Chapter 1

More Perl

Second Notes: Perl for Bioinformatics

D. Gusfield, K. Stevens

1.0.1 Another useful IO idiom

Perl has several classic "idioms", i.e., very useful, widely used constructs whose meaning is not immediately obvious. One of the most important idioms is:

```
while (<>) { a block of code }
```

To explain this idiom, assume that it is in a program called whidiom.pl and that the command line that calls whidiom.pl also contains several file names after it "whidiom.pl". For example, the command line might be

```
>perl whidiom.pl file1 file2 file3
```

When the above code is executed in whidiom.pl, each line in the files file1, file2 and file3 is consecutively read and processed in the block of code that follows the `while` statement. As a complete example, the following program will read and print every line of every file named on the command line.

```
# This is program whidiom.pl
while (<>) {
  print "$_";
}
```

Of course, the block of code following the `while` will generally be more complex, as we will see later.

More on multiple files

The above idiom is one way to process multiple files, but it requires that the files be named on the command line. That is not always convenient. Suppose instead we have a file, called say *filenames*, that lists the names of many files, each one on a seperate line. We want to again sequentially open each file and process each line of each of those files. For concreteness, let's just try to print out each line of each file. The following program will do that.

```
# read and print each line of each file in a file called filenames.
open (IN, 'filenames');
while ($file = <IN>) { # start of first while block
  open (FILE, $file);
     while ($line = <FILE>) { #start of second while block
         print "$line";
     } # end of second while block
  close (FILE);
} # end of first while block
```

There are several new things to point out here. Note the use of a second `while` statement nested inside of the first one. The indentation of the code and the placement of the curly brackets helps to see the nesting. Such indentation is crucial in creating readable (and understandable) programs.

Exercise 1.1 *Modify the above program so that the file that contains the file names is asked for by the program, and the user then types the name in during the execution of the program. Also, do you see any Perl shortcuts that can be used in this program?*

Chapter 2

Regular Expressions and Pattern Matching

Regular Expressions are one of the key features that make Perl so useful in text processing, and hence in bioinformatics.

2.1 Introduction to Regular Expressions

A *regular expression* is a string, also called a *pattern* or a *motif* in bioinformatics contexts, that defines a *set* of strings. The defined set might be as small as a single string, or it might be as large as an infinite set of strings. Any specific string that is in the defined set is said to *match* the regular expression, and we say that the regular expression *matches* that string.

Regular expressions are most often used in Perl together with *match* and *substitute* operators, to efficiently find occurrences of (complex) patterns in long strings, and then to extract or modify those occurrences. This is similar but to, but much more powerful than, the *find* and *replace* options built into most word processors. Regular expressions find many other uses in Perl as well.

In this section, we will learn the basic elements of regular expressions in Perl, and illustrate these elements with examples from bioinformatics.

2.1.1 A first simple example

The simplest kind of regular expression (or pattern) is a *literal string*, such as *TATA* or *ELVIS* (which appears in 27 protein sequences listed in the database SWISSPROT in 2002). A literal string is a string that does not contain any of the symbols

```
. \ | ( ) [ ] { } ^ $ * + ?
```

Those symbols are called *meta-characters*, and the special role of these symbols will be explained throughout this chapter.

We said above that a regular expression defines or matches a *set* of strings. In the case of a regular expression that is a literal string, the set of matched strings is just that single literal string. So, the set of strings that the regular expression *TATA* matches consists only of itself. Such simple regular expressions are not very interesting, but they are still quite usefull because any literal string can be used to search for matches in a longer string. For example, the following *match* statement causes Perl to search for the literal string *TATA* in whatever string is held in the variable $varstring:

```
$varstring =~ m/TATA/;
```

The character `m` after the `=~` is called the *match* operator, and the symbols `=~` bind the variable `$varstring` to the match operator. The two forward slashes, enclose the regular expression that is being matched. This is similar to its use in the `tr` function.

Be careful to always write the tilde `~` *immediately* after the equal sign `=`. Try it with a space in between to convince yourself that it doesn't work. We have seen many students make this error and waste a lot catchin finding the bug. Of course, you should always run your program with the `-w` option. Perl will then alert you that something odd is happening in the line that has `= ~`. The actual message you get in this case is not very meaningful, but you will be alerted to look closely at the line.

The match statement returns a value of *true* if the pattern *TATA* is contained somewhere in the string held in `$varstring`, and returns *false* otherwise. More formally, the statement returns true if and only if the pattern matches some substring of the string held in `$varstring`.

A match statement by itself is not interesting unless you use the result it returns. The simplest way to use the result is inside of an *if* statement as follows:

```
if ($varstring =~ m/TATA/) {
   print "success\n";
}
```

The meaning of this statement should be obvious even without much explanation of the *if* statement. The match statement is enclosed in parentheses and returns either true or false. If "true" is returned, then the code in the block that follows (enclosed in matching curly brackets) is executed. If the value returned is "false", then the code in the following block is skipped. As a result, the word `success` is printed if and only if the string held in `$varstring` contains *TATA*.

A simpler way to write this code is:

```
print "success\n" if $varstring =~ m/TATA/;
```

Many people find it more natural to allow the `if` clause to come after the print statemtment. Here is a complete program to read in a string, search for *TATA* and print the input string if *TATA* is found in it:

```
print "Input a string please\n";
$varstring = <>;
print "$varstring\n" if $varstring =~ m/TATA/;
```

Try out the program with different choices of input strings.

Before going on, we note another Perl shortcut. The following program does the same thing as the previous one.

```
print "Input a string please\n";
$_ = <>;
print if /TATA/;
```

The program uses the default input symbol `$_` and also leaves out the `m` in the match statement. We will explain later.

2.2 The basic elements of a regular expression

Now that we have seen a simple regular expression and an example of its use in pattern matching, we present a more complete definition of regular expressions, and give additional examples of their uses.

Regular expression have the nice property that they can be defined *recursively*. That means that complex regular expressions can be defined in terms of simpler ones, and that more complex regular expressions can be built up from simpler ones.

Regular expression are widely used outside of Perl, as well as in Perl, so we begin with recursive rules that coincide with the broadest notion of what a regular expression is. Those rules will be sufficient to define many of the regular expressions used in Perl, but not all. Later, we will show several ways that Perl can extend those regular expressions, making it much easier to write complex expressions.

Rule 1 (literal characters): Any character from the literal alphabet is a regular expression. The literal alphabet is any character other than a metacharacter, i.e., other than the characters:

```
. \ | ( ) [ ] { } ^ $ * + ?
```

So, for example, the single character `T` is a regular expression, as is the single character *A*.

Rule 2 (concatenation): Any concatenation of two regular expressions is a regular expression.

This is our first example of a *recursive definition*. Using it repeatedly we see that any string consisting of characters from the literal alphabet is a regular expression. We have already seen one such example, the literal string *TATA*, which is the concantenation of four smaller regular expressions.

Rule 3 (choice) Two regular expressions which are seperated by the symbol `|` form a regular expression. The symbol `|` has the meaning of "or".

For example, *TATA* and *tata* are two different literal strings, and hence are two different regular expressions, so the string `TATA|tata` is a regular expres-

sion. The Perl match statement

```
$varstring =~ m/TATA|tata/;
```

will return *true* if the string held in `$varstring` contains either `TATA` or `tata`. Of course, it could also contain both strings. The program

```
if ($varstring =~ m/TATA|tata/) { print "$varstring\n"; }
```

prints the whole string contained in `$varstring` if and only if it contains *TATA* or *tata*. Try this out before going on.

Rule 4 (parenthesising) Any regular expression enclosed by parentheses (one left and one right, as used to enclose this phrase) is a regular expression.

For example, `(TATA)` is a regular expression, as is `(TATA|tata)`. There are several reasons for parenthesising a regular expression. The first one is discussed in the next section.

Rule 5 (repetition) Any regular expression followed immediatly by the symbol `*` or `+` is a regular expression.

The meaning of `*` is "repeat the closest preceeding regular expression *zero* or more times".

The meaning of `+` is "repeat the closest preceeding regular expression *one* or more times".

For example, `TA+GG` is a regular expression that matches any string consisting of one T followed by at least one A followed by two G's. So if `$varstring` has the value *ATATAAAGGGA*, the statement:

```
if ($varstring =~ m/TA+GG/) { print "YES\n"; }
```

will print `YES`, because `$varstring` contains the substring *TAAAGG*, which matches the regular expression.

The repetition symbol `+` applies to the regular expression *A* and not to the longer regular expression *TA* because *A* is the preceeding regular expression

closest to `+`. So if `$varstring` holds *TATATAGG*, then the regular expression matches only *TAGG*, rather than the whole string. We will see shortly how to modify the regular expression so that it matches the whole string.

Exercise 2.1 *Write a full program to take in a string and then determine if the string contains a substring that matches* `TA+GG`. *Try the input ATATAAAGGGA; then try out the input string ATATGGGA. Then change the symbol* `+` *to* `*` *in the regular expression and try out the two input strings again. Explain the results.*

The above five rules define the basic elements of regular expressions and allow one to define surprisingly complex sets of strings. These basic elements are common to almost all definitions of regular expressions, whether in Perl or outside of Perl. However, Perl has several ways to enrich the *syntax* of regular expressions, allowing one to write complex regular expressions more easily. Perl also has several ways to extend the *semantics* of the basic regular expressions, i.e., to define sets of strings that are not definable using only the five rules discussed above.

We will discuss most of the enriching syntax of Perl regular expressions and some of the semantic extentions as well. But first, we will look at examples of what can be done just with the basic elements of regular expressions defined so far.

2.2.1 Parenthesization with Repetition

Our first example of the use of parentheses in regular expressions is to specify a longer or more complex pattern for repetition.

In the regular expression `GAT+A`, the repetition symbol `+` applies only to the letter `T`, the closest regular expression preceeding `+`. But suppose we want the repetition to apply to `AT`, i.e., we want to specify a set of strings consisting of the character `G` followed by one or more concatenated copies of `AT` followed by `A`. That set is specified by the regular expression: `G(AT)+A`. The parentheses around `AT` make `(AT)` the closest regular expression preceeding `+`, and so `+` applies to `(AT)`. Then, for example, if `$varstring` holds *ATAGATATATATAAG*, the statement

```
$varstring =~ m/G(AT)+A/;
```

matches the substring *GATATATATA* and returns the value of *true*;

Exercise 2.2 *Modify the regular expression presented in the discussion of Rule 5, so that the new regular expression matches all of TATATAGG, and not only TAGG.*

Note that neither symbol (nor) are in the string *ATAGATATATATAAG*. This shows that the match statement above does not look for those parentheses in a string. The parenthases are not used as *literal* characters in this regular expression. Rather, they are used as *meta-characters* that help to specify a more complex set of strings.

Later, we will discuss what to do if we want to match a literal left or right parenthasis in a string.

As a more realistic example of parenthesization with repeition, consider the desciption of "low-complexity" regions in a DNA string. Briefly, a low-complexity region is a region in the DNA that is highly repetitious, often consisting of many repeats of a simple string. For example, the two-nucleotice string `AC` is often repeated hundreds of times in a row in DNA. Such CA-repeat regions have little importance in our understanding of molecular biology, and hence many sequence analysis or search tools will "mask out" these regions, replacing them with a single symbol such as an X.

The following program determines if the string in `$varstring` contains an `AC` repeat region consisting of at least one hundred consequitive copies of `AC`.

```
if ($varstring) =~ m/(AC){100}/) {
   print "Yes the string contains at least 100 consecutive copies of AC\n"
   }
```

2.2.2 Examples of Regular Expressions from PROSITE

PROSITE is a database of regular expressions used to describe and identify important protein families and proteins domains. Proteins or protein domains belonging to a particular family generally share functional attributes

and are derived from a relatively recent common ancestor. Each family or domain is described by a regular expression. Presently there are roughly two thousand regular expressions in the Prosite database.

Each regular expression in Prosite is derived from a set of sequences by using the fact that some regions in the protein sequences have been better conserved than others during evolution. Those regions are generally important for the function of a protein and/or for the maintenance of its three-dimensional structure. By analyzing the constant and variable regions of such sequences, it is possible to derive a regular expression for a given protein family. Ideally, that regular expression should match a substring in each member of the family, but not match any substring in the sequences of proteins not in that family. Thus, the regular expression can be used to search new protein sequences to determine if a newly sequenced protein is a likely member of the family. Proteins identified in this way can then be explored by addition investigation to see if they really have the functional or structural features that are characteristic of that family.

As a first example from Prosite, consider the Endoplasmic Reticulum Targeting Sequence described as follows in the PROSITE documentation (It's Prosite accession number is PDOC00014):

> Proteins that permanently reside in the lumen of the endoplasmic reticulum (ER) seem to be distinguished from newly synthesized secretory proteins by the presence of the C-terminal sequence Lys-Asp-Glu-Leu (KDEL). While KDEL is the preferred signal in many species, variants of that signal are used by different species.

Thus, `KDEL` is a regular expression consiting of a literal string, but it can identify many ER proteins. However, because there are known varients of the KDEL motif in proteins that reside in the ER, a more general regular expression for the targeting sequence has been developed in Prosite. That expression consists of one amino acid from the set {K,R,H,Q,S,A} followed by one amino acid from {D,E,N,Q} followed by EL. Thus there are $6 \times 4 = 24$ amino acid sequences described this way. This set of 24 sequences can be defined by the following Perl regular expression:

```
(K|R|H|Q|A)(D|E|N|Q)EL
```

Exercise 2.3 *According to the Prosite documentation, all known ER protiens, except for liver esterases, contain one of the 24 strings defined by the regular expression above. The liver esterases are a bit different. Their motif is H followed by one of {T,V,I} followed by EL. Write a single Perl regular expression for that motif; then write a single Perl regular expression for the "or" of the two regular expressions for ER proteins; then write a Perl program that takes in a sequence and checks to see if it contains any ER motif defined in this latter regular expression. How many different ER motifs are defined by that regular exprssion?*

2.3 Extensions of Perl regular expression syntax

There are many ways that Perl allows one to write complex regular expressions more simply than by using only the five regular expression rules described so far. We now introduce several of the more important simplifications.

1. The Dot Symbol

The *dot* symbol `.` in a Perl regular expression is interpreted as "any single character", and it matches any single character other than the *newline* sequence (see section X). The dot is also called a "wild-card" and is a metacharacter when used in this way.

For example, `a.t` is the regular expression that matches any string of three characters starting with `a` and ending with `t`. (You really have to keep your wits about you when reading this. The . at the end of the previous sentence is not part of the regular expression, but just an ordinary period at the end of a sentence. That kind of ambiguity will arise again, so beware.) As another example, the regular expression `a.+t` matches any string starting with `a` and ending with `t` and containing *at least* one character in between them. Because there is no limit on the length of the intermediate substring, the set of strings defined by this regular expression is infinite. The set of strings defined by `a.*t` contains all the strings defined by `a.+t`, but also contains the string `at`.

2. Character Classes

The regular expression consisting of the "or" of individual literal characters can be written in Perl more easily than was done earlier. Any set of literal characters can be listed without seperation, and enclosed by left and right square brackets `[ ]` to define a *character class*. When used in a regular expression, the meaning of the class is "take exactly one of the characters in this class". In this context, the brackets are meta-characters.

For example, instead of `(K|R|H|Q|A)`, we can write `[KRHQA]`. So, the regular expression for the ER proteins we saw earlier, `(K|R|H|Q|A)(D|E|N|Q)EL`, can be written more simply as `[KRHQA][DENQ]EL`.

Here is an example of a Perl regular expression for a Prosite pattern called a *C3HC4 type Zinc finger*, also known as a *Ring finger*:

```
C.H.[LIVMFY]C..C[LIVMYA]
```

That regular expression has four wild-card characters and two character classes. Every string defined by it has exactly ten characters.

As another example, consider the string `PPPPPPNCAG`, where `P` stands for any pyrimidine (i.e., T or C) and N stands for any DNA nucleotide (A, T, C or G). Sequences that match that regular expression mark the downstream intron-exon boundries in Eukaryotic DNA [**?**]. A Perl regular expression for that set of strings is:

```
[TC][TC][TC][TC][TC][TC][ATCG]AG
```

Exercise 2.4 *In the "universal" genetic code, there are six DNA triplets (codons) that specify the amino acid arginine. They are CGT, CGC, CGA, CGG, AGA, AGG. A regular expression that matches these six codons is* `CG[ATCG]|AG[AG]`. *Look up the triplets that code for the amino acid serine and write a regular expression that matches the codons for serine. Then write a regular expression that matches any codon for arginine or serine. Of course, one could just put an "or" between the two seperate regular expressions for arginine and serine, but try instead to write a more compact regular expression for them.*

Character Ranges

Frequently, one wants to specify a character class that defines a natural range of characters. The most common example is the range of digits 0 through 9 in order. That class can be specified in Perl as `[0-9]` instead of the longer specification `[0123456789]`. We have already seen examples of this syntax when discussing the `tr` statement in the previous chapter.

In general, any contiguous range of characters from the ASCII alphabet (see Appendix X) can be specified as a character class, as follows: Inside square brackets, specify the first character in the range, followed by a dash, followed by the last character of the range.

A character range can be used together with other ranges or individual characters to specify a large character class in the natural way. For example, the character group `[AC-IK-NP-TVWY]` specifies the twenty character amino acid alphabet, which differs from the English alphabet by the omission of the characters B,J,O,U,X,Z. That character class is specified by using three character ranges and four individual characters.

Negation of classes

Sometimes it is easier to specify the members of a character class by specifying the characters *not* in the class. That is done by writing a carot `^` at the start of the class definition. For example, `[^atcguATCGU]` is the class of all characters other than the characters used to denote DNA or RNA nucleotides. The class `[^a-z]` consists of all characters not in the lower case English alphabet.

Predefined character classes

Some sets of characters are used so frequently that they have been given a shorthand notation in Perl. The most important of these are: `\d`, `\w`, `\s`, which we explain below. Additional predifined classes can be found in appendix XX.

The two-symbol notation `\d` is shorthand notation for the digits 0 through

9. It is equivalent to `[0-9]`.

For example, the regular expression `\d+` matches any string consisting of one or more consecutive digits.

The negation of this class is also predefined in Perl and is `\D`. That is, `\D` defines the class `[^0-9]`.

The two-symbol notation `\w` is shorthand for the class of "word characters", i.e, characters that are typically found in text. This class is equivalently specified as `[a-zA-Z0-9_]`. The symbol `_` is included in this predefined class because it is a commonly-used symbol in programming variable names. Note also that the class contains the digits 0 through 9.

The negation of `\w` is `\W`.

The two-symbol notation `\s` is shorthand for the class of "white-space characters", i.e., characters that typically follow the end of a word. The most common of these is a space, but sometimes the end of a word is followed by the newline sequence `\n`, or a tab character `\t`. The entire class specified by `\s` is equivalently defined by `[ \n\r\f\t]`. Note the space at the start of the class. The characters `\r` and `\f` are rarely used and we don't know exactly what they do.

The negation of `\s` is `\S`.

Repetition of character classes

Repetition can be used on character classes. For example, the regular expression `[AT]+` defines any string that contains only characters `A` and `T`. Two examples are `AATA` and `TATA`. Hence, the meaning of `[AT]+` is not that one selects a single character from the class and then repeats that character one or more times, but rather that one repeatedly selects a character from the class.

3. Quantified Repetition

So far, the only repetition operators we have seen are `+` and `*`. However, in Perl, one can specify an *exact* number of repetitions, or a *range* of numbers.

A regular expression followed by {i} means that the regular expression should be repeated i times. A regular expression followed by {i,j}, where i and j stand for two specific numbers, and $i < j$, means that the regular expression should be repeated at least i times, and at most j times.

For example, the regular expression given above for Eukaryotic intro-exon boundaries can be written more simply as

```
[TC]{6}[ATCG]AG
```

As another example, the regular expression

(AC){6} defines the string ACACACACACAC consisting of six repetitions of the dinucleotide AC. Note the difference between using parenthases around AC and using square brackets. The regular expression (AC){6} defines a single string, while the regular expression [AC]{6} defines a set of 64 strings.

Exercise 2.5 *Recall our earlier discussion on masking out CA-repeats. We don't want to replace each* CA *with an* X*, but only those occurrences of ¶CA that are found consecutively, with say at least five copies. Try out the following code on different examples to determine if the following code will do this. Hint: it doesn't. But explain what it does do, and explain why it doesn't do what is desired. Later we will see a way to fix it.*

```
$varstring =~ s/(CA){5}(CA)*/X/g;
```

A more interesting example is the "cross-brace" pattern as defined in the Prosite database:

> C-x2-C-x(9 to 39)-Cx(1 to 3)-H-x(2 to 3)-C-x2-C-x(4 to 48)-C-x2-C

In Prosite notation, character x denotes a wild-card; repetition ranges are stated explicitly enclosed in parenthesese, and individual elements of the pattern are seperated by dashes. The cross-brace Prosite pattern can be written as the following Perl regular expression:

```
C..C.{9,39}C.{1,3}H.{2,3}C..C.{4,48}C..C
```

Here is another example: In Genbank, the major U.S. repository of publicly available DNA and protein sequences, each published sequence is given a unique *accession number*, which could be more appropriately called an accession ID. Standard Genbank accession numbers either consist of one capital letter followed by five digits, or two capital letters followed by six digits. A Perl regular expression that matches a Genbank accession number of either format is:

```
(\s\w\d{5}\s|\s\w{2}\d{6}\s)
```

Exercise 2.6 *Would it be correct to write the regular expession for Genbank accession numbers as* `\s\w{1,2}\d{5,6}\s`*? Explain.*

Exercise 2.7 *Restriction enzymes are proteins that recognize specific DNA sequences and cut the DNA molecule at a specific location in that sequence. One such recognition sequence is* `GCCNNNNNGGC`*, where the character N stands for any of the four DNA nucleotides A, T, C or G. This sequence is recognized by the restriction enzyme called BGl1. Write a Perl regular expression that matches any BGl1 recognition sequence, and write a Perl program that takes in a string and determines if it contains a BGl1 recognition sequence.*

The quantifier clause `{i,}` means that the closest preceeding regular expression should be repeated *at least i* times, but there is no upper limit on the number of times it can be repeated.

More biological examples of quantified repetition

The following example of a set of DNA transcription factors is given in [**?**]:

```
TGTGGWWWG
```

where `W` stands for either `A` or `T`. There are many ways we can write a Perl regular expression that matches this set of strings. One is

```
(TG){2}G[AT]{3}G
```

although `TGTGG[AT]{3}G` actually uses fewer keystrokes.

As another example, several diseases are related to excessive repetition of a three-nucleotide sequence in an individual's DNA. For example, Huntington's disease is associated with the expansion of the triplet `CAG` in a particular gene. Normally, it is repeated in tandem from 10 to 35 times, while in individuals with Huntington's disease, it is repeated from 36 to 121 times [**?**]. The Perl regular expression for the diseases version is: `(CAG){36,121}`

One more repetition symbol

There is one more specialized repetition symbol that is used in Perl regular expressions. It is the question mark `?`, and it can be used after any regular expression in the same places that `+` or `*` can be used. In this contex, the meaning of `?` is "use the preceeding regular expression *zero or one* times". Another way to say this is "the preceeding regular expression is optional".

As an example, we recently wanted to search an online article for any mention of regions of DNA that contain a high density of adjacent nucleotide pairs CG, i.e., C followed by G. Such a region is sometimes called "CG-rich" and sometimes called a "CpG island", and is belived to be a region that marks the start of a gene-rich segment of DNA in vertibrates. A single regular expression that captures both *CG* and *CpG* is `Cp?G`.

This example is not terribly compelling, because we could have made two seperate searches, one for CG and one for CpG. We will see the quantifier `?` used in more complex examples later.

Another (sketchy) example is in searching for amino acid sequences that vary due to *alternative splicing*. Recall that genes in Eukaryotes contain two types of DNA intervals, *introns* and *exons*. Normally, after a gene is transcribed from DNA to RNA, the introns are *spliced out* and the exons are concatenated. The concatenated exons forms the shorter sequence that is used in the creation of the protein associated with the gene (that process is called *translation*). However, in a significant number of genes, there are variations in what is spliced out. The general phenomena is called *alternative splicing* and is involved both in generating a variety of useful proteins, and in producing defective proteins. In one form of alternative splicing, some of the exons or portions of exons are spliced out in addition to the introns, so that compared to the normal protein, the resulting protein has some intervals of

amino acids missing and/or changed.

One such case is the *Ultrabithorax* (Ubx) gene in *Drosophila*. In that gene there are at least four exons that are alternatively spliced in different combinations. Different combinations of splices lead to different modified Ubx proteins, with different biological functions. Most of the combinations are functional and are seen in nature. We will concentrate on three of the alternatively spliced exons, the ones where the alternative splice causes a deletion of an interval in the corresponding amino acid sequence. For example, the first alternative splice causes the deletion of the nine-character substring *ECPEDPTKS*. The other two deletions are of length 17 each and are shown below. Hence, there are eight variations of the Ubx protein, depending on which of these three substrings have been deleted. Six of those variations are known to occur in nature.

Now to connect this Ubx story to regular expression pattern matching, consider a situation where we want to search a string of amino acids for the Ubx protein, and we want to allow for each of these eight variations. Abstractly, we would create the regular expression

```
w(ECPEDPTKS)?x(KIRSDLTQYGGISTDMG)?y(KRYSESLAGSLLPDWLG)?z
```

where `w` denotes the amino acid sequence before the first alternative spliced region, `z` denotes the amino acid sequence after the third region, and `x`, `y` denote amino acid sequences between the three alternatively spliced regions. By using `?` three times, that single regular expression matches any of the eight possible variants of the Ubx protein sequence.

Synopsis of repetition operators

Max	Min	Range	Example
`+`	`+?`	match one or more times, same as {1,}	`/a+/` match a, aa, aaa ...
`?`	`??`	match zero or once, same as {0,1}	`/ab?/` match a or ab
`.`	`.`	matches zero or more times, same as {0,}	`/ab*/` match a, ab, abb ...
`{N}`	`{N}`	match exactly N times	`/a{3}/` match aaa
`{N,}`	`{N,}?`	match N or more times	`/a{2,}/` match aa, aaa, ...
`{N,M}`	`{N,M}?`	match between N and M times	`/a{2,3}/` match aa or aaa

2.3.1 Some Examples Exersises with Answers

Example: Write regular expression that will identify an integer of any length within a text:

Answer:

`/\d+/ or /[0-9]{1.}/ and other variations`

4. Anchors

An *anchor* in a Perl regular expression specifies, or influences, where in a string a pattern should occur. The most useful anchors are `^`, which specifies the *start* of the string, and `$`, which specifies the *end* of a string.

For example, if `$varstring` contains *abcXYZ*, then the code

```
if ($varstring =~ m/^[a-z]+[A-Z]+$/) { print "Yes\n";}
```

will print "Yes", but

```
if ($varstring =~ m/^[A-Z]+$/) { print "Yes\n";}
```

will not print "Yes" because the capital letters in `$varstring` do not start at the beginning of the string, even though there is a substring of captial letters in `$varstring`.

The regular expression `^$` matches a line that is blank, something that one often wants to check for.

Exercise 2.8 *Write a regular expression that matches any string starting with lower case letters, or ending with upper case letters, and containing any characters in between the two ends.*

There is actually one small subtlety in the way the anchor `$` works. If the string ends with a newline sequence, as it would if it was read in from the terminal or a file, then the newline sequence is ignored when matching with `$`. In other words, the anchor `$` matches the end of the line just before the newline sequence. To better understand this, try out the following code:

```
print "Please type in the string dna\n";
$string = <STDIN>;
if ($string =~ m/^dna$/) {
  print "Found the match: $string";
}
chomp $string;
print "After the chomp: $string";
print "Last print";
```

Note that there is no explicit newline sequence `\n` in the three print statements. After you run the code, note also that the match was found, and that the output of the first print statement is on its own line, while the output of the next two print statments are together on one line. This shows that initially `$string` contains a newline sequence at its end (because the first print does move to a newline), but `$` matches the end of the line before the newline sequence (otherwise the first print would not have executed). After `chomp`, `$string` no longer contains a newline sequence, so the last two print statements cause the output to be put on the same line.

Exercise 2.9 *Change the statement* `if ($string =~ m/^dna$/)` *to* `if ($string =~ m/^dna` *and run the code. Explain the result.*

A real example using anchors

Here is a realistic example showing the importance of using anchors. We recently received a file containing genetic information from a collection of individuals. The actual names of the individuals had been replaced by identifiers for annonymity and simplicity. Each identifier was either a number consisting of at most nine digits, or a single letter followed by a number

consisting of at most eight digits. Each identifier was written on a line of its own, and started at the left end of the line. The program that processes the genetic information repeatedly reads one line of the file and must determine if the line contains just an identifier, or the actual genetic data. The genetic data itself also contains numbers and numbers preceeded by letters, so a match statement that just looks for numbers or numbers preceded by a letter would not distinguish identifiers from genetic data. Instead, we use the fact that each identifier is on its own line. Assuming that a line is read into the variable `$line`, the following code checks whether the line only contains an identifier.

```
chomp $line;
if ($line =~ m/^\w\d{1,8}$/) {
  print "Just an ID: $line\n";
}
else {
  print "Not just an ID: $line";
}
```

Exercise 2.10 *One might think that the regular expression used to match an identifier should be* `{\w?\d{1,9}}` *instead of* `\w\d{1,8}`*. Would that regular expression work? Would* `{\w?\d{1,8}` *work?*

In the code above, we have used a new element of the `if` statement, the `else` clause. If the match statement returns "true" then the code in the following block is executed. Otherwise (if the match statement returns "false") the code in the block following `else` is executed.

Exercise 2.11 *Modify the above code to handle the case that a line containing only an identifier might have spaces before or after the identifier. Be sure that the code also handles the original case of no spaces.*

Exercise 2.12 *Transcription of genes in E. Coli bacteria is facilitated by a promoter sequence that is roughly 35 positions upstream of the start of the gene. While the promoter sequences can differ in each gene, they typically contain two nearly identical substrings (sometimes called "boxes") seperated*

by 18 nucleotides. The single "consensus" or "plurality" promoter sequence that characterises the set of E. Coli promoter sequences is: the box `TTGACA` *followed by any 18 DNA nucleotides (A, T, C or G) followed by the box* `TATATT`.

Write a regular expression that defines the E. Coli consensus promoter sequence.

Exercise 2.13 *In the E. Coli lactose gene (or operon), the two box sequences are* `TTTACA` *and* `TATGTT`*, which are slightly different from the boxes in the consensus sequence described in the previous exercise. Write a regular expression that matches either the consensus promoter sequence or the specific lactose promoter sequence.*

Make the regular expressions as short as possible. Write a full program that reads in a long string and determines if that string contains an E. Coli consensus promoter sequence or a lactose promoter sequence. Of course, get the program running correctly.

Exercise 2.14 *In Eukaryotes (higher organisms whose cells have a nucleose), promoter sequences are more variable than in Prokaryotes, such as the bacteria E. coli. In Eukaryotes, there are several different RNA polymerases that recognized different Eukaryotic promoter sequences. One type of RNA polymerase, the RNA polymerase II, recognizes DNA sequences that*

> *consist of two segments, the ... TATA box (consensus TATAWAW, where W is either A or T) and the initiator sequence (consensus YYCARR), where Y is either C or T and R is either A or G). [?]*

The two segments in these promoter sequences are seperated by 17 nucleotides. Write a regular expression that matches this RNA polymerase II promoter sequence.

Synopsis

The following is a synopsis of some meta-characters that provide additional functionality to a regular expression.

Pattern	Description	Examples
.	matches an arbitrary character	/b.g/ matches bag and bug
^	matches the beginning of a line/string	/^bag/ at the beginning of a line
$	matches the end of the line/string	/^bug$/ line with only one bug
[␣]	denotes a class of characters to match	/[Tt]he/ match the or The
[^␣]	a negated class of characters	/[^aeiou]/ match non-vowels
(␣|␣|␣)	match one of the given alternatives	/g(oo|ee)se/ match goose or geese
\w	matches alphanumeric, including _	same as [a-zA-Z_0-9]
\s	matches whitespace	same as [/n/t/f/c]
\d	matches numeric: 0-9	same as [0-9]

The dot . is used in regular expressions as a wildcard character. It will match any single character, this means that /a.a/ will match aba, a4a, etc.

The ^ character is guaranteed to match at only the beginning of the string, the $ character at only the end (or before the newline at the end). This means that embedded newlines will not be matched by ^ or $. You may, however, wish to treat a string as a multi-line buffer, such that the ^ will match after any newline within the string, and $ will match before any newline. You can do this by using the m option on the pattern matching functions.