Introduction to Perl Programming

Recitation

Week 7

Adapted from Dr. Jeff Solka's BINF634 class slides

Outline

- Installation
- Variables
- Match operators
- Substitution
- Transliteration
- String functions
  - length, reverse

How to install Perl on my PC?

First check if perl is already installed on your PC.

Open command line and type «perl -v»

If it says «This is perl X, version Y,…», the good news is you already have Perl on your PC!

For Windows & Mac OS users:

- Go to: http://www.activestate.com/activeperl/downloads and download the suitable (32 or 64 bit) Perl version
- Next->Next->Next->Install (do not change destination folder)
- Add Perl bin folder into System Variable «Path»: https://www.youtube.com/watch?v=z2C3ZrawHuY

Why Perl?

- Ease of Programming
  - Excellent pattern matching features
  - Good for gluing other programs together
  - Easy to learn (enough to get started)
- Rapid Prototyping
  - Few lines of code needed for many problems
  - One-liners

Variables

- Named location that stores a value
- The types of Perl variables are indicated by the initial symbol:

`$var` stores a scalar (a single string or number)

```
$x = 10;
$s = "ATTGCGT";
$x = 3.1417;
```

`@var` stores an array (a list of values)

```
@a = (10, 20, 30);
@a = (100, $x, "Jones", $s);
print "@a\n";  # prints "100 3.1417 Jones ATTGCGT"
```

`%var` stores a hash (associative array)

```
%ages = ( John => 30, Mary => 22, Lakshmi => 27 );
print $ages{"Mary"}, "\n";   # prints 22
```

Declaring Variables

`use strict;`

- Putting use strict; at the top of your programs will tell perl to slap your hands with a fatal error whenever you break certain rules.
- Requires us to declare all variables
- Avoids creating variable by typos

- variables may be declaring using `my`, `our` or `local`
- for now, we only need to use `my`:

```
my $a;                  # value of $a is undef
my ($a, $b, $c);        # $a, $b, $c are all undef
my @array;          # value of @array is ()
```

- Can combine declaration and initialization:

```
my @array = qw/A list of words/;
my $a = "A string";
```

Scalar and List Context

- All operations in Perl are evaluated in either **scalar** or **list context**, and may behave differently depending on context

```
@array = ('one', 'two', 'three');
$a = @array;      # scalar context for assignment, return size
print $a;  # prints 3

($a) = @array;   # list context for assignment
print $a;        # prints 'one'

($a, $b) = @array;
print "$a, $b";        # prints 'one, two'
($a, $b, $c, $d) = @array;  # $d is undefined
```

In computer science a list is an ordered collection of values

String Operations

- Ways to concatenate strings

```
$DNA1 = "ATG";
$DNA2 = "CCC";
$DNA3 = $DNA1 . $DNA2;    # concatenation operator
$DNA3 = "$DNA1$DNA2";     # string interpolation
print "$DNA3";       # prints ATGCCC

$DNA3 = '$DNA1$DNA2';     # no string interpolation
print "$DNA3";       # prints $DNA1$DNA2
```

Arrays

- An array stores an ordered list of scalars:

- @gene_array = (‘EGF1’, ‘TFEC’, ‘CFTR’, ‘LOC1691’);
- print “@gene_array\n”;

- Output:
- EGF1 TFEC CFTR LOC1691

- # there’s more than one way to do it (see previous slide on declaring variables)
- @gene_array = qw/EGF1 TFEC CFTR LOC1691/;

The 'quote word' function `qw()` is used to generate a list of words. If takes a string such as:

```
    tempfile tempdir
```

and returns a quoted list:

```
    'tempfile', 'tempdir'
```

Arrays

- An array stores an ordered list of scalars:

  @a = (‘one’, ‘two’, ‘three’, ‘four’);

- The array is indexed by integers starting

  with 0: print “$a[1] $a[0] $a[3]\n”;

prints:

  two one four

- Notice: $a[i] is a scalar since we used the $ method of addressing the variable

Match Operator

```
$dna = "ATGCATTT";
if ($dna =~ /ATT/) {
    print "$dna contains ATT\n";
}
else {
    print "$dna doesn't contain ATT\n";
}
```

Output of code snippet:

```
ATGCATTT contains ATT
```

```
# matching a pattern
$dna = "ATGAAATTT";
$pattern = "GGG";
if ($dna =~ /$pattern/) {
    print "$dna contains $pattern\n";
}
else {
    print "$dna doesn't contain
    $pattern\n";
}
print "\n";
```

```
ATGAAATTT doesn't contain GGG
```

Substitution

```
print "substitution
example:\n";
$dna = "ATGCATTT";
print "Old DNA: $dna\n";
$dna =~ s/TGC/gggagc/;
print "New DNA:
$dna\n\n";
```

```
substitution example:
Old DNA: ATGCATTT
New DNA: AgggagcATT
```

```
print "single substitution:\n";
$dna = "ATGCATTT";
print "Old DNA: $dna\n";
$dna =~ s/T/t/;
print "New DNA: $dna\n\n";
```

```
single substitution:
Old DNA: ATGCATTT
New DNA: AtGCATTT
```

```
print "global substitution:\n";
$dna = "ATGCATTT";
print "Old DNA: $dna\n";
$dna =~ s/T/t/g;
print "New DNA: $dna\n\n";
```

```
global substitution:
Old DNA: ATGCATTT
New DNA: AtGCAttt
```

Substitution

```
print "removing white space\n";
$dna = "ATG CATTT   CGCATAG";
print "Old DNA: $dna\n";
$dna =~ s/\s//g;
print "New DNA: $dna\n\n";
```

```
removing white space
Old DNA: ATG CATTT   CGCATAG
New DNA: ATGCATTTCGCATAG
```

```
print "substitution ignoring
  case\n";
$dna = "ATGCAttT";
print "Old DNA: $dna\n";
$dna =~ s/T/U/gi;
print "New DNA: $dna\n\n";
```

```
substitution ignoring case
Old DNA: ATGCAttT
New DNA: AUGCAUUU
```

Computing complementary DNA (with bug)

```
#!/usr/bin/perl -w
# File: complement1
# Calculating the complement of a strand of DNA (with bug)

# The DNA
$strand1 = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
print "strand1: $strand1 \n";

# Copy strand1 into strand2
$strand2 = $strand1;

# Replace all bases by their complements: A->T, T->A, G->C, C->G
$strand2 =~ s/A/T/g;
$strand2 =~ s/T/A/g;
$strand2 =~ s/G/C/g;
$strand2 =~ s/C/G/g;

print "strand2: $strand2 \n";
exit;

% complement1
strand1: ACGGGAGGACGGGAAAATTACTACGGCATTAGC
strand2: AGGGGAGGAGGGGAAAAAAAGAAGGGGAAAAGG
```

Can you find the bug?

Transliteration Operator

```
print "transliteration operator\n";
$dna = "ATGCAttT";
print "Old DNA: $dna\n";
$dna =~ tr/T/U/;
print "New DNA: $dna\n\n";
```

```
transliteration operator
Old DNA: ATGCAttT
New DNA: AUGCAttU
```

```
print "tr on multiple
  characters\n";
$dna = "ATGCAttT";
print "Old DNA: $dna\n";
$dna =~ tr/Tt/Uu/;
print "New DNA: $dna\n\n";
```

```
tr on multiple characters
Old DNA: ATGCAttT
New DNA: AUGCAuuU
```

DNA Complement

```
print "DNA complement strand\n";
$dna = "ATGCAttT";
$complement = $dna;
$complement =~ tr/AaTtGgCc/TtAaCcGg/;
print "$dna\n";
print "$complement\n\n";
```

```
DNA complement strand
ATGCAttT
TACGTaaA
```

Computing complementary DNA (without bug)

```
#!/usr/bin/perl -w
# File: complement2
# Calculating the complement of a strand of DNA

# The DNA
$strand1 = 'ACGGGAGGACGGGAAAATTACTACGGCATTAGC';
print "strand1: $strand1\n";

# Copy strand1 into strand2
$strand2 = $strand1;

# Replace all bases by their complements: A->T, T->A, G->C, C->G
# tr replaces each char in first part with char in second part
$strand2 =~ tr/ATGC/TACG/;

print "strand2: $strand2 \n";
exit;

% complement2
strand1: ACGGGAGGACGGGAAAATTACTACGGCATTAGC
strand2: TGCCCTCCTGCCCTTTTAATGATGCCGTAATCG
```

How have we eliminated the bug?

Length Function

```
print "length function\n";
$dna = "ATGCAttT";
$size = length($dna);
print "DNA $dna has length $size\n\n";
```

```
length function
DNA ATGCAttT has length 8
```

Reverse function

```
print "reverse function\n";
$dna = "ATGCAttT";
$reverse_dna = reverse($dna);
print "DNA: $dna\n";
print "Reverse DNA:
  $reverse_dna\n\n";

reverse function
DNA: ATGCAttT
Reverse DNA: TttACGTA
```

```
print "reverse complement\n";
$dna = "ATGCAttT";
$rev_comp = reverse($dna);
$rev_comp =~
  tr/AaTtGgCc/TtAaCcGg/;
print "$dna\n";
print "$rev_comp\n\n";

reverse complement
ATGCAttT
AaaTGCAT
```