-4. A program with a bug or two

```
#!/usr/bin/perl
# A program with a bug or two
#
# An optional argument, for where to start printing the
sequence,
#  is a two-base subsequence.
#
# Print everything from the subsequence ( or TA if no
subsequence
# is given as an argument) to the end of the DNA.
# declare and initialize variables
my $dna = 'CGACGTCTTCTAAGGCGA';
my @dna;
my $receivingcommittment;
my $previousbase = '';
my$subsequence = '';
if (@ARGV) {
    my$subsequence = $ARGV[0];
}else{
    $subsequence = 'TA';

}

my $base1 = substr($subsequence, 0, 1);
my $base2 = substr($subsequence, 1, 1);
# explode DNA
@dna = split ( '', $dna );
######### Pseudocode of the following loop:
#
# If you've received a committment, print the base and
```

```
continue.  Otherwise:
#
# If the previous base was $base1, and this base is $base2,
print them.
#   You have now received a committment to print the rest
of the string.
#
# At each loop, save the previous base.
foreach (@dna) {
    if ($receivingcommittment) {

print;

        next;
    } elsif ($previousbase eq $base1) {
        if ( /$base2/ ) {
            print $base1, $base2;
            $recievingcommitment = 1;

} }

    $previousbase = $_;
}
print "\n";
exit;
```

Here's the output of two runs of Example 6-1: `$ perl example 6-4 AA`

```
$ perl example 6-4
TA
```

Huh? It should have printed out `AAGGCGA` when called with the argument `AA`, and `TAAGGCGA` when called with no arguments. There must be a bug in this program. But, if you look it over, there isn't anything obviously wrong. It's time to fire up the debugger. What follows is an actual debugging session on Example 6-4, interspersed with comments to explain what's happening and why.

How to start and stop the debugger

The debugger runs interactively, and you control it from the keyboard.[6] The most common way to start it is by giving the *-d* switch to Perl at the command line. Since you're using buggy Example 6-4 to demonstrate the debugger, here's how to start that program:

[6] You also can run it automatically to produce a trace of the program in a file. `perl -d example6-4`

Alternatively, you could have added a *-d* flag to the command interpreter:

```
#!/usr/bin/perl -d
```

On systems such as Unix and Linux where command interpretation works, this starts the debugger automatically.

To stop the debugger, simply type `q`.

Debugger command summary

First, let's try to find the bug in Example 6-4 when it's called with no arguments:

```
$ perl
-d example6-4
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(example6-4:11):    my $dna = 'CGACGTCTTCTAAGGCGA';
  DB<1>
```

Let's stop right here at the beginning and look at a few things. After some messages, which may not mean a whole lot right now, you get the excellent information that the commands `h` and `h h` give more help. Let's try `h h`:

```
  DB<1> h h
List/search source lines:
execution:
  l [ln|sub]  List source code
trace
Control script
- or .      List previous/current line  s [expr]
Single step [in expr]
  w [line]    List around line
steps over subs
  f filename  View source in file
Repeat last n or s
  /pattern/ ?patt?   Search forw/backw
Return from subroutine
  v          Show versions of modules
Continue until position
n [expr]
<CR/Enter>
r
c [ln|sub]
Debugger controls:                        L            List
break/watch/actions
  O [...]     Set debugger options        t [expr]
Toggle trace [trace expr]
  <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub]
[cnd] Set breakpoint
  ! [N|pat]   Redo a previous command
Delete a/all breakpoints
  H [-num]    Display last num commands
```

```
cmd before line
  = [a val]   Define/list an alias
watch expression
  h [db_cmd]  Get help on command
Delete all actions/watch
  |[|]db_cmd  Send output to pager

cmd in a subprocess qor^D Quit

d [ln] orD
a [ln] cmd Do
W expr Add a A orW
![!] syscmd Run R

Attempt a restart
Data Examination:       expr     Execute perl code, also
see: s,n,t expr
  x|m expr      Evals expr in list context, dumps the
result or lists methods.
  p expr        Print expression (uses script's current
package).
  S [[!]pat]    List subroutine names [not] matching
pattern
  V [Pk [Vars]] List Variables in Package.  Vars can be
~pattern or !pattern.
  X [Vars]     Same as "V current_package [Vars]".
For more help, type h cmd_letter, or run man perldebug for
all docs.

DB<2>
```

It's a bit hard to read, but you have a concise summary of the debugger commands. You can also use the `h` command, which gives several screens worth of information. The `| h` command displays those several pages one at a time; the pipe at the beginning of a debugger command pipes the output through a pager, which typically advances a page when you hit the spacebar on your keyboard. You should try those out. Right now, however, let's focus on a few of the most useful commands. But remember that typing `h` *`command`* can give you help about the command.

Stepping through statements with the debugger

Back to the immediate problem. When you started up the debugger, you saw that it stopped on the first line of real Perl code:

```
main::(example6-4:11):    my $dna = 'CGACGTCTTCTAAGGCGA';
```

There's an important point about the debugger you should understand right away. It shows the line it's about to execute, not the line it just executed.

So really, Example 6-4 hasn't done anything yet. You can see from the command summary that `p` tells the debugger to print out values. If you ask it to print the value of `$dna`, you'll find:

```
DB<2> p $dna

DB<3>
```

It didn't show anything because there's nothing to show; it hasn't even seen the variable `$dna` yet. So you should execute the statement. There are two commands to use: `n` or `s` both execute the statement being displayed. (The difference is that `n` or "next" skips the plunge into a subroutine call, treating it like a single statement; `s` or "single step" enters a subroutine and single step you through that code as well.) Once you've given one of these commands, you can just hit Enter to repeat the same command.

Since there aren't any subroutines, you needn't worry about choosing between `n` and `s`, so let's use `n`:

```
 DB<3> n
main::(example6-4:12):     my @dna;

DB<3>
```

This shows the next line (you can see the line numbers of the Perl program at the end of the prompt). If you wish to see more lines, the `w` or "window" command will serve:

```
DB<3> w 9

10 # declare and initialize variables 11: my $dna =
'CGACGTCTTCTAAGGCGA'; 12==> my @dna;
```

13. `13:   my $receivingcommittment;`
14. `14:   my $previousbase = '';`

```
15
16: my $subsequence = ''; 17

18: if (@ARGV) { DB<3>
```

The current line—the line that will be executed next—is highlighted with an arrow (`==>`). The `w` seems like a useful thing. Let's get more information about it with the help

command `h w`:

```
  DB<3> h w
w [line]

DB<4>
```

```
List window around line.
```

Actually, there's more—hitting `w` repeatedly keeps showing more of the program; a minus sign backs up a screen. But enough of that.

Now that `$dna` has been declared and initialized, the program seems wrong on the first statement:

```
  DB<4> p $dna
CGACGTCTTCTAAGGCGA

DB<5>
```

That's exactly what was expected. There's no bug, so let's continue examining the lines, printing out values here and there:

```
  DB<5> n
main::(example6-4:13):
  DB<5> n
main::(example6-4:14):
  DB<5> n
main::(example6-4:16):
  DB<5> n
main::(example6-4:18):

DB<5> p @ARGV

DB<6> w 15

my $receivingcommittment;
my $previousbase = '';
my $subsequence = '';
if (@ARGV) {
16:
17
18==>
19:
20
21:
22
23
24:
my $subsequence = '';
if (@ARGV) {
    my $subsequence =

}else{

$ARGV[0];
```

```
    $subsequence = 'TA';
}
    my $base1 = substr($subsequence, 0, 1);
DB<6> n
main::(example6-4:21):
  DB<6> n
main::(example6-4:24):
1);
  DB<6> p $subsequence
TA
  DB<7> n
main::(example6-4:25):
1);
  DB<7> n
main::(example6-4:28):
  DB<7> p $base1

$subsequence = 'TA';
my $base1 = substr($subsequence, 0,

my $base2 = substr($subsequence, 1, @dna = split ( '', $dna );

T
  DB<8> p $base2

A DB<9>
```

So far, everything is as expected; the default subsequence `TA` is being used, and the `$base1` and `$base2` variables are set to `T` and `A`, the first and second bases of the subsequence. Let's continue:

```
  DB<9> n
main::(example6-4:39):
  DB<9> p @dna
CGACGTCTTCTAAGGCGA
foreach (@dna) {

DB<10> p "@dna" CGACGTCTTCTAAGGCGA

DB<11>
```

This shows a trick with Perl and printing arrays: normally they are printed without any spacing between the elements, but enclosing an array in double quotes in a `print` statement causes it to be displayed with spaces between the elements.

Again, everything seems okay, and we're about to enter a loop. Let's look at the whole loop first:

```
DB<11> w 36 #
```

```
37 # At each loop, save the previous base. 38
39==> foreach (@dna) {

40:
41:
42:
43
44:
45:

DB<11> w 43

    if ($receivingcommittment) {
        print;
        next;
    } elsif ($previousbase eq $base1) {
        if ( /$base2/ ) {
            print $base1, $base2;
    } elsif ($previousbase eq $base1) {
        if ( /$base2/ ) {
            print $base1, $base2;
            $recievingcommitment = 1;
    $previousbase = $_;
}

44:
45:
46:
47 } 48 }

49: 50 51 52:

    print "\n";
DB<11>
```

Despite the few repeated lines resulting from the `w` command, you can see the whole loop. Now you know something in here is going wrong: when you tested the program without giving it an argument, as it's running now, it took the default argument `TA`, and so far it

seemed okay. However, all it actually did in your test was to print out the `TA` when it was supposed to print out everything in the string starting with the first occurrence of `TA`. What's going wrong?

Setting breakpoints

To figure out what's wrong, you can set a breakpoint in your code. A breakpoint is a spot in your program where you tell the debugger to stop execution so you can poke around in the code. The Perl debugger lets you set breakpoints in various ways. They let you run the program, stopping only to examine it when a statement with a breakpoint is reached. That way, you don't have to step through every line of code. (If you have 5,000 lines of code, and the error happens when you hit a line of code that's first used when you're reading the 12,000th line of input, you'll be happy about this feature.)

Notice that the part of this loop that prints out the rest of the string, once the starting two bases have been found, is the if block starting at line 40:

```
    if ($receivingcommittment) {
        print;

next; }
```

Let's look at that $receivingcommittment variable.
Here's one way to do this. Let's set a breakpoint at line 40. Type b 40 and then c to

continue, and the program proceeds until it hits line 40:

```
DB<11> b 40

  DB<12> c
main::(example6-4:40):

DB<12> p C

DB<12>

if ($receivingcommittment) {
```

The last command, p , prints out the element from the @dna array you reached in the foreach loop. Since you didn't specify a variable for the loop, it used the default $_ variable. Many Perl commands such as print or pattern matching operate on the default $_ variable if no other variable is given. (It's the cousin of the @_ default array subroutines used to hold their parameters.) So the p debugger command shows that you're operating on C from the @dna array, which is the first character.

All well and good. But it would be good to have the program break when the variable $receivingcommittment has a change in its value, and then single step from there, to see why the program isn't printing out the rest of the string. Recall that this variable is the flag whose change tells the program to print the rest of the string. First let's delete all other breakpoints:

```
DB<12> D

Deleting all breakpoints...
```

You can "watch" the variable with W like so:

```
  DB<12> W $receivingcommittment

DB<13> c TA
```

```
Debugged program terminated. Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program

termination,
  h q, h R or h O to get additional info.
  DB<13>
```

Wait a minute! The `W` command should indicate when `$receivingcommittment` changes value. But when the program continued running with the `c` command, it ran to the end, meaning that `$receivingcommittment` never changed value. So let's start up the program again and break on the line that changes its value:

```
DB<13> R
Warning: some settings and command-line options may be lost!
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(example6-4:11): my $dna = 'CGACGTCTTCTAAGGCGA';

  DB<13> w 45
42:
    next;
} elsif ($previousbase eq $base1) {

43
44:
45:
46:
47 }
48 }
49: $previousbase = $_; 50 }

51
  DB<14> b 46
  DB<15> c
TAmain::(example6-4:46):
1;
  DB<15> n
main::(example6-4:49):
if ( /$base2/ ) {
    print $base1, $base2;
    $recievingcommitment = 1;
                           $previousbase = $_;
DB<15> p $receivingcommittment

DB<16>
```

Huh? The code says it's assigning the variable a value of 1, but after you execute the code, with the `n` and try to print out the value, it doesn't print anything.

If you stare harder at the program, you see that at line 66 you misspelled `$receivingcommittment` as `$recievingcommitment`. That explains everything; fix it and run it again:

```
$ perl example6-4
TAAGGCGA
```

Success!

Fixing another bug

Now, did that fix the other bug when you ran Example 6-4 with an argument?

```
$ perl example6-4 AA
GACGTCTTCTAAGGCGA
```

Again, huh? You expected `AAGGCGA`. Can there be another bug in the program? Let's try the debugger again:

```
$ perl -d example6-4 AA
Default die handler restored.
Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.


main::(example6-4:11):
  DB<1> n
main::(example6-4:12):
  DB<1> n
main::(example6-4:13):
  DB<1> n
main::(example6-4:14):
  DB<1> n
main::(example6-4:16):
  DB<1> n
main::(example6-4:18):
  DB<1> n
main::(example6-4:19):
  DB<1> n
main::(example6-4:24):
1);
  DB<1> n
main::(example6-4:25):
1);
  DB<1> n
main::(example6-4:28):
  DB<1> p $subsequence
```

```
 my $dna = 'CGACGTCTTCTAAGGCGA';
 my @dna;
 my $receivingcommittment;
 my $previousbase = '';
 my $subsequence = '';
 if (@ARGV) {

my $subsequence = $ARGV[0];
my $base1 = substr($subsequence, 0,

my $base2 = substr($subsequence, 1, @dna = split ( '', $dna );
```

Fixing another bug

Now, did that fix the other bug when you ran Example 6-4 with an argument?

```
$ perl example6-4 AA
GACGTCTTCTAAGGCGA
```

Again, huh? You expected AAGGCGA. Can there be another bug in the program? Let's try the debugger again:

```
$ perl -d example6-4 AA
Default die handler restored.
Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.


main::(example6-4:11):
  DB<1> n
main::(example6-4:12):
  DB<1> n
main::(example6-4:13):
  DB<1> n
main::(example6-4:14):
  DB<1> n
main::(example6-4:16):
  DB<1> n
main::(example6-4:18):
  DB<1> n
main::(example6-4:19):
  DB<1> n
main::(example6-4:24):
1);
  DB<1> n
main::(example6-4:25):
1);
  DB<1> n
main::(example6-4:28):
  DB<1> p $subsequence
 my $dna = 'CGACGTCTTCTAAGGCGA';
 my @dna;
 my $receivingcommittment;
 my $previousbase = '';
 my $subsequence = '';
 if (@ARGV) {

my $subsequence = $ARGV[0];
my $base1 = substr($subsequence, 0,

my $base2 = substr($subsequence, 1, @dna = split ( '', $dna );

DB<2> p $base1
  DB<3> p $base2
  DB<4>
```

Okay, for some reason the `$subsequence`, and therefore the `$base1` and `$base2` variables, are not getting set right. How come?

Check out line 19 where you declared a new `my` variable in the block of the `if` statement with the same name, `$subsequence`. That's the variable you're setting, but it's disappearing as soon as the `if` statement is over, because it's scoped in the block since it's a `my` variable.

So again, you fix that problem by removing the `my` declaration on line 19 and instead inserting an assignment `$subsequence = $ARGV[0];` and run the program again:

```
  $ perl example6-4
TAAGGCGA
$ perl example6-4 AA
AAGGCGA
```

Here, finally, is success.

use warnings; and use strict; redux

Example 6-4 was somewhat artificial. It turns out that these problems would have been reported easily if warnings had been used. So let's see an actual example of the benefits of `use strict;` and `use warnings;`, as discussed earlier in this chapter.

If you go back to the original Example 6-4 and add the `use warnings;` directive near the top of the program, you get the following output:

```
$ perl example6-4
Name "main::recievingcommitment" used only once: possible
typo at example6-4 line 47.
TA
```

As you see, the warnings found the first bug immediately. They noticed there was a variable that was used only once, usually a sign of a misspelled variable. (I can never spell "receiving" or "commitment" properly.) So fix the misspelling at line 66, and run it again:

```
$ perl example6-4
TAAGGCGA
$ perl example6-4 AA
substr outside of string at example6-4 line 26.
Use of uninitialized value in regexp compilation at
example6-4 line 45.
Use of uninitialized value in print at example6-4 line 46.
GACGTCTTCTAAGGCGA
```

So, the first bug is fixed. The second bug remains with a few warnings that are, perhaps, hard to understand. But focus on the first error message, and see that it complains about line 26:

```
my $base2 = substr($subsequence, 1, 1);
```

So, there's something wrong with `$subsequence`. Often, error messages will be off by one line, so it may well be that the error starts on the line before, the first time `$subsequence` is operated on by the substr. But that's not the case here.

Nonetheless, the warnings have pointed directly to the problem. In this case, you still have to take a little initiative; look back at the `$subsequence` variable and notice the extra `my` declaration within the `if` block on line 20 that is preventing the variable from being initialized properly. Now this is not necessarily always a bug—declaring a variable scoped within a block and that overrides another variable of the same name that is outside the block. In fact, it's perfectly legal, so the programmers who wrote the warnings did not flag it as an obvious error. However, it seems to have caused a real problem here!

One final point: if you go back to the original, buggy program, notice there's no `use strict;` in the program. If you add that and run the program without arguments, you get the following:

```
$ perl example6-4
Global symbol "$recievingcommitment" requires explicit
package name at example6-4 line 47.
Execution of example6-4 aborted due to compilation errors.
```

Fixing the misspelled variable, and running the program with the argument, you get:

```
$ perl example6-4 AA
GACGTCTTCTAAGGCGA
```

You can see that `use strict;` didn't help for the other bug. Remember, it's best to employ both `use strict;` and `use warnings;`.

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF BIO AND CHEMICAL ENGINEERING

DEPARTMENT OF BIOINFORMATICS

UNIT – IV - Perl for Bioinformatics – SBIA1304

UNIT IV MANIPULATING MACROMOLECULES AND DATABASES

Mutations and Randomization - A Program Using Randomization - A Program to Simulate DNA Mutation - Generating Random DNA - Analyzing DNA - The Genetic Code – Hashes - Data Structures and Algorithms for Biology - A Gene Expression Database - Gene Expression Data Using Unsorted Arrays - Gene Expression Data Using Hashes - Relational Databases - Translating Codons to Amino Acids - Translating DNA into Proteins - Reading DNA from Files in FASTA Format - Reading Frames - Translating Reading Frames

Mutations and Randomization

As every biologist knows, mutation is a fundamental topic in biology. Mutations in DNA occur all the time in cells. Most of them don't affect the actions of proteins and are benign. Some of them do affect the proteins and may result in diseases such as cancer. Mutations can also lead to nonviable offspring that dies during development; occasionally they can lead to evolutionary change. Many cells have very complex mechanisms to repair mutations.

Mutations in DNA can arise from radiation, chemical agents, replication errors, and other causes. We're going to model mutations as random events, using Perl's random number generator.

Randomization is a computer technique that crops up regularly in everyday programs, most commonly in cryptography, such as when you want to generate a hard-to-guess password. But it's also an important branch of algorithms: many of the fastest algorithms employ randomization.

Using randomization, it's possible to simulate and investigate the mechanisms of mutations in DNA and their effect upon the biological activity of their associated proteins. Simulation is a powerful tool for studying systems and predicting what they will do; randomization allows you to better simulate the "ordered chaos" of a biological system. The ability to simulate mutations with computer programs can aid in the study of evolution, disease, and basic cellular processes such as division and DNA repair mechanisms. Computer models of cell development and function, now in their early stages, will become much more accurate and useful in coming years, and mutation is a basic biological mechanism these models will incorporate.

From the standpoint of programming technique, as well as from the standpoint of modeling evolution, mutation, and disease, randomization is a powerful—and, luckily for us, easy-to-use—programming skill.

Here's a breakdown of what we will accomplish in this chapter:

Randomly select an index into an array and a position in a string: these are the basic tools for picking random locations in DNA (or other data)

Model mutation with random numbers by learning how to randomly select a nucleotide in DNA and then mutate it to some other (random) nucleotide

Use random numbers to generate DNA sequence data sets, which can be used to study the extent of randomness in actual genomes

Repeatedly mutate DNA to study the effect of mutations accumulating over time during evolution

Use random numbers to generate DNA sequence data sets, which can be used to study the extent of randomness in actual genomes

Repeatedly mutate DNA to study the effect of mutations accumulating over time during evolution

A Program to Simulate DNA Mutation

Example 7-1 gave you the tools you'll need to mutate DNA. In the following examples, you'll represent DNA, as usual, by a string made out of the alphabet A, C, G, and T. You'll randomly select positions in the string and then use the *substr* function to alter the DNA.

This time, let's go about things a little differently and first compose some of the useful subroutines you'll need before showing the whole program.

Pseudocode Design

Starting with simple pseudocode, here's a design for a subroutine that mutates a random position in DNA to a random nucleotide:

```
Select a random position in the string of DNA.
Choose a random nucleotide.
Substitute the random nucleotide into the random position in the DNA.
```

This seems short and to the point. So you decide to make each of the first two sentences into a subroutine.

Select a random position in a string

How can you randomly select a position in a string? Recall that the built-in function *length* returns the length of a string. Also recall that positions in strings are numbered from `0` to `length-1`, just like positions in arrays. So you can use the same general idea as in Example 7-1, and make a subroutine:

```
# randomposition
#
# A subroutine to randomly select a position in a string.
#
# WARNING: make sure you call srand to seed the
#  random number generator before you call this function.
sub randomposition {
    my($string) = @_;
    # This expression returns a random number between 0 and
length-1,
    # which is how the positions in a string are numbered
in Perl.
    return int(rand(length($string)));
}
```

randomposition is really a short function, if you don't count the comments. It's just like the idea in Example 7-1 to select a random array element.

Of course, if you were really writing this code, you'd make a little test to see if your subroutine worked:

```
#!/usr/bin/perl -w
# Test the randomposition subroutine
my $dna = 'AACCGTTAATGGGCATCGATGCTATGCGAGCT';
srand(time|$$);
for (my $i=0 ; $i < 20 ; ++$i ) {
    print randomposition($dna), " ";
}
print "\n";
exit;
sub randomposition {
    my($string) = @_;
    return int rand length $string;
}
```

Here's some representative output of the test (your results should vary):

```
28 26 20 1 29 7 1 27 2 24 8 1 23 7 13 14 2 12 13 27
```

Notice the new look of the `for` loop:

```
for (my $i=0 ; $i < 20 ; ++$i ) {
```

This shows how you can localize the counter variables (in this case, `$i`) to the loop by declaring them with `my` inside the `for` loop.

Choose a random nucleotide

Next, let's write a subroutine that randomly chooses one of the four nucleotides:

```
# randomnucleotide
#
# A subroutine to randomly select a nucleotide
#
# WARNING: make sure you call srand to seed the
#  random number generator before you call this function.
sub randomnucleotide {
  my(@nucs) = @_;
  # scalar returns the size of an array.
  # The elements of the array are numbered 0 to size-1
  return $nucs[rand @nucs];
}
```

Again, this subroutine is short and sweet. (Most useful subroutines are; although writing a short subroutine is no guarantee it will be useful. In fact, you'll see in a bit how you can improve this one.)

Let's test this one too:

```
#!/usr/bin/perl -w
# Test the randomnucleotide subroutine
my @nucleotides = ('A', 'C', 'G', 'T');
srand(time|$$);
for (my $i=0 ; $i < 20 ; ++$i ) {
print randomnucleotide(@nucleotides), " ";
}

print "\n";

exit;

sub randomnucleotide {
    my(@nucs) = @_;
    return $nucs[rand @nucs];
}
```

Here's some typical output (it's random, of course, so there's a high probability your output will differ):

```
CAAAATTTTTACACTAAGGG
```

Place a random nucleotide into a random position

Now for the third and final subroutine, that actually does the mutation. Here's the code:

```
# mutate
#
# A subroutine to perform a mutation in a string of DNA
#

sub mutate {

    my($dna) = @_;
    my(@nucleotides) = ('A', 'C', 'G', 'T');
    # Pick a random position in the DNA
    my($position) = randomposition($dna);
    # Pick a random nucleotide
    my($newbase) = randomnucleotide(@nucleotides);
    # Insert the random nucleotide into the random position
in the DNA.
    # The substr arguments mean the following:
```

```
    #  In the string $dna at position $position change 1
character to
    #  the string in $newbase
    substr($dna,$position,1,$newbase);
    return $dna;
}
```

Here, again, is a short program. As you look it over, notice that it's relatively easy to read and understand. You mutate by picking a random position then selecting a nucleotide at random and substituting that nucleotide at that position in the string. (If you've forgotten how *substr* works, refer to Appendix B or other Perl documentation. If you're like me, you probably have to do that a lot, especially to get the order of the arguments right.)

There's a slightly different style used here for declaring variables. Whereas you've been declaring them at the beginning of a program, here you're declaring each variable the first time it's used. There are pros and cons for each programming style. Having all the variables at the top of the program gives good organization and can help in reading; declaring them on-the-fly can seem like a more natural way to write. The choice is yours.

Also, notice how this subroutine is mostly built from other subroutines, with a little bit added. That has a lot to do with its readability. At this point, you may be thinking that you've actually decomposed the problem pretty well, and the pieces are fairly easy to build and, in the end, they fit together well. But do they?

The Genetic Code

Up to this point we've used Perl to search for motifs, simulate DNA mutations, generate random sequences, and transcribe DNA to RNA. These are all important activities, and they serve as a good introduction to the computational techniques you can use to study biological systems.

In this chapter, we'll write Perl programs to simulate how the genetic code directs the translation of DNA into protein. I will start by introducing the hash datatype. Then, after a brief discussion of how different data structures (hashes, arrays, and databases) can store and access experimental information, we will write a program to translate DNA to protein. We'll also continue exploring regular expressions and write code to handle FASTA files.

Hashes

There are three main datatypes in Perl. You've already seen two: scalar variables and arrays. Now we'll start to use the third: hashes (also called associative arrays).

A hash provides very fast lookup of the value associated with a key. As an example, say you have a hash called `%english_dictionary`. (Yes, hashes start with the percent sign.) If you want to look up the definition of the word "recreant," you say:

```
$definition = $english_dictionary{'recreant'};
```

The scalar `'recreant'` is the key, and the scalar definition that's returned is the value. As you see from this example, hashes (like arrays) change their leading character to a dollar sign when you access a single element, because the value returned from a hash lookup is a scalar value. You can tell a hash lookup from an array element by the type of braces they use: arrays use square brackets [ ]; hashes use curly braces { }.

If you want to assign a value to a key, it's similarly an easy, single statement:

```
$english_dictionary{'recreant'} = "One who calls out in
surrender.";
```

Also, if you want to initialize a hash with some key-value pairs, it's done much like initializing arrays, but every pair becomes a key-value:

```
%classification = (
    'dog',       'mammal',
    'robin',     'bird',
    'asp',       'reptile',

);
```

which initializes the key `'dog'` with the value `'mammal'`, and so on. There's another way of writing this, which shows what's happening a little more clearly. The following

does exactly the same thing as the preceding code, while showing the key-value relationship more clearly:

```
%classification = (

    'dog'   => 'mammal',
    'robin' => 'bird',
    'asp',  => 'reptile',

);
```

You can get an array of all the keys of a hash:

```
@keys  = keys %my_hash;
```

You can get an array of all the values of a hash:

```
@values  = values %my_hash;
```

You use hashes in lots of different situations, especially when your data is in the form of key-value or you need to look up the value of a key fast. For instance, later in this chapter, we'll develop programs that use hashes to retrieve information about a gene. The gene name is the key; the information about the gene is the value of that key. Mathematically, a Perl hash always represents a finite function.

The name "hash" comes from something called a hash function, which practically any book on algorithms will define, if you've a mind to look it up. Let's skip the details of how they work under the hood and just talk about their behavior.

Data Structures and Algorithms for Biology

Biologists explore biological data and try to figure out how to do things with it based on its existing structure in living systems. Bioinformatics is often used to model that existing structure as closely as possible. (Bear with me; I'm speaking in generalities!)

Bioinformatics also can take a slightly different approach. It thinks about what it wants to do with the data and then tries to figure out how to organize it to accomplish that goal. In other words, it tries to produce an algorithm by representing the data in a convenient data structure.

Now that you've got the three datatypes of Perl in hand—namely scalars, arrays, and hashes—it's time to take a look at these interrelated topics of algorithms and data structures. We've already talked about algorithms in Chapter 3. The present discussion highlights the importance of the organization of the data for algorithms, in other words, the data structures for the algorithm.

The most important point here is that different algorithms often require different data structures.

A Gene Expression Database

Let's consider a typical problem. Say you're studying an organism that has a total of about 30,000 genes. (Yep, you're right, it's human.) Say you're looking at a type of cell that's never been well characterized under certain interesting environmental conditions, and you are determining, for each gene, whether it's being expressed.[1] You have a nice microarray facility that has given you the expression information for that cell. Now, for each gene, you need to look up whether it's expressed in the cell. You have to put this look-up capability on your web site, so visitors who read your results in your upcoming paper can find the expression data for the genes.

[1] For the nonbiologists: a gene is expressed when it is transcribed into RNA, so that a protein can be made from it.

There are several ways to proceed. Let's look at a few alternatives as a short and gentle introduction to the art and science of algorithms and data structures.

What is your data? For simplicity, let's say you have the names for all the genes in the organism and a number for the expressed genes indicating the level of the expression in your experiment; the unexpressed genes have the number 0.

Gene Expression Data Using Unsorted Arrays

Now let's suppose you want to know if the genes were expressed, but not the expression levels, and you want to solve this programming problem using arrays. After all, you are somewhat familiar with arrays by this point. How do you proceed?

You might store in the array only the names of the genes that are being expressed and discard the other gene names. Say there were 8,000 expressed genes. Then, for any query, the answer requires looking through the array and comparing the query with each gene in the array until either you find it or get to the end of the array without finding it.

That works, but there are problems. Mainly, it's kind of slow. This isn't bad if you just do it now and then, but if you've got a lot of people hitting your web site asking questions about this new expression data, it can be a problem. On average, a lookup for an expressed gene requires looking through 4,000 gene names. A lookup for an unexpressed gene takes 8,000 comparisons.

Also, if someone asked about a gene missing from your study, you couldn't respond, since you discarded the unexpressed gene names. The query gives a negative response, not an error message saying the gene being searched for isn't part of your experiment. This might even be a false negative if the query gene that wasn't part of your study actually is expressed in the cell type (but you just missed it). You'd prefer it if your program would report to the user that no gene by that name was studied.

So you decide to keep all 30,000 genes in the array. (Of course, now a search will be slower.) But how to distinguish the expressed from the unexpressed genes? You can load each gene's name into the array and then append the expression measurement after the name of each gene. Then you will definitely know if a gene is missing from your experiment.

However, the program is still a bit slow. You still have to search through the entire array until you find the gene or determine that it wasn't studied. You may find it right away if it's the first element in the array, or you may have to wait until the last element. On average, you have to search through half of the array. Plus, you have to compare the name of the searched-for gene with the names of the genes in the array one by one. It will average 15,000 comparisons per query: slow. (Actually, on a modern computer, not too horribly slow, really, but I'm making a point. These sorts of things do add up with a program that runs too slowly.)

Another problem is that you're now keeping two values in one scalar: the gene name and the expression measurement. To do anything with this data, you have to also separate the gene name from the measurement of the expression of the gene.

Despite these drawbacks, this method will work. Now, let's think about alternatives.

Relational Databases

Databases are programs that store and retrieve large amounts of data. They provide the most common forms of datatypes to use in algorithms. There are several popular databases. Some good ones that are free of charge (the best ones are very expensive), and Perl provides access to all the most popular ones. The Perl/DBI modules, for instance, provide convenient access to relational databases from Perl programs.

Most databases are called relational, which describes how they store data. Another common name for these types of databases is relational database management systems, or RDMS.

Relational databases store data organized in tables. The data is usually entered and extracted with a query language called Structured Query Language , or SQL, which is a

fairly simple language for accessing the data in the tables and following links between the tables.

Relational databases are the most popular way to store and retrieve large amounts of data, but they do require a fair bit of learning. Programming with relational databases is beyond the scope of this book, but if you end up doing a lot of programming with Perl, you'll find that knowing the basics of using a database is a valuable skill. See the discussion in Chapter 13.

In particular, it's perfectly reasonable to store your gene expression data in a relational database and use that in your program to respond to queries made on your web site.

The Genetic Code

The genetic code is how a cell translates the information contained in its DNA into amino acids and then proteins, which do the real work in the cell.

Background

Herein is a short introduction for the nonbiologists.

As stated earlier, DNA encodes the primary structure (i.e., the amino acid sequence) of proteins. DNA has four nucleotides, and proteins have 20 amino acids. The encoding works by taking each group of three nucleotides from the DNA and "translating" them to an amino acid or a stop signal. Each group of three nucleotides is called a codon. We'll see in detail how this coding and translation works.

Actually, transcription first uses DNA to make RNA, and then translation uses RNA to make proteins. This is called the central dogma of molecular biology. But in this course, I'll abbreviate the process and somewhat inaccurately call the entire process from DNA to protein "translation."

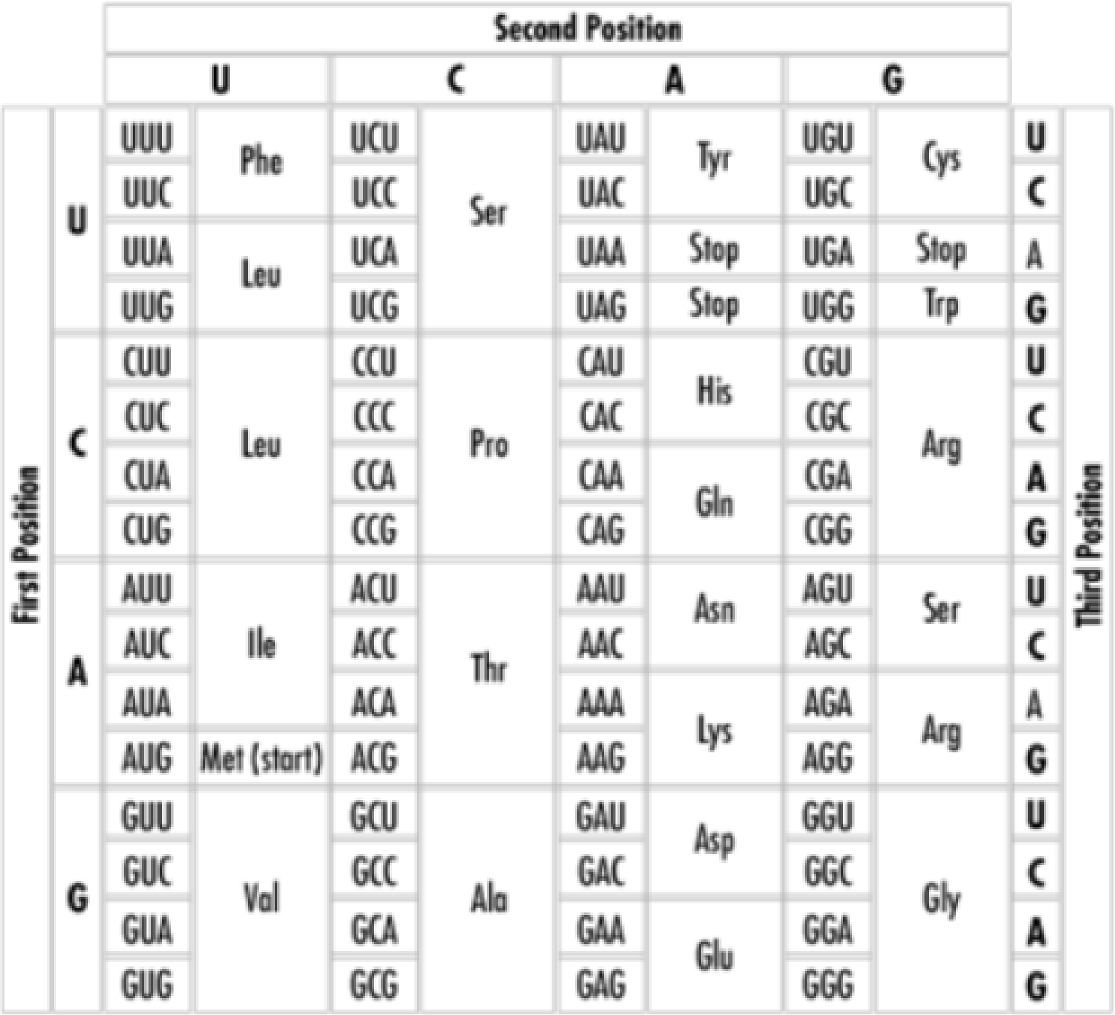
The reason for this cavalier distinction is that the whole business is much easier to simulate on computer using strings to represent the DNA, RNA, and proteins. In fact, as shown in Chapter 4, transcribing DNA to RNA is very easy indeed. In your computer simulations, you can simply skip that step, since it's just a matter of changing one letter to another. (The actual process in the cell, of course, is much more complex.)

Note that with four kinds of bases, each group of three bases of DNA can represent as many as 4 x 4 x 4 = 64 possible amino acids. Since there are only 20 amino acids plus a stop signal, the genetic code has evolved some redundancy, so that some amino acids are represented by more than one codon. Every possible three bases of DNA—each codon— represents some amino acid (apart from the three codons that represent a stop signal).

The chart in Figure 8-1 shows how the various bases combine to form amino acids. There are many interesting things to note about the genetic code. For our purposes, the most important is redundancy—the way more than one codon translates to the same amino acid. We'll program this using character classes and regular expressions, as you'll soon see.[2]

[2] Also note that the genetic code in Figure 8-1 is properly based on RNA, where uracil appears instead of thymine. In our programs, we're going to go directly from DNA to amino acids, so our codons will use thymine instead of uracil.

Figure 8-1. The genetic code



First Position	Second Position								Third Position
	U		C		A		G		
U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U
	UUC		UCC		UAC		UGC		C
	UUA	Leu	UCA		UAA	Stop	UGA	Stop	A
	UUG		UCG		UAG	Stop	UGG	Trp	G
C	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U
	CUC		CCC		CAC		CGC		C
	CUA		CCA		CAA	Gln	CGA		A
	CUG		CCG		CAG		CGG		G
A	AUU	Ile	ACU	Thr	AAU	Asn	AGU	Ser	U
	AUC		ACC		AAC		AGC		C
	AUA		ACA		AAA	Lys	AGA	Arg	A
	AUG	Met (start)	ACG		AAG		AGG		G
G	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U
	GUC		GCC		GAC		GGC		C
	GUA		GCA		GAA	Glu	GGA		A
	GUG		GCG		GAG		GGG		G

The machinery of the cell actually starts at some point along the RNA and "reads" the sequences codon after codon, attaching the encoded amino acid to the end of the growing protein sequence. Example 8-1 simulates this, reading the string of DNA three bases at a time and concatenating the symbol for the encoded amino acid to the end of the growing protein string. In the cell, the process stops when a codon is encountered.

Translating Codons to Amino Acids

The first task is to enable the following programs to do the translation from the three-nucleotide codons to the amino acids. This is the most important step in implementing the genetic code, which is the encoding of amino acids by three-nucleotide codons.

Here's a subroutine that returns an amino acid (represented by a one-letter abbreviation) given a three-letter DNA codon:

```
# codon2aa
#
```

```
# A subroutine to translate a DNA 3-character codon to an
amino acid
sub codon2aa {
    my($codon) = @_;
       if ( $codon =~ /TCA/i )
Serine
    elsif ( $codon =~ /TCC/i )
Serine
    elsif ( $codon =~ /TCG/i )
Serine
    elsif ( $codon =~ /TCT/i )
Serine
    elsif ( $codon =~ /TTC/i )
Phenylalanine
    elsif ( $codon =~ /TTT/i )
Phenylalanine
    elsif ( $codon =~ /TTA/i )
Leucine
    elsif ( $codon =~ /TTG/i )
Leucine
    elsif ( $codon =~ /TAC/i )
Tyrosine
    elsif ( $codon =~ /TAT/i )
Tyrosine
    elsif ( $codon =~ /TAA/i )
    elsif ( $codon =~ /TAG/i )
    elsif ( $codon =~ /TGC/i )
Cysteine
    elsif ( $codon =~ /TGT/i )
Cysteine
    elsif ( $codon =~ /TGA/i )
    elsif ( $codon =~ /TGG/i )
Tryptophan
    elsif ( $codon =~ /CTA/i )

Leucine

{ return 'S' }

{ return 'S' }

{ return 'S' }

{ return 'S' }

{ return 'F' }

{ return 'F' }

{ return 'L' }
{ return 'L' }

{ return 'Y' }

{ return 'Y' }

{ return '_' }
{ return '_' }
{ return 'C' }

{ return 'C' }

{ return '_' }
{ return 'W' }

{ return 'L' }

# # # # # # # # # #

# Stop # Stop #

#

# Stop #

#

elsif ( $codon =~ /CTC/i )
Leucine
    elsif ( $codon =~ /CTG/i )
Leucine
    elsif ( $codon =~ /CTT/i )
Leucine
    elsif ( $codon =~ /CCA/i )
Proline
    elsif ( $codon =~ /CCC/i )
Proline
    elsif ( $codon =~ /CCG/i )
Proline
    elsif ( $codon =~ /CCT/i )
Proline
    elsif ( $codon =~ /CAC/i )
Histidine
    elsif ( $codon =~ /CAT/i )
Histidine
    elsif ( $codon =~ /CAA/i )
Glutamine
    elsif ( $codon =~ /CAG/i )
Glutamine
    elsif ( $codon =~ /CGA/i )
```

```
Arginine
    elsif ( $codon =~ /CGC/i )
Arginine
    elsif ( $codon =~ /CGG/i )
Arginine
    elsif ( $codon =~ /CGT/i )
Arginine
    elsif ( $codon =~ /ATA/i )
Isoleucine
    elsif ( $codon =~ /ATC/i )
Isoleucine
    elsif ( $codon =~ /ATT/i )
Isoleucine
    elsif ( $codon =~ /ATG/i )
Methionine
    elsif ( $codon =~ /ACA/i )
Threonine
    elsif ( $codon =~ /ACC/i )
Threonine
    elsif ( $codon =~ /ACG/i )
Threonine
    elsif ( $codon =~ /ACT/i )
Threonine
{ return 'L' }     #
{ return 'L' }     #
{ return 'L' }     #
{ return 'P' }     #
{ return 'P' }     #
{ return 'P' }     #
{ return 'P' }     #
{ return 'H' }     #
{ return 'H' }     #
{ return 'Q' }     #
{ return 'Q' }     #
{ return 'R' }     #
{ return 'R' }     #
{ return 'R' }     #
{ return 'R' }     #
{ return 'I' }     #
{ return 'I' }     #
{ return 'I' }     #
{ return 'M' }     #
{ return 'T' }     #
{ return 'T' }     #
{ return 'T' }     #
{ return 'T' }     #
elsif ( $codon =~ /AAC/i )
Asparagine
    elsif ( $codon =~ /AAT/i )
Asparagine
    elsif ( $codon =~ /AAA/i )
```

```
Lysine
    elsif ( $codon =~ /AAG/i )
Lysine
    elsif ( $codon =~ /AGC/i )
Serine
    elsif ( $codon =~ /AGT/i )
Serine
    elsif ( $codon =~ /AGA/i )
Arginine
    elsif ( $codon =~ /AGG/i )
Arginine
    elsif ( $codon =~ /GTA/i )
Valine
    elsif ( $codon =~ /GTC/i )
Valine
    elsif ( $codon =~ /GTG/i )
Valine
    elsif ( $codon =~ /GTT/i )
Valine
    elsif ( $codon =~ /GCA/i )
Alanine
    elsif ( $codon =~ /GCC/i )
Alanine
    elsif ( $codon =~ /GCG/i )
Alanine
    elsif ( $codon =~ /GCT/i )
Alanine
    elsif ( $codon =~ /GAC/i )
Aspartic Acid
    elsif ( $codon =~ /GAT/i )
Aspartic Acid
    elsif ( $codon =~ /GAA/i )
Glutamic Acid
    elsif ( $codon =~ /GAG/i )
Glutamic Acid
    elsif ( $codon =~ /GGA/i )
Glycine
    elsif ( $codon =~ /GGC/i )
Glycine
    elsif ( $codon =~ /GGG/i )
Glycine
{ return 'N' }    #
{ return 'N' }    #
{ return 'K' }    #
{ return 'K' }    #
{ return 'S' }    #
{ return 'S' }    #
{ return 'R' }    #
{ return 'R' }    #
{ return 'V' }    #
{ return 'V' }    #
{ return 'V' }    #
{ return 'V' }    #
{ return 'A' }    #
{ return 'A' }    #
{ return 'A' }    #
{ return 'A' }    #
{ return 'D' }    #
{ return 'D' }    #
{ return 'E' }    #
{ return 'E' }    #
{ return 'G' }    #
{ return 'G' }    #
{ return 'G' }    #
elsif ( $codon =~ /GGT/i )    { return 'G' }    #
Glycine
    else {
        print STDERR "Bad codon \"$codon\"!!\n";
        exit;

} }
```

Translating DNA into Proteins

Example 8-1 shows how the new *codon2aa* subroutine translates a whole DNA sequence into protein.

Translate DNA into protein

```
#!/usr/bin/perl
# Translate DNA into protein
use strict;
use warnings;
use BeginPerlBioinfo;     # see Chapter 6 about this module
# Initialize variables
my $dna = 'CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC';
my $protein = '';
my $codon;
# Translate each three-base codon into an amino acid, and
append to a protein
for(my $i=0; $i < (length($dna) - 2) ; $i += 3) {
$codon = substr($dna,$i,3);
    $protein .= codon2aa($codon);
}
print "I translated the DNA\n\n$dna\n\n  into the
protein\n\n$protein\n\n";

exit;
```

To make this work, you'll need the BeginPerlBioinfo.pm module for your subroutines in a separate file the program can find, as discussed in Chapter 6. You also have to add the codon2aa subroutine to it. Alternatively, you can add the code for the subroutine condon2aa directly to the program in Example 8-1 and remove the reference to the BeginPerlBioinfo.pm module.

Here's the output from Example 8-1: `I translated the DNA`

```
CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC
  into the protein
RRLRTGLARVGR
```

You've seen all the elements in Example 8-1 before, except for the way it loops through the DNA with this statement:

```
for(my $i=0; $i < (length($dna) - 2) ; $i += 3) {
```

Recall that a `for` loop has three parts, delimited by the two semicolons. The first part initializes a counter: `my $i=0` statically scopes the `$i` variable so it's visible only inside this block, and any other `$i` elsewhere in the code (well, in this case, there aren't any, but it can happen) is now invisible inside the block. The third part of the `for` loop increments the counter after all the statements in the block are executed and before returning to the beginning of the loop:

```
$i += 3
```

Since you're trying to march through the DNA three bases at a shot, you increment by three.

The second, middle part of the `for` loop tests whether the loop should continue:

```
$i < (length($dna) - 2)
```

The point is that if there are none, one, or two bases left, you should quit, because there's not enough to make a codon. Now, the positions in a string of DNA of a certain length are numbered from `0` to `length-1`. So if the position counter `$i` has reached `length-2`, there's only two more bases (at positions `length-2` and `length-1`), and you should quit. Only if the position counter `$i` is less than `length-2` will you still have at least three bases left, enough for a codon. So the test succeeds only if:

```
$i < (length($dna) -2)
```

(Notice also how the whole expression to the right of the less-than sign is enclosed in parentheses; we'll discuss this in Chapter 9 in Section 9.3.1.)

The line of code:

```
$codon = substr ($dna, $i 3);
```

actually extracts the 3-base codon from the DNA. The call to the `substr` function specifies a substring of `$dna` at position `$i` of length `3`, and saves it in the variable `$codon`.

If you know you'll need to do this DNA-to-protein translation a lot, you can turn Example 8-1 into a subroutine. Whenever you write a subroutine, you have to think about which arguments you may want to give the subroutine. So you realize, there may come a time when you'll have some large DNA sequence but only want to translate a given part of it. Should you add two arguments to the subroutine as beginning and end points? You could, but decide not to. It's a judgment call—part of the art of decomposing a collection of code into useful fragments. But it might be better to have a subroutine that just translates; then you can make it part of a larger subroutine that picks endpoints in the sequence, if needed. The thinking is that you'll usually just translate the whole thing and always typing in `0` for the start and `length($dna)-1` at the end, would be an annoyance. Of course, this depends on what you're doing, so this particular choice just illustrates your thinking when you write the code.

You should also remove the informative `print` statement at the end, because it's more suited to a main program than a subroutine.

Anyway, you've now thought through the design and just want a subroutine that takes one argument containing DNA and returns a peptide translation:

```
# dna2peptide
#
# A subroutine to translate DNA sequence into a peptide

sub dna2peptide {
    my($dna) = @_;
    use strict;
    use warnings;
    use BeginPerlBioinfo;

module

    # Initialize variables
    my $protein = '';
# see Chapter 6 about this

      # Translate each three-base codon to an amino acid, and
append to a protein
    for(my $i=0; $i < (length($dna) - 2) ; $i += 3) {
        $protein .= codon2aa( substr($dna,$i,3) );
    }
    return $protein;
}
```

Reading DNA from Files in FASTA Format

Over the fairly short history of bioinformatics, several different biologists and programmers invented several ways to format sequence data in computer files, and so

bioinformaticians must deal with these different formats. We need to extract the sequence data and the annotations from these files, which requires writing code to deal with each different format.

There are many such formats, perhaps as many as 20 in regular use for DNA alone. The very multiplicity of these formats can be an annoyance when you're analyzing a sequence in the lab: it becomes necessary to translate from one format to another for the various programs you use to examine the sequence. Here are some of the most popular:

FASTA

The FASTA and Basic Local Alignment Search Technique (BLAST) programs are popular; they both use the FASTA format. Because of its simplicity, the FASTA format is perhaps the most widely used of all formats, aside from GenBank.

Genetic Sequence Data Bank (GenBank)

GenBank is a collection of all publicly released genetic data. It includes lots of information in addition to the DNA sequence. It's very important, and we'll be looking closely at GenBank files in Chapter 10.

European Molecular Biology Laboratory (EMBL)

The EMBL database has substantially the same data as the GenBank and the DDBJ (DNA Data Bank of Japan), but the format is somewhat different.

Simple data, or Applied Biosystems (ABI) sequencer output

This is DNA sequence data that has no formatting whatsoever, just the characters that represent the bases; it is output into files by the sequencing machines from ABI and from other machines and programs.

Protein Identification Resource (PIR)

PIR is a well-curated collection of protein sequence data.

Genetics Computer Group (GCG)

The GCG program (a.k.a. the GCG Wisconsin package) from Accelrys is used at many large research institutions. Data must be in GCG format to be usable by their programs.

Of these six sequence formats, GenBank and FASTA are by far the most common. The next few sections take you through the process of reading and manipulating data in FASTA.

FASTA Format

Let's write a subroutine that can handle FASTA-style data. This is useful in its own right and as a warm-up for the upcoming chapters on GenBank, PDB, and BLAST.

FASTA format is basically just lines of sequence data with newlines at the end so it can

be printed on a page or displayed on a computer screen. The length of the lines isn't specified, but for compatibility, it's best to limit them to 80 characters in length. There is also header information, a line or lines at the beginning of the file that start with the greater-than > character, that can contain any text whatsoever (or no text). Typically, a header line contains the name of the DNA or the gene it comes from, often separated by a vertical bar from additional information about the sequence, the experiment that produced it, or other, nonsequence information of that nature.

Much FASTA-aware software insists that there must be only one header line; others permit several lines. Our subroutine will accept either one or several header lines plus comments beginning with #.

The following is a FASTA file. We'll call it *sample.dna* and use it in several programs. You should copy it, download it from this book's web site, or make up your own file with your own data.

```
> sample dna | (This is a typical fasta header.)
agatggcggcgctgaggggtcttgggggctctaggccggccacctactgg
tttgcagcggagacgacgcatggggcctgcgcaataggagtacgctgcct
gggaggcgtgactagaagcggaagtagttgtgggcgcctttgcaaccgcc
tgggacgccgccgagtggtctgtgcaggttcgcgggtcgctggcgggggt
cgtgagggagtgcgccgggagcggagatatggagggagatggttcagacc
cagagcctccagatgccggggaggacagcaagtccgagaatggggagaat
gcgcccatctactgcatctgccgcaaaccggacatcaactgcttcatgat
cgggtgtgacaactgcaatgagtggttccatggggactgcatccggatca
ctgagaagatggccaaggccatccgggagtggtactgtcgggagtgcaga
gagaaagaccccaagctagagattcgctatcggcacaagaagtcacggga
gcgggatggcaatgagcgggacagcagtgagccccgggatgagggtggag
ggcgcaagaggcctgtccctgatccagacctgcagcgccgggcagggtca
gggacaggggttggggccatgcttgctcggggctctgcttcgccccacaa
atcctctccgcagcccttggtggccacacccagccagcatcaccagcagc
agcagcagcagatcaaacggtcagcccgcatgtgtggtgagtgtgaggca
tgtcggcgcactgaggactgtggtcactgtgatttctgtcgggacatgaa
gaagttcgggggccccaacaagatccggcagaagtgccggctgcgccagt
gccagctgcgggcccgggaatcgtacaagtacttcccttcctcgctctca
ccagtgacgccctcagagtccctgccaaggccccgccggccactgcccac
ccaacagcagccacagccatcacagaagttagggcgcatccgtgaagatg
agggggcagtggcgtcatcaacagtcaaggagcctcctgaggctacagcc
acacctgagccactctcagatgaggaccta
```

A Design to Read FASTA Files

In Chapter 4, you learned how to read in sequence data; here, you just have to extend that method to deal with the header lines. You'll also learn how to discard empty lines and lines that begin with the pound sign #, i.e., comments in Perl and other languages and file formats. (These don't appear in the FASTA file *sample.dna* just shown.)

There are two choices when reading in the data. You can read from the open file one line at a time, making decisions as you go. Or, you can slurp the whole file into an array and then operate on the array. For very big files, it's sometimes best to read them one line at a time, especially if you're looking for some small bit of information. (This is because reading a large file into an array uses a large amount of memory. If your system isn't robust enough, it may crash.)

For smaller, normal-sized files, the advantage to reading all the data into an array is that you can then easily look through at the data and do operations on it. That's what we'll do with our subroutine, but remember, this approach can cause memory space problems with larger files, and there are other ways of proceeding.

Let's write a subroutine that, given as an argument a filename containing FASTA- formatted data, returns the sequence data.

Before doing so you should think about whether you should have just one subroutine, or perhaps one subroutine that opens and reads a file, called by another subroutine that extracts the sequence data. Let's use two subroutines, keeping in mind that you can reuse the subroutine that deals with arbitrary files every time you need to write such a program for other formats.

Let's start with some pseudocode:

```
subroutine get data from a file
    argument = filename
    open file
        if can't open, print error message and exit
    read in data and
    return @data
}
Subroutine extract sequence data from fasta file
    argument = array of file data in fasta format
       Discard all header lines
       (and blank and comment lines for good measure)
       If first character of first line is >, discard it
    Read in the rest of the file, join in a scalar,
        edit out nonsequence data
    return sequence
}
```

In the first subroutine that gets data from a file, there's a question as to what's the best thing to do when the file can't be read. Here, we're taking the drastic approach: yelling "Fire!" and exiting. But you wouldn't necessarily want your program to just stop whenever it can't open a file. Maybe you're asking for filenames from the user at the keyboard or on a web page, and you'd like to give them three chances to type in the filename correctly. Or maybe, if the file can't be opened, you want a default file instead.

Maybe you can return a `false` value, such as an empty array, if you can't open the file. Then a program that calls this subroutine can exit, try again, or whatever it wants. But what if you successfully open the file, but it was absolutely empty? Then you'd have succeeded and returned an empty array, and the program calling this subroutine would think incorrectly, that the file couldn't be opened. So, that wouldn't work.

There are other options, such as returning the special "undefined" value. Let's keep what we've got, but it's important to remember that handling errors can be an important, and sometimes tricky, part of writing robust code, code that responds well in unusual circumstances.

The second subroutine takes the array of FASTA-formatted sequence and returns just the unformatted sequence in a string.

Reading Frames

The biologist knows that, given a sequence of DNA, it is necessary to examine all six reading frames of the DNA to find the coding regions the cell uses to make proteins.

What Are Reading Frames?

Very often you won't know where in the DNA you're studying the cell actually begins translating the DNA into protein. Only about 1-1.5% of human DNA is in genes, which are the parts of DNA used for the translation into proteins. Furthermore, genes very often occur in pieces that are spliced together during the transcription/translation process.

If you don't know where the translation starts, you have to consider the six possible reading frames. Since the codons are three bases long, the translation happens in three "frames," for instance starting at the first base, or the second, or perhaps the third. (The fourth would be the same as starting from the first.) Each starting place gives a different series of codons, and, as a result, a different series of amino acids.

Also, transcription and translation can happen on either strand of the DNA; that is, either the DNA sequence, or its reverse complement, might contain DNA code that is actually translated. The reverse complement can also be read in any one of three frames. So a total of six reading frames have to be considered when looking for coding regions , that part of the DNA that encodes proteins.

It is therefore quite common to examine all six reading frames of a DNA sequence and to look at the resulting protein translations for long stretches of amino acids that lack stop codons.

The *stop codons* are definite breaks in the DNA →protein translation process. During translation (actually of RNA to protein, but I'm being deliberately informal and vague about the biochemistry), if a stop codon is reached, the translation stops, and the growing peptide chain grows no more.

Long stretches of DNA that don't contain any stop codons are called open reading frames (ORFs) and are important clues to the presence of a gene in the DNA under study. So gene finder programs need to perform the type of reading frame analysis we'll do in this chapter.

Translating Reading Frames

Based on the facts just presented, let's write some code that translates the DNA in all six reading frames.

In the real world, you'd look around for some subroutines that are already written to do

that task. Given the basic nature of the task—something anyone who studies DNA has to do—you'd likely find something. But this is a tutorial, not the real world, so let's soldier on.

This problem doesn't sound too daunting. So, take stock of the subroutines at your disposal, think of where you are and how you can get to your destination.

Looking through the subroutines we've already written, recall dna2peptide. You may recall considering adding some arguments to specify starting and end points. Let's do this now.

Remember that although we calculated reverse complements back in Chapter 4, we never made a subroutine out of it. So let's start there:

```
# revcom
#
# A subroutine to compute the reverse complement of DNA
sequence
sub revcom {
    my($dna) = @_;
    # First reverse the sequence
    my($revcom) = reverse($dna);
    # Next, complement the sequence, dealing with upper and
lower case
    # A->T, T->A, C->G, G->C
    $revcom =~ tr/ACGTacgt/TGCAtgca/;
    return $revcom;
}
```

Now, a little pseudocode to sketch an idea for the subroutine that will translate specific ranges of DNA:

```
Given DNA sequence
subroutine translate_frame ( DNA, start, end)

return dna2peptide( substr( DNA, start, end - start + 1))

}
```

That went well! Luckily, the substr built-in Perl function made it easy to apply the desired start and end points, while passing the DNA into the already written dna2peptide

subroutine.

Note that the length of the sequence is `end-start+1`. To give a small example: if you start at position 3 and end at position 5, you've got the bases at positions 3, 4, and 5, three bases in all, which is exactly what 5 - 3 + 1 equals.

Dealing with indices like this has to be done carefully, or the code won't work. For many programs, this is the worst the mathematics gets.

You have to decide if you wish to keep the numbering of positions from 0, which is Perl's way to do it, or the first character of the sequence is in position 1, which is the biologist's way to do it. Let's do it the biologist's way. The positions will be decreased by one when passed to the Perl function *substr*, which, of course, does it Perl's way.

The corrected pseudocode looks like this:

```
Given DNA sequence
subroutine translate_frame ( DNA, start, end)
    # start and end are numbering the sequence from 1 to
```

```
length

return dna2peptide( substr( DNA, start - 1, end - start +1))

}
```

The length of the desired sequence doesn't change with the change in indices, since:

```
 (end - 1) - (start - 1) + 1 = end - start + 1
```

So let's write this subroutine:

```
# translate_frame
#
# A subroutine to translate a frame of DNA
sub translate_frame {
    my($seq, $start, $end) = @_;
    my $protein;
# To make the subroutine easier to use, you won't need
to specify
    #  the end point--it will just go to the end of the
sequence
    #  by default.
    unless($end) {
        $end = length($seq);
    }
    # Finally, calculate and return the translation
        return dna2peptide ( substr ( $seq, $start - 1,

$end -$start + 1) );
}
```

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF BIO AND CHEMICAL ENGINEERING

DEPARTMENT OF BIOINFORMATICS

UNIT – V - Perl for Bioinformatics – SBIA1304

UNIT V RESTRICTION MAPS, REGULAR EXPRESSIONS AND MORE

Regular Expressions - Logical Operators and the Range Operator - Finding the Restriction Sites - Perl Operations – GenBank - GenBank Files - Libraries - Separating Sequence and Annotation - Parsing Annotations - When to Use Regular Expressions - Protein Data Bank – Recursion - PDB Files - PDB File Format - Parsing PDB Files – BLAST - String Matching and Homology - BLAST Output Files - Parsing BLAST Output - The *printf* Function – Bioperl - Bioperl Tutorial Script

Regular Expressions

We've been dealing with regular expressions for a while now. This section fills in some background an.d ties together the somewhat scattered discussions of regular expressions from earlier parts of the book.

Regular expressions are interesting, important, and rich in capabilities. Jeffrey Friedl's book *Mastering Regular Expressions* (O'Reilly) is entirely devoted to them. Perl makes particularly good use of regular expressions, and the Perl documentation explains them well. Regular expressions are useful when programming with biological data such as sequence, or with GenBank, PDB, and BLAST files.

Regular expressions are ways of representing—and searching for—many strings with one string. Although they are not strictly the same thing, it's useful to think of regular expressions as a kind of highly developed set of wildcards. The special characters in regular expressions are more properly known as metacharacters.

Most people are familiar with wildcards, which are found in search engines or in the game of poker. You might find the reference to every word that starts with `biolog` by typing `biolog*`, for instance. Or you may find yourself holding five aces. (Different situations may use different wildcards. Perl regular expressions use * to mean "0 or more of the preceding item," not "followed by anything" as in the wildcard example just given.)

In computer science, these kinds of wildcards or metacharacters have an important history, both practically and theoretically. The asterisk character in particular is called the Kleene closure after the eminent logician who invented it. As a nod to the theory, I'll mention there is a simple model of a computer, less powerful than a Turing machine, that can deal with exactly the same kinds of languages that can be described by regular expressions. This machine model is called a finite state automaton. But enough theory for fundamental ideas we've just seen—repetition, alternation, and concatenation. For instance, the character class shown earlier can be written using alternation as `(C|G|T)`. Another common feature is the period, which can stand for any character, except a newline. So `ACG.*GCA` stands for any DNA that starts with `ACG` and ends with `GCA`. In English, this reads as: `ACG` followed by 0 or more characters followed by `GCA`.

In Perl, regular expressions are usually enclosed within forward slashes and are used as pattern-matching specifiers. Check the documentation (or Appendix B), for `m//,` which includes some options that affect the behavior of the regular expressions. Regular expressions are also used in many of Perl's built-in commands, as you will see.

The Perl documentation is essential: start with the *perlre* section of the Perl manual at http://www.perldoc.com/perl5.6/pod/perlre.html#top.

now.

We've already seen many examples that use regular expressions to find things in a DNA or protein sequence. Here I'll talk briefly about the fundamental ideas behind regular expressions as an introduction to some terminology. There is a useful summary of regular-expression features in Appendix B. Finally, we'll see how to learn more about them in the Perl documentation.

So let's start with a practical example that should be familiar by now to those who have been reading this text sequentially: using character classes to search DNA. Let's say there is a small motif you'd like to find in your library of DNA that is six basepairs long: CT followed by C or G or T followed by ACG. The third nucleotide in this motif is never A, but it can be C, G, or T. You can make a regular expression by letting the character class [CGT] stand for the variable position. The motif can then be represented by a regular expression that looks like this: CT[CGT]ACG. This is a motif that is six base pairs long with a C,G, or T in the third position. If your DNA was in a scalar variable `$dna`, you can test for the presence of the motif by using the regular expression as a conditional test in a pattern-matching statement, like so:

```
if( $dna =~ /CT[CGT]ACG/ ) {
    print "I found the motif!!\n";

}
```

Regular expressions are based on three fundamental ideas:

Repetition (or closure)

The asterisk (*), also called Kleene closure or star, indicates 0 or more repetitions of the character just before it. For example, `abc*` matches any of these strings: `ab`, `abc`, `abcc`, `abccc`, `abcccc`, and so on. The regular expression matches an infinite number of strings.

Alternation

In Perl, the pattern `(a|b)` (read: `a or b`) matches the string `a` or the string `b`.

Concatenation

This is a real obvious one. In Perl, the string `ab` means the character `a` followed by (concatenated with) the character `b`.

The use of parentheses for grouping is important: they are also metacharacters. So, for instance, the string `(abc|def)z*x` matches such strings as `abcx`, `abczx`, `abczzx`, `defx`, `defzx`, `defzzzzzx`, and so on. In English, it matches either `abc` or `def` followed by zero or more z's, and ending with an x. This example combines the ideas of grouping, alternation, closure, and concatenation. The real power of regular expressions is seen in this combining of the three fundamental ideas.

Perl has many regular-expression features. They are basically shortcuts for the three fundamental ideas we've just seen—repetition, alternation, and concatenation. For instance, the character class shown earlier can be written using alternation as (C|G|T). Another common feature is the period, which can stand for any character, except a newline. So ACG.*GCA stands for any DNA that starts with ACG and ends with GCA. In English, this reads as: ACG followed by 0 or more characters followed by GCA.

In Perl, regular expressions are usually enclosed within forward slashes and are used as pattern-matching specifiers. Check the documentation (or Appendix B), for m//, which includes some options that affect the behavior of the regular expressions. Regular expressions are also used in many of Perl's built-in commands, as you will see.

The Perl documentation is essential: start with the *perlre* section of the Perl manual at http://www.perldoc.com/perl5.6/pod/perlre.html#top.

Finding the Restriction Sites

So now it's time to write a main program and see our code in action. Let's start with a little pseudocode to see what still needs to be done:

```
# Get DNA
#
get_file_data
extract_sequence_from_fasta_data
#
# Get the REBASE data into a hash, from file "bionet"
#
parseREBASE('bionet');
for each user query
    If query is defined in the hash
        Get positions of query in DNA
    Report on positions, if any

}
```

You now need to write a subroutine that finds the positions of the query in the DNA. Remember that trick of putting a global search in a while loop from Example 5-7

and take heart. No sooner said than:

```
Given arguments $query and $dna
while ( $dna =~ /$query/ig ) {
    save the position of the match

}
```

```
return @positions
```

When you used this trick before, you just counted how many matches there were, not what the positions were. Let's check the documentation for clues, specifically the list of built-in functions in the documentation. It looks like the `pos` function will solve the problem. It gives the location of the last match of a variable in an `m//g` search. Example 9-3 shows the main program followed by the required subroutine. It's a simple subroutine, given the Perl functions like *pos* that make it easy.

Make restriction map from user queries

```
#!/usr/bin/perl
# Make restriction map from user queries on names of
restriction enzymes
use strict;
use warnings;
use BeginPerlBioinfo;     # see Chapter 6 about this module
# Declare and initialize variables
my %rebase_hash = (  );
my @file_data = (  );
my $query = '';
my $dna = '';
my $recognition_site = '';
my $regexp = '';
my @locations = (  );
# Read in the file "sample.dna"
@file_data = get_file_data("sample.dna");
# Extract the DNA sequence data from the contents of the
file "sample.dna"
$dna = extract_sequence_from_fasta_data(@file_data);
# Get the REBASE data into a hash, from file "bionet"
%rebase_hash = parseREBASE('bionet');
# Prompt user for restriction enzyme names, create
restriction map
do {
    print "Search for what restriction site for (or quit)?:
";
    $query = <STDIN>;
    chomp $query;
    # Exit if empty query
    if ($query =~ /^\s*$/ ) {

exit; }

    # Perform the search in the DNA sequence
    if ( exists $rebase_hash{$query} ) {
        ($recognition_site, $regexp) = split ( " ",
$rebase_hash{$query});
        # Create the restriction map
        @locations = match_positions($regexp, $dna);
    # Report the restriction map to the user
    if (@locations) {
print "Searching for $query $recognition_site
print "A restriction site for $query at
locations:\n";
      print join(" ", @locations), "\n";
    } else {
      print "A restriction site for $query is not in
the DNA:\n";

} }

  print "\n";
} until ( $query =~ /quit/ );

exit;

###########################################################
#####################
#
# Subroutine
#
# Find locations of a match of a regular expression in a
string
#
#
# return an array of positions where the regular expression
#  appears in the string
#
sub match_positions {
  my($regexp, $sequence) = @_;

use strict;

  use BeginPerlBioinfo;
module
  #
  # Declare variables
  #
  my @positions = (  );
# see Chapter 6 about this
```

```
#
# Determine positions of regular expression matches
#
while ( $sequence =~ /$regexp/ig ) {
```

```
push ( @positions, pos($sequence) - length($&) + 1); }

    return @positions;
}
```

Here is some sample output from Example 9-3:

```
Search for what restriction enzyme (or quit)?: AceI Searching
for AceI G^CWGC GC[AT]GC
A restriction site for AceI at locations:
54 94 582 660 696 702 840 855 957

Search for what restriction enzyme (or quit)?: AccII
Searching for AccII CG^CG CGCG
A restriction site for AccII at locations:
181
Search for what restriction enzyme (or quit)?: AaeI
A restriction site for AaeI is not in the DNA:

Search for what restriction enzyme (or quit)?: quit
```

Notice the `length($&)` in the subroutine *match_positions*. That `$&` is a special variable that's set after a successful regular-expression match. It stands for the sequence that matched the regular expression. Since *pos* gives the position of the first base *following* the match, you have to subtract the length of the matching sequences, plus one (to make the bases start at position 1 instead of position 0) to report the starting position of the match. Other special variables include `$`` which contains everything in the string before the successful match; and `$ ´,` which contains everything in the string after the successful match. So, for example: `'123456'=~ /34/` succeeds at setting these special variables like so: `$`= '12'`, $`& = '34'`, and `$ ´= '56'`.

What we have here is admittedly bare bones, but it does work. See the exercises at the end of the chapter for ways to extend this code.

Perl Operations

We've made it pretty far in this introductory programming book without talking about basic arithmetic operations, because you haven't really needed much more than addition to increment counters.

However, an important part of any programming language, Perl included, is the ability to do mathematical calculations. Look at Appendix B, which shows the basic operations available in Perl.

Precedence of Operations and Parentheses

Operations have rules of precedence. These enable the language to decide which

operations should be done first when there are a few of them in a row. The order of operations can change the result, as the following example demonstrates.

Say you have the code`8+4/2`. If you did the division first, you'd get`8+2`, or `10`.However, if you did the addition first, you'd get `12 / 2`, or `6`.

Now programming languages assign precedences to operations. If you know these, you can write expressions such as`8+4/2`, and you'd know what to expect. But this is a slippery slope.

For one thing, what if you get it wrong? Or, what if someone else has to read the code who doesn't have the memorization powers you do? Or, what if you memorize it for one language and Perl does it differently? (Different languages do indeed have different precedence rules.)

There is a solution, and it's called *using parentheses*. For Example 9-3, if you simply add parentheses: `(8 + ( 4 / 2 ))`, it's clear to you, other readers, and the Perl program, that you want to do the division first. Note that "inner" parentheses, contained within another pair of parentheses, are evaluated first.

Remember to use parentheses in complicated expressions to specify the order of operations. Among other things, it will save you some long debugging sessions!

GenBank

GenBank (Genetic Sequence Data Bank) is a rapidly growing international repository of known genetic sequences from a variety of organisms. Its use is central to modern biology and to bioinformatics.

This chapter shows you how to write Perl programs to extract information from GenBank files and libraries. Exercises include looking for patterns; creating special libraries; and parsing the flat-file format to extract the DNA, annotation, and features. You will learn how to make a DBM database to create your own rapid-access lookups on selected data in a GenBank library.

Perl is a great tool for dealing with GenBank files. It enables you to extract and use any of the detailed data in the sequence and in the annotation, such as in the FEATURES table and elsewhere. When I first started using Perl, I wrote a program that searched GenBank for all sequence records annotated as being located on human chromosome 22. I found many genes where that information was so deeply buried within the annotation, that the major gene mapping database, Genome Database (GDB), hadn't included them in their chromosome map. I think you'll discover the same feeling of power over the information when you start applying Perl to GenBank files.

Most biologists are familiar with GenBank. Researchers can perform a search, e.g., a BLAST search on some query sequence, and collect a set of GenBank files of related sequences as a result. Because the GenBank records are maintained by the individual scientists who discovered the sequences, if you find some new sequence of interest, you can publish it in GenBank.

GenBank files have a great deal of information in them in addition to sequence data, including identifiers such as accession numbers and gene names, phylogenetic classification, and references to published literature. A GenBank file may also include a detailed

FEATURES table that summarizes facts about the sequence, such as the location of the regulatory regions, the protein translation, and exons and introns.

GenBank is sometimes referred to as a databank or data store, which is different from a *database*. Databases typically have a relational structure imposed upon the data, including associated indices and links and a query language. GenBank in comparison is a flat file, that is, an ASCII text file that is easily readable by humans.[1]

[1] GenBank is also distributed in ASN.1 format, for which you need specialized tools, provided by NCBI.

From its humble beginnings GenBank has rapidly grown, and the flat-file format has seen signs of strain during the growth. With a quickly advancing body of knowledge, especially one that's growing as quickly as genetic data, it's difficult for the design of a databank to keep up. Several reworkings of GenBank have been done, but the flat-file format—in all its frustrating glory—still remains.

Due to a certain flexibility in the content of some sections of a GenBank record, extracting the information you're looking for can be tricky. This flexibility is good, in that it allows you to put what you think is most important into the data's annotation. It's bad, because that same flexibility makes it harder to write programs that to find and extract the desired annotations. As a result, the trend has been towards more structure in the annotations.

Since Perl's data structures and its use of regular expressions make it a good tool for manipulating flat files, Perl is especially well-suited to deal with GenBank data. Using these features in Perl and building on the skills you've developed from previous chapters, you can write programs to access the accumulated genetic knowledge of the scientific community in GenBank.

Since this is a beginning book that requires no programming experience, you should not expect to find the most finished, multipurpose software here. Instead you'll find a solid introduction to parsing and building fast lookup tables for GenBank files. If you've never done so, I strongly recommend you explore the National Center for Biotechnology Information (NCBI) at the National Institutes of Health (NIH) (http://www.ncbi.nlm.nih.gov). While you're at it, stop by the European Bioinformatics Institute (EBI) at http://www.ebi.ac.uk and the bioinformatics arm of the European Molecular Biology Laboratory (EMBL) at http://www.embl-heidelberg.de/. These are large, heavily funded governmental bioinformatics powerhouses, and they have (and distribute) a great deal of state-of-the-art bioinformatics software.

```
$regexp\n";
```

GenBank Files

The primary repositories for genetic information are the NCBI GenBank, EMBL in Europe, and the DNA Data Bank of Japan (DDBJ). All have almost identical information due to international cooperative agreements. Each entry or record in GenBank or its mirror sites may contain identifying, descriptive, and genetic information in ASCII- format files. Each record is written in a specific standard format, organized so that both humans and computer programs can extract the desired information with reasonable ease.

Let's look at a relatively short GenBank record and at how the fields are defined, before writing any code. I'll save this information in a file called *record.gb*, for use in later programs.

```
LOCUS       AB031069     2487 bp    mRNA            PRI
27-MAY-2000
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing
CXXC domain 1,
            complete cds.
ACCESSION   AB031069
VERSION     AB031069.1  GI:8100074
KEYWORDS    .
SOURCE      Homo sapiens embryo male lung fibroblast
cell_line:HuS-L12 cDNA to
            mRNA.
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata;
Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Primates; Catarrhini;
Hominidae; Homo.
REFERENCE   1  (sites)
  AUTHORS   Fujino,T., Hasegawa,M., Shibata,S.,
Kishimoto,T., Imai,Si. and
            Takano,T.
  TITLE     PCCX1, a novel DNA-binding protein with PHD
finger and CXXC domain,
            is regulated by proteolysis
  JOURNAL   Biochem. Biophys. Res. Commun. 271 (2), 305-310
(2000)
  MEDLINE   20261256
REFERENCE   2  (bases 1 to 2487)
  AUTHORS   Fujino,T., Hasegawa,M., Shibata,S.,
Kishimoto,T., Imai,S. and
            Takano,T.
  TITLE     Direct Submission
  JOURNAL   Submitted (15-AUG-1999) to the
DDBJ/EMBL/GenBank databases.
            Tadahiro Fujino, Keio University School of
Medicine, Department of
            Microbiology; Shinanomachi 35, Shinjuku-ku,
Tokyo 160-8582, Japan
            (E-mail:fujino@microb.med.keio.ac.jp,
            Tel:+81-3-3353-1211(ex.62692), Fax:+81-3-5360-
1508)
FEATURES             Location/Qualifiers
     source          1..2487
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
                     /sex="male"
                     /cell_line="HuS-L12"
                     /cell_type="lung fibroblast"
                     /dev_stage="embryo"
     gene            229..2199
                     /gene="PCCX1"
     CDS             229..2199
                     /gene="PCCX1"
                     /note="a nuclear protein carrying a
PHD finger and a CXXC

domain"

/codon_start=1
/product="protein containing CXXC
/protein_id="BAA96307.1"
/db_xref="GI:8100075"
/translation="MEGDGSDPEPPDAGEDSKSENGENAPIYCICRKPDINCFMIGCD
NCNEWFHGDCIRITEKMAKAIREWYCRECREKDPKLEIRYRHKKSRERDGNERDSSEP
RDEGGGRKRPVPDPDLQRRAGSGTGVGAMLARGSASPHKSSPQPLVATPSQHHQQQQQ
QIKRSARMCGECEACRRTEDCGHCDFCRDMKKFGGPNKIRQKCRLRQCQLRARESYKY
FPSSLSPVTPSESLPRPRRPLPTQQQPQPSQKLGRIREDEGAVASSTVKEPPEATATP
EPLSDEDLPLDPDLYQDFCAGAFDDHGLPWMSDTEESPFLDPALRKRAVKVKHVKRRE
KKSEKKKEERYKRHRQKQKHKDKWKHPERADAKDPASLPQCLGPGCVRPAQPSSKYCS
DDCGMKLAANRIYEILPQRIQQWQQSPCIAEEHGKKLLERIRREQQSARTRLQEMERR
FHELEAIILRAKQQAVREDEESNEGDSDDTDLQIFCVSCGHPINPRVALRHMERCYAK
YESQTSFGSMYPTRIEGATRLFCDVYNPQSKTYCKRLQVLCPEHSRDPKVPADEVCGC
PLVRDVFELTGDFCRLPKRQCNRHYCWEKLRRAEVDLERVRVWYKLDELFEQERNVRT
                     AMTNRAGLLALMLHQTIQHDPLTTDLRSSADR"

BASECOUNT 564a 715c 768g 440t ORIGIN

        1 agatggcggc gctgaggggt cttgggggct ctaggccggc
cacctactgg tttgcagcgg
       61 agacgacgca tggggcctgc gcaataggag tacgctgcct
gggaggcgtg actagaagcg
      121 gaagtagttg tgggcgcctt tgcaaccgcc tgggacgccg
ccgagtggtc tgtgcaggtt
      181 cgcgggtcgc tggcgggggt cgtgagggag tgcgccggga
gcggagatat ggagggagat
      241 ggttcagacc cagagcctcc agatgccggg gaggacagca
agtccgagaa tggggagaat
      301 gcgcccatct actgcatctg ccgcaaaccg gacatcaact
gcttcatgat cgggtgtgac
      361 aactgcaatg agtggttcca tggggactgc atccggatca
ctgagaagat ggccaaggcc
      421 atccgggagt ggtactgtcg ggagtgcaga gagaaagacc
ccaagctaga gattcgctat
481 cggcacaaga agtcacggga gcgggatggc aatgagcggg
acagcagtga gccccgggat
      541 gagggtggag ggcgcaagag gcctgtccct gatccagacc
tgcagcgccg ggcagggtca
      601 gggacagggg ttggggccat gcttgctcgg ggctctgctt
```

```
cgccccacaa atcctctccg
      661 cagcccttgg tggccacacc cagccagcat caccagcagc
agcagcagca gatcaaacgg
      721 tcagcccgca tgtgtggtga gtgtgaggca tgtcggcgca
ctgaggactg tggtcactgt
      781 gatttctgtc gggacatgaa gaagttcggg ggccccaaca
agatccggca gaagtgccgg
      841 ctgcgccagt gccagctgcg ggcccgggaa tcgtacaagt
acttcccttc ctcgctctca
      901 ccagtgacgc cctcagagtc cctgccaagg ccccgccggc
cactgcccac ccaacagcag
      961 ccacagccat cacagaagtt agggcgcatc cgtgaagatg
agggggcagt ggcgtcatca
     1021 acagtcaagg agcctcctga ggctacagcc acacctgagc
cactctcaga tgaggaccta
     1081 cctctggatc ctgacctgta tcaggacttc tgtgcagggg
cctttgatga ccatggcctg
     1141 ccctggatga gcgacacaga agagtcccca ttcctggacc
ccgcgctgcg gaagagggca
     1201 gtgaaagtga agcatgtgaa gcgtcgggag aagaagtctg
agaagaagaa ggaggagcga
     1261 tacaagcggc atcggcagaa gcagaagcac aaggataaat
ggaaacaccc agagagggct
     1321 gatgccaagg accctgcgtc actgccccag tgcctggggc
ccggctgtgt gcgccccgcc
     1381 cagcccagct ccaagtattg ctcagatgac tgtggcatga
agctggcagc caaccgcatc
     1441 tacgagatcc tcccccagcg catccagcag tggcagcaga
gcccttgcat tgctgaagag
     1501 cacggcaaga agctgctcga acgcattcgc cgagagcagc
agagtgcccg cactcgcctt
     1561 caggaaatgg aacgccgatt ccatgagctt gaggccatca
ttctacgtgc caagcagcag
     1621 gctgtgcgcg aggatgagga gagcaacgag ggtgacagtg
atgacacaga cctgcagatc
     1681 ttctgtgttt cctgtgggca ccccatcaac ccacgtgttg
ccttgcgcca catggagcgc
     1741 tgctacgcca agtatgagag ccagacgtcc tttgggtcca
tgtaccccac acgcattgaa
     1801 ggggccacac gactcttctg tgatgtgtat aatcctcaga
gcaaaacata ctgtaagcgg
1861 ctccaggtgc tgtgccccga gcactcacgg gaccccaaag
tgccagctga cgaggtatgc
     1921 gggtgccccc ttgtacgtga tgtctttgag ctcacgggtg
acttctgccg cctgcccaag
     1981 cgccagtgca atcgccatta ctgctgggag aagctgcggc
gtgcggaagt ggacttggag
     2041 cgcgtgcgtg tgtggtacaa gctggacgag ctgtttgagc
aggagcgcaa tgtgcgcaca
     2101 gccatgacaa accgcgcggg attgctggcc ctgatgctgc
accagacgat ccagcacgat
```

```
     2161 cccctcacta ccgacctgcg ctccagtgcc gaccgctgag
cctcctggcc cggacccctt
     2221 acaccctgca ttccagatgg gggagccgcc cggtgcccgt
gtgtccgttc ctccactcat
     2281 ctgtttctcc ggttctccct gtgcccatcc accggttgac
cgcccatctg cctttatcag
     2341 agggactgtc cccgtcgaca tgttcagtgc ctggtggggc
tgcggagtcc actcatcctt
     2401 gcctcctctc cctgggtttt gttaataaaa ttttgaagaa
accaaaaaaa aaaaaaaaaa
     2461 aaaaaaaaaa aaaaaaaaaa aaaaaaa
//
```

Even if you're used to seeing GenBank files, it's worth taking the time to look one over, while considering how you would write a program to extract various parts of the data. For instance, how would you extract the sequence data? What's the format of the FEATURES table and its various subfields?

There's a lot of information packed into a typical GenBank entry, and it's important to be able to separate the different parts. For instance, if you can extract the sequence, you can search for motifs, calculate statistics on the sequence, look for similarity with other sequences, and so forth. Similarly, you'll want to separate out—or parse—the various parts of the data annotation. In GenBank, this includes ID numbers, gene names, genus and species, publications, etc. The FEATURES table part of the annotation can include specific information about the DNA, such as the locations of exons, regulatory regions, important mutations, and so on.

The format specification of GenBank files and a great deal of other information about GenBank can be found in the GenBank release notes, *gbrel.txt,* on the GenBank web site at ftp://ncbi.nlm.nih.gov/genbank/gbrel.txt.
gbrel.txt gives complete detail about the structure of GenBank files to help programmers, so you may want to refer to it as your searches become more complex. As a Perl programmer, you won't need all of the detail because you can parse data using regular expressions or the *split* function. You need to get the data out and make it available to your programs. The code that accomplishes this task can be fairly simple, as you will see in this chapter.

GenBank Libraries

GenBank is distributed as a set of libraries—flat files containing many records in succession.[2] As of GenBank release 125.0, August 2001, there are 243 files, most of which are over 200 MB in size. Altogether, GenBank contains 12,813516 loci and 13,543,364,296 bases from 12,813,516 reported sequences. The libraries are usually distributed compressed, which means you can download somewhat smaller files, but you need to uncompress them after you received them. Uncompressed, this amounts to about 50 GB of data. Since 1982, the number of sequences in GenBank has doubled about every 14 months.

[2] The data is also distributed in the ASN.1 format.

GenBank libraries are further organized into divisions by the classification of the sequences they contain, either phylogenetically or by sequencing technology. Here are the divisions:

PRI: primate sequences
ROD: rodent sequences
MAM: other mammalian sequences
VRT: other vertebrate sequences
INV: invertebrate sequences
PLN: plant, fungal, and algal sequences
BCT: bacterial sequences
VRL: viral sequences
PHG: bacteriophage sequences
SYN: synthetic and chimeric sequences
UNA: unannotated sequences
EST: EST sequences (expressed sequence tags)
PAT: patent sequences
STS: STS sequences (sequence tagged sites)
GSS: GSS sequences (genome survey sequences)
HTG: HTGS sequences (high throughput genomic sequencing data)

HTC: HTC sequences (high throughput cDNA sequencing data)

Some divisions are very large: the largest, the EST, or expressed sequence tag division, is comprised of 123 library files! A portion of human DNA is stored in the PRI division, which contains (as of this writing) 13 library files, for a total of almost 3.5 GB of data. Human data is also stored in the STS, GSS, HTGS, and HTC divisions. Human data alone in GenBank makes up almost 5 million record entries with over 8 trillion bases of sequence.

The public database servers such as Entrez or BLAST at http://www.ncbi.nlm.nih.gov/ give you access to well-maintained and updated sequence data and programs, but many researchers find that they need to write their own programs to manipulate and analyze the data. The problem is, there's so much data. For many purposes, you can download a selected set of records from NCBI or other locations, but sometimes you need the whole dataset.

It's possible to set up a desktop workstation (Windows, Mac, Unix, or Linux) that contains all of GenBank; just be sure to buy a very large hard disk! Getting all that data onto your hard drive, however, is more difficult. A Perl program called mirror.pl helps to address this need. Downloading it, even with a university-standard, high-speed Internet connection can be time-consuming; downloading an entire dataset with a modem can be an exercise in frustration. The best solution is to download only the files you need, in compressed form. The EST data, for example, is about half the entire database; don't download it unless you really need to. If you need to download GenBank, I recommend contacting the help desk at NCBI. They'll help you get the most up-to-date information.

Since you're learning to program, it makes more sense to practice on a tiny, five-record library file, but the programs you'll write will work just fine on the real files.

Separating Sequence and Annotation

In previous chapters you saw how to examine the lines of a file using Perl's array operations. Usually, you do this by saving the data in an array with each appearing as an element of the array.

Let's look at two methods to extract the annotation and the DNA from a GenBank file. In the first method, you'll slurp the file into an array and look through the lines, as in previous programs. In the second, you'll put the whole GenBank record into a scalar variable and use regular expressions to parse the information. Is one approach better than the other? Not necessarily: it depends on the data. There are advantages and disadvantages to each, but both get the job done.

I've put five GenBank records in a file called *library.gb*. As before, you can download the file from this book's web site. You'll use this datafile and the file *record.gb* in the next few examples.

Separating annotations from sequence

Now that you've met the pattern-matching modifiers and regular expressions that will be your main tools for parsing a GenBank file as a scalar, let's try separating the annotations from the sequence.

The first step is to get the GenBank record stored as a scalar variable. Recall that a GenBank record starts with a line beginning with the word "LOCUS" and ends with the end-of-record separator: a line containing two forward slashes.

First you want to read a GenBank record and store it in a scalar variable. There's a device called an input record separator denoted by the special variable `$/` that lets you do exactly that. The input record separator is usually set to a newline, so each call to read a scalar from a filehandle gets one line. Set it to the GenBank end-of-record separator like so:

```
$/ = "//\n";
```

A call to read a scalar from a filehandle takes all the data up to the GenBank end-of- record separator. So the line `$record = <GBFILE>` in Example 10-2 stores the multiline GenBank record into the scalar variable `$record`. Later, you'll see that you can keep repeating this call in order to read in successive GenBank records from a GenBank library file.

After reading in the record, you'll parse it into the annotation and sequence parts making use of `/s` and `/m` pattern modifiers. Extracting the annotation and sequence is the easy part; parsing the annotation will occupy most of the remainder of the chapter.

Extract annotation and sequence from Genbank record

```
#!/usr/bin/perl
# Extract the annotation and sequence sections from the
first
#   record of a GenBank library
use strict;
use warnings;
use BeginPerlBioinfo;     # see Chapter 6 about this module
# Declare and initialize variables
```

```
my $annotation = '';
my $dna = '';
my $record = '';
my $filename = 'record.gb';
my $save_input_separator = $/;
# Open GenBank library file
unless (open(GBFILE, $filename)) {
print "Cannot open GenBank file \"$filename\"\n\n";

exit; }

# Set input separator to "//\n" and read in a record to a
scalar
$/ = "//\n";
$record = <GBFILE>;
# reset input separator
$/ = $save_input_separator;
# Now separate the annotation from the sequence data
($annotation, $dna) = ($record =~
/^(LOCUS.*ORIGIN\s*\n)(.*)\/\/\n/s);
# Print the two pieces, which should give us the same as
the
#  original GenBank file, minus the // at the end
print $annotation, $dna;

exit;
```

The output from this program is the same as the GenBank file listed previously, minus the last line, which is the end-of-record separator `//`.

Let's focus on the regular expression that parses the annotation and sequence out of the `$record` variable. This is the most complicated regular expression so far:

```
$record = /^(LOCUS.*ORIGIN\s*\n)(.*)\/\/\n/s.
```

There are two pairs of parentheses in the regular expression: `(LOCUS.*ORIGIN\s*\n)` and `(.*)`. The parentheses are metacharacters whose purpose is to remember the parts of the data that match the pattern within the parentheses, namely, the annotation and the sequence. Also note that the pattern match returns an array whose elements are the matched parenthetical patterns. After you match the annotation and the sequence within the pairs of parentheses in the regular expression, you simply assign the matched patterns to the two variables `$annotation` and `$dna`, like so:

```
($annotation, $dna) = ($record =~
/^(LOCUS.*ORIGIN\s*\n)(.*)\/\/\n/s);
```

Notice that at the end of the pattern, we've added the `/s` pattern matching modifier, which, as you've seen earlier, allows a dot to match any character including an embedded newline. (Of course, since we've got a whole GenBank record in the `$record` scalar, there are a lot of embedded newlines.)

Next, look at the first pair of parentheses:

```
(LOCUS.*ORIGIN\s*\n)
```

This whole expression is anchored at the beginning of the string by preceding it with a ^

metacharacter. (/s doesn't change the meaning of the ^ character in a regular expression.)

Inside the parentheses, you match from where the string LOCUS appears at the beginning of the GenBank record, followed by any number of characters including newlines with .*, followed by the string ORIGIN, followed by possibly some whitespace with \s*, followed by a newline \n. This matches the annotation part of the GenBank record.

Now, look at the second parentheses and the remainder:

```
(.*)\/\/\n
```

This is easier. The .* matches any character, including newlines because of the /s pattern modifier at the end of the pattern match. The parentheses are followed by the end- of-record line, //, including the newline at the end, with the slashes preceded by backslashes to show that you want to match them exactly. They're not delimiters of the pattern matching operator. The end result is the GenBank record with the annotation and the sequence separated into the variables $annotation and $sequence. Although the regular expression I used requires a bit of explanation, the attractive thing about this approach is that it took only one line of Perl code to extract both annotation and sequence.

Parsing Annotations

Now that you've successfully extracted the sequence, let's look at parsing the annotations of a GenBank file.

Looking at a GenBank record, it's interesting to think about how to extract the useful information. The FEATURES table is certainly a key part of the story. It has considerable structure: what should be preserved, and what is unnecessary? For instance, sometimes you just want to see if a word such as "endonuclease" appears anywhere in the record. For this, you just need a subroutine that searches for any regular expression in the annotation. Sometimes this is enough, but when detailed surgery is necessary, Perl has the necessary tools to make the operation successful.

Protein Data Bank

The success of the Human Genome Project in decoding the DNA sequence of human genes has captured the public imagination, but another project has been quietly gaining momentum, and it promises equally revolutionary results. This project is an international effort to determine the 3D structure of a comprehensive range of proteins on a genome- wide level using high-throughput analytical technologies. This international effort is the foundation of the new field of structural genomics.

Recent and expected advances in technology promise an accelerating pace of protein structure determination. The storehouse for all of this data is the *Protein Data Bank* (PDB). The PDB may be found on the web at http://www.rcsb.org/pdb/.

Finding the amino acid or primary sequence is just the beginning of studying a protein. Proteins fold locally into secondary structures such as alpha helices, beta-strands, and turns. Two or three adjacent secondary structures might combine into common local folds called " motifs" or "supersecondary" structures such as beta sheets or alpha-alpha units. These building blocks then fold into the 3D or tertiary structure of a protein. Finally, one or more tertiary structures may be combined as subunits into a quaternary structure such as an enzyme or a virus.

Without knowing how a protein folds into a 3D structure, you are less likely to know what the protein does or how it does it. Even if you know that the protein is implicated in a disease, knowledge of its tertiary structure is usually needed to find a possible treatment. Knowing the tertiary conformation of the *active* site of a protein (which may involve amino acids that are far apart in terms of the primary sequence but which are brought together by the folding of the protein) is critical to guide the selection of targets for new drugs.

Now that the basic genetic information of a number of organisms, including humans, has been decoded, a primary challenge facing biologists is to learn as much as possible about the proteins those genes produce and how they interact.

In fact, one of the great questions of modern biology is how the primary amino acid sequence of a protein determines its ultimate 3D shape. If a computational method can be found to reliably predict the fold of a protein from its amino acid sequence, the effect on biology and medicine would be profound.

In this chapter, you'll learn the basics of PDB files and how to parse out selected information form them. You'll also explore interesting Perl techniques for finding and iterating over lots of files, as well as controlling other bioinformatics programs from a Perl program. The exercises at the end of the chapter challenge you to extend the introductory material presented here to gain access to more of the PDB data.

BLAST

In biological research, the search for sequence similarity is very important. For instance, a researcher who has discovered a potentially important DNA or protein sequence wants to know if it's already been identified and characterized by another researcher. If it hasn't, the researcher wants to know if it resembles any known sequence from any organism. This information can provide vital clues as to the role of the sequence in the organism.

The Basic Local Alignment Search Tool (BLAST) is one of the most popular software tools in biological research. It tests a query sequence against a library of known sequences in order to find similarity. BLAST is actually a collection of programs with versions for query-to-database pairs such as nucleotide-nucleotide, protein-nucleotide, protein-protein, nucleotide-protein, and more.

This chapter examines the output from the nucleotide-nucleotide version of the program, *BLASTN* . For simplicity's sake, I'll simply refer to it here as BLAST. The main goal of this chapter is to show how to write code to parse a BLAST output file using regular expressions. The code is simple and basic, but it does the job. Once you understand the basics, you can build more features into your parser or obtain one of the fancier BLAST output parsers that's available via the Web. In either case, you'll know enough about output parsers to use or extend them.

This chapter also gives you a brief introduction to Bioperl, which is a collection of Perl bioinformatics modules. The Bioperl project is an example of an open source project that you, the Perl bioinformatics programmer, can put to good use. The Perl programming language is itself an open source project. The program and its source code are available for use and modification with only very reasonable restrictions and at no cost.

Obtaining BLAST

There are a several implementations of BLAST. The most popular is probably the one offered free of charge by the National Center for Biotechnology Information (NCBI): http://www.ncbi.nlm.nih.gov/BLAST/. The NCBI web site features a publicly available BLAST server, a comprehensive set of databases, and a well-organized collection of documents and tutorials, in addition to the BLAST software available for downloading.

Also popular is the WU-BLAST implementation from Washington University. The main web site, including a list of other WU-BLAST servers, can be found at http://blast.wustl.edu. Older versions of WU-BLAST are available at no charge. Newer versions are free if you qualify as a research or nonprofit organization and agree to the licensing arrangements from Washington University where the program is developed and maintained. If you work at a major research organization, you may already have a site license for the WU-BLAST program. If you are a for-profit company, there is a rather hefty charge for the newer WU-BLAST program (older versions are freely

available if you want to run BLAST on your own computer). Pennsylvania State University also develops some BLAST programs, available at http://bio.cse.psu.edu/. In addition to NCBI and WU-BLAST, many other BLAST server web sites are available. A Google search (http://www.google.com) on "BLAST server" will bring up many hits.

A big question that faces researchers when they use BLAST is whether to use a public BLAST server or to run it locally. There are significant advantages to using a public server, the largest being that the databases (such as GenBank) used by the BLAST server are always up to date. To keep your own up-to-date copy of these databases requires a significant amount of hard-disk space, a computer with a fairly high-end processor and a lot of memory (to run the BLAST engine), a high-capacity network link, and a lot of time setting up and overseeing the software that updates the databases. On the other hand, perhaps you have your own library of sequences that you want to use in BLAST searches, you do frequent or large searches, or you have other reasons to run your own in-house BLAST engine. If that's the case, it makes sense to invest in the hardware and run it locally.

The online documentation for BLAST is fairly extensive and includes details on the statistical methods the program uses to calculate similarity. In the next section, I touch briefly on some of those points, but you should refer to the BLAST home page and to the excellent material at the NCBI web site for the whole story and detailed references. Our interest here is not the theory, but rather to parse the output of the program.

String Matching and Homology

String matching is the computer-science term for algorithms that find one string embedded in another. It has a fairly long and fruitful history, and many string-matching algorithms have been developed using a variety of techniques and for different cases. (See the Gusfield book in Appendix A for an excellent treatment with a biological emphasis.) We've already done a fair amount of string matching, using the binding operator to search for motifs and other text with regular expressions.

BLAST is basically a string-matching program. Details of the string-matching algorithms, and of the algorithms used in BLAST in particular, are beyond the scope of this book. But first I want to define some terms that are frequently confused or used interchangeably. I also briefly introduce the BLAST statistics.

Biological string matching looks for similarity as an indication of homology. Similarity between the query and the sequences in the database may be measured by the percent identity, or the number of bases in the query that exactly match a corresponding region of a sequence from the database. It may also be measured by the degree of conservation, which finds matches between equivalent (redundant) codons or between amino acid residues with similar properties that don't alter the function of a protein (see Chapter 8). Homology between sequences means the sequences are related evolutionarily. Two sequences are or are not homologous; there's no degree of homology.

At the risk of oversimplifying a complex topic, I'll summarize a few facts about BLAST statistics. (See the BLAST documentation for a complete picture.) The output of a BLAST search reports a set of scores and statistics on the matches it has found based on the raw score S, various parameters of the scoring algorithm, and properties of the query and database. The raw score S is a measure of similarity and the size of the match. The BLAST output lists the hits ranked by their E value. The E (expect) value of a match measures, roughly, the chances that the string matching (allowing for gaps) occurs in a randomly generated database of the same size and composition. The closer to 0 the E value is, the less likely it occurred by chance. In other words, the lower the E value, the better the match. As a general rule of thumb for BLASTN, an E value less than 1 may be a solid hit, and an E value of less than 10 may be worth looking at, but this is not a hard and fast rule. (Of course, proteins can be homologous with even a very small percent identity; the percent similarity is typically higher for homologous DNA.)

Now that you have the basics, let's write code to parse BLAST output. First, you separate the hits, then extract the sequence, and finally, you find the annotation showing the E value statistic.

Bioperl

The *Bioperl* project is an important collection of Perl code for bioinformatics that has been in development since 1998. Although Bioperl uses the more advanced object- oriented style of Perl program design, it's possible to take an introductory look here at how it's organized and used.

The main focus of Bioperl modules is to perform sequence manipulation, provide access to various biology databases (both local and web-based), and parse the output of various programs.

Bioperl is available at http://www.bioperl.org/. Some of its features rely on having additional Perl modules—available from CPAN (http://www.cpan.org/)—installed. This situation is quite common, and as you do more Perl programming, you'll become familiar with installing modules from CPAN. The Bioperl tutorials include information on installing Bioperl and additional modules for the three major operating systems: Unix or Linux, Mac, and Windows.

Bioperl doesn't provide complete programs. Rather, it provides a fairly large—and growing—set of modules for accomplishing common tasks, including some tasks you've seen in this book. You're responsible for writing the code that holds the modules together. By providing these ready and (usually) easy-to-use modules, Bioperl makes developing bioinformatics applications in Perl faster and easier. There are example programs for

most of the modules, which can be examined and modified to get started.

Like many open source projects, Bioperl has suffered from fragmentation and uneven documentation, due to the strictly volunteer and geographically dispersed group of contributors. But recent work on the project leading up to Release 0.7 in March 2001 has significantly improved the project. In particular, there is now enough tutorial information on using the modules to enable you to make good use of the code.

Some difficulties still remain. Most of the code has been developed on Unix or Linux systems. Not all of it works on Macs or Windows operating systems, but most will. There are some documents available at the Bioperl web site that discuss using Bioperl on non- Unix computers, but the bottom line is that you might find that some things don't work.

If you're going to give Bioperl a try (and I strongly recommend you do), you should make sure you have a fairly recent version of Perl installed. You'll need at least Version 5.004; it would be much better to install the latest stable release from the Perl web site http://www.perl.com.

Bioperl Tutorial Script

Bioperl has a tutorial script to help you try out various parts of the package. In this section, I'll show how to start up and run some example computations.

I've mentioned already that you should learn how to download code from CPAN in order to add modules such as Bioperl. A great deal of the usefulness of the Perl programming environment now resides in these modules available on CPAN. This was a design decision: by concentrating on the core Perl language, the Perl designers can focus on making the language as good as they can. The Perl module developers can then concentrate on their many modules. By all means, take a look around the CPAN web site for an idea of the wealth of Perl modules available to you.

I won't give the details of how to install Bioperl here: as mentioned, they are available at the Bioperl web site, or you can visit the CPAN web site for information.

So, let's assume you've installed the Bioperl module and looked over the tutorial at the Bioperl web site. Now, let's see how to try out some Bioperl programs.

Go to the directory where the Bioperl software has been built on your system. For instance, on my Linux computer, I put the download file *bioperl-0.7.0.tar.gz* into the

directory */usr/local/src*, and then unpacked it with the command: `tar xvzf bioperl-0.7.0.tar.gz`

which creates the source directory /usr/local/src/bioperl-0.7.0. After installing the module (check the documentation), you're ready to run the tutorial script.

Change to the source directory and type `perl bptutorial.pl`. Here's the result (I've shown the head of the tutorial to give the author and copyright information):

```
% head bptutorial.pl
# $Id: ch12,v 1.44 2001/10/10 20:37:42 troutman Exp mam $
=head1  BioPerl Tutorial
  Cared for by Peter Schattner <schattner@alum.mit.edu>
  Copyright Peter Schattner
   This tutorial includes "snippets" of code and text from
various
   Bioperl documents including module documentation,
example scripts
% perl bptutorial.pl
The following numeric arguments can be passed to run the
corresponding demo-script.
1 => access_remote_db ,
2 => index_local_db ,
3 => fetch_local_db ,
run with demo 2)
4 => sequence_manipulations ,
5 => seqstats_and_seqwords ,
6 => restriction_and_sigcleave ,
7 => other_seq_utilities ,
8 => run_standaloneblast ,
9 => blast_parser ,
10 => bplite_parsing ,
11 => hmmer_parsing ,
12 => run_clustalw_tcoffee ,
(# NOTE: needs to be
13 => run_psw_bl2seq ,
14 => simplealign_univaln ,
15 => gene_prediction_parsing ,
16 => sequence_annotation ,
17 => largeseqs ,
18 => liveseqs ,
19 => demo_variations ,
20 => demo_xml ,
In addition the argument "100" followed by the name of a
single
bioperl object will display a list of all the public
methods
```

```
available from that object and from what object they are
inherited.
Using the parameter "0" will run all tests.
Using any other argument (or no argument) will run this
display.
So typical command lines might be:
To run all demo scripts:
 > perl -w  bptutorial.pl 0
or to just run the local indexing demos:
 > perl -w  bptutorial.pl 2 3
or to list all the methods available for object
Bio::Tools::SeqStats -
 > perl -w  bptutorial.pl 100 Bio::Tools::SeqStats

%
```

Now let's try option 9, the BLAST parser, and option 1, `access_remote_db`. So here goes, starting with the BLAST parser:

```
% perl bptutorial.pl 9
Beginning blast.pm parser example...
QUERY NAME
QUERY DESC
LENGTH
FILE
DATE
PROGRAM
VERSION
DB-NAME
sequences
DB-RELEASE
DB-LETTERS
: gi|1401126
: UNKNOWN
: 504
: t/blast.report
: Thu, 16 Apr 1998 18:56:18 -0400
: TBLASTN
: 2.0.4 [Feb-24-1998]</b>
: Non-redundant GenBank+EMBL+DDBJ+PDB
: Apr 16, 1998  9:38 AM
: 677679054
DB-SEQUENCES
GAPPED
TOTAL HITS
CHECKED ALL
FILT FUNC
SIGNIF HITS
SIGNIF CUTOFF  : 1.0e-05 (EXPECT-VALUE)
LOWEST EXPECT  : 0.0
HIGHEST EXPECT : 1e-05
HIGHEST EXPECT : 7.6 (OVERALL)
MATRIX
FILTER
EXPECT
LAMBDA, K, H
WORD SIZE

: BLOSUM62
: NONE
:10
: 0.270, 0.0470, 0.230 (SHARED STATS) :13

S
GAP CREATION
GAP EXTENSION  : 1

: 336723 : YES
: 100
: YES :NO :4

: 42, 74 (SHARED STATS) :11

Number of hits is 4
Fraction identical for hit 1 is 0.25
Sequence identities for hsp of hit 1 are 66-68 70 73 76 79
80 87-89 114 117
119 131 144 146 149 150 152 156 162 165 168 170 171 176
178-182 184 187 190
191 205-207 211 214 217 222 226 241 244 245 249 256 266-268
270 278 284 291
296 304 306 309 311 316 319 324
%
```

This is an interesting way to parse BLAST output! Now let's look at the access of the remote DB:

```
% perl bptutorial.pl 1
Beginning remote database access example...
seq1 display id is MUSIGHBA1
seq2 display id is AF303112
Display id of first sequence in stream is AF041456
%
```

Well, that was less informative as an output, but it seems you can infer that the remote DB access was successful. (By the way, if you're unsuccessful with this, it may be that you're behind a firewall which is denying access—a not uncommon occurrence in universities or large companies.)

The documentation suggests running the *bptutorial.pl* script under the Perl debugger to watch what happens step by step. I concur with that suggestion but won't include the output here. Try it yourself!

Since that last example wasn't much fun, let's try one more: here's the sequence manipulation tutorial:

```
% perl bptutorial.pl 4
Beginning sequence_manipulations and SeqIO example...
First sequence in fasta format...
>Test1
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGT
C
TGATAGCAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAG
G
TCACTAAATACTTTAACCAATATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTA
C
ACAACATCCATGAAACGCATTAGCACCACC
Seq object display id is Test1
Sequence is
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGT
CTGATAG
CAGCTTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACT
AAATACTTTAACCAATATA
GGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAACATCCATGAAACGCATTAG
CACCACC
Sequence from 5 to 10 is TTTCAT
Acc num is unknown
Moltype is dna
Primary id is Test1
Truncated Seq object sequence is TTTCAT
Reverse complemented sequence 5 to 10  is GTGCTA
Translated sequence 6 to 15 is LQRAICLCVD

Beginning 3-frame and alternate codon translation example...
ctgagaaaataa translated using method defaults : LRK*
ctgagaaaataa translated as a coding region (CDS): MRK

Translating in all six frames:
 frame: 0 forward: LRK*
 frame: 0 reverse-complement: LFSQ
 frame: 1 forward: *ENX
 frame: 1 reverse-complement: YFLX
 frame: 2 forward: EKI
 frame: 2 reverse-complement: IFS
Translating with all codon tables using method defaults:
1 : LRK*
2 : L*K*
3 : TRK*
4 : LRK*
5 : LSK*
6 : LRKQ
9 : LSN*
```

```
10 : LRK*
11 : LRK*
12 : SRK*
13 : LGK*
14 : LSNY
15 : LRK*
16 : LRK*
21 : LSN*
%
```

That was more fun, because this part of Bioperl is doing several things we've done in this book.

I hope this brief look at Bioperl has whetted your appetite for more. It's a good idea to explore this set of modules. A Perl module for parsing BLAST output called BPLite.pm may also be of interest: it's now part of the Bioperl project.