Perl for Bioinformatics

D. Gusfield, K. Stevens

Chapter 1

Formerly, Perl for Bioinformatics I

1.1 Why?

Why Program?

Shortly, we will discuss why we want to learn the programming language Perl. But first, why learn to program at all? There are hundreds, maybe thousands of computer programs already available for use in bioinformatics. Many of these have been written by professionals, many have beautiful graphics, many are free, and many are all three - free, beautiful and professionally written. What can you add to that by learning to write simple programs? Plenty!

Despite the extensive availability of high-quality, even free, software for bioinformatics, programming is still essential for several reasons.

First, because of ubiquitous problems in handling bioinformatics data. These problems range from the merely annoying to ones that can stop a project entirely. All too often the data is in the wrong format for the program you want to use, or there is too much of it to be submitted easily to the program, or it has duplications or junk that has to be located and removed, or the output from the program is overwhelming or organized badly, or it has lower caps when you want upper caps, or it is lined up wrong on the page, etc.

etc. etc. These problems are mundane, and dealing with them is not science. Still, fixing the problems by hand or by word-processor wastes enormous time and energy – scientists should not do such trivial things. But often a simple program or series of programs can quickly resolve the problems. It may take you a few hours to write the programs, but once written, the programs often fix the problems in seconds and are then available for future use by you and others.

Second, because knowing how to program liberates creative, scientific energies. Once you have the ability to program, you may start thinking of new ways you want to examine, chop up, combine, or correlate available data. This kind of exploration of data is science. Having the ability to get a program working yourself gives you the freedom to start thinking of new things you want programmed. You may think up and try out some crazy (what the heck!) new way to explore the data – follow an idea that you wouldn't bother with if you had to pay someone else to implement it.

Third, by understanding even a bit about computer programming, seeing firsthand the components of a program and the logical thinking that is required to create a program, one develops a more realistic view of what can and cannot be done by a computer. Non-programmers often have unrealistic expectations (either too pessimistic or too optimistic) about what tasks computers and software can accomplish.

Finally, programming allows (or forces) you to really examine and understand the mechanics of certain algorithms and techniques commonly used in bioinformatics. You can't program the Smith-Waterman alignment algorithm, for example, with only a "warm, fuzzy" feeling about how it works. After programming it, you still might not fully understand its logic (although we will study that as well), but when you program it, you will at least have to pay close attention to its mechanics. It's like the difference between learning how to drive a car, or learning to take the engine apart and rebuild it. When you rebuild the engine (and make the car run again), you learn a lot more, even if that exercise doesn't make you a professional mechanic.

To make the point that some programming knowledge is needed by biologists who extensively use sequence data, let's do a thought experiment. Imagine that graduate and undergraduate education in molecular biology required no chemistry courses. Imagine you have no knowledge of atoms, charge, energy,

bonding, PH, ion-transport, protein structure, folding, or any other aspect of chemistry. And why not? The argument can be made that molecular biologists aren't chemists. Genetics developed in isolation of bio-chemistry, didn't it? And in the lab, biologists increasingly use commercial kits anyway - don't they just need to know what a kit is for and how to follow the directions? If they occasionally need something special, they can always ask a chemist or a commercial company to make it for them.

Ridiculous, right? Molecularly oriented biologists must know some chemistry so that they have a reasonable intuition about what a molecule is and does, and what is happening at the chemical level in the biological processes they study. Chemistry is too central in life science to leave it entirely to others. Without some knowledge of chemistry, biologists can't even talk sensibly to chemists. And without some skills, biologists would always be at the mercy of strangers to help them solve even simple chemical problems in the lab.

The same is true with computers and computing. As computation becomes more central in the way biology is done, biologists have to understand more about it. It is not good enough just to know how to use computer packages (kits). One has to have some intuition about what really goes on in that kit, why the kit does or does not do what is claimed, and what new kits could plausibley be created. The development of skills and intuition starts with learning some simple computer programming. It won't make you a professional programmer, but it should start the process of understanding and of skill development.

Why Learn Perl?

Perl, an acronym for Practical Extraction and Report Language, has become the main programming language for simple computational tasks (and often complex ones as well) in genomics and bioinformatics. Although bioinformatics groups also use other "high level" languages such as Python, Tcl, Smalltalk, or Java, and it is not easy to completely avoid C (or C++), nothing comes close to Perl's popularity in bioinformatics. Why is this?

Perhaps the main reason, and certainly the reason we recommend and use Perl for bioinformatics, is that many many bioinformatics tasks can be effectively programmed with a very small subset of Perl. For typical sequence-

related tasks, a little Perl goes a long way. In most other computer languages, you have to learn a lot before you can do even a little. In more detail:

1. Perl is remarkably good for comparing, slicing, dicing, twisting, wringing, smoothing, summarizing and otherwise mangling text. And text is central in bioinformatics. It takes the form of DNA and protein sequences, clone names, annotations, comments, bibliographic references, whole papers, web pages, and much more. Although biologists also generate and examine non-text data such as numbers and representations of molecular structures, the most crucial data in bioinformatics is still textual (bio-molecular sequence data). Perl's powerful *regular expression* matching and string manipulation operators (which we will study in detail) simplify the handling of text in a way that is unequaled by most other computer languages.

2. Perl has *associative arrays (hashes)* that allow one to use text (rather than numbers) to connect items of data. Often this tremendously simplifies a programming task. We will use this extensively in the programs developed in this book.

3. Perl is well suited to handling lists that can contain mixtures of text and numbers. Moreover, that mixture can change dynamically when the program runs, in ways that the programmer does not have to anticipate. That makes many programming tasks simper, and is a hard trick to duplicate in a language like C.

4. Perl does the memory management for you. You don't have to guess how much space your program might require and when.

   Try as we might, we have never gotten a "segmentation fault" when running a Perl program. (For those of you who have not had the pleasure of programming in C, this is a very common error in C, indicating that you are trying to use more memory than you initially said you would.)

5. Perl is "loose and forgiving". Other languages are "uptight and rigid" in comparison. Biological data is often incomplete, or generally messed up in some way. A field may be missing, or data that is expected to be present once occurs several times, or a field that was expected to hold a number actually holds text, or, or, or ...

Perl doesn't particularly mind if a value is empty or contains odd characters. Regular expressions can be written to pick up and correct a variety of common errors in data entry. It is not even very difficult to handle data that is out of an expected order.

6. Perl allows data and data structures to expand and contract dynamically, requiring little effort on the part of the programmer.

7. Once mastered, Perl programs are easy to write and to get right. They are usually small compared to programs for the same task written in other languages. So, for the right applications, the whole programming process is smoother and quicker in Perl.

8. Perl is a good prototyping language. Because Perl is "quick and dirty", it often makes sense to quickly try out and explore a new algorithm or idea using Perl, before making the effort to implement it in a language that executes faster. Human time is important, especially when you are not exactly sure what you want the computation to do. This is actually common in bioinformatics - you use the program to *explore* the data rather than *process* it, successively modifying the program and the data, letting the results guide you to the next modification.

   Once you have the algorithm you want, if you need faster execution you can usually rewrite a small core of the program in a compiled language such as C, then compile it as a dynamically loaded module or external executable, and leave the rest of the program in Perl. (Don't worry if that last part makes no sense to you. We will explain this later.)

9. Perl is a good language for Web programming, allowing one to do things with the web that are inefficient or labor-intensive to do with a web browser alone. This is of growing importance in genomics and bioinformatics as the Web is absolutely indespensible in those fields. (We will only briefly touch on Perl as a language for Web programming.)

10. Perl is fun.

For all these reasons, and more, we recommend and use Perl for bioinformatics.

Why a Specialized Book on Perl for Bioinformatics?

Why not just get a good introduction to Perl, learn to program from it, and then go out and have at your bioinformatics applications? Three reasons.

First, Perl is a very versatile language that is often used for purposes that aren't central to bioinformatics. Therefore, there are many books that emphasize aspects of Perl programming (CGI programming or systems administration for example) that are pretty far from our interests in bioinformatics. For our uses, we like Perl as a general-purpose, stand-alone programming language. There are many books that don't look at Perl in that way.

Second, among books that do teach Perl as a general-purpose language, most are written for experienced programmers. Until recently, Perl was considered a language for sophisticated programmers, and there are some wonderful, even poetic, books for that audience. For the techno-nerd, some of these books are closest thing to fine literature they will ever read. But our intended audience includes many people who have never programmed before, in any language. The Great Books of Perl aren't appropriate for them.

Finally, even though we know of one good book that teaches Perl as a general-purpose language and aims at novice programmers, it doesn't have any applications or illustrations from bioinformatics (of course). In teaching Perl to bioinformatics students (most of whom are from biology), we have found that it is really hard for the novice programmer to transfer what they learn from a general book, to their specific needs in bioinformatics.

This book teaches Perl and programming through applications that really arise in bioinformatics. In that way, the student sees programs that are closer to their own needs, and that can be used as templates for their work. We are also able to teach some central concepts in bioinformatics in this way. We can introduce common data formats used in bioinformatics, talk about important databases, illustrate some important bioinformatics problems such as gene hunting, fully develop programs for classic algorithms such as the Needlaman-Wunch and Smith-Waterman algorithms for sequence alignment, and explore toy versions of classic programs such as BLAST and CLUSTAL.

1.2 Beginning to Program in Perl

A First Program

This is the first of several sections introducing the parts of Perl that we think are of most immediate use in bioinformatics. A programming background is not required, however basic computer skills are assumed. Our teaching style is the "see one, do one, teach one" approach, whereby you, the student, examine a program, then modify it to do a related task, and finally write a new, but similar, program for a different task. So, when there is an explicit exercise, or the text just says something like "Try this out", do it!.

We start with a simple, but complete, Perl program.

```
#!/usr/bin/perl
#
# A Simple Printing Program
#
print "Welcome to Perl\n";   # Print a greeting
print "TATA is a box\n";   # Print a second line
```

The first line in the program tells the computer where to find the Perl interpreter: in the above example, it is assumed to be in the directory /usr/bin. (An *interpreter* is a program that reads a Perl program and tells the computer how to execute each line). The word `perl` in the first line tells the computer that the program to follow is a Perl program. The exact location of the Perl interpreter varies on different systems (on mine, it is in the directory /usr/pkg, so each program on my system starts with the line `#!/usr/pkg/perl`). You can find the location of the Perl interpreter on your (Unix) system by typing:

```
which Perl
```

at the Unix prompt. The response you get back should be typed (following #!), in the first line of each of your Perl programs.

On windows, we use ActiveState Perl, downloaded from the website referenced at the start of Johnson p. 10. Those Perl programs actually run under

DOS, and it is sometimes easiest to open a DOS window to do all your editing and executions of Perl programs when using ActiveState Perl.

Don't have Unix, Linux or windows? Sorry, we can't help you.

Following the first line, a Perl program consists primarily of *statements* and *comments*. Statements make up the functional part of the program and are executed by the computer. The program above contains two simple statements (we will explain "simple" later), namely the two *print* statements. Each simple statement must end with a semicolon, as in the two statements above. Comments help the programmer or the reader of the program understand how it works. With experience, one learns how necessary it is to have comments in the program to explain (to humans) its logic and to remind the programmer of any details that are not obvious. Comments in Perl begin with a # and continue to the end of the line.

The above program contains two calls to the *print* function. Functions are subunits built into the Perl language that perform a predefined task on the input given to the function. Above, the first statement calls the print function, which outputs the string

Welcome to Perl

Before explaining more of the print statement, let's see how to run the above simple program.

Running The Program

Before you can run a Perl program, it must exist as a file on your local computer. Type or paste in the first program using a text editor, and then save it (as text) to a file. You can use any text editor (or word processor) you like, but some, such as Emacs, are attractive because they have a Perl mode that formats lines to aid legibility [ref perl mode]. But use whichever editor you're most comfortable with.

After you've entered and saved the above program in a file, you must make the file *executable*. In windows, this is automatic, but on a Unix system, this is done by typing the command

```
chmod u+x progname.pl
```

at the UNIX prompt, where **progname.pl** is the filename you used to save the program. You can do this without understanding it, but for a clue, chmod is a UNIX command to change the *protections* of a file. In this case, it changes the protections of the file progname.pl to allow the user to *execute* it. You don't have to use the specific extension .pl in the program name, but it is common to do so, and it reminds the user that this is a Perl program. You only have to use the chmod command once for each file, even if you later modify the program. Now to run the program just type

```
progname.pl
```

at the UNIX prompt. Shortly, you should see the following text on the screen:

```
Welcome to Perl
TATA is a box
```

Now enter the program and run it before going further.

Exercise 1.1 *As a simple modification exercise, remove the characters \n from the first line and rerun the program, noting how the output changes.*

Then try the program with the second \n also removed, and finally with the first one in but the second removed. Try inserting \t just after the open quote in one of the lines, to see the effect. Then insert it at different places in the lines, and note the effect. Next, replace the double quotes with single quotes (the key found below the double quote on most keyboards).

What was the effect? Next, try the program with no quotes at all, and note the consequences. Try to put together a coherent explanation for the outcomes of these experiments. We will explain the results in the next section.

Printing Strings

From the exercise above, you should have observed a few things about printing text in Perl. Now we begin a more complete explanation of what you observed.

The `print` function is usually the primary output tool in a Perl program. The first program printed out two strings. A *string* is simply a sequence of characters. For example, `Welcome to Perl\n` is the first string printed by the program. But notice that the \n in the print statement was not printed when the program was run. This is because \n is a *meta-character*, a character that tells the computer what to do, rather than what characters to print. In this case, \n tells the computer to advance the printing to the next line. This "new line" meta-character only works when \n is found between double-quotes. When the double quotes were replaced with single quotes, \n was literally printed on the screen at the end of the line, and the computer did not move to a new line before printing the second line.

In a `print` statement, the effect of a meta-characters depends on whether the string to be printed is enclosed in double-quotes or single- quotes. Strings enclosed in single quotes are printed literally (every character), even if the string contains some meta-characters. But strings enclosed in double-quotes are said to be *interpolated*, which means that the meta-characters inside the string are not printed at all, but rather are used to give the computer instructions.

Another frequently used meta-character is \t, representing a *tab advance*. The tab character is also interpolated when enclosed in double-quotes. Each time this meta-character is encountered in a print statement (and the string is enclosed in double-quotes), the printing advances to the next (pre-defined) tab position [ref quotes and strings].

If you need to literally print a meta-character inside a string that is enclosed in double quotes, simply put a backslash \ immediately before the meta-character. The effect is to suppress interpolation. This will also work if you want to literally print out other special characters (which we will introduce later) that would otherwise be interpolated inside double- quotes. For example:

```
print "\\n is a meta-character";
```

will print out

```
\n is a meta-character.
```

Synopsis of a Print statement

Below is the first of many *synopses* of Perl statements we include in this book. There are a few convensions you need to be aware of. Every synopsis begins with the name of the statement in lowercase, and is followed by a description of what comes after that name. Those descriptions are generic and are given in uppercase, with some greater detail written below the statement. For example, the word LIST is used in the print synopses, and then some additional detail is given below on what a LIST consists of. Also, in a synopsis, anything enclosed in brackets "[...]" represents an *optional* part of the statement. The brackets themselves are not part of the Perl statement, they are part of the synopsis. (Confusing? It will become clear with additional examples. Just remember that if you have a Perl statement that seems to follow the form of the synopsis, but is not working, see if you have brackets in the statement and try taking them out.)

SYNOPSIS *the* **print** *function*

print [FILEHANDLE] LIST;

Writes out LIST. The LIST is usually a single string or list of strings and variables, separated by commas (we will discuss lists and variables shortly). A FILEHANDLE is optionally used when writing the output list to a file (we will discuss this later). If no filehandle is given (as in the first program) the default output is to the screen.

Here is another example of using the print function. Compare the output to the what is in quotes in the original program.

```
#!/usr/bin/perl
# Print example
print 'hello\n';
print "Hello\n";
print "Hello\\\n", "World\n"; print "b\ty\te";
```

Here is the output.

```
hello\nHello
Hello\
World
b       y       e
```

Exercise 1.2 *Modify the Welcome program to produce the following output:*

```
When in double quotes:
Use \\ to output \
Use \t to output tabs
```

Of course, there are several ways to do this.

1.3 First Simple Tools for Text and Strings

You can do a lot more with strings in Perl than simply print them to the screen. Strings can also be placed into *variables*. As you might already know, a variable is a name for a location in memory that is used to hold data that may be retrieved or modified by the program [ref data types]. The programmer invents the name of the variable, and the computer system assigns the actual location in memory for that variable.

In Perl, each variable name begins with a symbol that tells the computer what *type* of variable it is, that is, what kind of data it is holding. This helps the interpreter figure out how to process that variable. Of most importance to us, variables that can contain strings start with $. For example, **$dnastring** is a variable that can hold a string (or a number – we will get to that in a bit). Then, the statement

```
$dnastring = 'actttgact';
```

assigns the string `actttgact` to the variable $dnastring.

Once assigned to a variable, a string can be referred to and manipulated by using the variable. For example, the program

```
$dnastring = 'actttgact';
$seconddna = 'tataccta';
print "$dnastring\n";
print "$dnastring \t $seconddna";
```

will print out

```
actttgact
actttgact       tataccta
```

The *value* of variable $dnastring is printed because the variable $dnastring is contained in double quotes, causing the variable to be interpolated. Try replacing the double quotes with single quotes, or removing quotes entirely, to see what happens.

Of course, we don't just want to print strings – Perl contains powerful tools for searching and manipulating strings – very, very useful for bioinformatics. We will now examine a simple one.

Translation

It is often useful to *translate* (that is, systematically change specified characters in a string into other specified characters). The Perl `tr` [ref translate] function is used for this purpose. It also has other uses, such as *deleting*

characters from a string, or counting the number of specific characters in a string.

We begin with the following program that converts a DNA string into the equivalent RNA string, printing out both the original and the translated string. The conversion is done with a single translate statement that leaves the characters a,A,c,C,g,G unchanged, but changes t and T into u and U respectively.

```
#!/usr/bin/perl
#
$nucstring = 'acCtagGgCCTTAcga';
print "$nucstring \n";
$nucstring =~ tr/tT/uU/;
print "$nucstring \n";
```

This use of the translate statement starts with a variable holding a string (**$nucstring** in the program above), followed by the two-character symbol `=~` (*equal sign* = followed by *tilde* ~) that is said to *bind* the variable to the translate function. Next comes `tr`, specifying the translate function itself. Finally, we have two *equal-length* lists of characters, enclosed and separated by forward slashes /. The first list is called the *search* list, and the second is called the *replacement* list.

When executed, the translate function searches the string held in the variable to find any occurrence of a character that is in the search list. If it finds such a character, it replaces it with the character in the corresponding position in the replacement list. So in the program above, any t in **$nucstring** gets replaced by u, and any T in $nucstring gets replaced by U. The value of $nucstring is now changed. You should run the above program to see the effect. Do it now.

Since the value of $nucstring is changed by the translate function, if you want to have both the DNA and the RNA strings, you should assign $nucstring to a new variable, say $dnastring, before the translate statement is executed. That assignment would be $dnastring = $nucstring; The effect is that the value of the $nucstring is assigned to the value of $dnastring, but the value of $nucstring is unchanged by that assignment. Then when the translate

statement is executed, the value of $nucstring is changed, but $dnastring continues to hold the original string.

Exercise 1.3 *Modify the above program to print the RNA string above the DNA string.*

Exercise 1.4 *Consider the following program:*

```
$silly = 'tttTTT';
$silly =~ tr/tT/Tt/;
print $silly;
```

If the tr function works by first changing every t in the string to T, and then every T in the current string to t, the end result will be ttttt. Is that your understanding of what the tr function should do? Try out the program? What do you learn from this experiment?

Now, suppose we had such a mixed-case string and wanted to convert all the characters to lowercase. We could use the translate function again (although we will later see an easier way in Perl) as follows:

```
$dnastring =~ tr/ATCG/atcg/;
```

This translate statement is pretty simple since the DNA alphabet only has four characters. But what would we do for a amino acid string? The amino acid alphabet has 20 characters (natural amino acids). We could explicitly list them all in the same way we did for the DNA alphabet, but that would be tedious. A similar problem arises if we have a normal English string over an alphabet of 26 letters. We wouldn't want to specify the entire alphabet (twice even) in the translate statement. Fortunately, the translate function allows one to easily specify a *range* of characters. For normal English strings we would write

```
$englishstring =~ tr/A-Z/a-z/;
```

where *$englishstring* is a variable holding some string that we have not shown.

The dash - between A and Z indicates that we are specifying all the characters that (naturally) come between A and Z in the ASCII alphabet [ref XXX]. So A-Z specifies all the uppercase English letters, and a-z specifies all the lowercase English letters.

The amino-acid alphabet has twenty of the normal English characters but omits some characters such as B. So the same statement will work fine to translate any of those twenty characters that occur in an amino-acid string. Assuming that the string is held in $proteinstring, the following statement works:

```
$proteinstring =~ tr/A-Z/a-z/;
```

You may wonder why a nucleotide or amino-acid string would have both upper and lowercase characters. Different cases are often used to indicate some information about the residue in that position. For example, an uppercase character might indicate a high-degree of certainty that the residue there is the correct one, or it might indicate that it is a highly conserved residue in many different organisms.

A Real Example

Here is a real situation that we recently encountered. We have a program, call it P, that takes in and proceses amino acid sequences containing both lower and upper case characters. Those sequences were derived from a *reference* amino acid sequence. An upper case character in an input sequence indicates that the character in that position is the same as the character in the corresponding position in the reference sequence. A lower case character indicates that the two corresponding characters not the same. Now for the problem. We recently obtained amino acid sequences with both upper

and lower characters, but the person who created those sequences used lower and uppercase characters exactly opposite to the way we do. That is, they use a lowercase character to indicate agreement with the reference sequence, and an uppercase character to indicate disagreement. Those sequences are unusable in our program P because P cares which characters agree with the reference sequence and which don't. So what to do?

The simplest thing is to change the given sequences, before feeding them to program P, so that every lowercase character changes to uppercase, and every uppercase character changes to lowercase.

Exercise 1.5 *Write a Perl program to do this conversion for the following sequence:*

VRNrIAEelslrrFMVALILdIKrTPgNKPriaemICDIDtYIvEa

Worrying about such mundane issues, as in this example, is not science, but it is the kind of annoying clerical thing one frequently has to take care of in order to do science. Perl makes it easy. For comparison, you might think about how you would try to do this conversion just using a standard word-processor, and how much time and aggrevation it would require.

More features of the translate function

The translate function has several additional useful features. To explain the first one, suppose we want to check for any "improper" characters in a protein string, i.e., characters other than the twenty natural amino acids. In fact, we will *count* the number of improper characters that occur. So we digress for a moment to talk about counting characters, and some of the issues that raises.

When the translate function executes, it also counts the number of times a translation occurs. That number can be explicitly assigned to a new variable by writing that new variable and an equal sign before the entire translate

statement. For example, if the variable $proteinstring holds a string, then the statement

```
$capcount = $proteinstring =~ tr/A-Z/a-z/;
print ``$capcount'';
```

will replace all the uppercase characters in $proteinstring with their lowercase equivalents, and keep a count of the number of replacements made. That count is then assigned to the variable **$capcount**. Note that the execution flow in this statement is *right to left*. Once the count is in $capcount, it can be used in any way a number can be used, or printed out with the statement:

```
print ``$capcount";
```

How does the count really get assigned to $capcount? Every function has some value that it *returns*, but it also may cause some changes that are considered as *side effects*. It is often helpful to keep this distinction clear. In the case of the translate function, the value it returns is the number of characters replaced. The actual replacing of characters is a side- effect of the translate execution.

If we make an explicit assignment of the translate statement to an explicit variable, as we did above, then the "return" is to that variable. But even without an explicit assignment, the translate statement "returns" a value that can be used. For example if print immediately precedes the translate statement, as in:

```
print $proteinstring =~ tr/A-Z/a-z/;
```

then the number returned from the translate statement is used as input to the print function, and the result is exactly the same as the previous code. Try it. This kind of shortcut (eliminating an explicit assignment to an explicit variable) is common in Perl and one of the ways that short, compact code is written in Perl. We will learn many such shortcuts.

Scalar Variables: a small digression

Before we return to the question of "improper" characters in an amino acid string, note that the variable $capcount above was assigned a *number* rather than a string. This is the first time we have seen such an assignment. In Perl, a variable that starts with $ is called a *scalar* variable, and it can hold *either* a string or a number. Moreover, the number can either be an integer or a real number with a decimal point. How does Perl know whether to treat it's value as a number or a string? The answer is *context*. We will see more examples of this later.

Back to counting improper amino acid characters

Now we are ready to address the question of counting the number of "improper" characters in an amino acid string. The improper characters are "B, J, N, O, U" and their lowercase equivalents. The following program counts and prints the number of improper characters in the string assigned to $proteinstring, without changing them.

```
$proteinstring = 'CLKJLDiCXLklJDMXLKJdLmxLEIKXMDLKJcMDLKJXMeNNlB';
print $proteinstring =~ tr/BJNOUbjnou/BJNOUbjnou/;
```

Exercise 1.6 *Often people divide the twenty amino acids into a few groups according to some properties of the amino acids. Each group is given a single symbol and each amino acid in a group is relabeled by its group symbol. This establishes a smaller alphabet for the amino acids, and several programs expect that the input amino acid sequences will be written in this alphabet. For example, one grouping of the amino acids uses properties of structural similarity, putting the residues {A, C, G, P, S, T, W, Y} into one group (the Ambivalent structure group), the residues {R, N, D, Q, E, H, K} into a second group (External group), and the residues {I, L, M, F, V} into a*

third group (Internal group). So we can recode the twenty amino acids into a three-letter alphabet {A, E, I} based on group membership.

Write a Perl program that translates a standard amino acid sequence into a sequence using this three-letter alphabet. Print out the initial and the translated sequences.

Try out the program on the following string:

SEETQMRLQLKRKLQRNRTSFTQEQIEALEKEFERTHYPDVFARERL

Yet more about the Translate Function

We've learned a lot about the translate function, but we aren't done yet. Up until now, we have used search and replacement lists that were of equal length. But the two lists are allowed to be of different lengths. How does Perl treat them?

If the replacement list is shorter than the search list, and there are no *optional characters* (which we will discuss next) in the translate statement, then Perl treats a shorter replacement list as if it has multiple copies of its last character. Often the last character is some default or catch-all character, and things are simplified by not having to write it out multiple times in the replacement list. For example, if we want to change all improper characters in an amino acid sequence $proteinstring to X, we could write:

```
$proteinstring =~ tr/BJNOUbjnou/X/;
```

In addition to this use of unequal search and replacement lists, Perl has three optional characters that can be appended to the end of `tr` to change the way it works. These are c, d, s. If you append a d at the end of the translate statement, after the third /, then a shorter replacement list causes a different effect: all characters found in the searchlist that do not have a corresponding character in the replacement list are deleted. To see this, try out

```
$proteinstring = 'CLKJLDiCXLklJDMXLKJdLmxLEIKXMDLKJcMDLKJXMeNNlB';
$proteinstring =~ tr/BJNOUbjnou//d;
print ``$proteinstring";
```

In this example, the replacement list is empty, so any character in the search list (an improper amino acid character) that is found in $proteinstring will be deleted.

Often, you want to translate, or delete, all the characters that are *not* in the search list. For that, you add the optional character c after the third / in the tr statement. For example, suppose the variable $dnastring should hold a lower-case DNA string, and you want to flag any position that doesn't contain one of the characters a, t, c, g. The following changes all such "improper" characters to X:

```
$dnastring =~ /atcg/X/c;
```

Try it out in a program to see it's effect.

If you want to delete all characters *except* those that are in the searchlist you can use the c and d options together with an empty replacement list.

Exercise 1.7 *Would it be a good idea to write code using the c option in order to delete improper amino acid characters?*

The third optional character s, replaces any *consecutive run* of identical replacement characters by a single copy of that character. This is called the *squash, squeeze or scrunch* option, hence s for short. The example below will help make its use clearer.

A small example

The following program first assigns a string to three variables (yes this is legal in Perl). Then, the first translate statement replaces all "improper" DNA

characters with X's. The second translate statement deletes the improper characters. The third translate statement replaces each consecutive run of "improper" characters with a single tilde character. Run this program.

```
$string3 = $string2 = $string1 =
'albctttpaglkmelkgghtaatctxmljeixjcaelkjxm';
$string1 =~ tr/atcg/X/c;
$string2 =~ tr/atcg//cd;
$string3 =~ tr/atcg/~/cs;
print "$string1 \n$string2 \n$string3";
```

Question What would be printed if the following statement were added to the end of the program above?

```
$string1 =~ tr/X/~/s;
print "$string1\n";
```

The following is a short synopsis of the translate function's syntax.

SYNOPSIS *the translation function:*`tr`

tr/SEARCHLIST/REPLACEMENTLIST/[c][d][s]

Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced.

OPTIONS
c - complement (i.e. invert) the SEARCHLIST
d - deletes all characters found in SEARCHLIST that do not have a corresponding character in REPLACEMENTLIST
s - squeezes any consecutive run of characters that are translated into the same character into one occurrence of this character.

To use `tr///` you provide it with a SEARCHLIST, a REPLACEMENTLIST, and a string to operate on. The searchlist is simply a set of characters each with a corresponding character in the replacement list. To save on typing Perl lets you use a dash (-) to designate a range of characters such as 0-9 or A-Z.

Exercise 1.8 *Write a program, using* `tr`*, to complement (in the Watson-Crick sense) any DNA sequence. That is, a,t,c,g or their upperclass equivalents, are changed to t,a,g,c respectively. Write another program that translates a DNA string into the RNA string that would be transcribed from it. Don't worry about 5'-3' orientation. That is, just translate characters left to right. (If you don't know about transcription of DNA into RNA, look it up in an introductory molecular biology text. If you don't know about 5'-3' orientation, don't worry about it here.)*

Exercise 1.9 *Using* `tr`*, write a program to count the number of* purines *in a DNA sequence, without changing the sequence?*

What's a purine? For our purposes, it is one of the characters A or G, as opposed to the pyramydines *C and T.*

A Little IO

In the programs we've seen so far, variables were assigned their values inside the program. For all the string assignments, the programmer wrote out a specific string in single quotes when the program was written. That is much too limiting to be of value. We want programs that can process any string at the time the program is *executed*, not just strings specified at the time the program is written. To do this requires learning a little bit of IO, *input-output*. The following program demonstrates a simple way to read in one string from the terminal and assign it to a variable. Notice that we have included comments in this program (we are starting to introduce good programming hygiene here).

```
#!/pkg/bin/perl
#This program reads in one string and then uses the tr function to
#count and replace all the uppercase characters with the
#character X.
#
print "Enter a DNA string where case matters, and hit return\n";
$text = <STDIN>; #Here is the crucial IO statement.
chomp $text;  #Remove any ending new-line character from the input string.
print "The original sequence is $text\n";

$count = $text =~ tr/ATCG/X/;
print "There are $count capital characters in the original sequence.
The sequence now is $text\n";
```

Exercise 1.10 *See what happens when the chomp line is commented out. Explain the result.*

Perl has funny, but very convenient, syntax for reading in a line of text. The basic elements of the syntax are the two *angle bracket* characters `< >`, either

written next to each other, or enclosing some word. In either case, when these two symbols are encountered in a Perl program, a single line of text (for now, and we will discuss exceptions later) is read in, from somewhere, to somewhere.

In the above program, the single line of text is read in from the keyboard. Perl knows when the line ends (even if your system uses wrap around lines) when the user presses the *enter* key. That keystroke is also entered as part of the input line, and is called the *newline* sequence (or sometimes a newline *character*). Perl knows it should read from the keyboard because of the word STDIN found between the two angle brackets. STDIN is a *special variable* in Perl whose value indicates where the STanDard INput is. By default, the value of STDIN is the keyboard, so unless you change it (and we won't), the statement

```
$text = <STDIN>;
```

causes one string typed at the keyboard and ended with the enter key, to be assigned to the variable $text.

Since that string includes the newline sequence at the end, if you don't want it, you have to explicitly delete it. That is what the chomp function does. Perl could have been designed to not include the newline sequence, but there are probably some good reasons to leave it in.

(Getting a little ahead of ourselves, a common problem that occurs when comparing two strings that one expects to be identical, is that one string still has the newline sequence in it, while the other does not. More times that we would like to admit, we've been stuck not knowing why our program doesn't work right, before discovering this simple error. So, it's a good idea to always chomp your strings on input.)

There is an alternative function `chop` that removes the last character of a string, regardless of what that character is. Older versions of Perl only had the chop function, but for inputing strings, `chomp` is safer since it is designed to only remove the newline sequence.

Trimmer, Slimmer Programs using Defaults

Below is a short program to read in one DNA string, replace any capital letters with X, and then print the resulting string.

```
#!/pkg/bin/perl
print "Input a DNA sequence\n";
$_ = <>;
chomp;
tr/ATCG/X/;
print;
```

Compared to the way the previous program read in, translated, and printed a string, this program has several significant differences.

First, we have replaced `<STDIN>` with `<>`. When the angle brackets have nothing inside them, Perl's default assumption is that STDIN is desired, and so it operates as if STDIN is explicitly written inside the brackets. Second, the input string is assigned to the funny-looking variable `$_`. This is the most important of several special variables built into Perl. It is the *input* special variable. Assigning a value to `$_` allows us to omit explicit variables in the next three statements. Whenever a function is missing an explicit *input*, Perl takes the input from the current value (assuming it has one) of the special variable `$_`. Hence the last three statements have the same effect as the following more explicit code:

```
chomp $_;
$_ =~ tr/ATCG/X/;
print "$_";
```

The *input* special variable `$_` takes its name from the fact that it's current value is used as *input* to any encountered function that calls for input, when no explicit input has been given to the function.

The use of default variables is one of the ways that people can write very short Perl code. A few keystrokes saved here, a few there, and it adds up over time.

The use of default variables also makes Perl more like a natural language because one can refer to a (default) variable implicitly as we did above. Perl knows by the context what default variable you mean. This is similar to a natural language where the subject of a discussion is sometimes established implicitly, and once the subject of discussion is established, one doesn't have to explicitly mention it in each sentence. If we are already talking about John, we can say "He went to the store and bought bread but forget butter", instead of "John went to the store, and John bought bread, but John forgot to buy butter". Context and defaults simplify natural language. Perl was designed by a linguist who built in the use of defaults and contextual smarts.

Default variables and side-effects allow more of the program to be specified implicitly, compared to super-explicit (some say uptight) languages such as C++. For many people, defaults and side-effects are some of Perl's most appealing qualities, but for others, they are some of Perl's most appalling qualities. We are in the first group.

A vital program development shortcut

When you are writing a program and trying to get it right, you often have to repeatedly try out the program with some data, check the results, and then modify the program. You sometimes even have to modify the data in order to test out all aspects of the program's behaviour. In this process, you don't want to repeatedly type in the required input from the keyboard - that is tedious and error prone. Even if you put the input in it's own file (we will learn how to read from a file next), you may then have to edit two files, or simultaniously examine two files, the program file and the data file. Perl allows a convenient way to put the data in the same file as the program, and have the read statements access that data. Simply write `__DATA__` at the end of the program; then any lines written below that point are considered lines of data. Next, in any read statement where you want an input line to come from that data, put the word DATA inside the angle brackets, i.e., `<DATA>`. Each execution of such a read statement inputs the first line of data that has not yet been read in. As usual, each execution of a read statement inputs a single line into a string variable.

Try out the following example:

```
#!/pkg/bin/perl
print "Input a DNA sequence\n";
$_ = <DATA>;
chomp;
tr/ATCG/X/;
print;
print "Input a protein sequence\n";
$_ = <DATA>;
chomp;
print;
__DATA__
acTGGgactATggCCGat
PRETEINSEQVENCE
```

1.3.1 A little file IO

At this point we can introduce simple reading and writing from and to a *file*. Even for developing simple programs, it is very useful to have the output of the program written to a file, so that you can study the output to find any errors in the program.

Suppose first we have a file named *myinputfile* in the current directory, and we want to read lines from it. In Perl, we must *open* the file and assign a *filehandle* to use in the program when refering to the file. We can make up any name we like for the filehandle. Let's use *IN* (not a very creative name, but commonly used). To open the file, we put the line

```
open IN, 'myinputfile'
```

anywhere before we try to read from the file. Then to read a line from *myinputfile*, simply put the filehandle `IN` inside the angle brackets of a read statement. This is similar to the way a line is read in using the handle `DATA`.

For example, the code

```
#!/pkg/bin/perl
open IN, 'myinputfile';
$_ = <IN>;
chomp;
tr/ATCG/X/;
print;
$_ = <IN>;
chomp;
print;
```

will result in exactly the same output as the previous code, as long as the first two lines in the file *myinputfile* are *acTGGgactATggCCGat* and *PRETEIN-SEQVENCE*. In general, each time a `read` statement is executed using the same file handle, the next line in the file is read. If there are no remaining lines to read, then the read fails (we will discuss what to do with failed reads later).

Note that in the above code, the name of the file to read is written explicitly in the program, so one would have to change the program to change the specified file. To make the program more versatile, one can specify the file to read at the time the program is executed, rather than at the time the program is written. To do this, you write the program so that it reads in the name of a file into a variable, say *$filename*, and then put the variable *$filename* in double quotes in the open statement. Because of the double quotes, Perl *interpolates* the variable `$filename` (remember interpolation?), and hence knows which file to open and associate with the file handle. Here is the begining of the previous program, done this way:

```
#!/pkg/bin/perl
print "Please input the name of a file to be read.\n";
$filename = <>;
open IN, "$filename";
$_ = <IN>;
```

Exercise 1.11 *Write the complete program to read in a file name, open the file, read and process two lines from the file as above. Don't use the file name $filename, and don't use the filehandle IN.*

Writing *to* a file is similar, but with some small differences. The first difference is in the `open` statement, when opening a file to print to. In that statement, you must put the character `>` before the name of the file, or the name of the variable holding the file name. The second difference is in the `print` statement. Between the word "print" and the string to be printed, you must write the file handle of the file where you want the output printed. The file-handle is written with no commas on either side of it.

For example, consider the following program:

```
#!/pkg/bin/perl -w
print "Please input the name of a file to be read.\n";
$infile = <>;
open IN, "$infile";
print "Please input the name of the file to print to.\n";
$outfile = <>;
open OUT, ">$outfile";
open OUT1, '>myoutputfile';
$line = <IN>;
$line =~ tr/ATCG/X/;
print OUT "$line";
print OUT1 "$line";
close (IN);
close (OUT);
close (OUT1);
```

The input comes from a file whose name is specified when the program is executed, and the output is sent to two different files, a file whose name is also specified at execution time, and another file whose name, `myoutputfile`, is literally written into the program. Note the use of double quotes and single

quotes in the different `open` statements. At the end of the program, all three files are closed.

Exercise 1.12 *Try out the above code, using file names and data that you create. Then change the double quotes to single quotes in the statement* `open OUT` *and rerun the program. Explain the result. Next change the single quotes to double quotes in the statement* `open OUT1`*, rerun and explain the results.*

1.4 A little iteration

Much of the power of a computer programming comes through the ability to repeat or *iterate* some statement or block of statements. This is called *iteration*. As in other programming languages, Perl has multiple ways to specify iteration, but the `while` statement in Perl is particularly easy to use when one needs to read data in from a file. This is a vital task in almost all bioinformatics applications, so here we introduce this use of the `while` statement; later we will discuss the `while` statement in more generality, and fully discuss other forms of iteration.

1.4.1 Reading until done

We have seen how to read and process a single line of text from a file. More often, we need to read and process all the lines in the file. That often involves repeating the same read and processing statements that we used for a single line, on all lines in the file. To do that, we put the original `read` statement in parentheses, place the word `while` in front of it, and then enclose in curly brackets, { and }, all the statements that are to be executed each time a new line is read. All the code between those two curly brackets creates a *block*. For example, we can modify the previous program in this way, to create the following code.

```
#!/pkg/bin/perl -w
print "Please input the name of a file to be read.\n";
$infile = <>;
open IN, "$infile";
print "Please input the name of the file to print to.\n";
$outfile = <>;
open OUT, ">$outfile";
open OUT1, '>myoutputfile';

while ($line = <IN>) { # the left bracket for the while block
     $line =~ tr/ATCG/X/; # note that we indent everything in the block
print OUT "$line";   # to make reading it easier.
print OUT1 "$line";
} # the right bracket for the while block

close (IN);
close (OUT);
close (OUT1);
```

The `while` statement works as follows: If the `read` statement succeeds, i.e., is able to read a line of text from the file, then all the code in the attached block is executed. If the `read` statement fails, then the code in the block is skipped, and the program moves to the line of code just after the end of the block.