Perl Programming Language:

Essential Scripting Basics for the 13CFLUX2 Advanced Course

This document is a quick introduction to the Perl language. Perl has many features, but this document is about the basics that are necessary to follow the 13CFLUX2 exercises. The coverage of the text is pretty quick, intended for people with a bit of programming experience. In the final chapter four examples are given. You should be able to write scripts for 13CFLUX2 if you understand these examples. Of cause, there are countless excellent textbooks or electronic sources about the Perl language available, e.g. http://www.perltutorial.org. Visit the Comprehensive Perl Archive Network (CPAN), http://www.cpan.org/, or http://www.perl.com/ to find a rich source Perl material available.

Parts of these text where just taken from internet sources specified at the end of the manuscript.

1. Introduction: What is Perl?

Perl is a free, open source programming language created by Larry Wall in the 90'ies. Perl stands for Practical Extraction and Reporting Language. Perl aims for adjectives like "practical" and "quick" and not so much words like "structured" or "elegant". Perl is probably best known for text processing -- dealing with files, strings, and regular expressions. Perl's quick, informal style makes it attractive for all sorts of little programs. That's the reason why we choose it in the context of this course.

1.1. Running Perl

A Perl program is just a text file. You edit the text of your Perl program, and the Perl interpreter reads that text file directly to "run" it. On Unix, the Perl interpreter is called "perl" and you run a Perl program by running the Perl interpreter and telling it which file contains your Perl program...

```
> perl [perlscriptname.pl] [ARGV1] [ARGV2] … [ARGVN]
```

The interpreter makes one pass of the file to analyze it and if there are no syntax or other obvious errors, the interpreter runs the Perl code. There is no "main" function -- the interpreter just executes the statements in the file starting at the top.

Every Perl script starts with the following first line (giving a hint to Linux to use the Perl interpreter to execute the code within this file):

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

By default, the Perl compiler does not warn about possibly erroneous code. Thus, you need to be careful to keep local and global variables straight. There are at least two tools to help Perl programmers write maintainable code: the strict pragma and the warnings pragma." `strict` and `warning` are probably the two most commonly used Perl pragmas, and are frequently used to catch "unsafe code." When Perl is set up to use these pragmas, the Perl compiler will check for, issue warnings against, and disallow certain programming constructs and techniques. In Perl (5.6.0 or later), pragmas are set up with the use command:

```
use strict;
use warnings;
```

The `strict` pragma checks for unsafe programming constructs. `strict` forces a programmer to declare all variables as package or lexically scoped variables. The programmer also needs to use quotes around all strings.

The `warning` pragma sends warnings when the Perl compiler detects a possible typographical error and looks for potential problems. There are a number of possible warnings (check the man pages or ActiveState's perldiag document page), but warnings mainly look for the most common syntax mistakes and common scripting bugs.

For short, without the use of these both pragmas, you will inevitably waste time debugging some trivial variable name mixup or syntax error.

Besides, at the end of every command line beside the first line a semicolon (;) is accepted.

2. Comments

Comments begin with a "#" and extend to the end of the line. The line is then ignored by the Perl interpreter and can be used to describe certain parts of code.

For instance:

```
# this is a comment
my $this_is_a_variable_definition = 'some value'; # here is another comment
```

3. Important Data Structures

There are three primary data structures in Perl: the *scalar*, the *array*, and the *associative array*, or short "hash". The latter data structures are not in focus of this course.

- Scalar variable:
  `(my) $scalar`
  declaration/name of a scalar variable, a single element (**number or string**), with name scalar

- Array/ List:
  `(my) @array`
  declaration/name of an array with name array

- Hash:
  `(my) %hash`
  declaration/name of an hash with name hash

If you first define either a scalar, array, or hash, you can use the `my` keyword to denote that it belongs to a particular scope in which it is defined. This will prevent you from many bugs in the future. It is not mandatory to use the `my` parameter, expecting `strict` pragma is active. All variables, whether they are a scalar, array, hash, or other type, are CASE-SENSITIVE, meaning that `$myvariable` and `$MYVARIABLE` are treated as two different variables.

3.1 Scalar variables

Scalar data is the one of the most basic and simplest data structure in Perl. Scalar data can be number or string. In Perl, string and number can be used nearly interchangeable. Scalar variable is used to hold scalar data. Scalar variable starts with dollar sign ($) followed by the Perl identifier. Perl identifier can contain alphanumeric characters and underscores. It is not allowed to start with a digit.

Example:

```
$var = 2; # scalar variable $var set to the number 2
```

3.1.1 Number

Perl uses double-precision floating point values for calculation. Perl internally cheats integer as floating-point value. I.e., if Perl has a number or other type when it wants a string, it just

silently converts the value to a string and continues. It works the other way too – a string such as "42" will evaluate to the integer 42 in an integer context. Perl uses a minus sign (-) to define negative number. Here is the code snippet to demonstrate scalar variable $x which holds the number 3.14 in Perl etc.:

```
#floating-point values
my $x = 3.14;
$y = -2.78;
#integer values
$a = 1000;
$b = -2000;
```

Perl also accepts string literal as a number for example:

```
$s = "2000"; # similar to $s = 2000;
```

In this case $s can be used as a number for calculation even though it is a string.

3.1.2 String

Perl defines a string as a sequence of characters. The shortest string contains no character or null string. The longest string can contain unlimited characters which is only limited to available memory of your computer. Strings constants are enclosed within double quotes (") or in single quotes ('). Strings in double quotes are treated specially -- special directives like \n (newline) and \x20 (hex 20) are expanded. More importantly, a variable, such as $x, inside a double quoted string is evaluated at run-time and the result is pasted into the string. This evaluation of variables into strings is called *interpolation* and it's a great Perl feature. Single quoted (') strings suppress all the special evaluation -- they do not evaluate \n or $x, and they may contain newlines.

For example:

```
$str = "this is a string in Perl"
$str2 = 'this is also a string in perl'
$filename = "willi.txt";
$a = "Could not open the file $filename."; # $fname evaluated
$b = 'Could not open the file $filename.'; # single quotes (') do no
special evaluation
# $a is now "Could not open the file willi.txt."
# $b is now "Could not open the file $filename."
```

3.1.3 Assigned Arguments

Assigned arguments are also scalar variables and can be addressed by:

```
$ARGV[0], $ARGV[1],…
```

3.1.4 Operations on scalar variables

Perl uses arithmetic operators like other languages as C, C++ or Java. Here is the code snippet to demonstrate operations on numerical scalar variables:

```
$x = 5 + 9; # Add 5 and 9, and then store the result in $x
$x = 30 - 4;# Subtract 4 from 30 and then store the result in $x
```

```
$x = 3 * 7; # Multiply 3 and 7 and then store the result in $x
$x = 6 / 2; # Divide 6 by 2
$x = 2 ** 8;# two to the power of 8
$x = 3 % 2; # Remainder of 3 divided by 2
$y = ++$x;  # Increase  $x by 1 and store $x in $y
$y = $x++;  # Store $x in $y then increase  $x by 1
$y = --$x;  # Decrease $x  by 1 and then store $x in $y
$y = $x--;  # Store $x in $y then decrease $x by 1
$x = $y;    # Assign $y to $x
$x += $y;   # Add $y to $x
$x -= $y;   # Subtract $y from $x
$x .= $y;   # Append $y onto $x
```

The dot operator (.) concatenates two strings.

```
$x = 3;
$c = "he ";
$s = $c x $x;
$b = "bye"; # $c repeated $x times
print $s . "\n"; #print s and start a new line
# similar to
print "$s\n";
$a = $s . $b; # Concatenate $s and $b
print $a;
```

3.2 Array -- @

Array constants are specified using parenthesis ( ) and the elements are separated with commas. Perl arrays are like lists or collections in other languages since they can grow and shrink, but in Perl they are just called "arrays". Array variable names begin with the at-sign (@). Unlike C, the assignment operator (=) works for arrays -- an independent copy of the array and its elements is made. Arrays may not contain other arrays as elements. Arrays work best if they just contain scalars (strings and numbers). The elements in an array do not all need to be the same type.
Example:

```
@str_array= ("Perl",1,2,5,"array");
```

Square brackets [ ] are used to refer to elements, so $a[6] is the element at index 6 in the array @a. Array indices start at zero (0).

```
$str_array[1];
```

Notice that for accessing an array element the dollar sign ($) is used instead of at sign

You can add or remove elements to/ from an array. Here is a list of functions that allows you to do common operations on an array:

`push(@array,$element)`	add $element to the end of an array @array
`pop(@array)`	remove the last element of an array @array and returns it.
`unshift(@array,$element)`	add $element to the start of an array @array

`shift(@array)` remove the first element from an array @array and returns it.

Be careful, Perl arrays are not bounds checked. If code attempts to read an element outside the array size, `undef` is returned. If code writes outside the array size, the array grows automatically to be big enough.

4 Loop and term/ condition structures

4.1 If Control structure

Perl's control syntax looks like C's control syntax. Blocks of statements are surrounded by curly braces `{ }`. Statements are terminated with semicolons (`;`). The parenthesis and curly braces are **required** in `if`/ `while`/ `for` forms.

```
{ #statements here
  { # nested block
    # statements here
  }
}
```

There is not a distinct "boolean" type, and there are no "true" or "false" keywords in the language. Instead, the empty string, the empty array, the number 0 and `undef` all evaluate to false, and everything else is true. The logical operators &&, ||, ! work as in C. There are also keyword equivalents (and, or, not) which are almost the same, but have lower precedence.

`if` control structure is used to execute a block of code based on a condition. The syntax of if control structure is given below:

```
if(condition){
   statements;
}
```

If the condition is true the “statements” inside the block will be executed, for example:

```
$x = 10;
$y = 10;
if($x == $y){
  print "$x is equal to $y";
}
```

In the line 1 and 2 we define two variables `$x` and `$y` with their values “10”. In line 4 we use an `if` statement to print a message if `$x` is equal `$y`. The message is only printed if the expression `$x == $y` is evaluated as “ true”. In Perl, everything is true except the number zero (0), empty strings, empty arrays, and `undef`.

If you need an alternative choice, Perl provides if-else control structure:

```
if(condition){
   if-statements;
}
else{
   else-statements;
```

```
}
```

If the condition is false the else-statements will be executed. Here is the code example:

```
$x = 5;
$y = 10;
if($x == $y){
  print "x is equal to y";
}
else{
  print "x is not equal to y";
}
```

And another source code example:

```
$x = 5;
$y = 10;
if($x > $y){
  print "x is greater than y";
}
else if ($x < $y){
  print "x is less than y";
}
else{
  print "x is equal to y";
}
```

Operator	Example	Defined	Result
==,eq	5 == 5 5 eq 5	Test: Is 5 equal to 5?	True
!=,ne	7 != 2 7 ne 2	Test: Is 7 not equal to 2?	True
<,lt	7 < 4 7 lt 4	Test: Is 7 less than 4?	False
>,gt	7 > 4 7 gt 4	Test: Is 7 greater than 4?	True
<=,le	7 <= 11 7 le 11	Test: Is 7 less than or equal to 11?	True
>=,ge	7 >= 11 7 ge 11	Test: Is 7 greater than or equal to 11?	False

Operator	Defined	Example
&&,and	Associates two variables using AND	if (($x && $y) == 5)...

||,or	Associates two variables using OR	if (($x || $y) == 5)...

Please note that you must use each different operator depending on whether or not you are comparing strings or numbers. In the table above, the black operators are for numbers and the red ones are for strings. The “greater” and “smaller than” operators refer to the position of the string in the alphabet.

4.2 For loop statement

In order to run a block of code iteratively, you use Perl`s `for` statement. It is useful for running a piece of code in a specific number of times. The following illustrates Perl `for` statement syntax:

```
for(initialization; test; increment){
   statements;
}
```

There are three elements in the `for` statement - initialization, test and increment - are separated by semicolons. Perl does the following sequential actions:

- Step 1. The initialization: “initialization” expression is evaluated- you can initialize a counter variable here.
- Step 2. The test expression is evaluated. If it is true, the block “statements” will be executed.
- Step 3. After the block was executed, the increment is performed and test is evaluated again. The process goes to step 2 until the test expression is false.
  If the test is never false, you encounter with an indefinite loop.

Here is a code snippet to print a message 10 times.

```
for($counter = 1; $counter <= 10; $counter++){
    print "for loop #$counter\n";
}
```

First the `$counter` is initialized. Next Perl checks if `$counter` is less than or equals ten or not. In this case, `$counter` is 1 so Perl executes the code inside block and increase the `$counter` by 1. The process takes place until the `$counter` variable is equal to ten, therefor the code block was executed 10 times.

5 Useful Functions/ Commands

5.1 Print Function

The print function prints string literals on standard-out by placing them directly following the print keyword, as follows:

```
print "Hello World!\n Welcome";
```

stdout:

```
Hello World
Welcome
```

Variables can be included in the output by placing them directly after the print statement or within double-quoted strings. (Variables within single-quoted strings will not be replaced.) The following example declares one variable called `$age` with a numeric value, another called message with a string value, then prints them:

Example1:

```
my $age = 22;
my $message = 'How old are you?';
print "Hi. I am $age. $message";
```

stdout:

```
Hi. I am 22. How old are you?
```

Example2:

```
my $message =’I am 22. How old are you?';
print $message
```

stdout:

```
I am 22. How old are you?
```

Example3:

```
my $message =’How old are you?';
print ”I am 22.”,$message
```

stdout:

```
I am 22. How old are you?
```

Whole lists/arrays can be included like in the following examples:

Example1:

```
my @int=(1,3,4,5)
print "That are the numbers\n:@int 6";
```

stdout:

```
That are the numbers:
1 3 4 5 6
```

Example2:

```
my @int=(1,3,4,5)
print "That are the numbers:\n",@int,"6";
```

stdout:

```
That are the numbers:
1 3 4 5 6
```

5.2 System Function

Both Perl's `exec()` function and `system()` function execute a system shell command.

```
exec(PROGRAM);
```

```
my $result = system("PROGRAM");
```

The big difference is that `system()` creates a fork process and waits to see if the command succeeds or fails - returning a value. `exec()` does not return anything, it simply executes the command.

Example1:

```
my $result =system("echo hallo");
```

stdout:

```
hallo
```

Example2:

```
my $result =system("echo $ARGV[0]");  # call: perl perlscript hallo2
```

stdout:

```
hallo2
```

If system returns a value of -1, it indicates a failure to start the program or an error of the system call. If the system's failure, you can check like this:

```
my $result =system("echo $ARGV[0]") ==0 or die "system @args failed: $?";
```

6 Examples

6.1 Script 1

```
#!/usr/bin/perl
use strict;
use warnings;
my $scalar1=1;
my $scalar2='The list contains the numbers';
my @list1=(1,3,5,2);
print "$scalar2 : @list1\n";
print "This is the list without the first argument with the
number $scalar1:\n $list1[1] $list1[2] $list1[3]\n"
```

output:

```
The list contains the numbers : 1 3 5 2
This is the list without the first argument with the number 1:
 3 5 2
```

6.2 Script 2

```
#!/usr/bin/perl
use strict;
use warnings;
my $scalar1=1;
my $scalar2='The list contains the numbers';
my @list1=(1,3,5,2);
my $program_value1;
my $program_value2;
$program_value1=system("echo $scalar2 : @list1\n");
$program_value2=system("echo This is the list without the
                    first argument with the number $scalar1:
                    $list1[1] $list1[2] $list1[3]");
```

output:

```
The list contains the numbers : 1 3 5 2
This is the list without
                  the first argument with the number 1: 3 5 2
```

6.3 Script 3

```
#!/usr/bin/perl
use strict;
use warnings;
my $scalar1=1;
my @list1=(1,3,5,2);
my $program_value1;
for(my $i=0; $i<=3; $i++)
{
   my $j=$i+1;
   $program_value1=system("echo the $j. number of the list is
                                                    : $list1[$i]");
}
```

output:

```
the 1. number of the list is : 1
the 2. number of the list is : 3
the 3. number of the list is : 5
the 4. number of the list is : 2
```

6.4 Script 4

```
#!/usr/bin/perl
use strict;
use warnings;
my $scalar1=1;
my $scalar2='number of the list is:';
my @list1=(1,3,5,2);
my @list2=(11, 15, 21, 86);
my $program_value1;
my $program_value2;
if ($ARGV[0]==1)
        {
        for(my $i=0; $i<=3; $i++)
                {
                my $j=$i+1;
                $program_value1=system("echo the $j.$scalar2
                                                $list1[$i]");
                }
        }
elsif ($ARGV[0]==2)
        {
        for(my $f=0; $f<=3; $f++)
                {
                my $j2=$f+1;
                $program_value2=system("echo the $j2.$scalar2
                                                $list2[$f]");
                }
        }
```

output:

```
the 1. number of the list is : 11
the 2. number of the list is : 15
the 3. number of the list is : 21
the 4. number of the list is : 86
```

7. References

http://www.perltutorial.org
http://www.developer.com/lang/perl/article.php/1478301/Perl-Strict-Warnings-and-Taint.htm
http://www.expertwebinstalls.com/cgi_tutorial/basic_perl_syntax_guide_variables.html
http://www.tizag.com/perlT/perloperators.php)
http://www.ehow.com/how_2095001_use-print-function-perl.html
http://perl.about.com/od/programmingperl/qt/perlexecsystem.htm