

Introduction to Programming and Perl

Alan M. Durham

Computer Science Department

University of São Paulo, Brazil

alan@ime.usp.br

Why do I want to learn perl?

- **Good question ;^)**
- **Perl is a powerfull language**
- **little can do lots**
 - **convert file formats**
 - **search a file for something you need**
 - **change things in a file**
 - **run a program and select just some lines of the output**
 - **process your sequences**
 - **build a pipeline that runs on many different systems**
- **you can learn a little now, a lot later if you want**

general structure of a computer program:

- **initialization : preparing the task ,allocating resources**
- **input: getting the actual data, only boring programs do not have input**
- **main task**
- **output: we want to know what happened**
- **cleaning up: sometimes we have to generate a lot of axiliary data and lock resources**

The programming cycle

- you pick the problem
- ******find a solution ******
- write the program: use a text editor to actually type the text of the program
- “run” the program
- correct the program....

Basic elements of a computer program: expressions, variables, functions.

- Expressions indicate calculations to make:
 - $5 * 7$
 - $8+2$
 - $8+9*3$
 - $(8+9)*3$

Let us try it

- i will use a perl script that runs perl on my input

perl demo.pl

- type the expressions

– **5 * 7**

– **8+2**

– **8+9*3**

– **(8+9)*3**

- and see the results

Let us try it

- create a directory “perl” on your computer
- copy inside the new directory the file
`cp /home/alan/classes/demo.pl perl/demo.pl`
- this is a perl script that runs perl on your input
- run the program
perl demo.pl
- now type the expressions and look at the result...
 - **5 * 7**
 - **8+2**
 - **8+9*3**
 - **(8+9)*3**

What went wrong?

- computers are VERY dumb
- they only do what you tell them to do
- you never told perl to show you the results....
- to tell perl to print a result, we write *print*
- again
 - **print (5 * 7)**
 - **print (8+2)**
 - **print (8+9*3)**
 - **print ((8+9)*3)**

Basic elements of a computer program: Variables and assignments

- Variables are “places” to put value
- variables are indicated by a name beginning with ‘\$’
- variables cannot have blanks in their names, but can have some special characters (“_”)
- variable names are case sensitive
- ex: \$name, \$a_place, \$anotherPlace, \$anotherplace
- assigning a value to a variable: “=”

\$one_hundred = 100

\$my_own_sequence = “ttattagcc”

Using expressions and variables: a simple program

```
$sequences_analyzed = 200
```

```
$new_sequences = 22
```

```
$percent_new_sequences = $new_sequences / $sequences_analyzed * 100
```

```
print $percent_new_sequences
```

Commands

- **commands are individual orders to the computer**
- **assignments are an example of a command**
- **after each command we need to put a semicolon (“;”),**
- **semicolons are important for the computer to know when a command ends**
- **commands can use more than one line:**

```
$percent_new_sequences = $new_sequences /  
$sequences_analyzed  
* 100;
```

The program, again:

```
$sequences_analyzed = 200 ;  
$new_sequences = 22 ; #now we will do the work  
$percentage_new_sequences = $new_sequences /  
                             $sequences_analyzed *100 ;  
print $percentage_new_sequences;
```

Comments

- we can put comments in programs, that is, text that is ignored by perl
- any text in a program line after # is ignored by perl

The program, again:

```
#this program computes the percentage of success in  
#obtaining new sequences  
$sequences_analyzed = 200 ; #this number can be big  
$new_sequences = 22 ; #this number is generally small  
$percentage_new_sequences = $new_sequences /  
                             $sequences_analyzed *100 ;  
print $percentage_new_sequences;
```

Exercise

- use emacs to create the program in file first.pl:

```
#this program computes the percentage of success in  
#obtaining new sequences
```

```
$sequences_analyzed = 200 ; #this number can be big
```

```
$new_sequences = 22 ; #this number is generally small
```

```
$percentage_new_sequences = $new_sequences /
```

```
    $sequences_analyzed *100 ;
```

```
print $percentage_new_sequences;
```

- use emacs to create the program in file first.pl:save the file (“save buffer” option in the “file” menu of emacs)
- run the program
 - Go to the terminal
 - type “*perl first.pl*”

Entering and printing data (input and output):

- reading data: *< STDIN >*, *< >*

```
$input_line = <STDIN>;
```

```
$another_input = <>;
```

- outputting data: *print*.

```
Print $input_line;
```

Example:

```
#!/usr/bin/perl
```

```
print "type the total number of sequences:";
```

```
$sequences_analyzed = <>;
```

```
print "type the number of new sequences:";
```

```
$new_sequences = <>;
```

```
$percentage_new_sequences = $new_sequences /  
                             $sequences_analyzed *100;
```

```
print "the result is:";
```

```
print $percentage_new_sequences;
```

```
print "percent\n";
```

obs: to change the printing line one have to use "\n"

How do we “run” a perl program

- tell perl to do it:

perl name_of_the_file

- Set the file to be “executable” and inside file indicate perl is used:

- program text (using a text editor)

```
#!/usr/bin/perl
```

```
$sequences_analyzed = 200;
```

```
$new_sequences = 22;
```

```
$percentage_new_sequences = $new_sequences / $sequences_analyzed * 100
```

```
print “the result is ” $percentage_new_sequences;
```

- **unix:**

```
chmod u+x name_of_the_file
```

```
./name_of_the_file
```

Input and output redirection

- in unix programs generally read from the keyboard and write on the screen
- we say that the keyboard is the *standard input* and the screen is the *standard output*
- we can “trick” programs in unix, substituting the standard input for a file (the same can be done with the standard output)
- if we create a file “my_file” with input, I can avoid typing it again using the command:
`my_program.pl < my_file`
- the same happens for output
`my_program.pl > out_file`
- try `ls > out`, look now at the contents of out...

Exercises

- 1) write a program in file `ex1.pl` that reads three numbers and output their average.**
- 2) run the program using `perl`**
- 3) run the program using the “sh-bang”**
- 4) run program for other data**
- 5) write a file `ex1.in` with the input to the program and use input redirection to run the program again**
- 6) use output redirection to send the output of your program to the file `ex1.out`**

Emacs makes your life easier

- **Emacs is a modal editor**
- **That means it can help you program in perl (or any other language)**
- **Generally in Unix emacs automatically enters “perl mode”**
 - **See if the bar above the minibuffer indicates it**
 - **If not type:**
 - M-x perl-mode**
- **Emacs can also color your program to help you**
 - M-x font-lock-mode**
- **Colors will indicate variables, commands, and strings**
- **Emacs automatically tabs your program**

conditionals and conditional expressions

- **programs that treat all data the same are boring and not so useful**
- **in order to perform alternative tasks we have conditional statements**
- **we do this in everyday life:**
 - **“if you need money, go to the bank”**
 - **if you passed the course, go on vacations, otherwise stay home and study more**

Conditionals in Perl

- in Perl, we can determine conditional execution of commands using the command *if*:

```
if ( condition ) {  
    commands  
}
```

OR....

```
if (condition) {  
    commands  
}  
else {  
    commands  
}
```

Conditionals: example:

```
#!/usr/bin/perl
$grade = <>;
if ($grade < 7.00) {
    print "failed!\n";
}
else {
    print "passed!\n";
};
```

- conditionals can have or not the “else” part”

```
$moneyInBank = <>;
if ($moneyInBank <= 0) {
    print "stop spending!!!\n";
};
```

Dealing with text (strings): comparing

- *ne* (not equal), *ge* (greater or equal), *gt* (*greater than*), *le* (less or equal), *lt* (less than), *eq* (*equal*)
- *alphanumerical comparison*
- ex:

```
if ("dna" lt "rna") {  
    print "dna is better\n";  
};
```
- Try it!

Dealing with strings: concatenating

- “.” operator
- Example:

```
#!/usr/bin/perl
```

```
$sequence = <>;
```

```
$complete_seq = $sequence . “aaaaaaaaa”;
```

```
print “new sequence is $complete_seq \n”;
```

Some String Functions - I

- getting rid of the last character: *chomp()*
 - perl reads in everything we type, including the “return”
 - to get rid of unwanted returns read, *chomp* the last character

```
$name = <STDIN>;
```

```
chomp($name); #we ALWAYS should do this when reading
```

Some string functions - II

- **getting a substring**

```
$sequence = "Durham is good for nothing";
```

```
$new_sequence = substr($sequence, 3);
```

```
print $new_sequence;    #"ham is good for nothing"
```

```
$new_sequence = substr($sequence, 0, 14);
```

```
print $new_sequence;    #"Durham is good"
```

- **separating a string in many: *split()***

- we will see later, with *arrays*

- **joining many strings in one (different from simple concatenation)**

- later, with *arrays*.

Searching something in a string

- we can try to find or change patterns of characters in a string
- this is performed by the operation: `=~`
- to find a pattern: `string =~ m/PATTERN/`

```
$someText = <STDIN>;  
if ($someText =~ m/MONEY/){  
    print "IT HAS MONEY!!!\n";  
}; #checks if what I read contains "MONEY"
```

```
$sequence = <STDIN>;  
$sequence =~ m/TATA/ ; #look if $sequence has sequence "TATA"
```

Replacing something in a string

- to replace a pattern:

string =~ s/OLD_PATTERN/NEW_PATTERN/

- example:

\$sequence =~ s/DNA/RNA/;

- to replace ALL occurrences (General replacement), add a “g” at the end:

\$sequence =~ s/DNA/RNA/g;

Example

- write a perl program that reads a small nucleotide sequence, a fasta sequence and masks all the occurrences of that first sequence in the second one.

Solution

```
#!/usr/bin/perl
print "type the sequence to search:";
$masked_sequence = <>;
chomp($masked_sequence);
print "give me the fasta:\n";
$fasta_comment = <>;
chomp($fasta_comment);
$main_sequence = <STDIN>;
chomp($main_sequence );
$main_sequence =~ s/$masked_sequence/XXXX/g;
print "new sequence:\n";
print "$fasta_comment \n";
print "$main_sequence \n";
```

The reading loop:

- we generally need to do things a repeated number of times.
- repetitions in programs are called “loops”
- simplest type of loop is when I want to read many lines and do something with each one

```
$fastaComment = <>;  
chomp($fasta_comment);  
while ($line = <STDIN>){  
    chomp($line);  
    $my_sequence = $my_sequence.$line;
```

The example, revisited

```
#!/usr/bin/perl
print "type the sequence to search:";
$masked_sequence = <>;
chomp($masked_sequence);
print "give me the fasta:\n";
$fasta_comment = <>;
chomp($fasta_comment);

while ($line = <STDIN>){
    chomp($line);
    $main_sequence = $main_sequence . $line;
};
$main_sequence =~ s/$masked_sequence/XXXX/g;
print "new sequence:\n";
print "$fasta_comment \n";
print "$main_sequence \n";
```

Exercise - 1

- 1) write perl program that
 - reads a FASTA sequence
 - checks if it has the subsequence “tataccc”,
 - And warns the user if this happens
- 2)create a directory lotsasequences
 - **mkdir lotsasequences**
- 3)copy (using cp):
 - **cd lotsasequences**
 - **scp workshop@192.168.2.27:lotsasequences/* .**
 - **password:whotdr05**
- 4)now use the program you wrote tell which of the files in the directory **lotsasequences/** have the mentioned subsequence

Solution 1

```
$fasta_header = <>;
$sequence = “”;
while ($line = <STDIN>){
    chomp($line);
    $sequence = $sequence . $line;

}
if ($sequence =~ m/tataccc/){
    print “oh,oh, sequence with tataccc.\n”;
};
```

Solution

```
$fasta_header = <>;  
$sequence = “”;  
while ($line =<STDIN>){  
    chomp($line);  
    #$sequence = $sequence . $line;  
    $sequence .= $line;  
}  
if ($sequence =~ m/tataccc/){  
    print “oh,oh, sequence with tataccc.\n”;  
};
```

Solution: case insensitive search

```
$fasta_header = <>;  
$sequence = "";  
while ($line = <STDIN>){  
    chomp($line);  
    # $sequence = $sequence . $line;  
    $sequence .= $line;  
}  
if ($sequence =~ m/tataccc/i){  
    print "oh,oh, sequence with tataccc.\n";  
};
```

Exercise – 2

- **1) write a perl program that reads some text from the standard input, substitute all occurrences of “Alan” by “Dr. Durham”, and print the result in the standard output**
- **2) copy the file**
 - **scp workshop@192.168.2.27:exampleText.txt .**
 - **password:whotdr05**
- **3) run this program using as input the above file and check the output**

Solution 2

```
$sequence = “”;  
while ($line = <STDIN>){  
    $sequence .= $line;  
}  
$sequence =~ s/Alan/Dr. Durham/g;  
print “final text:\n $sequence”;
```

Dealing with files

- we have seen how to use unix shell operator to make perl treat files instead of keyboard input
- however perl can read directly from files
- to do this we need two things
 - to associate a *file handle* to the file
 - to *open* the file

```
open(FILEHANDLE,string_with_name_of_file)  
    or die “message”;
```

- after we finish, we should release the handle
- ```
close(FILEHANDLE)
```

# We can now rewrite the exercise

```
open(SEQFILE, “/home/<your user name>/lostasequences/seq1.txt”)
 or die “could not find the file \n”;
$fasta_header = <SEQFILE>;
while ($line = <SEQFILE>){
 $sequence .= $line;
}
if ($sequence =~ m/TATACCC/i) {
 print “it has TATACCC!!!!\n”;
}
close(SEQFILE);
```

# We can also input the file name

```
print “type name of file to be processed:”;
$file_name = <STDIN>;
chomp($file_name);
open(SEQFILE, $file_name) or die “could not find the file $file_name \n”;
$fasta_header = <SEQFILE>;
while ($line = <TEXTFILE>){
 $sequence .= $line;
}
if ($sequence =~ m/TATACCC/i){
 print “it has a TATACCC!!!!\n”;
}
close(SEQFILE);
```

# What if we want to work with many files?

- We need to read each file name.
- With each file name we:
  - Open the file
  - Process the data
  - Close the file
- Therefore we need something like

```
while ($file_name = <STDIN>){
 <open file>
 <do stuff>
 <close file>
}
```

# Even fancier: exercise 2

```
$fasta_header = <SEQFILE>;
$sequence = “”;
while ($line =<SEQFILE>){
 chomp($line);
 $sequence .= $line;
}
if ($sequence =~ m/TATACCC/i){
 print “$fasta_header”;
 print “oh,oh, sequence with TATACCC.\n”;
};
```

# Even fancier: exercise 2

```
open (SEQFILE, $file_name)
 or die "could not find the file $file_name \n";
$fasta_header = <SEQFILE>;
$sequence = "";
while ($line = <SEQFILE>){
 chomp($line);
 $sequence .= $line;
}
if ($sequence =~ m/TATACCC/i){
 print "$fasta_header";
 print "oh,oh, sequence with TATACCC.\n";
};
close(SEQFILE);
```

# Even fancier: exercise 2

```
while ($file_name = <STDIN>) {
 open (SEQFILE, $file_name)
 or die "could not find the file $file_name \n";
 $fasta_header = <SEQFILE>;
 $sequence = "";
 while ($line = <SEQFILE>){
 chomp($line);
 $sequence .= $line;
 }
 if ($sequence =~ m/TATACCC/i){
 print "$fasta_header";
 print "oh,oh, sequence with TATACCC.\n";
 };
 close(SEQFILE);
};
```

# Useful setting: changing the “record boundary”

- Perl actually does not read lines but “records”
- Normally a record is something limited by a newline character (“\n”)
- We can change the record boundary used by perl, ex:  
$$\$/ = ">" ;$$
- Now the perl program will read an entire fasta entry at a time.
- **HOWEVER:** the “>” character of the next entry will be read with the previous one
- *Chomp* will remove record boundary

# Let's try

```
$/ = ">";
while ($entry = <>){
 print "-----\n";
 print "$entry \n";
}
```

- the output is not good, the “>” is printed at the end of the fasta

# Let's try

```
$/ = ">";
while ($entry = <>){
 chomp($entry);
 print "\n-----\n";
 print ">";
 print $entry;
}
```

- now it is better, but the first sequence printed still does not have the “>”, and we forgot to change lines at the end of the sequence

# Let's try

```
$/ = "\n>";
while ($entry = <>){
 chomp($entry);
 print "-----\n";
 print ">";
 print $entry;
 print "\n";
}
```

– now we have a good output

# Patterns, regular expressions

- searching for individual sequences is not enough
- we need a more general way of describing sequences
- we want to describe sets of sequences with a short descriptions
- one way is to use more general patterns

# How do we describe a more general pattern?

- *Regular Expressions!!!!*
- Regular expressions are short ways to describe a set of sequences
- Regular expressions in perl are similar to Unix, but syntax is different
- We have to remember that this is a *conceptual description*, what we see is a description of a SET of sequences

# Building regular expressions

- each letter and number is a pattern that describes itself
- a selection of characters: [`<list of characters>`]  
[cgat] =====> any nucleotides  
[cgatCGAT] =====> any nucleotide, small and uppercase

# More regular expressions

- **a range of characters:** *<inicial character>-<final character>*
  - a-z** ==> same as abcdefg...z
  - 0-9** ==> same as 0123456789
- **repeating patterns zero or more times:** \*
  - [cgat]\*** ==> any number of nucleotides
  - examples: cttac, cggg, cg, c, tta
- **repeating patterns one or more times:** +
  - [0-9]+**
- **Any character:** .
  - >.\*\n** ==> a fasta header in a line, that is “>” folowed by any caracter (“.”) , repeated any number of times (“.\*”), with an “enter”(“\n”) at the end

# More patterns...

- range of repetitions: {N,M}  
`[cgatCGAT]{100,400}` =====> any nucleotide sequence that has between 100 and 400 nucleotides  
`[cgatCGAT]{100,}` =====> any nucleotide sequence that has AT LEAST 100 nucleotides  
`[cgatCGAT]{,400}` =====> any nucleotide sequence that has AT MOST 400 nucleotides
- `\d` ==> any numerical characters (this is the same as [0-9])
- `\s` ==> space
- `\t` =====> a tab
- grouping many patterns in one: (...)  
`(gcc)` =====> a specific codon
- alternative patterns: ...|...|...  
`(uc|ucc|uca|ucg)` =====> rna sequences for serine  
`(uc|ucc|uca|ucg)+` =====> at least one serine codon

# Pattern examples

- sequence that starts with a lowercase letter and is followed by one or more digits and letters

`[a-z][a-z0-9]+`

examples: a1, bbb, z2cc, z12345, d1aa

- sequence with a poli-a tail (at least 9 “a”)

`[cgat]+aaaaaaaaa[a]*`

`[gcat]+aaaaaaaaa[a]+`

`[atcg]+a{9,}`

- guess the next one

`[cgatCGAT]*[aA]{9,}`

====>sequence with poli-a tail (more than 9 “a”), but with lower or upper case letters

# Examples

- checking if a string contains either “accta” or “cgтта”

```
if ($sequence =~ m/(accta|cgтта)/) {

}
```
- detecting more general pattern

```
if ($sequence =~ m/tatatatatata(ta)*[cgat]{6,8}cgatta/){
 print “found pattern\n”;
}
```

  - prints the message “found pattern” if the \$sequence contains at “ta” at least 7 times, followed by a conserved pattern “cgatta” after 6 or 8 nucleotides

# Anchoring searches: searching in the beginning or the end

- if you want something to appear in the beginning of a string type “^” at the start of the pattern
  - `m/^>/`  $\implies$  checks if the string starts with “>”
- if you want something to appear at the end of the string, type “\$” at the end of the pattern
  - `m/;$/`  $\implies$  checks if the string has a semicolon as its last character
  - `m/the end$/`  $\implies$  checks if the string has “the end” as te last 7 characters

# Regular Expression Example

```
While ($sequence = <STDIN>){
 if ($sequence =~ m/^>/){
 print "fasta comment line\n";
 }
 else{
 if ($sequence =~ m/(ucu|ucc|uca|ucg)/){
 print "found another serine"
 }
 }
}
```

# Exercise

- 1) write a perl program that reads genebank records and print only those that are from the organism Homo Sapiens
- 2) Copy the file Sequence.bg from the data in the course site into your computer
- 3) test your program in that file

# Exercise

- write a perl program `facility.pl` that reads a multi fasta file and print only the sequences that come from our laboratory ( they should have the text “`bioinfo_cbag`” followed by numbers and a blank in the fasta header)
- copy the file `multiseq.fas` from `test@cbag.bioinfo.org` into your account and try your program in it

# Exercises

- write a perl program that reads a fasta sequence, and finds out if it has a tata box. If it has, prints the sequence, but with the text “| tata box” added to the fasta header.
  - for the sake of this exercise suppose a tata box is:
    - TATA
    - 12 to 16 bases of any kind
    - 3 to 8 “C’s
- (difficult) write a perl program that reads many fasta sequences, detect if each one has a tata box, and print the comment lines of the ones that do

# Example

```
open(GENOME, “/home/alan/dog/genome.fasta”)
 or die “could not find genome file \n”;
$comment = <GENOME>;
$sequence = “”;
while ($line = <GENOME>){
 chomp($line);
 if ($line =~ /^>/) {
 #new sequence, treat old first
 #mask out vector sequence
 $sequence =~ s/cccattggt/xxxxxxxxxx/g ;
 print “$comment $sequence \n”;
 #now we have a new fasta
 $comment = $line;
 $sequence = “”;
 }
 else {$seq = $seq.$line;}
}
```

# Arrays: storing many things of the same type

- when we have to deal with a big number of things of the same type
- instead of creating a variable for each value we can use an *indexed* variable, or *array*

```
@instructors = (“alan”, “chuong”,
“jessica”);
```

```
print $instructors[0] , “\n”; #alan
```

# Using *split* to separate fields

- if we have a string that contains many fields and there is a character that separates the fields, for example

alan durham#durham@ime.usp.br#(55)(11)3091-6299

====> 3 fields separated by “#”

- we can use the *split* operation separate the fields of a string in an array.
- the *split* operation needs to know the string to separate and the field separator

`split( ^#/, $entry)`

# Split: example 1

- We want to read a genbank entry and get rid of the bases
- Read the genbank entry, separate what is before and after “ORIGIN”, print only what is before

```
$/ = “\n/\n” ; #read a whole entry;
```

```
$entry = <>;
```

```
chomp($entry); #get rid of “/\n”
```

```
($comments,$bases) = split(“\nORIGIN\n“, $entry);
```

```
print $comments;
```

- let us try it.

# Example

- write a perl program that reads a table with fields separated by blanks or tabs (\t) and print just the first, third and ninth columns

```
while ($line = <>){
 @fields = split(/[\s\t]+/, $line);
 print "$fields[0] \t $fields [2] \t $fields[8];
}
```

- write this example as splitexample.pl and run it on the output of “ls -l”  
– ls -l | perl splitexample.pl

# Exercise

- write a program `split_ex.pl` that does a similar task, that is reads lines and select columns number 0, 2 and 7
- but this time read the name of a file to be filtered and read the numbers of the 3 columns to be printed
- put the result of “`ls -l`” in a file named “out”
- run your program on that file, selecting columns number 0,1 and 7

# Solution

```
print "file name:";
$file = <>;
chomp($file);
print "first collumn to be printed:";
$coll_1 = <>;
chomp($coll_1);
print "second collumn to be printed:";
$coll_2 = <>;
chomp($coll_2);
print "third collumn to be printed:";
$coll_3 = <>;
chomp($coll_3);
open (THEFILE, $file) or die "sorry,cannot open the file \n";
while ($line = <THEFILE>){
 @fields = split(("[s\t]+/", $line);
 print "$fields[$coll_1] \t $fields[$coll_2] \t $fields[$coll_3]\n";
};
close (THEFILE);
```

# Example

- 1) write the previous example in a file named “onlyCommentsGenebank.pl” and test it in a real genebank file
- 2) To get an sample genebank entry do a remote copy from the server
  - `scp test@cbag.bioinfo.org:seq/geneBankEntry.gbk .`
- 3) rewrite your program to work with not just one, but many genebank entries (that is, use a “while”)

# Solution

```
$/ = "\n/\n" ; #read a whole entry;
while($entry = <>){
 chomp($entry); #get rid of "\n"
 ($comments,$bases) = split("\nORIGIN\n", $entry);
 print $comments;
}
```

# Arrays: large number of data under the same name.

- what if we want to split an entry that has an unknown number of fields?
- we know we can split, but where do we put the fields?
- we can use *arrays*

# Arrays: definition

- arrays are *indexed* variables, that is variables that store more than one value, and use indexes to access them.
- remember the algorithm workshop: we had many boxes and just used the name *box*, with an index
  - `box[1]`, `box[2]`, `box[i]`, etc.
- in perl:
  - `$box[1]`, `$box[2]`, `$box[i]`

# Arrays: using with split

- if we do not know how many fields the string has, we put the result of the split into an array
  - @lines = split(“\n”, \$genebank\_entry)
- the above example separates the lines of a genebank entry in different positions
- therefore
  - \$line[0] contains the first line
  - \$line[1] contains the second line,
  - etc.

# Example

- let us rewrite our program to read a genbank entry and print only the access number (first line of the genbank entry, and the definition (second line of the genbank entry)

```
#!/usr/in/perl
$/ = "\n//\n";
while ($entry = <>){
 ($comments, $bases) = split(/ORIGIN/, $entry); #separate the two parts
 @all_lines = split("\n", $comments); #separate the lines of first part in array
 #”@all_lines”
 print $linhas[0], "\n", $linhas[1], "\n"; #print first line ($all_lines[0]) and
 #second line ($all_lines[1])
}

```

# Exercise

- modify your program so it writes a fasta that contains the access number and definition in the fasta header
- hint:
  - you already separated the bases, now you only need to get rid of the unwanted characters .....
  - eliminating something is the same as replacing them for nothing

# The code

```
#!/usr/in/perl
#change the record separator to read the whole genbank entry
$/ = "\n/\n";
entry = <>;
chomp($entry);
#now we separate each part
($comment,$bases) = split("ORIGIN",$entry);
#get accession code to generate fasta header
#we need to separate the lines, and
#look in each one, trying to find what we want
@lines = split("\n",$comment); #separate the lines
$accession_line = $lines[0];
$definition_line = $lines[3];
#get rid of unwanted characters in the bases part
$bases =~ s/[0123456789\t\s]//g;
#print the results
print ">$accession_line $definition_line\n";
print "$bases \n\n";
```

# Review: what we have learned so far

- what is a program
- what is an algorithm
- programming is to *describe an algorithm in an formal, clear way that a computer can understand*

# Review: what we have learned so far - II

- perl basic concepts
  - variable
  - conditional statement
  - reading, from keyboard
  - printing in the screen
  - reading from files

# Processing an array one item at a time

- So far we know how to get specific entries in an array
- what if we want to do the same thing for all the entries in an array and we do not know its size?
- this problem is similar to the one processing many entry lines
- we can use the “*foreach*” command
  - `foreach <variable for one> (@array_name)`

- example

```
foreach $line (@lines) {
 if ($line =~ m/DEFINITION/){
 $line =~ s/DEFINITION\s*//;
 $def= $line;
 }
}
```

# Exercise

- since *foreach* treats all entries, you can now rewrite your program, so it does not depend on the order of the lines of the first part...

```

#change the record separator to read the whole genebank entry
$/ = "\n";
entry = <>;
chomp($entry);
#now we separate each part
($comment,$bases) = split("ORIGIN",$entry);
#get accession code to generate fasta header
#we need to separate the lines, and
#look in each one, trying to find what we want
@lines = split("\n",$comment); #separate the lines
foreach $line (@lines){
 if ($line =~m/DEFINITION/) {
 $definition_line = $lines;
 }
 if ($line =~ m/ACCESSION/){
 $acession_line = $line;
 }
}
$definition_line =~ s/DEFINITION\s*//;
$acession_line =~s/ACCESSION\s*//;
#get rid of unwanted characters in the bases part
$bases =~ s/[0123456789\n\s]//g;
#print the results
print ">$acession_line $definition_line\n";
print "$bases \n\n";

```

# Exercise

- write a perl script that reads the name of a species and the name of a file containing a multiple genebank entries
- this program should select all entries that are of sequences in the specified organism and print:
  - the access number
  - the title of the articles where the sequence appeared

# Using unix within perl: *system*

- we can call any unix command from inside perl using the function “system”
  - `system(“cp /home/alan/ibi5011/data/*.fasta .”);`
- we can also call using backquotes and get the result as a string
  - `$files = `ls``
  - `print $files;`
- using *system* we can build perl programs that run other programs in the system
- using back quote we can grab the standard output of a program and process it withing perl

# Exercise

- *write a perl program find\_mouse.pl that:*
  - reads a directory name,
  - look for mouse sequences within all fasta files in that directory
  - and put these in a file “mouse.fasta”
  - try it for files in /home/alan/ibi5011/data/



# Solution

```
$directory = <>;
chomp($directory);
$fileString = `ls $directory/*.fasta`;
open(SAIDA, ">mouse.fasta") or die "cannot open output file\n";
@files = split(/\n/, $fileString);
foreach $file (@files){
 open(FASTA, $file) or die "something weird, cannot open $file";
```

```
 close(FASTA)
}
close(SAIDA);
```

# Solution

```
$directory = <>;
chomp($directory);
$fileString = `ls $directory/*.fasta`;
open(SAIDA, ">mouse.fasta") or die "cannot open output file\n";
@files = split(/\n/,$fileString);
foreach $file (@files){
 open(FASTA, $file) or die "something weird, cannot open $file";
 $/ = ">";
 <FASTA>;
 while ($sum_fasta = <FASTA>){
 chomp($sum_fasta);
 if ($sum_fasta =~ m/Mus[\t\s]+musculus/){
 print SAIDA ">$sumFasta";
 }
 }
 close(FASTA);
}
close(SAIDA);
```

# Printing to files:opening the file

- when printing into files, we also need to open and close them
- however the format of the open string is slightly different

```
open(FILEHANDLE, ">name_of_the_file");
```

- the string with the file name has to start with ">"

# Printing to files: the print command

- we print as before, however just after the “print” word, we insert the file handle:

```
print FILEHANDLE $stuff_to_be_printed
```

- be carefull: there are only spaces around the file handle

# Let's try

```
open(OUTFILE, ">generatedFile.txt")
 or die "could not open file\n";
while ($entry = <>){
 chomp($entry);
 print OUTFILE "$entry\n";
}
close (OUTFILE);
```

# Exercise:

write a program named “substitutueInMultifasta.pl” that reads a sequence and the name of a multifasta file.

- for each sequence in the multifasta file, the program should mask the sequence with “XXXX”.
- the program should generate a multifasta file “result.mfasta” with the new fastas. In the new file insert a blank line between each fasta

# Solution

```
print "give me the sequence to be masked:";
$sequence_to_be_masked = <>;
print "type the name of the input file:";
$input_file = <>;
chomp($input_file)
open(INPUT, $input_file)
 or die "cannot open input file";
chomp($sequence_to_be_masked);
open (RESULT, ">result.mfasta")
 or die "could not open result file\n";
$/ = ">";
<INPUT> ; #get rid of the first empty read
while ($fasta = <INPUT>){
 chomp($fasta);
 $fasta =~ s/$sequence_to_be_masked/XXXX/gi;
 print RESULT ">", $fasta, "\n";
}
close (INPUT);
close (RESULT);
}
```

# Hashes: arrays with arbitrary indexes

- many times in computing you need to index things by a string
- to use names as indexes we need *hashes*
- hashes are like arrays, but we use “{...}” instead of “[...]” and we use “%” instead of “@”.

- example

```
$grades{“alan”} = “C”;
$grades{“luciano”} = “A”;
print “luciano:”, $grades{“luciano”}, “alan:”, $grades{“alan”}, “\n”;
```

- another way:

```
%grades = (“alan” => “C”, “luciano” => “A”);
print “luciano”, $grades{“luciano”}, “alan:”, $grades{“alan”}, “\n”;
```

- we can also write

```
$x = “alan”;
$y = “luciano”;
$grades{$x} = “C”;
```

# I can do things to one entry at a time too

- `keys(<hash_name>)` returns an array with the hash's keys

- Example:

```
%final_grades = (“alan durham” => 10,
 “joao e. ferreira” => 8,
 “ariane machado” => 5);
```

```
print “name\tfinal grade\n”; #using tab
```

```
foreach $key (keys(%final_grades)){
```

```
 print STDOUT $key,“\t”, $final_grades{$key}, “\n”;
```

```
}
```

Try it!!

# We can read a hash table

- try to modify the previous program to make it read the hash table

```
print “please type the table in the format key-value:\n”;
```

```
print “name\tfinal grade\n”; #using tab
```

```
foreach $chave (keys(%final_grades)){
```

```
 print STDOUT $chave,“\t”, $final_grades{$chave}, “\n”;
```

```
}
```

# We can read a hash table

- try to modify the previous program to make it read the hash table

```
print "please type the table in the format key-value:\n";
while ($line = <>){
 chomp($line);
 ($chave, $valor) = split("-", $line);
 $final_grades{$chave} = $valor;
}
print "name\tfinal grade\n"; #using tab
foreach $chave (keys(%final_grades)){
 print STDOUT $chave,"\t", $final_grades{$chave}, "\n";
}
```

# We can test a hash to see if some entry exists

- exists(<hash entry>)
- exemplo:

```
%nomes = (“alan” => “durham”, “junior” => “barrera”);
if (exists($nomes{“alan”})) {
 print “good, alan is here!\n”
}
if (!exists($nomes{“Junior”})) {
 print “bad, Junior is not here.\n”;
}
```
- Try it!!!!

# Example: counting the number of entries

- read a bunch of numbers, tell me how many are bigger than 5

# Example: counting the number of entries for each organism

- Problem: read a multi-genebank file and list the organisms of the sequences and the number of sequences of each organism
- first step: read one entry
- second step: isolate the organism's name
- third step(use hash): if new organism, count one, if repeated organism, add one
- print the organism names and counts
- do it!

# Solution

```
$/ = “\n/\n”;
%organism = ();
while ($entry = <>){ #read the entry
 ($first_part, $sequence) = split(/\nORIGIN/, $entry);
 @lines = split(/\n/, $first_part);
 #look for the line that contains the organism
 foreach $line (@lines){
 if ($line =~ m/ORGANISM/) {
 #clean the line
 $line =~ s/ORGANISM[\s\t]*//;
 #check if is already in hash, add if so, set to one if not
 if (exists($organisms{$line})){
 $organisms{$line} += 1;
 }
 else {
 $organisms{$line} = 1;
 }
 }
 }
}
foreach $nome (keys(%organisms)){
 print “Organism: $nome ==> $organisms{$nome} copies\n”
}
```

# A useful perl function: sort

- you can use the function `sort` to put an array in order:
  - `sort(@array);`
- try

```
@original = ("alan", "alberto", "aab", "zilda", "pedro");
@ordered = sort(@original);
foreach $entry (@ordered){
 print "$entry\n";
}
```
- the “default” sort is alphabetical:
  - `sort( (1, 5, 10, 15, 25, 8))`

# Example

- now we can repeat the previous example, but printing the organisms in alphabetical order
- what do you have to change in the previous exercise?
- you have to use  
`foreach $chave (sort(keys(%organisms)))`
- try it!

# Solution

```
$/ = "\n/\n";
while ($entry = <>){ #read the entry
 ($first_part, $sequence) = split(/\nORIGIN/, $entry);
 @lines = split(/\n/, $first_part);
 #look for the line that contains the organism
 foreach $line (@lines){
 if ($line =~ m/ORGANISM/) {
 #clean the line
 $line =~ s/ORGANISM[\\s\\t]*//;
 #check if is already in hash, add if so, set to one if not
 if (exists($organisms{$line})){
 $organisms{$line} += 1;
 }
 else {
 $organisms{$line} = 1;
 }
 }
 }
}
foreach $chave (sort(keys(%organisms))) {
 print "Organism: $chave ==> $organisms{$chave} copies\n";
}
```

# General sorting: the code block

- alphabetical is the “default” order,
- however you can define any order you want by giving a “code block”
- `sort {<comparison code>} @array`
- in `< comparison code>` you use `$a`, and `$b` as variables to designate what do you want to do with the first and second number.
- the “code block” should produce 3 values
  - 1 indicating `$b` comes before `$a`
  - 0 indicating `$a` and `$b` are equivalent
  - -1 indicating `$a` comes before `$b`.

# Operators for comparison

- `<=>` compares two numbers, returning -1 if the first one is smaller, 0 if they are equal, 1 if the first one is bigger
  - `{ $a <=> $b }` can be used to sort numbers in ascending order
  - `{ $b <=> $a }` can be used to sort numbers in descending order
- `cmp` is similar, but for strings
  - `{ $a cmp $b }` can be used to sort ascending alphabetical order order (same as the default sorting)
  - `{ $b cmp $a }` can be used to sort in descending alphabetical order

# Exercise

- change your program to print the list in 2 different orders:
  - ascending by organism name
  - descending by organism name

# Getting the size of arrays;

- what if we want to know the size of an array?
- just assign the array to a scalar
- try:  

```
@array = ("alan", "peter", "kim");
$num = @array;
print "$num \n";
```
- question: how do we get the size of a hash?
- answer: `$size = keys(%my_hash);`

# Match: getting the matches

- when we use perl to find regular expressions, actually perl generates more data that we have been using
- `<string> =~ m/<pattern>/g;`
- the code above generates an array with all the instances of the regular expression
- try:  

```
$string = "the student was stupid enough so stagger";
@array = ($string =~ m/st[a-z]+/g);
print @array;
```





# Perl can be used to run things in Linux

- any command of the underlying system can be performed from perl
- example

```
system("blastall seq.fasta") or die ("cannot run blast\n");
```

- more interesting example

```
while ($command = <STDIN>){
 sytem ($command)
 or die "bad command!!! \n");
}
```

- try the second one!

# You can use perl to automate tasks building a *pipeline*

- pipeline is a term used in Bioinformatics a lot
- generally means a set of tasks performed incrementally on some data
- instead of performing manually from the shell, we can use perl
- sometimes this is done using unix *shellscript*
  - not as flexible and powerful as perl
- perl programs describing pipelines can become very complex

# So what?

- perl can be used (actually IS used) to write pipelines
- because we can insert variable names in the system calls, we can write program that perform a task for an arbitrary file, for example
- example

```
$basicName = <STDIN>;
$chromatFile = $basicName.“abi”;
system(“phred $chromatFile > $basicName.phd”);
system(“convertToFasta $basicName.phd $basicName.fasta”);
```

# Many ways of doing loops

- in bioinformatics we want repetition
  - blast all the 10.000 est sequences of my genome project
  - finding all the copies of a primer in my genome and reporting which is their position
  - getting a list of ORF positions, separate each one from the genomic sequence and put them in a multifasta file
  - mask all occurrences of vector code in a sequence

# Many Loops in Perl

- perl is a very flexible language with many ways of describing repetitive processes
- substitution operation
- reading loops
- generic while loops
- for loops
- translate operations

# perl has many ways of specifying repetition:

- while,
- do...while
- for
- foreach
- we will look at the last one, you should investigate the others

# “foreach” : processing the elements of an array

- many times we want to do a specific task to all elements of an array
  - printing
  - querying
  - using the string as a filename
- to do this we can use the *foreach* command

```
foreach $element (@array) {
 #use $element to perform tasks
 print “here is another element $element \n”;
}
```

# Example: processing the *ls* command

- read directory name, open all files in the directory and check which ones have the a tata-box

```
print "type the directory name:";
$directory = <>;
chomp($directory);
$file_list = `ls $directory`;
@files = split("\n", $file_list);
foreach $file_name (@files){
 open (FILE, $file_name);
 $fasta_comment = <FILE>;
 $sequence = "";
 while ($line = <FILE>){
 $sequence .= $line;
 }
 if ($sequence =~ m/tata(ta)*[acgt]{10,20}aug/){
 print "file $directory/$file has a tata-box";
 }
}
```

# Some important notes

-

# Subroutines

- you can separate perl code to do specific tasks you your program (split, chomp are subroutines)
- this separation can make your program easier to read and to maintain
- routines well written can be eventually reused in other programs

# Examples of subroutine tasks

- get the header of a fasta
- get the bases of a fasta
- obtain specific fields of a genebank entry

# Characteristics of a subroutine

- produces one value only: strings, numbers, etc.
- uses *arguments*:
  - values considered to produce the desired result
  - ex: a string with a fasta, a string with a gbk entry, some numbers, name of a field, etc
- is used in a program like a value:
  - $\$a = \text{substr}(\$s1, 1, \$c);$
  - name: substr,
  - arguments: \$s1, 1, \$c
  - result is stored in \$a

# Syntax:

- *sub* <name> {  
    my \$arg1 = shift;  
    ....  
    my \$argn = shift;  
    <code that produces the result, say, in variable \$r>  
    return \$r  
}

# Example of a subroutine

- writing

```
sub get_fasta_header {
 my $fasta_string = shift;
 @lines = split(/\n/,$fasta_string);
 return $lines[0];
}
```

- using

```
$/ = ">";
while ($entry = <>){
 print ">",get_fasta_header($entry),"\n";
}
```

# Exercise

- write a perl subroutine that gets as argument a string with a complete genebank entry and returns a string with the bases of that entry
- hint: look at the code we have already written and copy parts of it

# Modules: getting lots of interesting routines together

- sets of routines and variables that can be packaged together to be used by other programs
- to create a module we write a file with the “.pm” suffix
- the contents of the file are a set of variables and subroutines and a package declaration
  - packages are sets of perl names, you can read more about it in the books, but for here you can just assume you have to write an extra line.
  - the package should have the same name as the file

# Example: file GBKEntry.pm

```
package GBKEntry;
sub get_bases {
...
}
sub get_species{
...
}
sub generate_fasta_header {
....
}
```

# Using a package

- include a *use* directive in the beginning of the program  
use GBKEntry;
- use the variables and functions

```
$="\n//";
```

```
while ($gbk_entry = <>){
```

```
 $header = GBKEntry::generate_fasta_header($gbk_entry);
```

```
 $seq = GBKEntry::get_bases($gbk_entry);
```

```
 print "$header\n$seq\n";
```

```
}
```

# Exercise

- create the package GBKEntry.pm with the functions above, rewrite your programs to use it.

# Strict

- strict
  - helps debug your program
  - every variable has to be declared with *my* before it is used, otherwise error

```
my $number;
my @lines;
my %grades;
```
  - every time undeclared variable is used, program complains
  - specially useful in finding typing errors
  - try to **ALWAYS** use it, I do.

# Bioperl

- modules, packages, scripts with more code that you will ever use
- most format conversions and information extraction available
- will reduce you need to write perl code, most things can be found as scripts, others modules you can use to make your program much smaller
- <http://bioperl.org>

# What's next?

- we have seen a lot but this is just a taste of perl
- you know have the basics of perl, but more study is necessary
- with just a little more you should be able to build very useful programs

# Where?

- internet tutorials
  - <http://www.sanbi.ac.za/tdrcourse/>
  - <http://www.dbbm.fiocruz.br/class/schedule.html>  
(look for Cris Mungall's lectures in the schedule, there are links to each topic)
- books
  - Perl in a Nutshell
  - Perl for Bioinformatics
  - Learning Perl
  - The Perl Cookbook
  - Core Perl

# Case study: building modular pipelines

- as said before, perl can be used to implement pipelines
- use `system, `...``, and perls text processing capabilities to run programs and parse their output
- high-throughput projects need automatic processing, manual processing is unfeasible
- standard approaches:
  - you can write ONE big perl script to do everything
  - you can write SOME big perl scripts to do big phases

# Building modular pipelines

- previous approaches mean you have to write a lot of code for each new pipeline
- alternative approach: modular pipelines

# Modular pipelines

- find basic component of pipelines
- make them look similar
- create standard of communication between components
- create pipelines by getting the components together

# Modular pipelines

- much harder to build
- need more software development expertise
- take longer to build
- save time in the long run by making future pipelines easy to create

# Egene: a modular pipeline

- objective: processing chromatograms and getting clean sequences
- many tasks: filtering (against databases, quality, size), masking, base-calling, end-trimming, assembling, building reports
- visual language to specify how each component work and communication between them
- program to run specification

# Egene: a modular pipeline

