Ruby-Programming Course
Programming in Bioinformatics
Lecture notes on a course held in the winter 2013/2014

Stefan Kurtz

Research Group for Genome Informatics
Center for Bioinformatics Hamburg
University of Hamburg

December 9, 2013
Note: a large part of the material presented here was adapted from the book:
Beginning Perl for Bioinformatics
by James Tisdall, O'Reilly Media, 2001.

Contents

Some features of Ruby

- useful whenever file processing is important
- e.g. Bioinformatics and WEB-programming
- it is for free
  - if you do not have Ruby on your computer, then install it
  - see `http://www.ruby-lang.org`
- runs on all commonly used operating systems
- two meanings of Ruby: programming language and translator
- translator turns Ruby program into instructions understood by the computer
- two variations of translator
  - interactive Ruby shell `irb` directly executes the Ruby command on your computer
  - Ruby interpreter reads Ruby scripts and executes them
- other notions: Ruby script, Ruby code

Low and Long Learning Curve

- get started quickly
- many useful programs can be written without much experience
- learning all of Ruby will take a while
- Ruby is object oriented
- everything you manipulate is an object and the results of those manipulations are objects as well
- each object is generated as an instance of a class
- a class consists of a state (with variable bindings) and a set of functions (called methods) to manipulate the state

Positive Aspects of Ruby

- ease of programming
  - Ruby contains features that simplify several common bioinformatics tasks
  - e.g. parse information from text files
  - e.g. manipulate medium size DNA or protein sequences
  - e.g. generate WEB-site content
  - e.g. glue different programs (in any language) into one large program
    - via system calls and output parsing
    - by accessing functions and datatypes of external language (possible for C-programs)
- rapid prototyping
  - explore an idea by writing a simple program
  - Ruby programs are often much shorter than programs in other languages
  - it takes less time to write a Ruby program (depending on the application)
  - if the program is not run too often, then shorter development time pays off

Portability, Speed, Space and Program Maintenance

- portability: Ruby program can run with (almost) no changes on different operating systems
- Ruby is a high level language $\Rightarrow$ independent of machine specific instructions $\Rightarrow$ highly portable
- Ruby is good when processing small and medium size data sets
- for large data sets it is better to use C (order of magnitude faster and needs less space)
- standard approach: first write the program in Ruby
- finally, implement time and space critical parts in C
- program maintenance: activity to keep the program working
  - i.e. bug fixing, adding features, changing input/output formats, porting to other platforms
- with some discipline one can write Ruby code that is easy to maintain

How to Run Ruby Programs on Unix or Linux

- make sure that `ruby` is in your path list

```
$ which ruby
/usr/bin/ruby
```

- suppose Ruby script is in file `myfile.rb`
- then run `ruby myfile.rb`
- you may have to give the path of `myfile.rb` if you are not in the same directory
- you can also put the following line at the top of your file

```
#!/usr/bin/env ruby
```

- magic string `#!` tells the Unix/Linux shell: use the given application to interpret the script
- but `myfile.rb` must be made an executable: `chmod u+x myfile.rb`
- typing `myfile.rb` runs your script
- but magic string requires that Ruby is in the given path

Need to learn a text editor (self study)

- Ruby scripts are ASCII files: you need to edit them
- possible editors:
  - `vi` or `emacs` (very powerful, have special highlighting mode, not easy to learn)
  - `xedit`, `gedit`, `joe`, `pico` (simple, not very powerful)
- start with one of the simpler editors
- sometimes learn `vi` or `emacs`, they can save you a lot of time

Finding Help

- Ruby comes with an online help `ri`: use it!
- for example: `ri File.new` will deliver:

```
-------------------------------------------------------------- File::new
     File.new(filename, mode="r")            => file
     File.new(filename [, mode [, perm]])    => file
------------------------------------------------------------------------
     Opens the file named by _filename_ according to _mode_ (default is
     ``r'') and returns a new +File+ object. See the description of
     class +IO+ for a description of _mode_. The file mode may
     optionally be specified as a +Fixnum+ by _or_-ing together the
     flags (O_RDONLY etc, again described under +IO+). Optional
     permission bits may be given in _perm_. These mode and permission
     bits are platform dependent; on Unix systems, see +open(2)+ for
     details.

        f = File.new("testfile", "r")
        f = File.new("newfile",  "w+")
        f = File.new("newfile", File::CREAT|File::TRUNC|File::RDWR, 0644)
```

- there are many tutorial available on the WEB: use them!

The Art of Programming (self study)

- programming is a hands-on-activity: you have to do it; otherwise you won't learn it
- learn the principles of Ruby
- study many examples
- attempt to write your own script
- this is the task of the exercises
- you may find the solutions to the given exercises on the WEB, but this won't help you
- Programming is a cyclic task:
  1. write/revise a script in the editor
  2. run the script
  3. watch its behavior
  4. compare results to the expected results
  5. continue with step 1.
- The steps 2-4 can (in most cases) be fully automated

Backup your files (self study)

- save your files regularly
- in the ZBH-computer pools we have regular backups
- but it does not hurt, if you *systematically* save your files somewhere else
- save versions of the script which already work, before “improving” the script
- this is supported by version control systems (e.g. git or svn)
- very important if more than one programmer is working on the same project

Error Messages

- typical errors involve mistyping, missing symbols or incorrect operators
- the Ruby system then reports error messages
- the messages are a guess (sometimes a good one) what is wrong, and *where* something is wrong
- it takes some experience to understand the messages
- start by looking at the first error message
- the other messages may just result from previous errors
- so fix the first problems, and then apply the Ruby-interpreter again

Debugging

- once your program is valid, run it and check the result
- more than often the program does not do what you expect
- some strategies to find the bugs:
  - print your program, sit down and read the program line by line
  - be critical about programming language features you have not used too often before (you may have misunderstood them)
  - read the documentation about these features
  - rethink your strategy for solving the problem
  - examine your program by adding print statements to show intermediate results

Open Source Programs (self study)

- programs are economically valuable ⇒ there is a long tradition to keep the source code hidden
- but many of the best and most used programs are freely available in source code form
- examples of important open source programs (see also `http://www.makemenoise.com/some-of-the-most-important-open-source-software-2/`
  - Linux OS
  - Apache Web Server
  - thunderbird and firefox
  - open office, PostgreSQL, ghostscript,
  - wine, cURL, MediaWiki, gimp
  - Ruby interpreter and Ruby libraries
- once you have some experience with programming, you may sometime look at the freely available code
- there you can learn how professional programmers write code

The Programming Process

- case study: solve the problem of counting regulatory elements in DNA
- go step by step as follows:
  - identify required inputs, such as data or information given by the user
  - make an overall design for the program, including the algorithm by which the program computes the output
  - decide how the outputs will print
  - refine the overall design by specifying more detail
  - write the Ruby code

The Design Phase

- collect necessary information from the user:
  - where does the input (DNA and regulatory elements) come from (e.g. filename, other programs etc.)?
  - in what format will the input be available?
  - what is the expected size of the input?
  - what should the output look like?
- develop an algorithm to perform the search
- algorithm: design or plan for the computation done by a computer
- algorithm works step by step and can be implemented in a programming language

The Design Phase (cont.)

– possible algorithm to the above problem:
  take every regulatory element and search it in the DNA
– solve string matching problem
– write down the algorithm in an informal way: pseudo code

```
get the name of the DNA file from the user

read in the DNA from the file

for each regulatory element
  if element is in DNA, then
    add one to the count

print count
```

Comments (self study)

– programs without comments are basically of no worth
– almost no maintenance possible
– use comments in your programs
– anything from `#` to the end of the line is comment
– exception: if the first line starts with a magic string

```
#!/usr/bin/env ruby
```

– many programming languages do not support commenting very much
– the comment needs extra syntax
– it should be the other way round
– literate programming style: the program parts are marked, comment is standard

Comments (self study) (cont.)

- comments should contain:
  - description of the overall purpose and design of the program ⇒ turn your pseudocode into a comment
  - examples of how to use the program
  - specify input and output formats
  - interspersed throughout the program: explain why the code is there and what it does
- another use of comments: debugging

  put the comment sign `#` at the beginning of a line which you do not want to be executed
- commenting this line may give you an idea of what went wrong

Syntax Rules

- statements appear on single lines
- semicolon at end of statement is **not** necessary
- syntax layout is up to the user
- but indentation of blocks by two blanks is common
- methods are defined by the keyword `def` followed by a method name and the method's parameters between parentheses `()`
- a sequence of compound statements is finished by the keyword `end`
- variables do not have to be declared; a variable springs into existence once we assigned to it
- the first character of an identifier indicates if the identifier is used for local variables
- method parameters, method names all start with a lower case letter or an underscore
- global variables are prefixed by the $-symbol

Syntax Rules (cont.)

- instance variables begin with one `@`-symbol, class variables with two (`@@`)
- class names, module names and constants must start with an uppercase letter, CONSTANTS usually consist of uppercase letters only
- following an initial character, an identifier can be any combination of letters, digits and underscores
- the symbol following `@` must not be a digit
- convention: multiword instance variables are written with underscores between the word
- convention: multiword class names are written with MixedCase (with each word capitalized)
- method names may end with the symbols `?`, `!`, and `=`

Representing Sequence Data

- goal: manipulate sequences of symbols representing DNA and proteins
- DNA consists of nucleic acids (nucleotides, bases)

A	Adenine
C	Cytosine
G	Cytosine
T	Thymine

Additionally:

U	Uracil (for RNA)
N	unknown base

- notation: DNA is sequence of bases in upper or lower case

Representing Sequence Data (cont.)

– protein consists of 20 aminoacids

`C`	Cysteine	Cys
`A`	Alanine	Ala
`R`	Arginine	Arg
`N`	Asparagine	Asn
`D`	Aspartic acid	Asp
`Q`	Glutamine	Gln
`E`	Glutamic acid	Glu
`G`	Glycine	Gly
`H`	Histidine	His
`I`	Isoleucine	Ile

`L`	Leucine	Leu
`K`	Lysine	Lys
`M`	Methionine	Met
`F`	Phenylalanine	Phe
`P`	Proline	Pro
`S`	Serine	Set
`T`	Threonine	Thr
`W`	Tryptophan	Trp
`Y`	Tyrosine	Tyr
`V`	Valine	Val

– notation: protein is sequence of aminoacids (one letter code and uppercase)

– sequence representation is often a simplification of reality

– but suffices for this course

Representing Sequence Data (cont.)

Some computer science terms:

– above tables define two *alphabets*, i.e. finite set of symbols

– *string*: sequence of symbols

– in general: computers use ASCII alphabet or superset thereof

– ASCII contains 128 characters numbered from 0 to 127

– each member of the ASCII alphabet denotes printable or non-printable character

– for example: ASCII 65 is `A`, ASCII 10 is newline, etc.

– Linux character set contains 256 characters (first 128 = ASCII alphabet)

A Ruby script to store a DNA sequence

```ruby
# Example 4-1   Storing DNA in a variable, and printing it out
# First we store the DNA in a variable called dna
dna = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'

# Next, we print the DNA onto the screen
puts dna

# Finally, we'll specifically tell the program to exit.
exit 0
```

- store the script in a textfile, say `example4-1.rb`
- on a Unix-Shell type the following commands

```
$ chmod ug+x example4-1.rb
$ example4-1.rb
ACGGGAGGACGGGAAAATTACTACGGCATTAGC
```

- statements of script are executed step by step from top to bottom
- `exit 0` terminates the script with return code 0
- comments begin with the symbol # and end at the end of the line
- first line: assignment statement; store the DNA in a variable `dna`

Variables and Assignments

- name of variable is arbitrary
- composed of upper & lower case letters, digits, and underscore _
- choose appropriate variable names
- name should reflect what the variable is for ⇒ self documenting code
- string is enclosed in single quotes
- double quotes would also work
- `=` is the assignment operator: variable to the left and expression to the right
- after assignment variable stores the assigned value
- use the variable to print the DNA sequence

Concatenating DNA Fragments

```
# Example 4-2   Concatenating DNA

# Store two DNA fragments into two variables called dna1 and dna2
dna1 = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'
dna2 = 'ATAGTGCCGTGAGAGTGATGTAGTA'

# Print the DNA onto the screen
print "Here are the original two DNA fragments:\n\n"

puts dna1
puts dna2, "\n"

# Concatenate DNA fragments into a third variable and print them
# using "string interpolation"
dna3 = "#{dna1}#{dna2}"

puts "Concatenation of the first two fragments (version 1):\n"

puts "#{dna3}\n"

# An alternative way using the concatenation operator:
# Concatenate DNA fragments into a third variable and print them
```

Concatenating DNA Fragments (cont.)

```
dna3 = dna1 + dna2

puts "Concatenation of the first two fragments (version 2):\n"

puts "#{dna3}\n"

# Print the same thing without using the variable dna3
puts "Concatenation of the first two fragments (version 3):\n"

print dna1, dna2, "\n"

exit 0
```

- print statements show the value of the variables in the different steps
- formatting using the newline symbol `\n`

Concatenating DNA Fragments (cont.)

```
Here are the original two DNA fragments:

ACGGGAGGACGGGAAAATTACTACGGCATTAGC
ATAGTGCCGTGAGAGTGATGTAGTA

Concatenation of the first two fragments (version 1):

ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA

Concatenation of the first two fragments (version 2):

ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA

Concatenation of the first two fragments (version 3):

ACGGGAGGACGGGAAAATTACTACGGCATTAGCATAGTGCCGTGAGAGTGATGTAGTA
```

Concatenating DNA Fragments (cont.)

- the statement `dna3 = "#{dna1}#{dna2}"` concatenates the contents of variable `dna1` and `dna2` and stores the result in `dna3`
- double quotes ⇒ expression `#{e}` is evaluated. The resulting value is inserted (string interpolation).
- a different way to do the concatenation uses operator `+`:

```
dna3 = dna1 + dna2
```

- operator: takes some arguments and does something with them
- e.g. addition, subtraction, multiplication, division operators in arithmetic expressions
- variable can hold a string (as in the example) but also an integer, a floating-point number, or boolean value

```
number1 = 42
number2 = 56

puts number1 + number2
```

Transcription: DNA to RNA

```
# Example 4-3   Transcribing DNA into RNA

# The DNA
dna = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'

# Print the DNA onto the screen
puts "Here is the starting DNA:\n"

puts "#{dna}\n"

# Transcribe the DNA to RNA by substituting all T's with U's.
rna = dna.gsub(/T/,'U')

# Print the RNA onto the screen
puts "Here is the result of transcribing the DNA to RNA:\n"

puts "#{rna}"

exit 0
```

Transcription: DNA to RNA (cont.)

- fourth statement makes copy of DNA sequence in variable `RNA`
- transcription happens in the statement

  `rna = dna.gsub(/T/,"U")`

- substitution method `gsub` performs substitution

```
     str.gsub(pattern, replacement)       => new_str
------------------------------------------------------------------------
     Returns a copy of _str_ with _all_ occurrences of _pattern_
     replaced with either _replacement_ or the value of the block. The
     _pattern_ will typically be a +Regexp+; if it is a +String+ then no
     regular expression metacharacters will be interpreted (that is
     +/\d/+ will match a digit, but +'\d'+ will match a backslash
     followed by a 'd').

        "hello".gsub(/[aeiou]/, '*')              #=> "h*ll*"
        "hello".gsub(/([aeiou])/, '<\1>')         #=> "h<e>ll<o>"
```

Calculating the Reverse Complement

```
# Example 4-4   Calculating the reverse complement of a DNA-strand

dna = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC'
puts "Here is the DNA:\n"
puts "#{dna}\n"

# Calculate the reverse complement Warning: this attempt will fail!
# First, copy the DNA into new variable revcom
# (short for REVerse COMplement)
# It doesn't matter if we first reverse the string and then do the
# complementation; or if we first do the complementation and
# then reverse the string. Same result each time. So when we
# make the copy we'll do the reverse in the same statement.

revcom = dna.reverse

# Next substitute all bases by their complements
revcom.gsub!(/A/,"T")
revcom.gsub!(/T/,"A")
revcom.gsub!(/G/,"C")
revcom.gsub!(/C/,"G")
```

Calculating the Reverse Complement (cont.)

```
puts "Here is the incorrect result:\n#{revcom}"

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

# Make a new copy of the DNA (see why we saved the original?)
revcom = dna.reverse

# See the text for a discussion of tr
revcom.tr!("ACGTacgt","TGCAtgca")

# Print the reverse complement DNA onto the screen
puts "Here is the reverse complement DNA:\n#{revcom}"
```

Calculating the Reverse Complement (cont.)

```
Here is the DNA:

ACGGGAGGACGGGAAAATTACTACGGCATTAGC

Here is the incorrect result:

GGAAAAGGGGAAGAAAAAAAGGGGAGGAGGGGA

Here is the reverse complement DNA:

GCTAATGCCGTAGTAATTTTCCCGTCCTCCCGT
```

Calculating the Reverse Complement (cont.)

- let's recapitulate the algorithmic idea:
- apply the substitution step by step
- look at each base one at a time, make the change to the complement
- then look at the next base in the DNA
- `tr`-method is exactly suited for this task:
  translates a set of characters into a new set, all at once
- each character in the first set is translated into the character at the same position in the second set

Calculating the Reverse Complement (cont.)

```
     str.tr!(from_str, to_str)   => str or nil
------------------------------------------------------------------------
     Translates _str_ in place, replaces characters in from_str by the
     corresponding characters in _to_str_. If _to_str_ is shorter than
     _from_str_, it is padded with its last character. Both strings may use
     the c1--c2 notation to denote ranges of characters, and _from_str_ may
     start with a +^+, which denotes all characters except those listed.

        "hello".tr!('aeiou', '*')    #=> "h*ll*"
        "hello".tr!('^aeiou', '*')   #=> "*e**o"
        "hello".tr!('el', 'ip')      #=> "hippo"
        "hello".tr!('a-y', 'b-z')    #=> "ifmmp"
```

Reading proteins in files

- previous examples: sequences were hard coded in the script
- but usually sequences are stored on a file
- for example, the file `NM_021964fragment.pep` stores

```
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

- filenames can be arbitrary, but they should somehow reflect the file contents
- e.g. file above is from Genbank: ID is `NM_021964`
- sequence is a fragment and protein sequence data or peptide

Reading proteins in files (cont.)

```
# Example 4-5 Reading protein sequence data from a file

# The filename of the file containing the protein sequence data
proteinfilename = "NM_021964fragment.pep"

# First we create a new File object.
# We name it "proteinfile" for readability.
proteinfile = File.new(proteinfilename, "r")

# Now we do the actual reading of the protein sequence data from
    the file
# by calling the "readline" method of the File object.
protein = proteinfile.readline

# Now that we've got our data, we can close the file.
proteinfile.close

# Print the protein onto the screen
puts "Here is the protein:\n#{protein}"
```

Reading proteins in files (cont.)

```
Here is the protein:
MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
```

- `new` method of class `File` opens the file and delivers a File object named `proteinfile`
- all interactions with the file are done via file object
- reading, writing, searching, erasing the file content
- `readline` method reads a single line of the file (namely the first line)
- this line is stored in the string object `protein` and then printed out

Reading proteins in files (cont.)

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

# The filename of the file containing the protein sequence data
proteinfilename = '../Basic/NM_021964fragment.pep'

# First we have create a new File object.
# We name it "proteinfile" for readability.
proteinfile = File.new(proteinfilename, "r")

# Now we reading the protein sequence data from the file
# by calling the "readline" method of the File object.
#
# Since the file has three lines, and since the read only is
# returning one line, we'll read a line and print it, three times.

# First line
protein = proteinfile.readline

# Print the protein onto the screen
puts "Here is the first line of the protein file:\n#{protein}"

# Second line
protein = proteinfile.readline
```

Reading proteins in files (cont.)

```
# Print the protein onto the screen
puts "Here is the second line of the protein file:\n#{protein}"

# Third line
protein = proteinfile.readline

# Print the protein onto the screen
puts "Here is the third line of the protein file:\n#{protein}"

# Now that we've got our data, we can close the file.
proteinfile.close

exit 0
```

Reading proteins in files (cont.)

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

MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD

Here is the second line of the protein file:

SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ

Here is the third line of the protein file:

GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

- script reads in sequence line by line
- every time the contents of the current line is bound to a variable
- the File object remembers where the previous read was
- drawback: each line in the file requires extra code

Reading proteins in files (cont.)

- a more elegant solution which
  - works for any file and
  - handles the case that the file cannot be opened, using exceptions
- A robust program will handle exceptions, that is, unexpected situations (like file that cannot be opened, or disks that are full)
- handle exceptions for blocks of code
- the block of code marked with begin executes until there is an exception, which causes control to be transferred to a block of error handling code, which is marked with rescue
- If no exception occurs, the rescue code is not used.

```
proteinfilename = 'NM_021964fragment.pep'
puts "Try to open \"#{proteinfilename}\""

# First we have to open the file, and in case the
# open fails, print an error message and exit the program.
begin
  proteinfile = File.new(proteinfilename,"r")
rescue => err
  STDERR.puts "Could not open file \"#{proteinfilename}\": #{err}"
  exit 1
end
```

Reading proteins in files (cont.)

- call to the method `new` is a system call: Ruby must ask for the file from the operating system
- in case of failure handle the error by printing an error message
- otherwise you will not be able to figure out where your script went wrong
- important to check for success or failure of anything that can go wrong
- `rescue` handles an exception and prints out the error message

```
# Read protein sequence data from file in a block
proteinfile.each do |line|
  puts "  ######  Here is the next line of the file:"
  print line
end
proteinfile.close
```

- using the method `each`, we iterate over all lines of the open file
- the local variable `line` contains the current line
- it is printed out

Arrays

- arrays allow to store a set of value (not necessarily of the same type)

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

# The filename of the file containing the protein sequence data
proteinfilename = 'NM_021964fragment.pep'

# First we create a file object
proteinfile  = File.new(proteinfilename, "r")

# Read protein sequence data from file, and store it into array
proteins = proteinfile.readlines

proteins.each_with_index do |line, idx|
  print "#{idx+1}: #{line}"   # Print line with linnumber
end
proteinfile.close     # Close the file.

exit 0
```

Arrays (cont.)

- the result:

```
1: MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
2: SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
3: GLQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

- advantage: only one read statement `proteins = proteinfile.readlines`
- each element in an array is referenced by its position

Arrays (cont.)

```
bases = ['A', 'C', 'G', 'T']
puts "The array elements:"
puts "First element: #{bases[0]}"
puts "Second element: #{bases[1]}"
puts "Third element: #{bases[2]}"
puts "Fourth element: #{bases[3]}"
```

```
Here are the array elements:
First element: A
Second element: C
Third element: G
Fourth element: T
```

Arrays (cont.)

– variation: print the elements one after each other:

```
bases = ['A', 'C', 'G', 'T']
puts "The array elements: #{bases}"
```

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

Arrays (cont.)

– variation: print the elements one after each other separated by spaces:

```
bases = ['A', 'C', 'G', 'T']
print "The array elements: "
puts bases.join(" ")
```

```
Here are the array elements: A C G T
```

Arrays (cont.)

- take an element off the end of the array with `pop`

```
bases = ['A', 'C', 'G', 'T']
base1 = bases.pop
puts "The element removed from the end: #{base1}"
puts "The remaining array of bases: #{bases}"
```

```
Here is an element removed from the end: T
Here is the remaining array of bases: ACG
```

Arrays (cont.)

- take a base of the beginning of the array with `shift`

```
bases = ['A', 'C', 'G', 'T']
base2 = bases.shift
puts "The element removed from the beginning: #{base2}"
puts "The remaining array of bases: #{bases}"
```

```
Here is an element removed from the beginning: A
Here is the remaining array of bases: CGT
```

Arrays (cont.)

- put an element at the beginning of the array with `unshift`

```
bases = ['A', 'C', 'G', 'T']
base1 = bases.pop
bases.unshift(base1)
puts "An element from the end put on the beginning: #{bases}"
```

```
Here is an element from the end put on the beginning: TACG
```

Arrays (cont.)

- put an element on the end of the array with `push`

```
bases = ['A', 'C', 'G', 'T']
base2 = bases.shift
bases.push(base2)
puts "An element from the beginning put on the end: #{bases}"
```

```
Here is an element from the beginning put on the end: CGTA
```

Arrays (cont.)

– get the length of the array with `length`:

```
bases = ['A', 'C', 'G', 'T']
puts "The length of the array: #{bases.length}"
```

```
Here is the length of the array: 4
```

Arrays (cont.)

– insert an element at an arbitrary place in an array with `insert`:

```
bases = ['A', 'C', 'G', 'T']
bases.insert(2,"X")
print "The array with an element inserted after the 2nd element: "
puts "#{bases}"
```

```
Here is the array with an element inserted after the 2nd element: ACXGT
```

Flow of Control

- script executes step by step in sequential order
- two ways to organize the execution in another way: conditional statements and loops
- same principle as in other languages with minimal syntax

  ```
  if 1 == 1
    puts "1 equals 1"
  end
  ```

  produces the output

  ```
  1 equals 1
  ```

- note that test for equality is written with the operator `==`
- `1` (as any other number not equal to 0) evaluates to `true`

Flow of Control (cont.)

```
if 1
  puts "1 evaluates to true"
end
```

- the `if` can optionally be followed by an `else`

  ```
  if 1 == 0
    puts "1 equals 0"
  else
    puts "1 does not equal 0"
  end
  ```

- `if not` can be expressed by `unless`

  ```
  unless 1 == 0
    puts "1 does not equal 0"
  end
  ```

- note that blocks are closed by the keyword `end`

Flow of Control (cont.)

- here is a (incomplete) list of operators to be used in conditions

==	equality
!=	inequality
<	lower than
>	greater than
<=	lower equal
>=	greater equal

Flow of Control (cont.)

```
word = 'MNIDDKL'
if word == 'QSTVSGE'
  puts "QSTVSGE"
elsif word == 'MRQQDMISHDEL'
  puts "MRQQDMISHDEL"
elsif word == 'MNIDDKL'
  puts "MNIDDKL-the magic word!"
else
  puts "Is \"#{word}\" a peptide? This cannot be decided."
end
exit 0
```

- prints the following result:

```
MNIDDKL-the magic word
```

- note the use of `\"`: it prints the doubleqote inside a double quoted string

Finding Motifs

- one of the most common things done in bioinformatics
- motif is a short segment of DNA or protein of special interest, e.g. regulatory elements
- motifs are usually not simple strings: some positions are unspecific $\Rightarrow$ does not matter what base or residue is there
- regular expressions are convenient to express motifs
- at the end of this section you find a formal definition of regular expressions
- best explained by a Ruby script which
  - reads in protein sequence data from file
  - puts in the sequence data into one string for easy searching
  - looks for motifs, the user types in at the keyboard

Finding Motifs (cont.)

```ruby
# Ask the user for the filename of the file containing
# the protein sequence data, and collect it from the keyboard
print "type filename of the protein sequence: "
proteinfilename = STDIN.gets

# Remove the trailing newline from the protein filename
proteinfilename.chomp!

# open the file, or exit
begin
  proteinfile = File.open(proteinfilename,"r")
rescue
  STDERR.puts "Could not open file #{proteinfilename}!"
  exit 1
end

# Read the protein sequence data from the file, and store it
# into the array variable "protein"
protein = proteinfile.readlines

# Close the file - we've read all the data into "protein" now.
proteinfile.close
```

Finding Motifs (cont.)

```
# Put the protein sequence data into a single string, as it's
# easier to search for a motif in a string than in an array
# of lines (what if the motif occurs over a line break?)
proteinseq = protein.join

# Remove whitespace and '>' symbols
proteinseq.gsub!(/(\s|\>)/,"")

# In a loop, ask the user for a motif, search for the motif,
# and report if it was found. break out of the loop if no
# motif is entered.
loop do
  print "Enter a motif to search for: "
  motif = STDIN.gets
  motif.chomp!
  puts "checking motif \"#{motif}\""
  if motif.match(/^\s*$/)
    break
  end
  if motif.match(/^quit$/)
    break
  end
```

Finding Motifs (cont.)

```
  puts "searching motif \"#{motif}\""
  if proteinseq.match(/#{motif}/)
    puts "I found it!"
  else
    puts "I couldn\'t find it."
  end
end
```

Finding Motifs (cont.)

- Here is an example of the output:

```
type filename of protein sequence: NM_021964fragment.pep
Enter a motif to search for: SVLQ
I found it!
Enter a motif to search for: jkl
I did not find it!
Enter a motif to search for:
```

- recall the following line:

```
motif = STDIN.gets
```

Finding Motifs (cont.)

- `STDIN` is a special file object which reads from standard input, e.g. from the keyboard
- in this case: read the file name
- user types in filename and a newline (enter key)
- newline is part of the string read from the keyboard
- Ruby function `chomp!` removes newlines from the end of a string
- protein sequence data is usually formatted into lines of some fixed length
- formatting newline characters must be removed from the input, before processing it
- use method `join` to combine all lines into a single string

```
proteinseq = protein.join
```

Finding Motifs (cont.)

- regular expression: enclosed by `/`
- match one or more strings using special wildcard characters
- for example `\s` matches any space character

```
protein.gsub!(/(\s|\>)/,"")
```

- removes any whitespace (space, tab, newline, carriage return, formfeed) in the protein sequence ⇒ `\s` can be written as

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

- now consider another example of a regular expression

```
if /^\s*$/.match(motif)
```

- test for a blank line in the variable `motif`
- match a string that, from the beginning to the end, contains only whitespaces

Finding Motifs (cont.)

- now consider the next line:

```
if /#{motif}/.match(protein)
```

- this interpolates the value of a variable into a string match
- simplest regular expression: just a string of characters, e.g. `AQQK`

```
A[DS]V
```

- search for an `A` followed by `D` or `S`, followed by `V`

```
KND*E{2,}
```

- search for `K`, followed by `N`, followed by zero or more `D`'s, and two or more `E`'s

```
EE.*EE
```

- search for two `E`'s followed by anything, followed by another two `E`'s

Counting Bases

```ruby
# Get the name of the file with the DNA sequence data
print "Please type the filename of the DNA sequence data: "
dnafilename = STDIN.gets

# Remove the newline from the DNA filename
dnafilename.chomp!

# open the file, or exit
if File.exist?(dnafilename)
  begin
    dnafile = File.new(dnafilename,"r")
  rescue => err
    STDERR.puts "Could not open file #{dnafilename}: #{err}"
    exit 1
  end
else
  STDERR.puts "File #{dnafilename} does not exist!"
  exit 1
end

# Read the DNA sequence data from the file, and store it
```

Counting Bases (cont.)

```ruby
# as a concatenated string in the variable "dna"
dna = dnafile.read

# Close the file
dnafile.close

# Remove whitespace
dna.gsub!(/\s/,"")

# Now explode the DNA into an array where each letter of the
# original string is now an element in the array.
# This will make it easy to look at each position.
dna = dna.split(//)

# Initialize the counts.
count_of_A = 0
count_of_C = 0
count_of_G = 0
count_of_T = 0
errors     = 0

# In a loop, look at each base in turn, determine which of the
# four types of nucleotides it is, and increment the
```

Counting Bases (cont.)

```
# appropriate count.
dna.each do |base|
  if base == 'A'
    count_of_A += 1
  elsif base == 'C'
    count_of_C += 1
  elsif base == 'G'
    count_of_G += 1
  elsif base == 'T'
    count_of_T += 1
  else
    STDERR.puts "Error - I don\'t recognize this base: #{base}"
    errors += 1
  end
end

puts "A = #{count_of_A}"
puts "C = #{count_of_C}"
puts "G = #{count_of_G}"
puts "T = #{count_of_T}"
puts "errors = #{errors}"
exit 0
```

An Alternative Way of Counting

– Use the `scan` method, after joining the array into a string

```
dna = dnafile.readlines.join

countA = dna.scan(/a/i).length
countC = dna.scan(/c/i).length
countG = dna.scan(/g/i).length
countT = dna.scan(/t/i).length
errors = dna.scan(/[^acgt]/i).length

puts "A=#{countA} C=#{countC} G=#{countG} T=#{countT}"
puts "errors=#{errors}"
```

An Alternative Way of Counting (cont.)

```
------------------------------------------------------------ String#scan
     str.scan(pattern)                         => array
------------------------------------------------------------------------
     Iterate through _str_, matching the pattern (which may
     be a +Regexp+ or a +String+). For each match, a result is generated
     and added to the result array. If the
     pattern contains no groups, each individual result consists of the
     matched string, +$&+. If the pattern contains groups, each
     individual result is itself an array containing one entry per
     group.

        a = "cruel world"
        a.scan(/\w+/)        #=> ["cruel", "world"]
        a.scan(/.../)        #=> ["cru", "el ", "wor"]
        a.scan(/(...)/)      #=> [["cru"], ["el "], ["wor"]]
        a.scan(/(..)(..)/)   #=> [["cr", "ue"], ["l ", "wo"]]
```

Write the Results to a File

```
outputfilename = "countbase"

# write the results to a file named countbase
begin
  countbase = File.new(outputfilename,"w")
rescue
  STDERR.puts "Could not open file #{outputfilename}!"
  exit 1
end

countbase.puts "A=#{countA} C=#{countC} G=#{countG} T=#{countT}"
countbase.close
```

– for each base and the non-base characters there is a separate while loop to count the number of occurrences
– string matching expression `dna.scan(/a/i)` is looking for regular expression `a`
– since option `i` is used, it matches case insensitive

Formal Definition of Regular Expressions

Ruby allows a convenient notation for regular expressions. To define regular expressions we first introduce the notion of atoms:
An atom represents a subset of the characters in the underlying alphabet Σ. It consists of one of the following items:

- A single character which matches itself.
- A wildcard . which matches any character in Σ.
- A character class denoted by $[c_0 \dots c_k]$ where $k \geq 0$ and $c_0, \dots, c_k$ are pairwise distinct characters from Σ. Such a character class matches any of the characters $c_0, \dots, c_k$.
- A complement character class denoted by $[\hat{}\, c_0 \dots c_k]$ where $k \geq 0$ and $c_0, \dots, c_k$ are pairwise distinct characters from Σ. Such a character class matches any of the characters from Σ, except for $c_0, \dots, c_k$.

Formal Definition of Regular Expressions (cont.)

Regular expressions are constructed according to the following rules:

- Any atom is a regular expression.
- If r and s are regular expressions, then so is the concatenation rs.
- If r and s are regular expressions, then so is the alternation $r|s$.
- If r is a regular expression, then so is (r).
- If r is a regular expression and i, j are non-negative numbers such that $i \leq j$, then $r\{i,j\}$, $r\{i,\}$, $r\{,j\}$, and $r\{i\}$ are regular expressions.
- If r is a regular expression then so is $r*$, $r+$, and $r?$.

Formal Definition of Regular Expressions (cont.)

For any $k \geq 0$ and any regular expression r, r^k denotes the regular expression consisting of the concatenation of k copies of r. For any regular expression r over alphabet Σ, the *language* $\mathcal{L}(r) \subseteq \Sigma^*$ of r is defined as follows:

$$\begin{array}{rcl}
\mathcal{L}(a) & = & \{a\} \text{ for any character } a \text{ in } \Sigma \\
\mathcal{L}(.) & = & \Sigma \\
\mathcal{L}([c_1 c_2 \ldots c_r]) & = & \{c_1, c_2, \ldots, c_r\} \\
\mathcal{L}([\hat{}\, c_1 c_2 \ldots c_r]) & = & \Sigma \setminus \{c_1, c_2, \ldots, c_r\} \\
\mathcal{L}(rs) & = & \{xy \mid x \in \mathcal{L}(r), y \in \mathcal{L}(s)\} \\
\mathcal{L}(r|s) & = & \mathcal{L}(r) \cup \mathcal{L}(s) \\
\mathcal{L}((r)) & = & \mathcal{L}(r)
\end{array}$$

Formal Definition of Regular Expressions (cont.)

$$\begin{array}{rcl}
\mathcal{L}(r\{i,j\}) & = & \bigcup_{i \leq k \leq j} \mathcal{L}(r^k) \\
\mathcal{L}(r\{i,\}) & = & \bigcup_{i \leq k} \mathcal{L}(r^k) \\
\mathcal{L}(r\{,j\}) & = & \bigcup_{0 \leq k \leq j} \mathcal{L}(r^k) \\
\mathcal{L}(r\{i\}) & = & \mathcal{L}(r^i) \\
\mathcal{L}(r*) & = & \{\varepsilon\} \cup \mathcal{L}(rr*) \\
\mathcal{L}(r+) & = & \mathcal{L}(r*) \setminus \{\varepsilon\} \\
\mathcal{L}(r?) & = & \mathcal{L}(r) \cup \{\varepsilon\}
\end{array}$$

For any regular expression r and any string $w \in \mathcal{L}(r)$, we say that r *matches* w.

Function Definition and Scoping

- Functions are an important concept to structure your program code
- advantage of using functions:
  - part of code becomes reusable (no paste and copy): faster to write and more reliable code
  - easier to test: functions can be tested separately
  - helps to organize and to abstract your ideas
  - improves readability

Function Definition and Scoping (cont.)

```
# Example 6-1  program with a function to append ACGT to DNA

def addACGT(dnaparam)
  dnaparam += 'ACGT'
  return dnaparam
end

# The original DNA
dna = 'TGCA'

# The call to the function "addACGT".
# The argument being passed in is "dna"
# the result is saved in "longer_dna"
longer_dna = addACGT(dna)

puts "I added ACGT to #{dna} and got #{longer_dna}"
```

- produces the following output:

```
I added ACGT to TGCA and got TGCAACGT
```

Function Definition and Scoping (cont.)

– function declaration starts with keyword `def` followed by a function name and a list of parameters
– function declarations ends with keyword `end`
– function returns value in a `return` statement

Command line arguments and Arrays

– until now we have always used STDIN to accessed information from the outside of a script e.g. names of file
– another way is to read command line arguments (as almost every Unix-command does)

```
# Example 6-3   Count number of G's in some DNA from command line

# Collect DNA from the arguments on the command line
# If no arguments are given, print a USAGE statement and exit.

def countG(dna)
  # return a count of the number of G's in the argument dna
  # Use the String object's built-in method "count" to determine
  # abundance of "g"s.
  return dna.downcase.count("g")
end

# $0 is a special variable that has the name of the program.
USAGE = "Usage: #{$0} DNA"
```

Command line arguments and Arrays (cont.)

```
# ARGV is an array containing all command-line arguments. If it is
# empty, the test will fail and the print USAGE and exit statements
# will be called.
if ARGV.length != 1
  STDERR.puts USAGE
  exit 1
end

# Read in the DNA from the argument on the command line.
dna = ARGV[0]

# Call the function, collect the result and print it.
num_of_Gs = countG(dna)
puts "The DNA #{dna} has #{num_of_Gs} G\'s in it!"
```

Command line arguments and Arrays (cont.)

- every Ruby script has two special variables:
- `ARGV` contains any command line arguments
- $0 is the name of the script as it was called from the command line
- `USAGE` defines an informative message on how the script should be called
- if `ARGV` has anything in it, `empty?` evaluates to `false`; otherwise to `true`

  ```
  dna = ARGV[0]
  ```

- extracts the first element in the array

Passing Data to Functions

```
# Here, the parameters are named, and the order when
# passing determines assignment. You can also do this:

def func1(a, b, c)
  puts "#{__method__}: #{a} #{b} #{c}"
end

func1(1, 2, 3)

# Here, the third parameter has a default, so it can be
# omitted if desired, in order to use the default value.

def func2(a, b, c=3)
  puts "#{__method__}: #{a} #{b} #{c}"
end

func2(1, 2)
func2(1, 2, 5)

# Next, the '*' operator lets you do variable argument
# lists. All "extra" arguments get put into an array
```

Passing Data to Functions (cont.)

```
# in the variable specified.

def func3(a, b, *otherargs)
  puts "#{__method__}: #{a} #{b} #{otherargs.join(' ')}"
end

# a=1, b=2, otherargs=[3,4,5]
func3(1, 2, 3, 4, 5)
# a=1, b=2, otherargs=[]
func3(1, 2)
```

Output:

```
func1: 1 2 3
func2: 1 2 3
func2: 1 2 5
func3: 1 2 3 4 5
func3: 1 2
```

Passing Data to Functions (cont.)

- Ruby uses call-by-value parameter passing, where the value is a reference to an object
- in other words: copy of the reference is passed to the method.

⇒ this allows the referred-to object to be manipulated via the reference copy

⇒ those changes would be reflected on return of the method

- BUT any changes made to the reference copy itself will not be reflected on return of the method (as the scope of this reference copy is just within the method).
- assigning a new object to the reference copy would only have the scope of that method, i.e. on return the original reference would be unchanged.

Passing Data to Functions (cont.)

```
def passintandstrings(param1,param2,param3)
  param1 = 5
  param2 = "five"
  param3.capitalize!
  print "in #{__method__}:"
  puts "param1=#{param1} param2=#{param2} param3=#{param3}"
end

intval = 3
strval1 = "three"
strval2 = "five"
passintandstrings(intval,strval1,strval2)
print "end:"
puts "intval=#{intval} strval1=#{strval1} strval1=#{strval2}"
```

Output:

```
in passintandstrings:param1=5 param2=five param3=Five
end:intval=3 strval1=three strval1=Five
```

Passing Data to Functions (cont.)

```ruby
def pushshift(itab,jtab)
  puts "in function: itab = #{itab}"
  puts "in function: jtab = #{jtab}"
  itab.push('4')
  jtab.shift
end

itab = ['1','2','3']
jtab = ['a','b','c']

puts "in main before calling pushshift: itab = #{itab}"
puts "in main before calling pushshift: jtab = #{jtab}"

pushshift(itab,jtab)

puts "in main after calling pushshift: itab = #{itab}"
puts "in main after calling pushshift: jtab = #{jtab}"
```

Passing Data to Functions (cont.)

– leads to the following result:

```
in main before calling function: itab = 123
in main program before calling function: jtab = abc
in function: itab = 123
in function: jtab = abc
in main after calling function: itab = 1234
in main after calling function: jtab = bc
```

Modules and Libraries of Functions

- to build reusable software it is necessary to distribute program code over modules and libraries
- put your function into a separate file, for example mylib.rb
- with a statement `require "mylib.rb"` in your main program you can use the functions there
- if you have `mylib.rb` in a different directory, then you can e.g. use

```
require "/home/joeuser/rubydir/mylibdir/mylib.rb"
```

Hashes

- main data types in Ruby: numbers, strings, arrays, and hashes
- this section introduces hashes (also called associative arrays)
- Hashes provide a fast lookup of the value associated with a key
- the values can be defined as follows:

```
english2german['pearl'] = "Perle"
```

- and lookup is done as follows:

```
germanword = english2german['pearl']
```

- `pearl` is the key-value, and the returned value is associated with the key

Hashes (cont.)

- a hash always introduces a finite mapping from keys to values, as reflected in the following

```
english2german =
{
  'dog'   => 'Hund',
  'robin' => 'Rotkehlchen',
  'asp'   => 'Natter'
}
```

- the keys of a hash are extracted with the function `keys`

```
translatedwords = english2german.keys
```

- the values of a hash are extracted into a list with the function `values`

```
translations = english2german.values
```

Translating a Codon into an Aminoacid

- take a group of three consecutive nucleotides (codon) from DNA and translate it into an aminoacid
- this view simplifies the central dogma of molecular biology: first transcription into RNA and then translation to proteins

```
def codon2aa(codon)
  codon.upcase!
  if    codon == 'TCA' return 'S'     # Serine
  elsif codon == 'TCC' return 'S'     # Serine
  elsif codon == 'TCG' return 'S'     # Serine
  elsif codon == 'TCT' return 'S'     # Serine
  ... 57 more similar lines
  elsif codon == 'GGC' return 'G'     # Glycine
  elsif codon == 'GGG' return 'G'     # Glycine
  elsif codon == 'GGT' return 'G'     # Glycine
  else
    STDERR.print "Bad codon \"#{codon}\"!!\n";
    exit 1
  end
end
```

Translating a Codon into an Aminoacid (cont.)

- code serves its purpose, but many checks are necessary to perform the translation
- 64 possible codons and 20 aminoacids
- translation is not injective, i.e. the different codons may translate into the same aminoacid
- genetic code is redundant ⇒ using regexps we can enumerate the codons which translate into the same aminoacid:

```
def codon2aa(codon)
  codon.upcase!
  if       /GC./.match(codon) return 'A'  # Alanine
  elsif /TG[TC]/.match(codon) return 'C'  # Cysteine
  elsif /GA[TC]/.match(codon) return 'D'  # Aspartic Acid
  elsif /GA[AG]/.match(codon) return 'E'  # Glutamic Acid
  elsif /TT[TC]/.match(codon) return 'F'  # Phenylalanine
  ...  7 more similar lines
```

Translating a Codon into an Aminoacid (cont.)

```
  elsif        /GT./.match(codon) return 'V'  # Valine
  elsif        /TGG/.match(codon) return 'W'  # Tryptophan
  elsif     /TA[TC]/.match(codon) return 'Y'  # Tyrosine
  elsif /TA[AG]|TGA/.match(codon) return '_'  # Stop
  else
    STDERR.print "Bad codon \"#{codon}\"!!\n";
    exit 1
  end
 end
```

- `/[TC]/` matches a single character, either `T` or `C`
- `/TC.|AG[TC]/` matches `/TC./` or `/AG[TC]/`

Translating a Codon into an Aminoacid (cont.)

– a third variant of `codon2aa` uses hashes:

```
def codon2aa(codon)
  genetic_code = {
    'TCA' => 'S',    # Serine
    'TCC' => 'S',    # Serine
    'TCG' => 'S',    # Serine
    'TCT' => 'S',    # Serine
    'TTC' => 'F',    # Phenylalanine
     ...  55 more similar lines
    'GGA' => 'G',    # Glycine
    'GGC' => 'G',    # Glycine
    'GGG' => 'G',    # Glycine
    'GGT' => 'G',    # Glycine
  }

 if genetic_code.has_key?(codon)
   return genetic_code[codon]
 else
   STDERR.print "Bad codon \"#{codon}\"!!\n"
   exit 1
 end
end
```

Translating a Codon into an Aminoacid (cont.)

– first part consists of defining a hash with 64 entries
– keys are the codons and value are the associated aminoacids
– function `has_key?` tests if the key `codon` exists in the hash

Translating DNA into proteins

```
dna = 'CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC'
protein = ''

# translate each codon into amino acid, and append to protein
dna.scan(/[GATC]{3}/) do |codon|
  protein += codon2aa(codon)
end

puts "translated DNA\n#{dna}\ninto protein\n#{protein}"
```

– this gives the result:

```
translated DNA
CGACGTCTTCGTACGGGACTAGCTCGTGTCGGTCGC
into protein
RRLRTGLARVGR
```

– `scan` marches over the DNA, matching each sequence of three consecutive bases

Translating DNA into proteins (cont.)

```
------------------------------------------------------------ String#scan
     str.scan(pattern) {|match, ...| block }   => str
------------------------------------------------------------------------
     Iterate through _str_, matching the pattern (which may
     be a +Regexp+ or a +String+). For each match, a result is
     passed to the block. If the
     pattern contains no groups, each individual result consists of the
     matched string, +$&+. If the pattern contains groups, each
     individual result is itself an array containing one entry per
     group.

        a = "cruel world"
        a.scan(/\w+/) {|w| print "<<#{w}>> " }
        print "\n"
        a.scan(/(.)(.)/) {|a,b| print b, a }
        print "\n"

     _produces:_

        <<cruel>> <<world>>
        rceu lowlr
```

Translating DNA into proteins (cont.)

- we use a block in `do` … `end` notation to access the codons one after the other from left to right
- each codon is translated into the corresponding amino acid, which is appended to the current protein
- the functionality from above will be used more than once
- make a function out of it

```
def dna2peptide(dna)
  protein = ''
  dna.scan(/[GATC]{3}/) do |codon|
    protein += codon2aa(codon)
  end
  return protein
end
```

Reading DNA from Files in FASTA format

- FASTA format is basically just a line of sequence data with newlines at the end
- length of line is not specified, but it is best to limit the line length to some constant ≤ 80
- each sequence in FASTA-formatted file has header line
- this is a line beginning with the character > followed by some or no text

```
>gi|16127994|ref|NC_000913.1| Escherichia coli K12 genome
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAA
#this is a comment
TTCTGAACTGGTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAG
CCCGCACCTGACAGTGCGGGCTTTTTTTTTCGACCAAAGGTAA
...
```

Reading DNA from Files in FASTA format (cont.)

– the following extracts the sequence data from a FASTA file named `filename`

```
def extract_sequence_from_fasta_data(filename)
  seq = ''
  File.open(filename,"r").each_line do |line|
    if not line.match(/^(>|\s*$|\s*#)/)
      seq += line
    end
  end
  return seq.gsub(/\s/,"")
end
```

– the following prints the sequence in lines of a given maximal length

```
def print_sequence(seq, linelength)
  pos = 0
  while pos < seq.length do
    puts seq[pos..pos+linelength-1]
    pos += linelength
  end
end
```

Reading DNA from Files in FASTA format (cont.)

– finally lets put everything together to read and write a fasta file

```
require "extract_sequence.rb"
require "print_sequence.rb"

dna = extract_sequence_from_fasta_data("sample.dna")
print_sequence(dna,60)
```

Mapping Restriction Enzymes

– restriction enzymes are proteins that cut DNA at short, specific sequences
– example: EcoIRI cuts between `G` and `A` where it finds `GAATTC`
– example: HindIII cuts between the `A`s where it finds `AAGCTT`
– there are about 1000 known restriction enzymes
– a restriction map shows all positions where a given restriction enzyme cuts
– they are very important for planning wet-lab experiments
– goal of this section: write a Ruby script that looks for restriction enzymes in a sequence
– the Restriction Enzyme Database available here `http://rebase.neb.com/rebase/rebase.html`

Mapping Restriction Enzymes (cont.)

– here are the first 20 lines of the restriction data file:

```
REBASE version 301                                              bionet.301

    =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
    REBASE, The Restriction Enzyme Database   http://rebase.neb.com
    Copyright (c)  Dr. Richard J. Roberts, 2002.   All rights reserved.
    =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

Rich Roberts                                                    Dec 27 2002

AaaI (XmaIII)                     C^GGCCG
AacI (BamHI)                      GGATCC
AaeI (BamHI)                      GGATCC
AagI (ClaI)                       AT^CGAT
AaqI (ApaLI)                      GTGCAC
AarI                              CACCTGCNNNN^
AarI                              ^NNNNNNNNGCAGGTG
AasI (DrdI)                       GACNNNN^NNGTC
AatI (StuI)                       AGG^CCT
AatII                             GACGT^C
```

Mapping Restriction Enzymes (cont.)

- the first ten lines up until the line beginning with `Ritch Robers` serves as a comment an can be discarded
- each of the remaining lines consists of two or three columns
- the first item in each line is the name of the restriction enzyme
- the names in brackets are synonyms which can be ignored for our purpose
- the last column specifies the recognition site with the symbol ˆ to denote the cut point
- the recognition site is given as a sequence of bases and additional symbols `N, S, Y, W, R, K, V, B, D, H, M`
- these are IUB ambiguity characters each matching a subset of set of {`A, C, G, T`}

Mapping Restriction Enzymes (cont.)

`R` means `G` or `A`

`Y` means `C` or `T`

`M` means `A` or `C`

`K` means `G` or `T`

`S` means `G` or `C`

`W` means `A` or `T`

`B` means not `A` (`C` or `G` or `T`)

`D` means not `C` (`A` or `G` or `T`)

`H` means not `G` (`A` or `C` or `T`)

`V` means not `T` (`A` or `C` or `G`)

`N` means `A` or `C` or `G` or `T`

Mapping Restriction Enzymes (cont.)

- to transform the recognition site into a valid regular expression we use the following function:

```
def iub_to_regexp(iub)
  iub2character_class = {
        'A' => 'A',
        'C' => 'C',
        'G' => 'G',
        'T' => 'T',
        'R' => '[GA]',
        'Y' => '[CT]',
        'M' => '[AC]',
        'K' => '[GT]',
        'S' => '[GC]',
        'W' => '[AT]',
        'B' => '[CGT]',
        'D' => '[AGT]',
        'H' => '[ACT]',
        'V' => '[ACG]',
        'N' => '[ACGT]'
    }
```

Mapping Restriction Enzymes (cont.)

```
    # Remove the ^ signs from the recognition sites
    iub.gsub!(/\^/,"")

    # Temporarily transform into an array
    regexp_string = iub.split(//)

    # Replace each IUB item with its character class
    regexp_string.map! do |iubchar|
      if iub2character_class.has_key?(iubchar)
        iub2character_class[iubchar]
      else
        STDERR.puts "#{$0}: unknown IUB-character #{iubchar}"
        exit 1
      end
    end

    # Concatenate to string
    return regexp_string.join
end
```

Mapping Restriction Enzymes (cont.)

```
------------------------------------------------------------- Array#map!
     array.collect! {|item| block }   ->   array
     array.map!     {|item| block }   ->   array
------------------------------------------------------------------------
     Invokes the block once for each element of _self_, replacing the
     element with the value returned by _block_. See also
     +Enumerable#collect+.

        a = [ "a", "b", "c", "d" ]
        a.collect! {|x| x + "!" }
        a             #=>  [ "a!", "b!", "c!", "d!" ]
```

Mapping Restriction Enzymes (cont.)

– The next function parses the repbase file:

```
def parseREBASE(rebasefile)
  rebasetab = Hash.new()   # hash to be returned
  File.open(rebasefile,"r").each_line do |line|
    if not line.match(/^(\s+|REBASE|Rich Roberts)/)
      fields = line.split(" ")      # split the 2 or 3 fields
      # Remove parenthesized names by not saving the middle
      # field (if any), just the first and last
      re_name = fields.shift          # extract first element
      re_site = fields.pop            # extract last element
      regex = iub_to_regexp(re_site)  # translate recog. site
      rebasetab[re_name] = "#{re_site} #{regex}"
    end
  end
  puts "parsed #{rebasetab.length} restriction enzymes"
  return rebasetab  # Return hash with reformatted REBASE
end
```

– The next function is used to match the regular expressions and obtain their start positions

Mapping Restriction Enzymes (cont.)

```
def match_positions_fwd(regexp, sequence)
  positions = Array.new()
  # match regexp against sequence, be case insensitive
  lastpos = 0
  loop do
    p = sequence.index(/#{regexp}/i,lastpos)
    if p.nil?
      break
    end
    positions.push(p)
    lastpos = p+1
  end
  return positions
end
```

Mapping Restriction Enzymes (cont.)

```
inputfilename = ARGV[0]
dna = extract_sequence_from_fasta_data(inputfilename)
rebase_hash = parseREBASE("REBASE.txt") # Get REBASE into hash
loop do
  print "Please enter name of restriction enzyme to search: "
  query = STDIN.gets
  if not query or query.chomp.match(/(^\s*$)|quit/)
    break
  end
  query.chomp!
  if rebase_hash.has_key?(query)
    recognition_site, regexp = rebase_hash[query].split(" ")
    locations = match_positions_fwd(regexp, dna)
    if locations.empty?
      puts "\"#{query}\" does not occur in DNA"
    else
      puts "\"#{regexp}\" was found at pos #{locations.join(', ')}"
    end
  else
    puts "\"#{query}\" is not a valid name"
  end
end
```

Mapping Restrict. Enz.: a class-implementation

```
class RestrictFind
  @@iub2character_class = { # class var: exists only once
    'A' => 'A',
    'C' => 'C',
    'G' => 'G',
    'T' => 'T',
    'R' => '[GA]',
    'Y' => '[CT]',
    'M' => '[AC]',
    'K' => '[GT]',
    'S' => '[GC]',
    'W' => '[AT]',
    'B' => '[CGT]',
    'D' => '[AGT]',
    'H' => '[ACT]',
    'V' => '[ACG]',
    'N' => '[ACGT]'
  }

  def initialize(fasta_file, rebase_file = "REBASE.txt")
    @rebase_hash = parseREBASE(rebase_file)
    @dna = extract_sequence_from_fasta_data(fasta_file)
```

Mapping Restrict. Enz.: a class-implementation (cont.)

```
  end

  def get_restriction_matches(query)
    if @rebase_hash.has_key?(query)
      recognition_site, regexp = @rebase_hash[query].split(" ")
      locations = match_positions_fwd(regexp)
    else
      raise "\"#{query}\" not a valid name"
    end
    return [recognition_site, locations]
  end

private

  def match_positions_fwd(regexp)
    positions = Array.new()
    lastpos = 0
    while not (p = @dna.index(/#{regexp}/i,lastpos)).nil?
      positions.push(p)
      lastpos = p+1
    end
    return positions
  end
```

Mapping Restrict. Enz.: a class-implementation (cont.)

```
  def iub_to_regexp(iub)
    regexp_string = iub.gsub(/\^/,"").split(//)
    regexp_string.map!{|iubchar| @@iub2character_class[iubchar]}
    return regexp_string.join
  end

  def parseREBASE(rebasefile)
    rebase_hash = Hash.new()
    File.open(rebasefile,"r").each_line do |line|
      if not line.match(/^(\s+|REBASE|Rich Roberts)/)
        fields = line.split(" ")     # split the 2 or 3 fields
        re_name = fields.shift       # extract first element
        re_site = fields.pop         # extract last element
        regex = iub_to_regexp(re_site)  # translate recog. site
        rebase_hash[re_name] = "#{re_site} #{regex}"
      end
    end
    return rebase_hash
  end

end   # of class
```

Mapping Restrict. Enz.: a class-implementation (cont.)

```
rf = RestrictFind.new(ARGV[0])
loop do
  print "Please enter name of restriction enzyme to search: "
  query = STDIN.gets
  if not query or query.chomp.match(/(^\s*$)|quit/)
    break
  end
  query.chomp!
  begin
    site, locations = rf.get_restriction_matches(query)
    if locations.empty?
      puts "\"#{query}\" is not in the DNA"
    else
      puts "\"#{site}\" was found at pos #{locations.join(', ')}"
    end
  rescue => err
    STDERR.puts "#{$0}: #{err}"
    next
  end
end
```

Parsing Genbank files

– Genbank (Genetic Sequence Data Bank) is a rapidly growing international repository of known genetic sequences
– this chapter describes Ruby script to parse and extract information from a Genbank file
– Genbank entry consists an annotation part and a sequence part
– the annotation part itself can be divided into a description part defining (among other thing):
    - accession numbers,
    - definitions of the kind of sequence in the entry,
    - the source where the sequence comes from,
    - references, and
    - a feature table.
– The following two pages show an excerpt of a Genbank entry:

Parsing Genbank files (cont.)

```
LOCUS       AB031069     2487 bp    mRNA            PRI       27-MAY-2000
DEFINITION  Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
            complete cds.
ACCESSION   AB031069
VERSION     AB031069.1  GI:8100074
KEYWORDS    .
SOURCE      Homo sapiens embryo male lung fibroblast cell_line:HuS-L12 cDNA to
            mRNA.
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
REFERENCE   1  (sites)
  AUTHORS   Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,Si. and
            Takano,T.
  TITLE     PCCX1, a novel DNA-binding protein with PHD finger and CXXC domain,
            is regulated by proteolysis
  JOURNAL   Biochem. Biophys. Res. Commun. 271 (2), 305-310 (2000)
  MEDLINE   20261256
REFERENCE   2  (bases 1 to 2487)
  AUTHORS   Fujino,T., Hasegawa,M., Shibata,S., Kishimoto,T., Imai,S. and
            Takano,T.
  TITLE     Direct Submission
  JOURNAL   Submitted (15-AUG-1999) to the DDBJ/EMBL/GenBank databases.
            Tadahiro Fujino, Keio University School of Medicine, Department of
            Microbiology; Shinanomachi 35, Shinjuku-ku, Tokyo 160-8582, Japan
            (E-mail:fujino@microb.med.keio.ac.jp,
            Tel:+81-3-3353-1211(ex.62692), Fax:+81-3-5360-1508)
FEATURES             Location/Qualifiers
     source          1..2487
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
```

Parsing Genbank files (cont.)

```
                         /sex="male"
                         /cell_line="HuS-L12"
                         /cell_type="lung fibroblast"
                         /dev_stage="embryo"
     gene                229..2199
                         /gene="PCCX1"
     CDS                 229..2199
                         /gene="PCCX1"
                         /note="a nuclear protein carrying a PHD finger and a CXXC
                         domain"
                         /codon_start=1
                         /product="protein containing CXXC domain 1"
                         /protein_id="BAA96307.1"
                         /db_xref="GI:8100075"
                         /translation="MEGDGSDPEPPDAGEDSKSENGENAPIYCICRKPDINCFMIGCD
                         NCNEWFHGDCIRITEKMAKAIREWYCRECREKDPKLEIRYRHKKSRERDGNERDSSEP

                         AMTNRAGLLALMLHQTIQHDPLTTDLRSSADR"
BASE COUNT      564 a    715 c    768 g    440 t
ORIGIN
        1 agatggcggc gctgaggggt cttgggggct ctaggccggc cacctactgg tttgcagcgg
       61 agacgacgca tggggcctgc gcaataggag tacgctgcct gggaggcgtg actagaagcg

     2401 gcctcctctc cctgggtttt gttaataaaa ttttgaagaa accaaaaaaa aaaaaaaaaa
     2461 aaaaaaaaaa aaaaaaaaaa aaaaaaa
//
```

gb2fasta.rb: extract sequence part from a Genbank file

```
def gb2fasta(filename)
  sequence = ''     # initialize sequence string
  inseq = false     # true iff currently a sequence line is parsed
  File.open(filename,"r").each_line do |line|
    if line.match(/^\/\/\n/)      # line is end-of-record line //\n
      break         # break out of the nearest enclosing loop
    elsif inseq    # we are in a sequence
      sequence += line    # add current line to concatenation
    elsif line.match(/^ORIGIN/)     # line before sequence part
      inseq = true                  # set the inseq flag
    end
  end
  return sequence.gsub(/[\s0-9]/,"")
end

ARGV.each do |filename|
  sequence = gb2fasta(filename)
  puts ">"
  print_sequence(sequence, 50) # print in FASTA format, width 50
end
```

gb2fields.rb: Get some annotation fields from Genbank-file

```
indef = false
definition = locus = accession = organism = ""

File.open(ARGV[0],"r").each_line do |line|
  if line.match(/^LOCUS/)
    line.sub!(/^LOCUS\s*/,"")        # delete field name at the beginning
    locus = line
  elsif line.match(/^DEFINITION/)
    line.sub!(/^DEFINITION\s*/,"")  # delete field name at the beginning
    definition = line
    indef = true                    # you are now inside the definition part
  elsif line.match(/^ACCESSION/)
    line.sub!(/^ACCESSION\s*/,"")   # delete field name at the beginning
    accession = line
    indef = false                   # now again outside the definition part
  elsif indef                       # still inside the definition
    definition.chomp!               # def part has been found, delete \n
    line.sub!(/^\s+/," ")           # repl. initial multispaces by single space
    definition += line
  elsif line.match(/^  ORGANISM/)   # delete field name at beginning
    line.sub!(/^\s*ORGANISM\s*/,"")
    organism = line
  end
end

puts "*** LOCUS ***\n#{locus}"
puts "*** DEFINITION ***\n#{definition}"
puts "*** ACCESSION ***\n#{accession}"
puts "*** ORGANISM ***\n#{organism}"
```

gb2annoseq.rb: extract annotation and sequence using regexps

```
# strategy: read entire gb-record into string, and process it
# using regexps. Usually file data is stored line by line,
# since the default input separator is newline.

# a gb-record begins with the word LOCUS and ends with // on a
# separate line. Read in record to a scalar, using new separator

filename = ARGV[0]
record = get_file_data(filename,"//\n")[0]
annotation = dna = nil
mo = record.match(/^(LOCUS.*ORIGIN\s*\n)(.*)\/\/\n/m)
if mo
  annotation = mo[1]
  dna = mo[2]
else
  STDERR.puts "#{filename}: Cannot separate annot./seq."
  exit 1
end

print "The annotation:\n#{annotation}", "the DNA:\n#{dna}"
```

gb2annoseq.rb: extract annotation and sequence using regexps (cont.)

- The standard meaning of the symbols ^, $, and . in regular expression is as follows:
  - ^ anchors the regexp to the beginning of the string and after a newline embedded in a string
  - $ anchors the regexp to the end of the string and before a newline embedded in a string
  - . matches any character except newline
- To handle multiline scalars appropriately, there is a modifier `/m` for regular expressions.
- The modifier `/m` makes the . match any character including newline.

⇒ This allows to treat the entire string as one single *line* with embedded newlines (multiline).

gb2annoseq.rb: extract annotation and sequence using regexps (cont.)

- Example 1: `"AAC\nGTT".match(/^.*$/)`
  This match is successful, from the beginning up to the first embedded newline. That is, AAC is the matching substring.
- Example 2: `"AAC\nGTT".match(/^.*$/m)`
  This match is successful, i.e. the entire string is matched, because the symbol . matches the newline.

gb2annoseq.rb: extract annotation and sequence using regexps (cont.)

Now separate the annotation from the sequence data using the expression `^(LOCUS.*ORIGIN\s*\n)(.*)\/\/\n/m`

- The first part (enclosed in the first pair of braces) begins with `LOCUS` and ends with `ORIGIN` followed by any number of spaces followed by a newline.
- The second part (enclosed in the second pair of braces) is the remaining text, up to the `//n`.
- The third part is the `//n` line. Due to the `/m` modifier, `record` is treated as a single line and `.` matches a newline.
- The first part of the match is assigned to the variable `annotation` in:

  `annotation = match[1]`
- The second part is assigned to the variable `dna` in

  `dna = match[2]`

splitGB.rb: split genbank entry, use regexps

```
def filename2fileobject(filename)
  begin
    file = File.open(filename,"r")
  rescue => err
    STDERR.puts "Cannot open file #{filename}: #{err}"
    exit 1
  end
  return file
end

# get next record of genbank library file given fileobject
def get_next_record(file)
  return file.readline("//\n")
end

# get annotation and DNA for given GB record
def get_annotation_and_dna(record)
  mo = record.match(/^(LOCUS.*ORIGIN\s*\n)(.*)\/\/\n/m)
  if mo
    annotation = mo[1]
    dna = mo[2]
  else
    STDERR.puts "Cannot separate annotation from sequence info"
    exit 1
  end
  dna.gsub!(/[\s0-9]/,"")   # clean the sequence of any whitespace or digits
  return annotation, dna
end
```

searchGB.rb: apply some searches to GB-library

```
# search sequence for regular expression
def search_sequence(sequence, regexp)
  positions = []
  lastpos = 0
  while ((p = sequence.index(/#{regexp}/i,lastpos)) != nil)
    positions.push(p)
    lastpos = p+1
  end
  return positions
end

# search annotation for regular expression
def search_annotation(annotation, regexp)
  positions = []
  # note the /m modifier-. matches any character including newline
  lastpos = 0
  while ((p = annotation.index(/#{regexp}/im,lastpos)) != nil)
    positions.push(p)
    lastpos = p+1
  end
  return positions
end

if ARGV.length == 1
  library = ARGV[0]
else
  STDERR.puts "Usage: #{$0} <genbankfile>"
  exit 1
end
```

searchGB.rb: apply some searches to GB-library (cont.)

```
fh = filename2fileobject(library)
# for given file object store next position which has not been read yet
offset = fh.pos

begin
  while record = get_next_record(fh)   # read records one after the other
    # split record into annotation and dna
    annotation, dna = get_annotation_and_dna(record)

    if not (search_sequence(dna, 'AAA[CG].').empty?)
      # show the first position of the record in GB-library
      puts "Sequence found in record at offset #{offset}"
    end
    if not (search_annotation(annotation, 'homo sapiens').empty?)
      puts "Annotation found in record at offset #{offset}"
    end
    offset = fh.pos
  end
rescue EOFError   # close file if end has been reached
  fh.close
end
```

parseAnno.rb: parse some annotations into a hash

Parsing method:

- given a GenBank annotation, return a hash with the field names as keys and field contents as values.
- annotation fields all begin with a keyword in capital letters at the beginning of a line.
- access these top level strings, treat them as keys and store the corresponding lines as values.
- each top level keyword, matched by `\n([A-Z])`, is prefixed with a special character not appearing in the original GenBank entry
- the whole string is then split into an array at the position where this marker appears.
- the keywords are then extracted by trying to match capital letter words only at the beginning of a line.
- these words are used to index the lines in a hash.

parseAnno.rb: parse some annotations into a hash (cont.)

```
def parse_annotation(annotation)
  results = Hash.new

  # mark beginnings with special character and split there into array
  sep = "\001"
  tops = annotation.gsub(/\n([A-Z])/, "\n#{sep}\\1").split(sep)

  # process annotation fields into keyword-indexed hash
  tops.each do |value|
    # the BASE COUNT has a space in it, treat separately
    if value.match(/^BASE COUNT/)
      results['BASE COUNT'] = value
    else
      # get key from line
      mo = value.match(/^([A-Z]+)/)
      if mo
        key = mo[1]
      else
        STDERR.puts "Cannot find key in line \"#{value}\""
        exit 1
      end
      results[key] =  value   # store the value in the hash
    end
  end
  return results
end
```

getAnno.rb: get the annotation from first genbank record

```
require "splitGB.rb"
require "parseAnno.rb"

if ARGV.length == 1
  library = ARGV[0]
else
  STDERR.puts "Usage: #{$0} <genbankfile>"
  exit 1
end

fh = filename2fileobject(library) # Open library and read a record

record = get_next_record(fh) # get the first record

annotation, dna = get_annotation_and_dna(record)

fields = parse_annotation(annotation) # Extract fields of the annotation

# Print the fields
fields.each_pair do |key, value|
    puts "******** #{key} *********"
    puts value
end

exit 0
```

parseFeatures.rb: extract features from the FEATURES field of a GB-record

Parsing method:

- The feature table can contain several entries (called feature entries), each of which is matched by the following match operation:

  `^( {5}\S.*\n(^ {21}\S.*\n)*)`

- A feature entry thus begins with a keyword after 5 white spaces ...
- followed by a none-white space, ...
- followed by any number of characters until a newline appears.
- The second part of a feature entry is optional. It begins with 21 white spaces and otherwise has the same structure as before.

```
def parse_features(features)
  featuretab = Array.new() # store the individual features here

  # Extract features
  features.scan(/^( {5}\S.*\n( {21}\S.*\n)*)/) do |entry|
    featuretab.push(entry[0])   # add it to end of the current featuretable
  end
  return featuretab
end
```

features.rb: extract feature entries from GB-library

```ruby
# Get the fields from the first GenBank record in a given file
fh = filename2fileobject(filename)

record = get_next_record(fh)

# split record into annotation and sequence
annotation, dna = get_annotation_and_dna(record)

# parse the annotation
fields = parse_annotation(annotation)

# Extract the feature entries from the FEATURES table
features = parse_features(fields['FEATURES'])

# Print the features
features.each do |featureentry|
  # extract the name of the feature
  # the following match will be successful. The expression of interest
  # (enclosed in a pair of braces) will be the nonspace sequence
  # following the 5 white spaces

  mo = featureentry.match(/^ {5}(\S+)/)
  if mo
    puts "******** #{mo[1]} *********"
    puts featureentry
  end
end
```

Parsing a genbank record in XML format

- XML is generic framework for storing text or data whose structure can be represented as a tree.
- text is enclosed between `<tag>` and `</tag>` where `tag` is an identifier.
- the following shows a part of a genbank record in XML

```xml
<Seq>
  <Seq_locus>AAU04286</Seq_locus>
  <Seq_length>436</Seq_length>
  <Seq_moltype>AA</Seq_moltype>
  <Seq_topology>linear</Seq_topology>
  <Seq_division>BCT</Seq_division>
  <Seq_update-date>27-AUG-2004</Seq_update-date>
  <Seq_create-date>19-AUG-2004</Seq_create-date>
  <Seq_definition>citrate (Si)-synthase; (R)-citric synthase.; Citrate
                  condensing enzyme.;
                  Citrate oxaloacetate-lyase, CoA-acetylating.;
                  Oxaloacetate transacetase.
                  [Rickettsia typhi str. Wilmington]</Seq_definition>
  <Seq_source>Rickettsia typhi str. Wilmington</Seq_source>
  <Seq_organism>Rickettsia typhi str. Wilmington</Seq_organism>
  <Seq_sequence>aacactgtgaattaagagcctatcacaacgagccatactacg</Seq_sequence>
</Seq>
```

Parsing a genbank record in XML format (cont.)

– the following shows how to parse an XML-file using the REXML-module which implements a class `Document`

```
require 'set'
require 'rexml/document'

# show tag and correpoding text whenever tag belongs to idset
def showfirstlevel(idset,record)
  record.elements.each do |x|
    if idset.member?(x.name)
      puts "#{x.name}=#{x.text}"
    end
  end
end

# specify for which tags the output will be shown
idset = Set.new ['Seq_locus','Seq_sequence','Seq_division']

File.open("Record.xml","r") do |file|
  xml = REXML::Document.new(file)
  xml.elements.each do |record|   # iterate over all records
    showfirstlevel(idset,record)
  end
  file.close_read
end
```

Iterators

– We have seen many examples in which iterator like `each`, `each_line`, `scan` are used in connection with blocks

– now we consider how to use own iterators

– Ruby iterator: a method that can invoke a block of code (i.e. a group of statements)

– block is written starting on the same line as the method's call, e.g.

```
three_times {puts "Hello"}
```

– code in the block is not executed at the time it is encountered

– but Ruby remembers the context in which the block appears

– within the method, the block may be invoked, as if it was a method itself, using the `yield` statement

```
def three_times
  yield
  yield
  yield
end
```

Iterators (cont.)

- whenever `yield` is executed, it invokes the code in the block
- when the block exits, control picks back up immediately after the `yield`

```
def three_times
  yield
  yield
  yield
end
three_times {puts "Hello"}
```

- the block (i.e. the code between the braces) is associated with the call to the method `three_times`.
- within this method, `yield` is called three times in a row.
- each time it invokes the code in the block.
- blocks can be passed parameters and one can receive parameters from them

Iterators (cont.)

- consider an example in which fibonacci numbers are generated:

```
def fib_up_to(max)
  i1 = 1
  i2 = 1
  while i1 <= max
    yield i1
    tmp = i1 + i2
    i1 = i2
    i2 = tmp
  end
end

fib_up_to(1000) do |f|
  print "#{f} "
end
```

Iterators (cont.)

- one can even leave the maximum value undefined
- instead decide in the executed block about the largest fibonacci number to be created

```
def fib_infinite
  i1 = 1
  i2 = 1
  loop do
    yield i1
    tmp = i1 + i2
    i1 = i2
    i2 = tmp
  end
end

fib_infinite do |f|
  if f > 1000
    break
  end
  print "#{f} "
end
```

Interacting with the file system: listing directories

```
def listdirectory(directory)
  # prepare regexp for entries to ignore
  # saves time for repeated regexp use, since it stays the same
  ignore_dirs = Regexp.compile(/^\.\.?$/)

  begin
    # Iterate over items in directory
    Dir.foreach(directory) do |entry|

      # We ignore current and parent directory entries here
      unless entry.match(ignore_dirs)
        # directory entry is a regular file
        if File.stat("#{directory}/#{entry}").file?
          puts "#{directory}/#{entry}"
        # directory entry is a subdirectory
        elsif File.stat("#{directory}/#{entry}").directory?
          # Here is the recursive call to this function
          listdirectory("#{directory}/#{entry}")
        end
      end
    end
  rescue
    # We got here because of some error. Report error and exit.
    STDERR.puts "Error accessing directory #{directory}."
    exit 1
  end
end
```

Interacting with the file system: a simple find command

```
#!/usr/bin/env ruby
# Demonstrate a recursive subroutine to list a subtree of a
    filesystem

require "listdir.rb"

if ARGV.length == 1
  filename = ARGV[0]
else
  STDERR.puts "Usage: #{$0} <file or directoryname>"
  exit 1
end

if File.stat(filename).file?
  puts filename
else
  if File.stat(filename).directory?
    listdirectory(filename)
  end
end

exit 0
```

Generating texts: here documents

- sometimes it is necessary to print several lines of text in some program
- usual way: use several print or printf statement with
  n at the end of each line
- here script simplify this

```
puts <<HEREDOC
Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs", Nucleic Acids Res. 25:3389-3402.
RID: 991533563-27495-9092
HEREDOC
```

- terminating string (in our case HEREDOC) can be any string

Generating texts: here documents (cont.)

– Here document allow to use interpolated variables to e.g. generate bulk letters

```
["GEZ\nKoeln", "Versicherung\nHamburg", "BFA\nBerlin"].each do |address|
print "------------------------------------------------------------\n";
print <<HEREDOC
                                                    Hamburg, den 1.11.2012

#{address}

Joe User
Universitaetsstrasse 25
33613 Bielefeld

Sehr geehrte Damen und Herren,

meine neue Adresse lautet: Bundesstrasse 43, 20146 Hamburg

MfG, Joe User
HEREDOC
end
```

– technique is also very suitable to generate code fragments with small changes

Showprint.rb: formatted printing

```
floatval = 2323.14159265
intval = 76
stringval = "hello world"

# use formatted print to show these lines:

printf "A float\t\t  \"%6.4f\"\n", floatval
printf "An integer\t  \"%-5d\"\n", intval
printf "A string\t  \"%s\"\n", stringval
```

– arguments of printf: format string contains text along with format directives
– each directive consists of % followed by conversion specifier
  - `f` means floating point value
  - `d` means integer value
  - `s` means string

Showprint.rb: formatted printing (cont.)

- between % and these characters there are optional flags to specify
  - minimum field width
  - precision (for floats)
  - length modifier
  - alignment modifier
- For example: `%6.4f` means that floating point is printed with minimum width 6 characters (padded with spaces if necessary) and at most 4 positions for the decimal part
- `%-5d` means that integer is printed in a field of 5 characters; `-` means left adjustment

Detecting pairs of similar sequences

A standard task in sequence analysis is the following:

- given the product of all genes (i.e. proteins) in two genomes G_1 and G_2
- determine all pairs of similar proteins from the two genomes: i.e. if $S(G_i)$ is the set of proteins in G_i, $i \in \{1, 2\}$, find all pairs

$$(p_1, p_2) \in S(G_1) \times S(G_2) \text{ satisfying } p_1 \approx p_2$$

  where $\approx$ denotes a similarity relation
- $\approx$ is usually defined in terms of scores of alignments, such that the higher the score, the more similar the sequences
- similarity usually means local similarity involving substrings of the considered sequences
- for example, the following alignments shows a short local similarity between two longer sequences

Detecting pairs of similar sequences (cont.)

```
NLGPSTKDDFLGK-ILGPSTKDDQKDDFLG
         |  |  ||  |
QNQLERSDNF-GKSINQLERSSNNKESQN
```

– the local alignments often allow to transfer information from one well known genome (e.g. G_1) to an unknown genome (e.g. G_2)

– e.g. if the function of the gene product p_1 in G_1 is known, then p_2 in G_2 may have a similar function, provided that the similarity covers a considerable stretch of the sequences

– one usually applies rules such as:

> *for each of both sequences the similarity must cover* 80% *of the sequence*

– we will know show how to compute such local similarities (using Blastp) and apply the selection rule (using a ruby script)

– Let us call G_1 the subject sequence and G_2 the query sequence

Detecting pairs of similar sequences (cont.)

– technical assumption: the sequence sets are available as files in Fasta-format, e.g. the subject sequence:

```
>gendb|HPB128_199g1| Helicobacter pylori B128 hypothetical protein
MKKIILACLMAFVGANLSAEPKWYSKAYNKTNTQKGYLYGSGSATSKEASKQKALADLVASISVVVNSQI
HIQKSRVDNKLKSSDSQTINLKTDDLELNNVEIVNQEAQKGIYYTRVRINQNLFLQGLRDKYNALYGQFS
LQSLLYKELKDYANKEGQGNTGL
>gendb|HPB128_199g2| Helicobacter pylori B128 hypothetical protein
MKRLAIALALVLGVAWGKSLPKWARDCSKEVQVEKTQTKDEKFLVCGMSDILLSDMDYSLSSARQNALEK
...
```

and the query sequence:

```
>HPB8_1
MDTNNNIEKEILALVKQNPKVSLIEYENYFSQLKYNPNASKSDIAFFYAPNQVLCTTITAKYGALLKEIL
SQNKVGMHLAHSVDVRIEVAPKIQISAQSNINYKAIKTSVKDSYTFENFVVGSCNNTVYEIAKKVAQSDT
KKMLEEEKSPFISSLREEIKNRLNELNDKKTAFNSSE
>HPB8_2
MKKTLCLSFFLTFSNPLQALVIELLEEIKTSPHKGTFKAKVLDSKEPRQVLGVYNISPHKKLTLTITHIS
...
```

– We will use NCBI-Blastp (available from `ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/` to compute local similarities

Detecting pairs of similar sequences (cont.)

- first step is to make a blast database from the subject file: `makeblastdb -in` *subjectfile* `-dbtype prot -out` *dbname*
- this creates three files *dbname*`.phr`, *dbname*`.psq`, *dbname*`.pin`, containing the headers, the sequence and some additional information.
- the second step runs the program `blastp` for determining the similarities:
  `blastp -query` *queryfile* `-outfmt 7 -db` *dbname*`-out hits.tab`
- the output of the program goes to the file `hits.tab`
- the program provides several different output formats: format 7 is tabular output with comment lines: e.g.

```
# Query: HPB8_1
# Database: B128
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens
    , q. start, q. end, s. start, s. end, evalue, bit score
# 18 hits found
HPB8_1  gendb|HPB128_199g88|  100.00  457  0  0  1  457  1  457  0.0   937
HPB8_1  gendb|HPB128_11g21| 28.47  144  63   8  39   177  68   176  0.063  30.8
...
```

Detecting pairs of similar sequences (cont.)

- The `Fields`-line explains the meaning to the values which are separated by tabulators
- As one can see, the length of the sequences involved in the match is not available in the output
- as it is required for computing the coverage of a hit (e.g. the percentage of bases covered by a hit), we must compute it first:

```
def mklengthtab(filename)
  lengthtab = Hash.new()
  FastaIterator.new(filename).each do |header,sequence|
    blastid = header.split(/\s/)[0]
    if lengthtab.has_key?(blastid)
      STDERR.puts "#{$0}: duplicated header \"#{blastid}\""
      exit 1
    end
    lengthtab[blastid] = sequence.length
  end
  return lengthtab
end
```

Detecting pairs of similar sequences (cont.)

- method `mklengthtab` makes use of the `FastaIterator`-class to be developed in the exercises
- each iteration delivers a pair of header and sequence, processed in a block
- Blastp uses the prefix of the header up to (but excluding) the first whitespace as identifier:

⇒ replace the initial >, split the header on whitespaces and take the first element of the resulting array

- for each sequence the `blastid` serves as key for a hash table `lengthtab` which stores the length of the sequence
- lets now consider how to use this function in the main ruby-script

Detecting pairs of similar sequences (cont.)

```ruby
if ARGV.length != 4
  STDERR.puts "Usage: #{$0} <queryfile> <subjectfile> " +
              " <blastoutfile> <mincoverage>"
  exit 1
end

queryfile=ARGV[0]
subjectfile=ARGV[1]
blastoutfile=ARGV[2]
mincov=ARGV[3].to_i

if mincov < 1 or mincov > 100
  STDERR.puts "#{$0}: coverage must be in the range 1 to 100"
  exit 1
end

querylengthtab = mklengthtab(queryfile)
STDERR.puts "# #{querylengthtab.length} sequences " +
            "in #{queryfile}"
subjectlengthtab = mklengthtab(subjectfile)
STDERR.puts "# #{subjectlengthtab.length} sequences " +
            "in #{subjectfile}"
```

Detecting pairs of similar sequences (cont.)

- to parse the table of blast hits, we develop a class `Blasttable`

```
class Blasttable
  @querylengthtab = nil
  @subjectlengthtab = nil
  @hitfile = nil
  def initialize(querylengthtab,subjectlengthtab,blastfile)
    @querylengthtab = querylengthtab
    @subjectlengthtab = subjectlengthtab
    begin
      @hitfile  = File.open(blastfile,"r")
    rescue => err
      STDERR.print "Could not open file \"#{blastfile}\": #{
          err}\n"
      exit 1
    end
  end
  def delete()
    @hitfile.close
  end
```

- the coverage is computed by the following method:

Detecting pairs of similar sequences (cont.)

```
  def coverage(hitstart,hitend,lengthtab,seqid)
    seqlength = nil
    if lengthtab.has_key?(seqid)
      seqlength = lengthtab[seqid]
    else
      STDERR.puts "#{$0}: no length for #{seqid}"
      exit 1
    end
    if hitend < hitstart
      STDERR.puts "#{$0}: hitend = #{hitend} < " +
                  "#{hitstart} = hitstart"
      exit 1
    end
    return 100.0 * (hitend - hitstart + 1).to_f/
           seqlength.to_f
  end
```

- the class `Blasttable` has an iterator `each` which ignores comment lines (those beginning with #), splits the remaining lines on tabulators and processes the resulting array to extract the values making up a blasthit:

Detecting pairs of similar sequences (cont.)

```
def each()
  @hitfile.each_line do |line|
  if line.match(/^[^\#]/)
    bla = line.split(/\t/)
    query, queryhitstart = bla[0], bla[6].to_i
    queryhitend = bla[7].to_i
    querycoverage = coverage(queryhitstart,
                                          queryhitend,
                                          @querylengthtab,
                                          query)
    subject, subjecthitstart = bla[1], bla[8].to_i
    subjecthitend = bla[9].to_i
    forwardstrand = true
    if subjecthitstart > subjecthitend
      tmp = subjecthitstart
      subjecthitstart = subjecthitend
      subjecthitend = tmp
      forwardstrand = false
    end
    subjectcoverage = coverage(subjecthitstart,
                                            subjecthitend,
                                            @subjectlengthtab,
                                            subject)
```

Detecting pairs of similar sequences (cont.)

– in each iteration `each` yields a structure `Blasthit` declared as follows:

```
Blasthit = Struct.new("Blasthit",:query,
                                 :subject,
                                 :identity,
                                 :alignlength,
                                 :mismatches,
                                 :gapopenings,
                                 :queryhitstart,
                                 :queryhitend,
                                 :subjecthitstart,
                                 :subjecthitend,
                                 :evalue,
                                 :bitscore,
                                 :forwardstrand,
                                 :querycoverage,
                                 :subjectcoverage)
```

– here is the corresponding `yield`-statement:

Detecting pairs of similar sequences (cont.)

```
      yield Blasthit.new(query,
                         subject,
                         bla[2].to_f,  # identity
                         bla[3].to_i,  # alignlength
                         bla[4].to_i,  # mismatches
                         bla[5].to_i,  # gapopenings
                         queryhitstart,
                         queryhitend,
                         subjecthitstart,
                         subjecthitend,
                         bla[10].to_f, # evalue
                         bla[11].to_i, # bitscore
                         forwardstrand,
                         querycoverage,
                         subjectcoverage)
```

– now lets use the class `Blasttable`:

Detecting pairs of similar sequences (cont.)

```
processed = 0
selected = 0
blhits = Blasttable.new(querylengthtab,subjectlengthtab,
                        blastoutfile)
blhits.each do |blasthit|
  if blasthit.querycoverage >= mincov and
     blasthit.subjectcoverage >= mincov
    showblasthit(blasthit)
    selected += 1
  end
  processed += 1
end
STDERR.puts "# #{processed} hits processed"
STDERR.puts "# #{selected} hits selected"
blhits.delete()
```

– in the next section we will consider how to process Blast-output involving alignments

Parsing Blast Alignment Output

- A Blast output consists of several sections:
- A header section:

```
BLASTN 2.1.3 [Apr-11-2001]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.
RID: 991533563-27495-9092
Query=
         (400 letters)

Database: nt
           868,831 sequences; 3,298,558,333 total letters
```

- An overview of the high scoring pairs:

```
                                                                  Score    E
Sequences producing significant alignments:                      (bits)  Value

dbj|AB031069.1|AB031069 Homo sapiens PCCX1 mRNA for protein cont...   793   0.0
ref|NM_014593.1| Homo sapiens CpG binding protein (CGBP), mRNA        779   0.0
gb|AF149758.1|AF149758 Homo sapiens CpG binding protein (CGBP) m...   779   0.0
ref|XM_008699.3| Homo sapiens CpG binding protein (CGBP), mRNA        765   0.0
```

Parsing Blast Alignment Output (cont.)

- A list of alignments for the high scoring pairs:

```
ALIGNMENTS
>dbj|AB031069.1|AB031069 Homo sapiens PCCX1 mRNA for protein containing CXXC domain 1,
           complete cds
          Length = 2487

 Score =  793 bits (400), Expect = 0.0
 Identities = 400/400 (100%)
 Strand = Plus / Plus


Query 1   agatggcggcgctgaggggtcttggggggctctaggccggccacctactggtttgcagcgg 60
          ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct 1   agatggcggcgctgaggggtcttggggggctctaggccggccacctactggtttgcagcgg 60


Query 61  agacgacgcatggggcctgcgcaataggagtacgctgcctgggaggcgtgactagaagcg 120
          ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct 61  agacgacgcatggggcctgcgcaataggagtacgctgcctgggaggcgtgactagaagcg 120


Query 121 gaagtagttgtgggcgcctttgcaaccgcctgggacgccgccgagtggtctgtgcaggtt 180
          ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct 121 gaagtagttgtgggcgcctttgcaaccgcctgggacgccgccgagtggtctgtgcaggtt 180


Query 181 cgcgggtcgctggcgggggtcgtgagggagtgcgccgggagcggagatatggagggagat 240
          ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct 181 cgcgggtcgctggcgggggtcgtgagggagtgcgccgggagcggagatatggagggagat 240
```

Parsing Blast Alignment Output (cont.)

```
Query 241 ggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggggagaat 300
          ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct 241 ggttcagacccagagcctccagatgccggggaggacagcaagtccgagaatggggagaat 300


Query 301 gcgcccatctactgcatctgccgcaaaccggacatcaactgcttcatgatcgggtgtgac 360
          ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct 301 gcgcccatctactgcatctgccgcaaaccggacatcaactgcttcatgatcgggtgtgac 360


Query 361 aactgcaatgagtggttccatggggactgcatccggatca 400
          ||||||||||||||||||||||||||||||||||||||||
Sbjct 361 aactgcaatgagtggttccatggggactgcatccggatca 400
```

– Note that the keyword `ALIGNMENTS` appears only once

Parsing Blast Alignment Output (cont.)

– Finally a footer section:

```
  Database: nt
    Posted date:  May 30, 2001  3:54 AM
  Number of letters in database: -996,408,959
  Number of sequences in database:  868,831

Lambda     K      H
    1.37    0.711     1.31

Gapped
Lambda     K      H
    1.37    0.711     1.31

Matrix: blastn matrix:1 -3
Gap Penalties: Existence: 5, Extension: 2
Number of Hits to DB: 436021
Number of Sequences: 868831
Number of extensions: 436021
Number of successful extensions: 7536
Number of sequences better than 10.0: 19
length of query: 400
length of database: 3,298,558,333
effective HSP length: 20
effective length of query: 380
effective length of database: 3,281,181,713
effective search space: 1246849050940
effective search space used: 1246849050940
```

splitBlast.rb: Split the blast output

```
# parse beginning and ending annotation, and alignments,
# from BLAST output file

def splitblastoutput(filename)
  # Get the BLAST program output into an array from a file

  blast_output_file = get_file_data(filename).join

  # Extract the beginning annotation, alignments, and ending annotation
  # beginning_annotation is everything up to line starting with ALIGNMENTS
  # alignment_section contains all alignments and goes until keyword Database
  # appears at line with two blanks indented. use modifier m: . matches \n

  matchData = blast_output_file.match(/(.*^ALIGNMENTS\n)(.*)(^  Database:.*)/m)

  if not matchData
    STDERR.puts "#{$0}: Illegal input in blast file #{filename}"
    exit 1
  end
  # Assign values for annotation and alignment from match results
  beginning_annotation, alignment_section, ending_annotation = matchData[1..3]
  alignments = parse_blast_alignment(alignment_section)
  return beginning_annotation, ending_annotation, alignments
end

# parse the alignments from a BLAST output file, return hash with
# key = ID and value = text of alignment
```

splitBlast.rb: Split the blast output (cont.)

```
def parse_blast_alignment (alignment_section)
  alignmenttable = Hash.new()

  # loop through the scalar containing the BLAST alignments,
  # extracting the ID and the alignment and storing in a hash

  # The regular expression matches a line beginning with > and containing
  # the ID between the first pair of | characters; followed by any number of
  # lines that don't begin with >. Here (?!>) is a negative lookahead assertion
  # meaning that the line is not allowed to begin with >

  alignment_section.scan(/(^>.*\n(^(?!>).*\n)+)/) do |entry|
    val = entry[0]
    keytab = val.split(/\|/)      # split it at pattern |
    key = keytab[1]               # extract second element, which is the
                                  # accession number of the sequence
    if alignmenttable.has_key?(key)
      STDERR.puts "#{$0}: there is already an alignment for key \"#{key}\""
      exit 1
    end
    alignmenttable[key] = val     # store the alignment in hash
  end
  return alignmenttable
end
```

blast1.rb: The module using splitBlast.rb

```ruby
require "splitBlast.rb"

if ARGV.length != 1
  STDERR.puts "Usage: #{$0} <blastoutputfile>"
  exit 1
end

blastoutputfile = ARGV[0]

beginning_annotation, ending_annotation, alignments = splitblastoutput(blastoutputfile)

puts "XXXXXXXXXXXXXXXXXXXX begin first part XXXXXXXXXXXXXXXX"
puts beginning_annotation
puts "XXXXXXXXXXXXXXXXXXXX end first part XXXXXXXXXXXXXXXX"

alignments.each_pair do |key, val|
  puts "XXXXXXXXX Alignment of query against #{key} XXXXXXX"
  puts val
  puts "XXXXXXXXXXXX end of Alignment XXXXXXXXXXX"
end

puts "XXXXXXXXXXXXXXXXXX begin last part XXXXXXXXXXXXXXXXXX"
puts ending_annotation
puts "XXXXXXXXXXXXXXXXXX end last part XXXXXXXXXXXXXXXXXX"
```

splitAlign.rb: Split the Blast alignment

```ruby
# Parse alignments from BLAST output file
# First parse beginning annotation, and HSPs, from BLAST alignment
# Return an array with first element set to the beginning annotation,
# and each successive element set to an HSP

def parse_blast_alignment_HSP (alignment)
  hsps = Array.new()

  # Extract the beginning annotation and HSPs
  matchdata = alignment.match(/(.*?)(^ Score =.*)/m)

  if not matchdata
    STDERR.puts "#{$0}: Cannot parse alignment"
    exit 1
  end
  beginning_annotation, hsp_section = matchdata[1..2]
  # .* is used with non-greedy or minimal matching operator "?"  which means
  # to match everything before the first appearance of the keyword Score

  # Store the value of beginning_annotation as first entry in array hsps
  hsps.push(beginning_annotation)

  # Parse the HSPs, store each HSP as an element in hsps. Each HSP
  # begins with Score = and extends until before the next appearance of
  # Score =. This is expressed via ?! stating a negative lookahead
  # assertion: match the following lines that do _not_ begin with Score =

  hsp_section.scan(/(^ Score =.*\n(^(?! Score =).*\n)+)/) do |entry|
    hsps.push(entry[0])
  end
```

splitAlign.rb: Split the Blast alignment (cont.)

```
  # Return array with first element = the beginning annotation,
  # and each successive element = an HSP
  return hsps
end

# parse HSP from BLAST output alignment section, return values:
# Expect value; Query string; Query range; Subject string; Subject range

def extract_HSP_information(hsp)

  # expect gets value to the right of Expect =
  matchdata = hsp.match(/Expect = (\S+)/)
  if matchdata == nil
    STDERR.puts "#{$0}: Cannot find occurrence of \"Expect = \""
    exit 1
  end
  expect = matchdata[1]

  # extract all lines beginning with Query: and concatenate them

  query = hsp.scan(/^Query(.*)\n/).join

  # extract all lines beginning with Subject: and concatenate them
  # this is done in the same way as extracting the query lines

  subject = hsp.scan(/^Sbjct(.*)\n/).join

  # select first and last number from all query lines
  # this is achieved by a pattern consisting of four parts:
  # part 1: match any number of digits
```

splitAlign.rb: Split the Blast alignment (cont.)

```
  # part 2: match any sequence of character including newline
  #         due to the use of the modifier m
  # part 3: match any character which is not digit
  # part 4: match any number of digits
  # as a result we get a list of two numbers which is concatenated with ..

  query_range = query.scan(/(\d+).*\D(\d+)/m).join('..')

  # select first and last number from all subject lines
  # in analogy to query_range

  subject_range = subject.scan(/(\d+).*\D(\d+)/m).join('..')

  # select everything which is a base. this is achieved by
  # substituting all characters except for bases

  query.gsub!(/[^acgt]/i,"")
  subject.gsub!(/[^acgt]/i,"")

  return expect, query, query_range, subject, subject_range
end
```

blast2.rb: the complete Blast parser

```
if ARGV.length != 1
  STDERR.puts "Usage: #{$0} <blastoutputfile>"
  exit 1
end

blastoutputfile = ARGV[0]

beginning_annotation, ending_annotation, alignments = splitblastoutput(blastoutputfile)

# process the alignments one after the other

alignments.each_pair do |key, alignment|
  hsps = parse_blast_alignment_HSP(alignment)

  hsps.shift
  hsps.each do |hsp|
    expect, query, query_range, subject, subject_range = extract_HSP_information(hsp)

    puts "###################"
    print ">align Query(#{query_range}) with "
    print "Subject=#{key}(#{subject_range}), expected = #{expect}\n"
    print_sequence(query,70)
    puts ">"
    print_sequence(subject,70)
  end
end
```