Chapter 1

More Perl - third notes

Third Notes: Perl for Bioinformatics

D. Gusfield, K. Stevens

1.1 More on Pattern Matching in Perl

We have already seen some uses of the *match* statement, but we now discuss matching in more detail, along with the related *substitution* statement. The following synopsys summarizes what we have seen so far, and lists *options* that can be set to modify the search.

SYNOPSYS
[EXPR `=~`] m/PATTERN/[g][i][m][o][v][x] The match function, searches EXPR for PATTERN. EXPR can be any string value or variable containing a string. The match function returns the number of matches found, if any. Any returned number greater than zero is interpreted as a TRUE when the match statement is used in a conditional statement; if no matches were made, the match statement returns FALSE. Common Options g - matches as many times as possible, but the matches are non-overlapping in EXPR i - searches in a case-insensitive manner m - treats the string as multiple lines s - treats the string as a single line Note that any option character you use is placed immediatly after the / For other options see `m//` section in appendix

SYNOPSYS
[$VAR =~] s/PATTERN/REPLACEMENT/[e][g][i][m][o][s][x] Substitute function, searches a string in $VAR for a pattern, and if found, replaces that pattern with the replacement text. It returns the number of substitutions made, if any; if no substitutions are made, it returns FALSE. Common Options g - replaces all occurrences of the pattern e - evaluates the replacement string as a Perl expression i - searches in a case-insensitive manner For other options see `s///` section in appendix

Example of Substitution: Masking Out CA-repeats

As discussed earlier, DNA sequences often contain long regions consiting of consecutively repeating CA substrings. These CA-repeats are often considered a nuscience, and are often masked out by replacing each CA with a single character, such as X, or by replacing a long region of CA repeats with a short substring.

In the next program fragment, we replace each occurrence of CA (case insensitive) in `$varstring` with the single symbol `X`.

```
$varstring =~ s/CA/X/gi;
```

Actually, we probably 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. Later, in section **??**, we will see how that can be done.

A more involved example: Extracting and converting DNA sequences from GenBank

Here we discuss a realistic and useful program that only uses simple elements of regular expression pattern matching, along with substitution.

GenBank is the NIH-sponsored and maintained repository of all publicly reported nucleic acid sequences. Sequences and assosiated data are retrieved as text files that contain a great deal of information in addition to the DNA sequence. However, often all one wants is a sequence identifier and the sequence itself. Moreover, when using a sequence as input to other programs, it is common that the sequence needs to be in a specific format, called FASTA format. FASTA format is a more compact representation of sequence data, widely used for both nucleotide and amino acid sequences. [ref file formats in appendix]

Problem Statement: We want to write a Perl program to extract the sequence name and the full DNA sequence from a GenBank report, and then convert the sequence from GenBank format to FASTA format. The output will look as follows: The first line of the output will begin with the character > followed by a comment. The comment is usually the name or the location (or both) of the sequence. In the following program, the comment will be taken from the DEFINITION field of the GenBank report. Following the comment line, each line of the DNA sequence data found after the word "ORIGIN", will be output, but with all digits and spaces removed from the GenBank-formatted sequence.

Example GenBank Entry in GenBank Format

```
LOCUS      HUMCKRASA     450 bp    mRNA            PRI       27-APR-1993
DEFINITION Human PR310 c-K-ras protein mRNA, 5' end.
ACCESSION  M35504
VERSION    M35504.1  GI:180591
KEYWORDS   c-K-ras oncogene; c-myc proto-oncogene.
SOURCE     Human (patient PR310) lung carcinoma, cDNA to mRNA.
ORGANISM   Homo sapiens
           Eukaryota; Metazoa; Chordata; Craniata; Vertebrata;
           Euteleostomi; Mammalia; Eutheria; Primates; Catarrhini;
           Hominidae; Homo.
REFERENCE  1  (bases 1 to 450)
AUTHORS  Yamamoto,F., Nakano,H., Neville,C. and Perucho,M.
TITLE    Structure and mechanisms of activation of c-K-ras
         oncogenes in human lung cancer
JOURNAL  Prog. Med. Virol. 32, 101-114 (1985)
MEDLINE  85271309
FEATURES      Location/Qualifiers
   source   1..450
            /organism="Homo sapiens"
            /db_xref="taxon:9606"
   CDS      1..>450
            /note="PR310 c-K-ras oncogene"
            /codon_start=1
            /protein_id="AAA35689.1"
            /db_xref="GI:180592"
            /translation="MTEYKLVVVGAGGVGKSALTIQLIDNHFVDEYDPTIEDSYRKQV
            VIDGETCLLDILDTAGHEEYSAMRDQYMRTGEGFLCVFAINNTKSFEDIHHYREQIKR
            VKDSEDVPMVLVGNKCDLPSRTVDTKQAQDLARSYGIPFIQTSAKTRQ"
BASE COUNT      155 a     71 c    106 g    118 t
ORIGIN
   1 atgactgaat ataaacttgt ggtagttgga gctggtggcg taggcaagag tgccttgacg
  61 atacagctaa ttgacaatca ttttgtggac gaatatgatc caacaataga ggattcctac
 121 aggaagcaag tagtaattga tggagaaacc tgtctcttgg atattctcga cacagcaggt
 181 catgaggagt acagtgcaat gagggaccag tacatgagga ctggggaggg ctttctttgt
 241 gtatttgcca taaataatac taaatcattt gaagatattc accattatag agaacaaatt
 301 aaaagagtta aggactctga agatgtacct atggtcctag taggaaataa atgtgatttg
```

```
 361 ccttctagaa cagtagacac aaaacaggct caggacttag caagaagtta tggaattcct
 421 tttattcaaa catcagcaaa gacaagacag
//
```

Example Output in FASTA Format

```
>Human PR310 c-K-ras protein mRNA, 5' end.
ATGACTGAATATAAACTTGTGGTAGTTGGAGCTGGTGGCGTAGGCAAGAGTGCCTTGACGATACAGCTAA
TTGACAATCATTTTGTGGACGAATATGATCCAACAATAGAGGATTCCTACAGGAAGCAAGTAGTAATTGA
TGGAGAAACCTGTCTCTTGGATATTCTCGACACAGCAGGTCATGAGGAGTACAGTGCAATGAGGGACCAG
TACATGAGGACTGGGGAGGGCTTTCTTTGTGTATTTGCCATAAATAATACTAAATCATTTGAAGATATTC
ACCATTATAGAGAACAAATTAAAAGAGTTAAGGACTCTGAAGATGTACCTATGGTCCTAGTAGGAAATAA
ATGTGATTTGCCTTCTAGAACAGTAGACACAAAACAGGCTCAGGACTTAGCAAGAAGTTATGGAATTCCT
TTTATTCAAACATCAGCAAAGACAAGACAG
```

Here is the program that does the extraction and format conversion.

```
#!/pkg/bin/perl -w
   #
   # Extract and Convert GenBank DNA strings to FASTA Format
   #
   # open the file
   open(INFO, 'genbankfile'); # file holding a GenBank report
   open(OUTFO, ">fastafile"); # output file

   # read and process each line of the input file.
   while ($line = <INFO>) {
     if ($line =~ m/^DEFINITION/ ) {   # search for "DEFINITION" at the
                                       # start of a line.

                                       # Remove "DEFINITION" and
                                       # any trailing  blanks.
       $line =~ s/DEFINITION +//;      # Remove "DEFINITION" and any
                                       # trailing spaces (at least one.)
       print ">$line";                 # print the first FASTA line.
                                       # notice no \n
     }
     if ($line =~ m/^[0-9 tagc]+$/) {  # this recognizes a line of
                                       # DNA in GenBank format
                                       # note the use of anchors to be sure
                                       # that the line contains only the
```

```
                                        # expected symbols.

      $line =~ tr/0-9 //d;              # remove digits and spaces.
      $line =~ tr/tagc/TAGC/;           # capitalize (optional)
      print $line;
    } # end of the if block
  }  # end of the while block
  close(INFO);
  close(OUTFO);
```

Exercise 1.1 *The above program is not very sophisticated in what it considers a permitted line of DNA data. For example, it will print a line consisting of intermixed digits and DNA letters such as aat045cg89 ccgt. Clearly, such a line is not part of any DNA sequence in GenBank. Modify the program so that it only prints lines where the digits precede the DNA characters. Then modify the program further, to make sure that the DNA characters being read are organized into groups of ten consecutive characters each, seperated by a space, until the last group which might have less than ten characters. This is to provide another check that the characters being read are really the DNA sequence in the GenBank report.*

1.1.1 Extracting a Found Pattern

One of the most powerful features of the match operator is that it allows you to extract matched substrings. The syntax for this is very simple: Enclose any part of the regular expression with parentheses. Anything that matches that part of the regular expression is assigned to a special Perl variable, `$1`. That variable can then be used in any way that a normal variable can be used. For example the following program fragment should print out *ctgaaACTaggg*.

```
$string = 'this looks like dna ctgaaACTaggg, but is it real?';
$string =~ m/([actgATCG]{3,})/;
print $1;
```

The parentheses are *meta-characters* in this context, i.e., they do not literally match themselves, but are used to identify the subpattern that will be extracted.

You can extract more than just one pattern found in a regular expression. The special variables are just `$1, $2, $3, ...`

For example, the following program extracts two subpatterns:

```
#!/pkg/bin/perl -w
#example of subpattern extraction
#
$string = 'What does accession number HU39876 represent?';
$count = $string =~ m/([A-Z]{1,2})(\d+){3,7}/;
if ($count) { print "I think the species code is $1,
the sequence number is $2,
and the whole accession number is $1$2 \n"; }
```

To figure out which special variable goes with which part of the regular expression, count left parentheses starting from the / in the regular expression. Everything from the i'th left parenthesis to its matching right parenthesis gets put into the special variable `$i`. For example, change the line

```
$count = $string =~ m/([A-Z]{1,2})(\d+)/;
```

to

```
$count = $string =~ m/(([A-Z])(\d+))/;
```

and see what happens. Then try to change the print statement so that you get the right output again. Note that this example shows that you can have a parenthesis pair *nested* inside another parenthesis pair.

In addition to capturing matched substrings by using parentheses and the special variables `$1, $2, ...` , Perl has three built-in special variables that are useful in extracting substrings: The special variable `$&` holds the most recently matched substring, `$‘` holds the part of the string to the left of that matched substring, and `$’` holds the part of the string to the right of the the matched substring. These variables get their values automatically when the match statement is executed. Try out the following program to see these special variables in action.

```
# Program specialmatch.pl
# test program to demonstrate the three special variables
# associated with regular expression matching.

$string = 'This might hold DNA, but maybe it is not dna at all';
if ($string =~ m/dna/i) {
    print "$string \n";
    print "$`\n";
    print "$&\n";
    print "$'\n";
}
```

1.1.2 The while-g idiom

At this point, we can describe another valuable Perl idiom, the *while-g* construct used in pattern matching. Suppose we want to find all substrings in a string that match a regular expression, and do something with each one. We have seen how to do this using the substitution operator, where each occurrence of a matching substring is changed into some other substring. But how do we do this if we don't want to make any substitutions? For example, suppose we just want to print out each matching substring.

As an illustration, suppose we want to find and print out each substring that looks like a GenBank accession number in a single line of text. The following code will only find and print the first accession number in `$line`.

```
if ($line =~ m/([A-Z]{1,2}\d{3,7})[ ,.;:?]/)  {
   print "The input line contains accession number $1.\n";
}
```

The next code fragment uses the *while-g* idiom and will find and print out every substring in the line that looks like an accession number.

```
while ($line =~ m/([A-Z]{1,2}\d{3,7})[ ,.;:?]/g)  {
   print "The input line contains accession number $1.\n";
}
```

What is different between this program fragment and the previous one is that we have replaced the `if` with a `while`, and we added the option `g` to the match statement.

With this idiom, matching is repeated as long as there is an additional match found in `$line`. After each match is found, the program executes the code in the block attached to the `while` statement. The matches are found successively, moving from the start of the string to the end.

Exercise 1.2 *Modify program specialmatch.pl so that it finds all the matches in* `$string` *and executes the code in the attached block after each match is found.*

DG put in the first kmer program here. DG Exercise break up a long line into lines of at most 40 chars for nice DG printing.

1.2 Greedy versus Non-Greedy Matching

Unless told to do otherwise, in regular expression pattern matching Perl does *greedy* matching.

Formal Definition of Greedy Matching When Perl searches for a substring matching a subexpression of a regular expression, it continues to match that subexpression as long as possible, *provided* that the rest of the regular expression matches further to the right.

As an example of greedy matching, suppose `$varstring` holds *gaaacccttcgg*. Then the statement:

```
if ($varstring =~ m/(a.*c)/) {
   print "$1";
}
```

will print

```
aaacccttc
```

even though shorter strings, such as `ac` also match. When `a.*c` matches `aaacccttc`, first the two `a's` match, and then `.*` matches `aaccctt`, and finally the two `c's` match. The key point is that Perl continues to match the subexpression `.*` as long as possible, provided that the remaining part of the regular expression, `c`, can match something. This is greedy, or *maximal* matching.

The formal definition of greedy matching may suggest that Perl tries to find the longest substring that matches the specified regular expression, but that interpretation is no correct. If you do not see this yet, you should explicitly work out how the formal definition of greedy matching applies, whenever you see examples of greedy matching. Soon, it will be clear, even though it is a bit subtle.

As another example of greedy matching, consider a program that wants to sucessively read in lines from a file looking for the first GenBank accession number in each line. To be more certain that what it has found is an accession number, the program requires that the word "accession" be found on the same line, somewhere before the accession number. In particular, as a line is read, the program checks if the line has the word "accession" followed by a space followed later on the line by something that fits the description of an accession number. If it finds one, it extracts it and prints it. Here is the code:

```
print "type the input file name please\n";
open (INFO, <>);  # open the input file and assign the handle INFO
print "type the output file name please\n";
$myoutfile = <>;  # Read in the file name for the output.
open (OUTFO, "> $myoutfile");

while ( $line = <INFO>) {     # Read one line at a time from INFO, until
                              # all lines have been read in.
                              # process each line in the block below.

if ($line =~ m/accession.* ([A-Z]{1,2}\d{3,7})[ ,.;:?]/i)  {
print OUTFO "The input line does contain an Accession number,
and it is $1.  \n";
}
```

This program will do fine if there is only one accession number in a line. But because of greedy matching, if there is more than one, it will report the *last* accession number found after the word "accession", instead of the first accession number. Shortly, we will see how to fix this – here, you should just understand how greedy matching causes this problem.

Greedy matching can also cause subtle errors that are hard to catch. For example, suppose in the above program we had written the `if` statement as:

```
if ($line =~ m/accession .*([A-Z]{1,2}\d{3,7})[ ,.;:?]/i)  {
```

That is, supose we move the space from after the `.*` to before it. This seems inocuous enough, but it can cause subtle problems. To illustrate, focus on a line that only has one accession number. Everything will work fine when the accession number starts with only a single letter. But when the accession number starts with two letters, only the second one of them will be printed. For example, if the accession number is `AB12345` then only `B12345` will be printed. Why?

The problem is greedy matching. A space is no longer required to be found just before the complete accession number, and the regular expression allows a match starting with only a single letter, so greedy matching matches the first letter of the accession number as part of `.*`, instead of matching it as part of `[A-Z]{1,2}`. This can't happen when we use the original regular expression, in the first `if` statement - if the first letter in the accession number is matched as part of `.*`, then the space before it would also be matched as part of `.*`, and so there would be no space in the string to match the space after `.*` in the regular expression. If this is confusing, try to apply the formal definition of greedy matching to this example.

STOP

1.2.1 Non-Greedy Matching

Consider again the program fragment

```
if ($varstring =~ m/(a.*c)/) {
```

```
   print "$1";
}
```

When `$varstring` holds *gaaacccttcgg* then the program will print `aaacccttc`. But other strings inside `$varstring` match the regular expression `a.*c`. For example, *aaac* matches it. This is the match that would result if Perl started at the left end of the string and *continued* matching the `.*` subexpression as *little* as possible, provided that the rest of the regular expression would match. This is called *non-greedy* or *minimal* matching.

Non-greedy matching is specified in Perl by the use of `?`. In Perl, whenever you place a question mark `?` after a *quantifier*, the result is a match that *continues* as little as possible to the right, *provided* that the rest of the regular expression matches. That is, using `?` after a quantifier instructs Perl to do non-greedy matching on that part of the regular expression. Therefore, the code

```
if ($varstring =~ m/(a.*?c)/) {
   print "$1";
}
```

will print *aaac*, when `$varstring` contains *gaaacccttcgg*.

Note that in the above explanation of non-greedy matching, we emphasized the word "continues". That is to emphasize the fact that non-greedy matching does not try to reduce the match size on the left end, but only on the right end of the matching substring. For example, the regular expression above matches the substring *ac*, which is the smallest substring matching the regular expression, but the program does not print *ac* because non-greedy matching only reduces the continuation of a match on the right end.

Exercise 1.3 *Paralleling our Formal Definition of Greedy Matching, write out a Formal Definition of Non-Greedy Matching.*

Note that this ues of `?` is different from its earlier use, i.e., where it specified that the closest preceeding regular expression was optional.Perl knows the

difference between these two uses of ?, because one follows a quantifier and the other does not.

We find the workings of non-greedy matching a bit difficult to grasp at first (some books get it wrong - they say it causes Perl to find the smallest matching substring). But with some experimentation, people usually figure out where to put the ? to get what they want. You should try out some examples of your own where you do and don't put in a ?, until you feel comfortable with it. If stuck, go back to the Formal Definitions of Greedy and Non-Greedy Matching. Below we give some more examples of non-greedy matching.

1.2.2 More examples of non-greedy matching

In sequence analysis applications, non-greedy matching is often essential in order to find a substring of interest. One simple way this happens is when we are looking for a substring that has several known pieces seperated by an unknown piece, and we want the two closest known pieces.

Example 1 As a first example, consider again the problem of finding things that look like accession numbers, where each one must be proceeded by its own copy of the word "accession". We want to find all of these in each line in a file. Here we show how to process an individual line.

```
#!/pkg/bin/perl
# Program allaccessions.pl
# This programs reads a line, finds each occurrence of
# the word "accession" (case insensitive),
# followed by any characters, followed by a space and an accession number.
# It extracts each accession number found and prints them. We use
# non-greedy matching to find the closest accession number following
# each "accession", and uses the while-g idiom to find them all.

   $line = <DATA>)
   chomp $line;
   $found = 0;
   while ($line =~ m/accession.*? ([A-Z]{1,2}\d{3,7})[ ,.;:?]/gi)  {
```

```
   $found = 1;
   print "The input line does contain an Accession number, and it
   is $1.  \n";
   }

   unless ($found) {
   print "The line \n $line \n does not contain any accession numbers.\n";
   }
}

__DATA__
First ACCESSION number is AB45678 and next is ACCESSION number C12345.
```

Program *allaccessions.pl* illustrates the use of non-greedy matching and the while-g idiom, but it also illustrates two other things, the use of *flags* (i.e., the variable `$found`) and the use of the `unless` statement (read about it in Johnson).

Exercise 1.4 *Extend program allaccessions.pl so that it reads and processes each line in a specified file.*

Example 2: As a second example of non-greedy matching, we consider the important task of ORF finding in DNA sequences.

The major problem in *annotating* newly sequenced DNA (i.e, identifying the interesting portions of the DNA) is the problem of identifying the genes in the sequence. One critical step in this process is identifying all the *open reading frames*, ORFs, which are candidate regions that may contain a gene. An ORF is defined as a region of DNA that starts with the *initiation codon* *ATG* and ends with the *first* stop codon *TAA, TAG* or *TGA*, provided that the number of nucleotides between the stop and the start codons is a multiple of three. That is, the start and the stop codons must be "in register".

Ultimately, we will write a Perl program that takes in a DNA sequence and finds all the ORFs in the sequence. But to start, we want to develop a program that finds and prints the first ORF found in the string. Try the following program:

```
#!/pkg/bin/perl -w
$string = <>;
if ($string =~ m/(atg(...)*(taa|tga|tag))/) {
    print "ORF found: $1\n";
}
else {
    print "No ORF found\n";
}
```

Will this program correctly print out the first ORF, if the string contains one, and always report "no ORF" when there are none? Try out the program with the following four input strings:

aatgccttgat

aatgccttgataa

aatgaaatatat

Explain the results before going on.

For the first two input strings, the above program says that an ORF has been found, and it correctly determines that the third string contains no ORF. However, in the case of the second string, the program prints out more than the ORF. The problem is greedy matching: the subexpression `(...)*` matches past the first stop codon because the second string has two stop codons in register with the start codon. To fix this, we add a `?` after the `*` to make that part of the regular expression non-greedy, as follows:

```
$string =  m/(atg(...)*?(taa|tga|tag))
```

This will find a single ORF in a string. Try it out.

Exercise 1.5 *One might think that the match statement*
$string = m/atg((.{3})(taa—tga—tag))?/;*

should work to find an ORF. It does not. Explain why?

*It may help to recall that the symbol ? has two meanings depending on the context it is used. When it comes after a quantifier such as * or +, it tells the*

regular expression to use non-greedy matching, i.e., the minimum quantification that allows the whole regular expression to match a substring. But when there is no quantifier before the ?, the symbol ? has the simpler meaning of "repeat the preceeding regular expression zero or one time",

Third Example Consider the E. coli *origin of replication* (a stretch of DNA where where DNA replication starts). Simplifying the description given in [**?**], the E. coli origin of replication contains eight substrings, seperated from each other by additional nucleotides. The total length of the origin of replication is roughly 245 nucleotides. The details of the eight substrings are as follows: First there are three copies of `GATCTNTTNTTTT`, where `N` stands for any nucleotide; then there are five copies of `TTWTWCAWA`, where `W` is either `A` or `T`.

In searching a string of E. coli DNA for the origin of repliction, we look for the eight substrings in the proper order. Ideally, we would also like to require that they be found in an interval of roughly 245 nucleotides. We will implement that requirement in a later program. For now, we use non-greedy matching to minimize the lengths of the intervals between the eight substrings, because they are close to each other in E. coli origin of replication sequences. Thus, we would use the following Perl regular expression in a match statement in order to find an E. coli origin of replication:

```
(GATCT[ATCG]TT[ATCG]T{4}.*?){3}.*?(TT[AT]T[AT]CA[AT]A.*?){5}
```

Exercise 1.6 *Try out the above regular expression in a match statement, both with and without non-greedy matching. Explain how the question marks work in the above regular expression.*

Exercise 1.7 *Now we complicate the previous program, to get closer to the true description of an E. coli origin of replication. In the E. coli origin of replication, each of the last five substrings actually appear either as shown above, or in the inverted configuration (*`AWACWTWTT`*). Write a modified Perl regular expression that incorporates that possibility. Keep it as simple as possible.*

1.3 Optional Project

Problem Statement: Write a Perl program that converts a PROSITE pattern field into a Perl regular expression.

A series of correctly chosen substitution and match statements will suffice for a complete conversion. Test your program with the examples given below.

Examples of PROSITE pattern entries:

```
PA   D-[SGN]-D-[PE]-[LIVM]-D-[LIVMGC].

PA   [AC]-x-V-x(4)-{ED}.
```

This pattern is translated as: [Ala or Cys]-any-Val-any-any-any-any-{any but Glu or Asp]

```
PA   <A-x-[ST](2)-x(0,1)-V.
```

This pattern, which must be in the N-terminal of the sequence (<), is translated as: Ala any-[Ser or Thr]-[Ser or Thr]-(any or none)-Val

Standard Pattern Conventions:

- The standard IUPAC one-letter codes for the amino acids are used.
- The symbol 'x' is used for a "wild-card" position where any amino acid is accepted.
- Ambiguities are indicated by listing the acceptable amino acids for a given position between square brackets '[ ]'. For example: [ALT] stands for Ala or Leu or Thr.
- Ambiguities are also indicated by listing between a pair of curly brackets '{ }' th amino acids that are not accepted at a given position. For example: {AM} stands fo any amino acid except Ala and Met.
- Each element in a pattern is separated from its neighbor by a dash '-'.
- Repetition of an element of the pattern is indicated by following that element with numerical value or a numerical range between parenthesis. Examples: x(3) correspond to x-x-x, x(2,4) corresponds to x-x or x-x-x or x-x-x-x.
- When a pattern is restricted to either the N- or C-terminal of a sequence, that patter either starts with a < symbol or respectively ends with a > symbol.
- A period ends the pattern.

Example Output

```
PROSITE PAttern Field:
PA   [AC]-x-V-x(4)-{ED}.

The Output Equivalent Perl Regular Expression:
[AC].V.(4)[^ED]
```