STUDIES IN LOGIC, GRAMMAR
AND RHETORIC 35 (48) 2013
DOI: 10.2478/slgr-2013-0032

Ruby vs. Perl – the Languages of Bioinformatics

Maciej Goliński[1], **Agnieszka Kitlas Golińska**[2]

[1] Department of Programming and Formal Methods, University of Bialystok, Poland
[2] Department of Medical Informatics, University of Bialystok, Poland

Abstract. Ruby and Perl are programming languages used in many fields. In this paper we would like to present their usefulness with regard to basic bioinformatic problems. We concentrate on a comparison of widely used Perl and relatively rarely used Ruby to show that Ruby can be a very efficient tool in bioinformatics. Both Perl and Ruby have a built-in regular expressions (or regexp) engine, which is essential in solving many problems in bioinformatics. We present some selected examples: printing the file content, removing comments from a FASTA file, using hashes, printing nucleotides included in a sequence, searching for a specific nucleotide in sequence and translating nucleotide sequences into protein sequences obtained in GenBank format. It is our belief that Ruby's popularity will rise because of its simple syntax and the richness of its methods. Programs in Ruby are very easy to read and therefore easier to maintain and debug, which are the most important characteristics for a programming language.

Introduction

It is our intent to show that a relatively rarely scientifically-used programming language – Ruby – can be a very efficient tool in the field of bioinformatics, much more so than widely used Perl, and that applications written in Ruby are much easier to read or maintain, and – most of all – easier to write. Ruby, compared to Perl, is a new language, still gaining popularity, while Perl has a well established position as a general-purpose programming language.

The Perl Language

Perl is a programming language developed in 1987 by Larry Wall. It is a dynamic, interpretive, general-purpose language. It incorporates features of other languages including AWK, shell scripting (sh), C, and Lisp (Schwartz et al., 2011).

Perl is sometimes called the hacker language because of its sometimes not easily readable syntax (Foy, 2007). Here is an example of a short, and relatively simple, program which finds the documentation on the *atan2* function and then formats it differently for printing, using a complicated regular expression, a tool which is explained later:

```
#!usr/bin/perl
@lines = 'perldoc -u -f atan2';
foreach (@lines)
{
  s/\w<([^>]+)>/\U$1/g;
  print;
}
```

One of the very important features of Perl languages is the regular expressions they use. Perl is a widely used tool in the field of bioinformatics, especially in the study of the structure and function of genes and proteins.

The Ruby Language

Ruby is a programming language developed in Japan in 1995 by Yukihiro Matsumoto. It is dynamic and reflective, which makes it a very efficient, general-purpose tool. Ruby supports many programming paradigms, including functional, object-oriented and imperative. It is also excellent for metaprogramming, an advanced programming concept. The language was influenced by Perl, Smalltalk, Eiffel and Lisp (Thomas et al., 2009).

One of the most basic ideas for Ruby is that everything is an object, including numbers, classes, and exceptions (Thomas et al., 2009). Thanks to that, a programmer can treat all constructs with a certain universality.

Another important feature of Ruby is a built-in regular expressions handler, which is extremely useful in problems of bioinformatics.

Ruby is very helpful in processing files. It saves the programmer the trouble of remembering to close opened files (which is a very common problem) (Thomas et al., 2009). In addition, it's very easy to manipulate long text files, like those containing nucleotide sequences in FASTA format.

The Regular Expressions

Both Perl and Ruby have a built-in regular expressions (or regexp) engine (Foy, 2007; Thomas et al., 2009), which is essential in solving many problems in bioinformatics. Regexps are an efficient tool in finding parts of a text (or other sequences of characters) that match a given pattern. To provide a pattern, one should use a special sub-language, created for that purpose. A regexp consists of a number of characters, as well as a few special ones. The pattern is usually placed between slashes "/". Here is a simple example:

```
/gene/
```

This regexp will match a single occurrence of a sequence "gene". This is no different from a natural text searching. To make regular expressions more interesting, we have to introduce a few special symbols.

The most basic symbol is a dot "`.`", which means "any single character". This means that the regexp:

```
/.at/
```

will match both "rat" and "cat", because they have a single character preceding "at". The pattern:

```
/Ru.y/
```

matches the word "Ruby", but not "Rugby", since there are two characters, where there should be only one.

The next symbol, a vertical line "`|`", indicates an alternative. The following pattern will match both the words "Perl" and "Ruby":

```
/Perl|Ruby/
```

The characters surrounded by parentheses are grouped together. Grouping and any alternatives often appear together:

```
/(r|c)at/
```

matches the same words as:

```
/(rat)|(cat)/
```

An important note: The following pattern does not mean the same thing as the previous one:

```
/rat|cat/
```

It means the same thing as this one:

```
/ra(t|c)at/
```

There are special characters that signal the beginning and the end of a string. The pattern:

```
/^Ruby/
```

matches the word "Ruby" only if it occurs at the beginning of the analyzed string, and the regexp:

```
/Perl$/
```

will match the word "Perl", only if it is at the end of the string.

Another very important feature of the regular expressions are repetitions. They are a symbol or a set of symbols indicating how many times the previous expression should be repeated in the text.

The question mark "?" matches the preceding element zero or one time. For example:

```
/-?15/
```

matches both "−15" and "15".

The "*" character matches the preceding pattern zero or more times. For example:

```
/10*1/
```

will match eg. "101", "11" or "10000001".

The plus sign "+" denotes one or more repetitions of the preceding pattern. For example:

```
/10+1/
```

will match "101", "100001", "1001", but not "11".

Regular expressions are a very useful tool in the field of bioinformatics, especially in parsing files in the FASTA format.

The FASTA Format and GenBank Format

FASTA format is a text-based format for representing peptide or nucleotide sequences (Baxevanis et al., 2004). In FASTA, amino acids or nucleotides are written in single-letter codes, which makes them easy to process.

A part of the file in FASTA format is presented below (Campylobacter jejuni subsp. jejuni NCTC 11168 complete genome) (National Center for Biotechnology Information, 2006):

>gi|30407139|emb|AL111168.1| Campylobacter jejuni subsp. jejuni NCTC 11168 complete genome
ATGAATCCAAGCCAAATACTTGAAAATTTAAAAAAAGAATTAAGTGAAAACGA
ATACGAAAACTATTTATCAAATTTAAAATTCAACGAAAAACAAAGCAAAGCAG

```
ATCTTTTAGTTTTTAATGCTCCAAATGAACTCATGGCTAAATTCATACAAACA
AAATACGGCAAAAAAATCGCGCATTTTTATGAAGTGCAAAGCGGAAATAAAG
CCATCATAAATATACAAGCACAAAGTGCTAAACAAAGCAACAAAAGCACAAAA
ATCGACATAGCTCATATAAAAGCACAAAGCACG
```

In the first line there is a description (comments) of the file and then lines of sequence data. Here we present only 5 lines, although there are many more in this file.

The GenBank (National Center for Biotechnology Information, 2009) is an open access nucleotide and protein sequence database. Files in GenBan format contain an extensive description, nucleotide sequence and its translation to protein sequence (Baxevanis et al., 2004).

A part of the file in GenBank format is presented below (Homo sapiens 43kDa acetylcholine receptor-associated protein (RAPSN) mRNA) (National Center for Biotechnology Information, 2001):

```
/translation="MGQDQTKQQIEKGLQLYQSNQTEKALQVWTKVLEKSSDLMGRFR
VLGCLVAHSEMGRYKEMLKFAVVQIDTARELEDADFLLESYLNLARSNEKLCEFH
KTISYCKTCLGLPGTRAGAQLGGQVSLSMGNAFLGLSVFQKALESFEKALRYAHN
NDDAMLECRVCCSLGSFYAQVKDYEKALFFPCKAAELVNNYGKGWSLKYRAMS
QYHMAVAYRLLGRLGSAMECCEESMKIALQHGDRPLQALCLLCFADIHRSRGDLE
TAFPRYDSAMSIMTEIGNRLGQVQALLGVAKCWVARKALDKALDAIERAQDLAE
EVGNKLSQLKLHCLSESIYRSKGLQRELRAHVVRFHECVEETELYCGLCGESIGE
KNSRLQALPCSHIFHLRCLQNNGTRSCPNCRRSSMKPGFV"

ORIGIN
        1 cccaactggc agcgacagct gcagacgggc tgaaccagct ttgttcccag ggtggcgcct
       61 gctctccatc caggccccat tccggctccc acccgacgct gcttttgttc ccacgtttcg
      121 gggggcagct ggcactgtga ttcctgcccc atgagtgcct agaggcacgg agccaccagg
      181 gatcacccca cgtgggacac agggcttggg gaggatgggg caggaccaga ccaagcagca
      241 gatcgagaag ggctccagc tgtaccagtc caaccagaca gagaaggcat tgcaggtgtg
      301 gacaaaggtg ctggagaaga gctcggacct catggggcgc ttccgcgtgc tgggctgcct
      361 ggtcacagcc cactcggaga tgggccgcta caaggagatg ctgaagttcg ctgtggtcca
      421 gatcgacacg gcccgggagc tggaggatgc cgacttcctc ctggagagct acctgaacct
      481 ggcacgcagc aacgagaagc tgtgcgagtt tcacaagacc atctcctact gcaagacctg
      541 ccttgggctg cctggtacca gggcaggtgc ccagctcgga ggccaggtca gcctgagcat
      601 gggcaatgcc ttcctgggcc tcagcgtctt ccagaaggcc ctggagagct tcgagaaggc
      661 cctgcgctac gcccacaaca atgatgacgc catgctcgag tgccgcgtgt gctgcagcct
      721 gggcagcttc tatgcccagg tcaaggacta cgagaaagcc ctgttcttcc cctgcaaggc
      781 ggcagagctt gtcaacaact atggcaaagg ctggagcctg aagtaccggg ccatgagcca
      841 gtaccacatg gccgtggcct atcgcctgct gggccgcctg ggcagtgcca tggagtgttg
      901 tgaggagtct atgaagatcg cgctgcagca cggggaccgg ccactgcagg cgctctgcct
      961 gctctgcttc gctgacatcc accggagccg tggggacctg gagacagcct tccccaggta
     1021 cgactccgcc atgagcatca tgaccgagat cggaaaccgc ctggggcagg tgcaggcgct
     1081 gctgggtgtg gccaagtgct gggtggccag gaaggcgctg gacaaggctc tggatgccat
     1141 cgagagagcc caggatctgg ccgaggaggt ggggaacaag ctgagccagc tcaagctgca
     1201 ctgtctgagc gagagcattt accgcagcaa agggctgcag cgggaactgc gggcgcacgt
     1261 tgtgaggttc cacgagtgcg tggaggagac ggagctctac tgcggcctgt gcggcgagtc
```

1321 cataggcgag aagaacagcc ggctgcaggc cctaccctgc tcccacatct tccacctcag
1381 gtgcctgcag aacaacggga cccggagctg tcccaactgc cgccgctcat ccatgaagcc
1441 tggctttgta tgactcctgg cagcaggcgt gggcttcctc ctcgccactc ctgctctttc
1501 tccactgcac gccagaggcc catttactcc tggggcagct gccaggtcgt cctcaccata
1561 gccaaggcct tggggcctgc ccagggctgc tcccctgggc ccagctcccc tccctgcctc
1621 tttgtacttt gctctttata gaaaaataaa ctgtttgtac ctggtcccag g

Selected examples

In this section, we present a few programs very useful in the field of bioinformatics written both in Perl and in Ruby. The purpose of these examples is to present an alternative for the commonly used Perl language, which in our opinion is simpler to write, simpler to read, and simpler to maintain.

The goal of this program is to open a simple text file, and print its contents on the console, line by line.

Perl:

```
open(F, "file.txt");
while($line = <F>)
{
  print "$line"
}
close F;
```

Ruby:

```
File.open("file.txt") do |f|
  while line = f.gets
    print line
  end
end
```

The program in Perl is fairly straightforward. First we open the file, than in a while loop we obtain each line separately, save it in a variable, and print it. The often forgotten part is closing the file, which is both unprofessional and potentially dangerous to the file. The Ruby approach takes care of the last problem automatically by the usage of blocks.

In this program we take a file in a FASTA format, than copy its contents to a second file, omitting the lines containing the comments.

Perl:

```
open (F, "seq.fa");
open (FF,">seq2.txt");
while (<F>)
{
  next if(/^>/);
  print FF;
}
close F;
close FF;
```

Ruby:

```
File.open("seq.fa") do |in|
  File.open("seq2.fa", "w+") do |out|
    while line = in.gets
      out << line unless line =~ /^>/
    end
  end
end
```

Both approaches utilize regular expressions to check if the line begins with a “>” sign. The program in Ruby is shorter, and there is no problem with unclosed files. Also, the part concerning copying the lines is much easier to understand.

The hash is a variation on the table, where instead of just numbers, anything can serve as an index called a key. This program shows the way to use hashes in both languages. The key in the hash is a name of a species, and the value is a gene count. The program prints the names with their gene counts.

Perl:

```
%gene_counts = ("Human" => 31000, "Fruit fly" => 13000,
"Mouse" => 30000, "Chickenpox virus" => 69, "Rice" => 40000,
"Tuberculosis bacteria" => 4000);
while ( ( $key, $value ) = each %gene_counts )
{
    print "$key has $value genes in its genome.\n";
}
```

Ruby:

```
gene_counts = ("Human" => 31000, "Fruit fly" => 13000,
"Mouse" => 30000, "Chickenpox virus" => 69, "Rice" => 40000,
"Tuberculosis bacteria" => 4000)
gene_counts.each_pair {|key, value| puts "#{key}has #{value}
genes in its genome."}
```

Both programs first define the hash. Then, in the Perl approach, we obtain the key-value pair in a while loop, and print the appropriate sentence. The Ruby program is again much simpler, and again, thanks to the use of code blocks.

This example prints the nucleotides that are included in a given sequence. It utilizes both a hash and regular expressions.

Perl:

```
%dict = (A => Adenine, T => Thymine, G => Guanine,
C => Cytosine);
$sequence = 'CTATGCGGTA';
while ( $sequence =~ /./g )
{
    print "$dict{$&}\n";
}
```

Ruby:

```
@dict = {"A" => "Adenine", "T" => "Thymine", "G" => "Guanine",
"C" => "Cytosine"}
sequence = "CTATGCGGTA"
sequence.scan(/./).each {|i| puts @dict[i]}
```

Both programs first define both the hash, which serves as a dictionary for the nucleotides' names, and a fragment of a DNA sequence. The Perl program uses a match operator with an additional "g" modifier, which allows for the scanning of the entire sequence, in order to match patterns to a string. Then, it prints the value corresponding to the letter obtained from the sequence. This method may be a bit difficult to understand. The Ruby approach is simpler thanks to the scan method, which is easier to use than the match operator.

This program is designed to count the occurrences of a specific nucleotide in a given sequence, in this case Adenine (A).

Perl:

```
$sequence="ATGAATCCAAGCCAAATACTTGAAAATTTAAAAAAAGAATTAAGTGAAAAC
GAATACGAAAACTATTTATCAAATTTAAAATTCAACGAAAAACAAAGCAAAGCAGATCTTTT
AGTTTTTAATGCTCCAAATGAACTCATGGCTAAATTCATACAAACAAAATACGGCAAAAAAA
TCGCGCATTTTTATGAAGTGCAAAGCGGAAATAAAGCCATCATAAATATACAAGCACAAAGT
GCTAAACAAAGCAACAAAAGCACAAAAATCGACATAGCTCATATAAAAGCACAAAGCACG";
$sum=0;
@tab=split('', $sequence);
foreach $i (@tab)
{
  $sum++ if $i eq 'A';
}
print $sum;
```

Ruby:

```
@sequence="ATGAATCCAAGCCAAATACTTGAAAATTTAAAAAAAGAATTAAGTGAAAAC
GAATACGAAAACTATTTATCAAATTTAAAATTCAACGAAAAACAAAGCAAAGCAGATCTTTT
AGTTTTTAATGCTCCAAATGAACTCATGGCTAAATTCATACAAACAAAATACGGCAAAAAAA
TCGCGCATTTTTATGAAGTGCAAAGCGGAAATAAAGCCATCATAAATATACAAGCACAAAGT
GCTAAACAAAGCAACAAAAGCACAAAAATCGACATAGCTCATATAAAAGCACAAAGCACG";
@sum=0
@sequence.each_char {|i| @sum+=1 if i == 'A'}
puts @sum
```

The program in Ruby is quite simple. It passes each character of the string into the block, where it is compared with the letter A. The Perl approach is more complicated, since Perl treats strings as a singular value. Therefore, it is impossible to iterate the string. It is necessary to split the string into a table with single characters as elements. This allows for an iteration, and counting of the letter A.

Files in GenBan format contain an extensive description, nucleotide sequence and its translation to protein sequence. How can we obtain this translation? First, we need a genetic code for translation and then we implement programs in Perl and Ruby as one can see below. For this analysis we selected the GenBank: AF111785.1 file (Homo sapiens myosin heavy chain IIx/d mRNA) (National Center for Biotechnology Information, 1998):

Perl:

```
%dict = ("TTT" => "F", "TTC" => "F", "TTA" => "L", "TTG" =>
"L", "CTT" => "L", "CTC" => "L", "CTA" => "L", "CTG" => "L",
```

```
"ATT" => "I", "ATC" => "I", "ATA" => "I", "ATG" => "M", "GTT"
=> "V", "GTC" => "V", "GTA" => "V", "GTG" => "V", "TCT" =>
"S", "TCC" => "S", "TCA" => "S", "TCG" => "S", "CCT" => "P",
"CCC" => "P", "CCA" => "P", "CCG" => "P", "ACT" => "T", "ACC"
=> "T", "ACA" => "T", "ACG" => "T", "GCT" => "A", "GCC" =>
"A", "GCA" => "A", "GCG" => "A", "TAT" => "Y", "TAC" => "Y",
"TAA" => "STOP", "TAG" => "STOP", "CAT" => "H", "CAC" => "H",
"CAA" => "Q", "CAG" => "Q", "AAT" => "N", "AAC" => "N", "AAA"
=> "K", "AAG" => "K", "GAT" => "D", "GAC" => "D", "GAA" =>
"E", "GAG" => "E", "TGT" => "C", "TGC" => "C", "TGA" =>
"STOP", "TGG" => "W", "CGT" => "R", "CGC" => "R", "CGA" =>
"R", "CGG" => "R", "AGT" => "S", "AGC" => "S", "AGA" => "R",
"AGG" => "R", "GGT" => "G", "GGC" => "G", "GGA" => "G", "GGG"
=> "G");
$sequence=uc("atgagttctgactctgagatggccatttttggggaggctgctcctttt
cctccgaaagtctgaaagggagcgaattgaagcccagaacaagccttttgatgccaagaca
tcagtctttgtggtggaccctaaggagtcctttgtgaaagcaacagtgcagagcagggaag
gggggaaggtgacagctaagaccgaagctggagctactgtaacagtgaaagatgaccaagt
cttccccatgaaccctcccaaatatgacaagatcgaggacatggccatgatgactcatcta
cacgagcctgctgtgctgtacaacctcaaagagcgctacgcagcctggatgatctacacct
actcaggc");
$goal="MSSDSEMAIFGEAAPFLRKSERERIEAQNKPFDAKTSVFVVDPKESFVKATVQS
REGGKVTAKTEAGATVTVKDDQVFPMNPPKYDKIEDMAMMTHLHEPAVLYNLKERYAAWMI
YTYSG";
$translation="";
while ($sequence =~ /.../g)
{
    $translation .= $dict{$&};
}
print "Success!" if ($translation eq $goal);
```

Ruby:

```
@dict = {"TTT" => "F", "TTC" => "F", "TTA" => "L", "TTG" =>
"L", "CTT" => "L", "CTC" => "L", "CTA" => "L", "CTG" => "L",
"ATT" => "I", "ATC" => "I", "ATA" => "I", "ATG" => "M", "GTT"
=> "V", "GTC" => "V", "GTA" => "V", "GTG" => "V", "TCT" =>
"S", "TCC" => "S", "TCA" => "S", "TCG" => "S", "CCT" => "P",
"CCC" => "P", "CCA" => "P", "CCG" => "P", "ACT" => "T", "ACC"
=> "T", "ACA" => "T", "ACG" => "T", "GCT" => "A", "GCC" =>
"A", "GCA" => "A", "GCG" => "A", "TAT" => "Y", "TAC" => "Y",
```

```
"TAA" => "STOP", "TAG" => "STOP", "CAT" => "H", "CAC" => "H",
"CAA" => "Q", "CAG" => "Q", "AAT" => "N", "AAC" => "N", "AAA"
=> "K", "AAG" => "K", "GAT" => "D", "GAC" => "D", "GAA" =>
"E", "GAG" => "E", "TGT" => "C", "TGC" => "C", "TGA" =>
"STOP", "TGG" => "W", "CGT" => "R", "CGC" => "R", "CGA" =>
"R", "CGG" => "R", "AGT" => "S", "AGC" => "S", "AGA" => "R",
"AGG" => "R", "GGT" => "G", "GGC" => "G", "GGA" => "G", "GGG"
=> "G"};
@sequence="atgagttctgactctgagatggccattttggggaggctgctcctttcct
ccgaaagtctgaaagggagcgaattgaagcccagaacaagccttttgatgccaagacatca
gtctttgtggtggaccctaaggagtcctttgtgaaagcaacagtgcagagcagggaagggg
ggaaggtgacagctaagaccgaagctggagctactgtaacagtgaaagatgaccaagtctt
ccccatgaaccctcccaaatatgacaagatcgaggacatggccatgatgactcatctacac
gagcctgctgtgctgtacaacctcaaagagcgctacgcagcctggatgatctacacctact
caggc";
@goal="MSSDSEMAIFGEAAPFLRKSERERIEAQNKPFDAKTSVFVVDPKESFVKATVQS
REGGKVTAKTEAGATVTVKDDQVFPMNPPKYDKIEDMAMMTHLHEPAVLYNLKERYAAWMI
YTYSG";
@translation=""
@sequence.upcase.scan(/.../).each
{
    |i| @translation << @dict[i]
}
puts "Success!" if @translation == @goal
```

Both approaches begin by defining a few variables. The first one is a hash serving as a dictionary for the translation. The second one is the given sequence in a GenBank format. The next variable contains the same sequence, but as a natural protein sequence, which is later used to check if the program is valid. The solution in Perl is far more difficult to comprehend than the one in Ruby. The *uc* function in Perl transforms the string into upper case.

A Quick Overview of Literature

There are many books or papers on the application of Perl in bioinformatics (Moorhouse et al., 2004; Tisdall, 2001), but not so many on the application of Ruby (Aerts et al., 2009). However, this is changing every day because Ruby is becoming more and more popular. We couldn't find papers in which Perl and Ruby were compared for simple and basic use in the field of bioinformatics.

It is worth mentioning that in recent years some scientists and programmers have developed libraries and tools for bioinformatics, molecular biology, genomics and life sciences, namely BioPerl (BioPerl, 2012; Stajich et al., 2002) and BioRuby (BioRuby, 2013; Goto et al., 2009). In our paper, we showed that one can perform basic bioinformatics analyses in Perl and Ruby without downloading and using these special libraries. We also compared programs written in both languages, in our opinion in favor of the Ruby language.

Conclusions

Perl and Ruby are useful tools in the field of bioinformatics. Both languages are general purpose, free to use, and popular. While Perl has a stable position in medical computer science, Ruby is still working its way into the field. It is our belief that Ruby's popularity will rise because of its simple syntax and the richness of its methods. The programs in Ruby are very easy to read and, therefore, easier to maintain, which are the most important characteristics for a programming language.

REFERENCES

Aerts, J., & Law, A. (2009). An introduction to scripting in Ruby for biologists. *BMC Bioinformatics*, 10(221), Retrieved July 30, 2013, from BioMed Central: http://www.biomedcentral.com/1471-2105/10/221. DOI: 10.1186/1471-2105-10-221.

Baxevanis, A. D., & Ouellette, B. F. F. (2004). *Bioinformatics: a practical guide to the analysis of genes and proteins.* USA: Wiley-Interscience.

BioPerl (2012). Retrieved July 30, 2013 from http://www.bioperl.org/.

BioRuby (2013). Retrieved July 30, 2013 from http://www.bioruby.org/.

Foy, B. (2007). *Mastering Perl* (2nd ed.). USA: O'Reilly Media.

Goto, N., Prins, P., Nakao, M., Bonnal, R., Aerts, J., & Katayama, T. (2009). BioRuby: Bioinformatics software for the Ruby programming language. *Bioinformatics*, 26(20), 2617–2619. DOI: 10.1093/bioinformatics/btq475.

Moorhouse, M., & Barry, P. (2004). *Bioinformatics, Biocomputing and Perl: an introduction to bioinformatics computing skills and practice*, USA: Wiley.

National Center for Biotechnology Information, U.S. National Library of Medicine. (1998). *Homo sapiens myosin heavy chain IIx/d mRNA, complete cds.* Retrieved July 30, 2013, from http://www.ncbi.nlm.nih.gov/nuccore/4808814?report=genbank#sequence_4808814.

National Center for Biotechnology Information, U.S. National Library of Medicine. (2001). *Homo sapiens 43kDa acetylcholine receptor-associated protein (RAPSN) mRNA, complete cds.* Retrieved July 30, 2013, from http://www.ncbi.nlm.nih.gov/nuccore/19310212?report=genbank.

National Center for Biotechnology Information, U.S. National Library of Medicine. (2006). *Campylobacter jejuni subsp. jejuni NCTC 11168 complete genome.* Retrieved July 30, 2013, from http://www.ncbi.nlm.nih.gov/nuccore/30407139?report=fasta.

National Center for Biotechnology Information, U.S. National Library of Medicine. (2009). Retrieved July 30, 2013, from http://www.ncbi.nlm.nih.gov/genbank/.

Schwartz, R., & Phoenix, T. (2011). *Learning Perl.* USA: O'Reilly Media.

Stajich, J. E., Block, D., Boulez, K., Brenner, S., Chervitz, S., Dagdigian, C., Fuellen, G., Gilbert, J. Korf, I., Lapp, H., Lehväslaiho, H., Matsalla, C., Mungall, C. J., Osborne, B. I., Pocock, M. R., Schattner, P., Senger, M., Stein, L. D., Stupka, E., Wilkinson, M. D., & Birney, E. (2002). The BioPerl Toolkit: Perl modules for the life sciences. *Genome Research*, 12(10), 1611-1618. DOI: 10.1101/gr.361602.

Thomas, D., Fowler, Ch., & Hunt, A. (2009). *Programming Ruby 1.9: the pragmatic programmers' guide.* USA: Pragmatic Bookshelf.

Tisdall, J. (2001). *Beginning Perl for bioinformatics.* USA: O'Reilly Media.