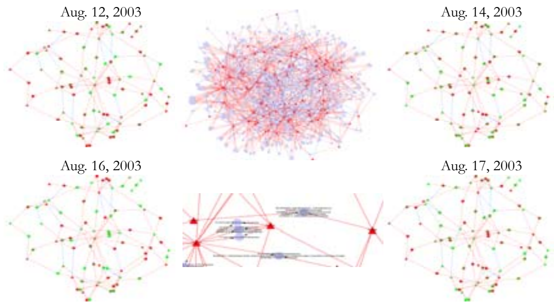
Lecture 1

Torgeir R. Hvidsten
Assistant professor in Bioinformatics
Umeå Plant Science Center (UPSC)
Computational Life Science Centre (CLiC)

My research interests

A systems biology approach to model the transcriptional network in trees



Course goals

At the end of this course you will know how to :

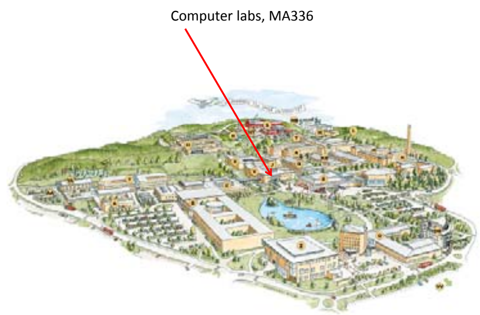
- write and debug basic Perl programs
- select the correct algorithm design for a given problem and do time/space complexity analysis
- use online resources with Perl and use online Perl libraries and interfaces
- use Perl to pipeline other programs (e.g. BLAST) and parse output
- combined the above to solve practical Bioinformatics problems

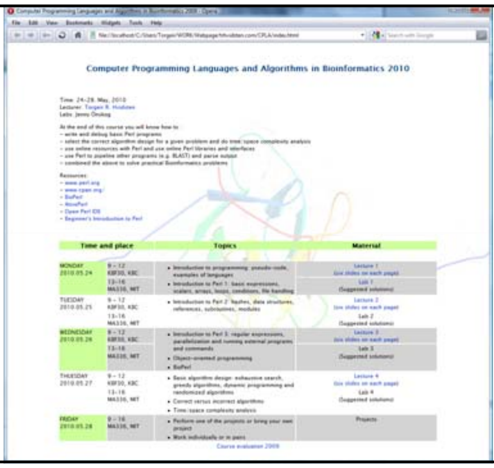
Course information (I)

- Monday - Thursday:
  - 9 – 12: lectures and lab summary
  - 13 – 16: computer labs
- Friday
  - 9 – 16: project in the computer lab
- Credit points: 2ECTS
- To pass:
  - attend lectures and labs
  - send lab and project documentation at the end of each day to: `jenny.onskog@plantphys.umu.se`

Course information (II)

- Course webpage:
  - `http://www.trhvidsten.com/CPLA/`
- Here you can find the
  - course plan
  - online resources
- and download
  - lecture slides
  - labs/project description
  - additional material





This lecture

- Introduction to programming:
  - programming languages
  - pseudo-code
- Introduction to Perl 1
  - basic expressions
  - scalars
  - arrays
  - loops
  - conditions
  - file handling

Algorithm

- Algorithm: a sequence of instructions that one must perform in order to solve a well-formulated problem
- Correct algorithm: translate every input instance into the correct output
- Incorrect algorithm: there is at least one input instance for which the algorithm does not produce the correct output
- Many successful algorithms in bioinformatics are incorrect

Programs

- Algorithms are implemented in a programming language to form programs
- Programs consists of:
  - Variables: names with values (float, integer, string) or arrays/tables/hashes of values
  - Conditional statements: IF-THEN-ELSE
  - Loops: while, for, until, etc.
  - Modularity: procedures/functions/sub-routines/objects/methods
- Pseudo-code: programming language-independent, often used to sketch a program using pen and paper

Pseudo-code

Sorting problem: Sort a list of n integers:
$\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$ e.g. $\boldsymbol{a}$ =(7,92,87,1,4,3,2,6)

SelectionSort($\boldsymbol{a}$,n)

for $i \leftarrow 1$ **to** n-1
  $j \leftarrow$ Index of the smallest element among $a_i, a_{i+1}, \ldots, a_n$
  Swap elements a_i and a_j
return $\boldsymbol{a}$

Example run

$i = 1$:	(**7**,92,87,1,4,3,2,6)
$i = 2$:	(1,**92**,87,7,4,3,2,6)
$i = 3$:	(1,2,**87**,7,4,3,92,6)
$i = 4$:	(1,2,3,**7**,4,87,92,6)
$i = 5$:	(1,2,3,4,**7**,87,92,6)
$i = 6$:	(1,2,3,4,6,**87**,92,7)
$i = 7$:	(1,2,3,4,6,7,**92**,87)
	(1,2,3,4,6,7,87,92)

Syntax versus semantics

- Syntax: the rules for constructing valid statements in a programming language
- Semantics: the meaning of a program

- A specific algorithm implemented in different programming languages would use different syntax, but have the same semantics
- Syntax is easy and can be checked before execution (the interpreter will tell you when you make syntax mistakes)
- Semantics is hard and "bugs" typically only reveal themselves at execution time

Programming languages

- Imperative programming: describes computation as statements that change a program state (e.g. Perl, Fortran, C, and Java)
- Functional programming: treats computation as the evaluation of (mathematical) functions, and often avoids state (e.g. LISP)
- Declarative programming: while imperative programs explicitly specify an algorithm to achieve a goal, declarative programs explicitly specify the goal and leave the implementation of the algorithm to the support software (e.g. PROLOG)

Sorting: imperative/procedural

Sorting problem: Sort a list of n integers:
$\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$

SelectionSort($\boldsymbol{a}$,n)

```
for i ← 1 to n-1
    j ← Index of the smallest element
    among a_i, a_{i+1}, …, a_n
    Swap elements a_i and a_j
return a
```

Pseudo-code hides ugly details such as

"Swap elements a_i and a_j"

```
tmp ← a_j
a_j ← a_i
a_i ← tmp
```

or

"$j \leftarrow$ Index of the smallest element among $a_i, a_{i+1}, \ldots, a_n$"

IndexOfMin(**array**,*first*,*last*)

```
index ← first
for k ← first + 1 to last
    if array_k < array_index
        index ← k
return index
```

Remember, though, that the devil is in the details!

Recursion

RecursiveSelectionSort($\boldsymbol{a}$,*first*,*last*)

```
if (first < last)
    index ← Index of the smallest element
    among a_first, a_first+1, …, a_last
    Swap elements a_first and a_index
    a ← RecursiveSelectionSort(a,first+1,last)
return a
```

Example I

Write pseudo-code for a program that solves a quadratic equation $ax^2 + bx + c = 0$:

QuadraticEquationSolver (a, b, c)

Remember that: $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
QuadraticEquationSolver(a, b, c)
root ← b²-4ac;
if root < 0
    return "No solution"
x1 ← (-b + √root) / 2a
x2 ← (-b - √root) / 2a
if x1 = x2
    output "Solution: x = x1"
else
    output "Solutions: x = x1 or x = x2"
```

Example II

Write pseudo-code for a program that removed duplicates in an array $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$

RemoveDuplicates (***a***)

E.g. $\boldsymbol{a} = (1, 2, 2, 4, 4)$ outputs $(1, 2, 4)$

```
RemoveDuplicates(list, n)
newlist ← ()
for i ← 1 to n
    m ← length of newlist
    foundDuplicate ← false
    for j ← 1 to m
        if list_i = newlist_j
            foundDuplicate = true
            break
    if foundDuplicate = false
        add list_i to newlist
return newlist
```

Example III

Write pseudo-code for a program that counts from 0 to $\boldsymbol{n} = (n_1, n_2, \ldots, n_k)$:

Count (***n***)

E.g. $\boldsymbol{n} = (1, 2)$ outputs:

00
01
02
10
11
12

```
Count(n, m)
c ← (0, 0, …, 0)
while forever
    for i ← m to 1
        if c_i = n_i
            c_i ← 0
        else
            c_i ← c_i + 1
            break
    output c
    if c = (0, 0, …, 0)
        break
```

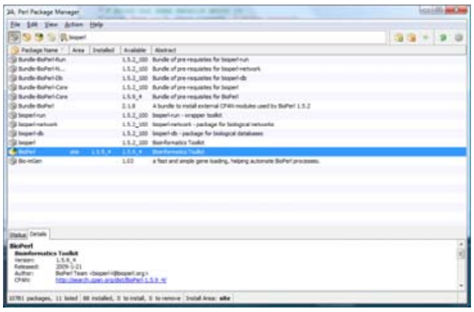
What is Perl ?

- Perl was created by Larry Wall
- Perl = Practical Extraction and Report Language
- Perl is an Open Source project
- Perl is a cross-platform programming language

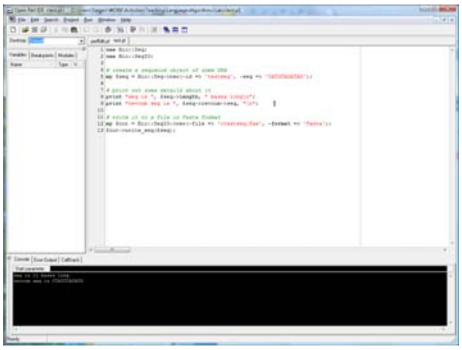
Why Perl

- Perl is a very popular programming language
- Perl allows a rapid development cycle
- Perl has strong text manipulation capabilities
- Perl can easily call other programs
- Existing Perl modules exists for nearly everything
  - http://www.bioperl.org
  - http://www.cpan.org/ (Comprehensive Perl Archive Network)

ActivePerl



Open Perl IDE



Our first Perl program

```
use strict;
use warnings;

print "Hello world!\n";
```

Hello world!

"use strict" makes it harder to write bad software

"use warnings" makes Perl complain at a huge variety of things that are almost always sources of bugs in your programs

"\n" prints a new line

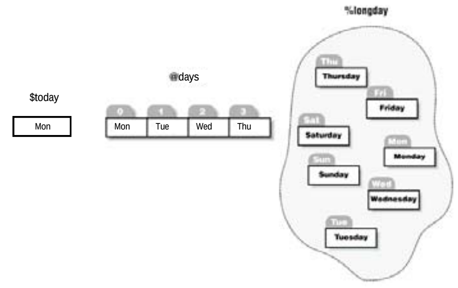
Perl scalars

- Perl variables that hold single values are called *Scalars*.
- Scalars hold values of many different types such as *strings, characters, floats, integers,* and *references*
- Scalars are written with a leading $, like: $sum
- Scalars, as all variables, are declared with my, like my $sum
- Perl is not a typed language: scalars can be strings, numbers, etc.
- You can reassign values of different types to a scalar:
  my $b = 42; $b = "forty-two"; print "$b\n";
  forty-two
- Perl will convert between strings and numbers for you:
  my $a = "42" + 8; print "$a\n";
  50
  my $a = "Perl" + 8; print "$a\n";
  Argument "Perl" isn't numeric in addition (+) at test.pl line 4.
  8

Perl scalars: some numerical operators

- $i++; # $i = $i + 1;
- $i--; # $i = $i - 1;
- $i+= 5; # $i = $i + 5;
- $i/=5; # $i = $i / 5;
- $i**3; # $i * $i * $i;
- $i = sqrt($i)

The three fundamental datatypes in Perl



- The *sigills* $,@,% must always be used.
- You can use different datatypes with the same name in the same program.

Perl Arrays

- *Arrays* hold multiple ordered values.
- Arrays are written with a leading @, like: @shopping_list
- Arrays can be initialized by lists.

```
my @s = ("milk","eggs","butter"); print "@s\n";
milk eggs butter
```

- Arrays are indexed by integer. The first scalar in an array has index 0 and no matter its size, the last scalar has index -1:

```
my @s = ("milk","eggs","butter"); print "$s[0] - $s[-1]\n";
milk - butter
```

- The sizes of arrays are not declared; they grow and shrink as necessary.

```
my @s = ("milk","eggs","butter"); $s[4] = "beer"; print "@s\n";
Use of uninitialized value in join or string at test.pl line 4.
milk eggs butter  beer
```

Perl Arrays

- Arrays can be iterated over in foreach loops. You don't need to know their size:

```
my @s = ("milk","eggs","butter");
foreach (@s) {
    print "$_\n";
}
milk
eggs
butter
```

$_ is known as the "default input and pattern matching variable".

This is all equivalent:

```
my @s = ("milk","eggs","butter");
foreach (@s) {
    print;
    print "\n";
}
```

```
my @s = ("milk","eggs","butter");
foreach my $item (@s) {
    print "$item\n";
}
```

```
my @s = ("milk","eggs","butter");
foreach (@s) {
    print "$_\n";
}
```

Perl Arrays

An array in scalar context evaluates to its size. You can easily get the index of the last item in an array.

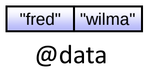
```
my @s = ("milk","eggs","butter");
my $length = @s;
print "$length\n";
3

my @s = ("milk","eggs","butter");
my $last_index = $#s;
print "$last_index\n";
2

my @s = ("milk","eggs","butter");
print "$s[$#s]\n";
butter
```

Perl Arrays

Special commands add or remove items to the front or back of arrays.
push and pop add to the back, making a stack.

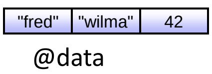
```
my @s = ("milk","eggs","butter");
push @s, "beer";
print "@s\n";
milk eggs butter beer

my @s = ("milk","eggs","butter");
pop @s;
print "@s\n";
milk eggs

my @s = ("milk","eggs","butter");
my $last_item = pop @s;
print "$last_item\n";
butter
```

Perl arrays grow or shrink as needed



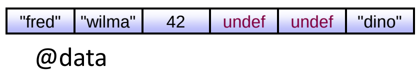

```
my @data = ("fred","wilma");
```

Perl arrays grow or shrink as needed




```
my @data = ("fred","wilma");
push @data, 42;
```

Perl arrays grow or shrink as needed




```
my @data = ("fred","wilma");
push @data, 42;
$data[5] = "dino";
```

undef

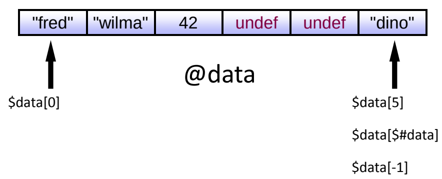
- The value of all uninitialized scalars (and scalar elements of arrays and hashes) has the special scalar value undef.
- undef evaluates as 0 when used as a number and "" when used as a string, which is why you most often don't have to initialize variables explicitly before you use them.

```
my $a; $a++; print "$a\n";
1
my @a = (1,2);
$a[3] = 23; print "@a\n";
Use of uninitialized value in join or string at test.pl line 4.
1 2  23
```

- Even after a scalar has been assigned, you can undefine them using the undef operator.

```
$a = undef;
undef @a;
```

Array indexing



Arrays and lists in assignments

"fred"	"wilma"	42	undef	undef	"dino"

@data

You can initialize or set arrays or lists by arrays or lists:

```
my ($man,$wmn) = ($data[0],$data[1]); print "$man $wmn\n";
fred wilma
my ($man,$wmn) = @data; print "$man $wmn\n";
fred wilma
@data = ("barney", "bambam"); print "@data\n";
barney bambam
my @mydata = @data; print "@data | @mydata\n";
barney bambam | barney bambam
```

You can swap elements without a temporary:

```
($data[1],$data[0]) = ($data[0],$data[1]); print "$data[0] $data[1]\n";
 bambam barney
```

Array slices

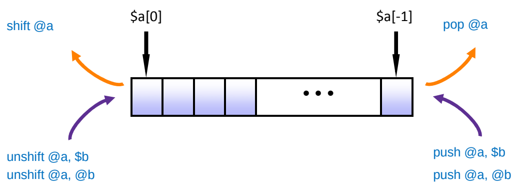
"fred"	"wilma"	42	undef	undef	"dino"

@data

You can select multiple elements from an array at once.

```
my ($man,$wmn) = @data[0..1]; print "$man $wmn\n";
fred wilma
@data[2,3] = ("barney","bambam"); print "@data\n";
fred wilma 42 barney bambam dino
my @mydata = @data[0..2,5]; print "@mydata\n";
fred wilma barney dino
@data[0,1] = @data[1,0]; print "@data\n";
wilma fred 42 barney bambam dino
```

Adding elements to array ends



Loops: Iterating over Arrays

```
for ($i = 0; $i  < @data ; $i++) { # c-style
    print "$data[$i]\n";
}

for (0..$#data) { # perl-style, default scalar is index
    print "$data[$_]\n"; # use when you need the indices explicitly
    print "The $_","th element is $data[$_]\n"; # like here
}

foreach (@data) { # perl-stylier, default scalar is element...
    print "$_\n";
}

while (@data) { # evaluates false when scalar(@data) == 0
    print shift @data, "\n"; # side-effect: removes 0th element
}
```

conditions

- if – else statements are used to test whether an expression is true or false

```
if ($a < 0) {
        print "$a is a negative number\n";
} elsif ($a == 0) {
        print "$a is zero\n";
} else {
        print "$a is a positive number\n";
}
```

- Use the function defined to test if a scalar has the value undef

```
if (defined $a) {
        $a++;
}
```

equivalent to

```
$a++ if defined $a;
```

The rules of truth in Perl

- Only Scalars can be True or False
- undef is False
- "" is False
- 0 is False
- 0.0 is False
- "0" is False
- Everything else is True (including "0.0" !)

Logical expression

```
$a == $b     # compare numbers, true if $a equal to $b
$a != $b     # compare numbers, true if $a is not equal to $b
$a eq $b     # compare strings, true if $a is equal to $b
$a ne $b     # compare strings, true if $a is not equal to $b
!$a          # boolean, true if $a is 0, false if $a is 1
```

Controlling loops: next and last

next skip to the next iteration

```
my @a = (1,2,5,6,7,0);

my @filtered;
foreach (@a) {
   next if $_ < 5;
   push @filtered, $_;
}
print "@filtered\n";
5 6 7
```

last ends the loop

```
my @a = (1,2,5,6,7,0);

my $found_zero = 0;
foreach (@a) {
   if ($_ == 0) {
      $found_zero = 1;
      last;
   }
}
print "$found_zero \n";
1
```

Sorting arrays

- Use the built in function `sort`
- The results may surprise you!

```
my @words = ("c","b","a","B");
@words = sort @words;
print "@words\n";
B a b c

my @numbers = (10,3,1,2,100);
@numbers = sort @numbers;
print "@numbers\n";
1 10 100 2 3
```

sort

- `sort` uses a default sorting operator `cmp` that sorts "ASCIIbetically", with capital letters ranking over lower-case letters, and then numbers.

  `sort @words;`
  is equivalent to:
  `sort {$a cmp $b} @words;`
- `cmp` is a function that returns three values:
  - -1 if `$a le $b`
  - 0 if `$a eq $b`
  - +1 if `$a ge $b`
- where `le`, `eq`, and `ge` are string comparison operators.
- `$a` and `$b` are special scalars that only have meaning inside the subroutine block argument of sort. They are aliases to the members of the list being sorted.

sort {$a <=> $b} @numbers

- `<=>` (the "spaceship operator") is the numerical equivalent to the `cmp` operator:
  - -1 if `$a < $b`
  - 0 if `$a == $b`
  - +1 if `$a > $b`
- You can provide your own named or anonymous comparison subroutine to `sort`:

```
my @numbers = (10,3,1,2,100);
@numbers = sort {$a <=> $b} @numbers;
print "@numbers\n";
1 2 3 10 100
@numbers = sort {$b <=> $a} @numbers;
print "@numbers\n";
100 10 3 2 1
```

Syntax summary: scalars

- Declare: `my $age;`
- Set: `$age = 29; $age = “twenty-nine”;`
- Access: `print “$age\n”;` twenty-nine

Syntax summary: arrays

- Declare: my @children;
- Set all: @children = (“Troy”,”Anea”);
- Set element: $children[0] = “Troy Alexander”;
- Access all: print “@children\n”; Troy Alexander Anea
- Access element: print “$children[1]\n”; Anea

Syntax summary: loops

```
foreach my $child (@children) {
   print “$child\n”;
}
Troy Alexander
Anea

for (my $i = 0; $i < @children; $i++) {
    print “$i: $children[$i]\n”;
}
0: Troy Alexander
1: Anea
```

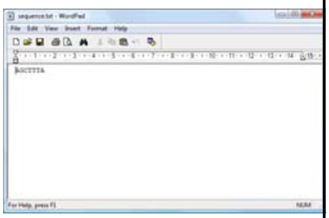
Syntax summary: conditions

```
foreach my $child (@children) {
   if (length($child) > 4) {
      print “$child\n”;
   }
}
Troy Alexander
```

Reading and writing to files

- open(A, ">sequence.txt") – creates a new file and opens it for writing
- open(A, ">>sequence.txt") – opens an existing file for writing
- open(A, "sequence.txt") – opens an allready existing file for reading

```
open(A , ">sequence.txt");
print A “AGCTTTA\n";
close(A);
```



Reading and writing to files

```
open(A , ">>sequence.txt");
print A "AGCTTTA\n";
close(A);
```

```
open(A , "sequence.txt");
my $line1 = readline *A;
my $line2 = readline *A;
close(A);
print "$line1 | $line2\n";
AGCTTTA
 | AGCTTTA
```

Reading files

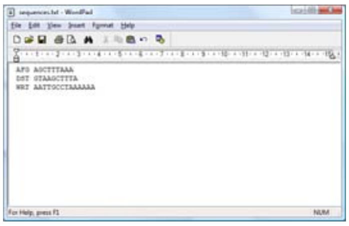
```
my @seqs;
open(A , "sequence.txt");
while (<A>) {
   chomp;
   push @seqs, $_;
}
close(A);
print ”@seqs\n";
AGCTTTA AGCTTTA
```

chomp removes ”\n” from the end of the line if it exists

Splitting strings: split

- You can split a string on any substrings that match a regular-expression with:
  - @array = split /PATTERN/, $string;
  - split /\s/, "do the twist"; # gives ("do","the","twist")
  - split //, "dice me"; # gives ("d","i","c","e"," ","m","e");
- Extremly useful when parsing files:

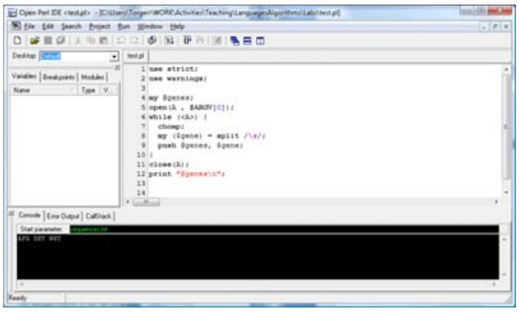
```
my @genes;
open(A , "sequences.txt");
while (<A>) {
    chomp;
    my ($gene) = split /\s/;
    push @genes, $gene;
}
close(A);
print "@genes\n";
AFG DST WRT
```



Extracting fragments: substr

```
my $string = "AC Milan";
my $fragment = substr $string, 3;
print "$fragment\n";
Milan

my $string = "F.C. Internazionale";
my $fragment = substr $string, 5, 5;
print "$fragment\n";
Inter

my $string = "F.C. Internazionale";
my $fragment = substr $string, -7, 4;
print "$fragment\n";
zion
```

@ARGV: command-line arguments



Acknowledgements

- Several slides were taken or re-worked from David Ardell and Yannick Pouliot.