Computing Languages for Bioinformatics: Perl

Giuseppe Agapito, Magna Græcia University, Catanzaro, Italy

Introduction

Perl is a general-purpose scripting language introduced by Larry Wall in 1987 (Wall *et al.*, 1994, 2000). Perl was developed to connect different languages and tools together by making compatible the various data format between them. The reasons to spur Wall to create Perl have been, to gather together all the best features of C, C++, Lisp, awk, Smalltalk 80, Pascal and Unix Shell languages without their disadvantages. Perl became very popular as server-side script language, but with the time extended its application domain from system administrator tasks, managing databases, as well as object-oriented programming, finance, bioinformatics, and Graphical User Interface (GUI) programming. Perl is not an acronym, although we can refer to PERL as "Practical Extraction and Reporting Language." The language used to develop Perl is C, Perl is a cross-platform language, and it is available to download under the General Public Licence (GNU). Perl is available to download at the following website: www.perl.org. At the time of writing, the current stable version of Perl is the 5.24.1. The major strength points of Perl are the following: (i) Perl is easy to learn and use, as it was designed to be easy to understand especially for humans rather than for computers. (ii) Perl is portable in the sense that, it is possible to run a script wrote in Windows operating system on several different other operating systems without change any line of code. Perl is a partially interpreted language since existing a compilation step through that a Perl script goes through before its execution. Before to execute a Perl script, is mandatory to compile the script that is translated in bytecode and finally, Perl interprets the bytecode. In Perl the compilation includes many of the same optimization steps like in Java, C, and C++ program, that is the elimination of unreachable code, replacing of constant expressions by their values, linking library and the built-in functions, and so on. Another characteristic of Perl is that variables do not have an intrinsic type in the sense that, conversely from languages such as Java, C or C++, a variable can be declared without any particular type. In this way, a variable previously used to store an integer can next be used to contain String or Double. Moreover, a variable can also contain an undefined value, by assigning to it the special object *undef*. The *undef* keyword in Perl's is the equivalent of the *null* object in Object Oriented Languages. A faster way to obtain further information regarding the Perl language features is to consult Perl's online documentation, commonly referred to as *perldoc*.

Perl is still broadly used for its original purpose: working like mediator among different tools, making data coming from one software in a format compatible with the format expected by the other tool. Going from processing and summarizing system logs (Newville, 2001), through manipulating databases (Wishart *et al.*, 2007), reformatting text files (Letunic and Bork, 2011), and simple search-and-replace operations, as well as in comparative studies (Warren *et al.*, 2010), life sciences data analysis (Ravel, 2001). handling data from the Human Genome Project as reported in Stein *et al.* (2002), Stein (1996), managing bioinformatics data see work of Arakawa *et al.* (2003), Lim and Zhang (1999) and all the tasks that require massive amounts of data manipulation.

Background

Perl is a language primarily intended to be used from command-line interface, shell or terminal because it was developed as a server-side scripting language (Wall *et al.*, 1994; Hall and Schwartz, 1998; Ousterhout, 1998). To take advantages of all the power of Perl, programmers are to know how to deal with a terminal interface. A terminal usually is a black/withe screen displaying the prompt that looks like: $, %, C:\>. After the prompt symbol, there is a flashing underscore meaning that the terminal is ready to get command. By using the terminal, it is possible to consult the Perl documentation by typing the command *perldoc*. From a terminal window type "*perldoc-h*" (the -h option prints more help) as conveyed in **Fig. 1**.

The *perldoc* command allows programmers to access to all the information about a particular function, including the implementation code. To get information about the *rename* function, the programmer has to type in the terminal: "*perldoc-f rename*" allowing to see the description and code of the rename function. As well as *perldoc* allows programmers to search for all the question-answer entries in the Perl FAQs for which the questions contain a particular keyword, for example, looking for perldoc-q substr" in this case the command allow to obtain more detailed information about *substr* function. The terminal is also used to write and execute Perl programs. Before to run Perl programs it is necessary to write them, by using an editor. Each operating system, Unix, OS X, Windows, and Linux, comes with several different text editors. Thus each programmer is free to use its favorite editor.

Install Perl on Your Machine

Perl has been developed to be used on many platforms. It will almost certainly build and run any UNIX-like systems such as Linux, Solaris, FreeBSD. Most other current operating systems are supported: Windows, OS/2, Apple Mac OS, and so on. Programmers can get the source release and/or the Binary distributions of Perl at the following web address https://www.perl.org/get.html.

```
os-x:~ mac$ perldoc -h
perldoc [options] PageName|ModuleName|ProgramName|URL...
perldoc [options] -f BuiltinFunction
perldoc [options] -q FAQRegex
perldoc [options] -v PerlVariable

Options:
    -h   Display this help message
    -V   Report version
    -r   Recursive search (slow)
    -i   Ignore case
    -t   Display pod using pod2text instead of Pod::Man and groff
             (-t is the default on win32 unless -n is specified)
    -u   Display unformatted pod text
    -m   Display module's file in its entirety
    -n   Specify replacement for groff
    -l   Display the module's file name
    -F   Arguments are file names, not modules
    -D   Verbosely describe what's going on
    -T   Send output to STDOUT without any pager
    -d output_filename_to_send_to
    -o output_format_name
    -M FormatterModuleNameToUse
    -w formatter_option:option_value
    -L translation_code   Choose doc translation (if any)
    -X   Use index if present (looks for pod.idx at /System/Library/Perl/5.18/darwin-thread-multi-2le
vel)
    -q   Search the text of questions (not answers) in perlfaq[1-9]
    -f   Search Perl built-in functions
    -v   Search predefined Perl variables
```

Fig. 1 Using terminal to display the Perl documentation by using perldoc command.

- **Install Perl on Windows:** Make sure you do not have any version of Perl already installed. If you do uninstall Perl be sure if you still have a folder in C:\Strawberry to delete it Download the Strawberry Perl version 5.12.3 from http://strawberryperl.com. Reboot your machine, after go to your start menu, then click the "*Perl command*" link to verify that the installation worked type: *perl-v*.
- **Install Perl on Unix/Linux:** Install a compiler (if not yet installed on your machine), such as *gcc* through your system package management (e.g., apt, yum). Open a Terminal and copy and paste the command "*curl-L* http://xrl.us/installperlnix | *bash*" then press *return key* to confirm.
- **Install Perl on OSX:** First, install "Command Line Tools for Xcode," directly through Xcode, or through the Apple Developer Downloads (free registration required). Xcode can also be installed through the App Store application. Launch the Terminal Applications, copy and paste the command "*curl-L* http://xrl.us/installperlnix | *bash*" then press *return key* to confirm.

Write and Execute Perl Program by Using Interactive Development Environment

Despite Perl was be intended to be used from terminal command line, now are available several tools that make it possible to write and run Perl program by using Interactive Development Environment (IDE). In Windows there are a plenty of IDE editors the most used are *Notepad* ++ , *Padre* and *Notepad* (Plain Text Editor) the link to download are available at the following web address https://learn.perl.org/installing/windows_tools.html. In Unix/Linux the most used IDE editor are *vim*, *emacs* and *Padre* you can get more information where download these editor to the following web address https://learn.perl.org/installing/unix_linux_tools.html. In Mac OSX the most used IDE editor are *vim*, *emacs*, TextEdit (Plain Text Editor), TextMate (Commercial) and *Padre*. More information about where to download these editor can get to the following web address https://learn.perl.org/installing/osx_tools.html.

Finally, users can use the NetBeans IDE wrote in Java making NetBeans platform independent that is, can run on each operating system in which Java is installed. NetBeans let users to quickly and easily develop Java desktop, mobile, and web applications, as well as HTML5, JavaScript, PHP, and CSS. The IDE also provides an excellent set of tools for PHP and C/C++ including Perl. They are free and open source and has a large community of users and developers around the world. To use Perl in NetBeans IDE is necessary to download and install the *Perl On NetBeans – plugin* from the NetBeans *Plugin Portal*. The *Perl On NetBeans – plugin* requires the following steps to be installed on your system for the IDE to work correctly:

- Perl v5.12 or greater installed on your machine;
- Java Runtime Environment 7 or higher installed on your computer;
- NetBeans 8 or higher installed on your computer;
- The Perl and Java binaries (in Windows system) should be available in the PATH variable.

After you match the following system requirements, the installation of *Perl On NetBeans* can be summarized in the following steps:

- Download the *Perl On NetBeans* plugin form the web site "http://plugins.netbeans.org/plugin/36183/perl-on-netbeans".
- In NetBeans 8 select *Tool* from the menu bar and select *Plugins*, showing the *Plugins Manager* window;
- In the *Plugins Manager* window select the *Downloaded* tab and click on the button "*Add Plugins …*" will show the file system navigation window, to locate the *Perl On NetBeans* file (download previously);
- Select the file to add to NetBeans 8.

Write and Execute a Perl Program by Using the Terminal

Assuming that Perl is installed on the machine (if not it is mandatory to install Perl by following the instruction provided in the Perl web site (www.perl.org)), the next step is to set up a directory for all the codes where to save all the Perl programs. A Perl program will look like as depicted in **Fig. 2**.

Let's look more in detail line by line the program presented in **Fig. 2**.

The first line "*#!/usr/bin/perl*" is called *interpreter directive* (also called "*shebang*", specifying to the operating system the suitable interpreter to execute the script. Perl treats all lines starting with # as a comment, ignoring it. However, the concatenation between "*#*" and "!" at the start of the first line tells the operating system that it is an executable file, and compatible with perl, which is located at the "*/usr/bin/*" directory. The second line "*usewarnings;*" activates warnings. The activation of warnings is useful because recall to the interpreter to highlight to the programmers possible mistakes that otherwise will be not visualized. As an example, suppose that we made the following error (print "Perl is $Awesome !!!") in the line 3 of the code presented in **Fig. 2**. Commenting the "use warning" line we get the following result in output (let see **Fig. 3**), without having any clue on why in the output misses the string "Awesome".

Instead, enabling the warnings visualization the perl interpreter give in out-put the message conveyed in **Fig. 4**. The interpreter informs the programmer that is using an uninitialized variable called "$Awesome" (in Perl the variables are preceded by the $ symbol).

Fundamentals

Perl is a programming language with which it is possible to guide the computer to solve problems. Problem-solving is related to data elaboration, to elaborate data it needs to use a language that can be understood by the machine. Programming

```
#!/usr/bin/perl

use warnings;

print "Perl is Awesome !!!\n";
~
~
"firstPerlProgram.pl" 5L, 63C
```

Fig. 2 A simple Perl program wrote by using vi editor.

```
[os-x:~ mac$ ./firstPerlProgram.pl ]
 Perl is  !!!
 os-x:~ mac$
```

Fig. 3 The output when we warnings are disabled.

```
[os-x:~ mac$ ./firstPerlProgram.pl                                                    ]
 Name "main::Awesome" used only once: possible typo at ./firstPerlProgram.pl line 5.
 Use of uninitialized value $Awesome in concatenation (.) or string at ./firstPerlProgram.pl line 5.
 Perl is  !!!
 os-x:~ mac$
```

Fig. 4 The output when warnings are enabled.

languages such as Perl provide problem-solving capabilities to the computer. Perl uses statements often grouped together into blocks, that are easy to write for humans as well as are easy to be understood by machines. A Perl statement tells the computer how to deal with data, ending with a semicolon ";". To gather together any number of statements it is necessary to surround statements by curly braces {…}, that in Perl is called *block*, here's an example: "{print "Hello Perl.\n"; print "That's a block."}", this statement prints on video the message Hello Perl and That's a block, on different rows (without quotes). Statements are not enough to elaborate data, because the machine needs to store data somewhere generally into the main memory, to deal with them when it is necessary during the whole elaboration process. The memory locations where program languages store data for simplicity are identified through variables. In Perl as well as in the other program languages, a variable is defined through a name. In particular, a variable name has to start with the symbols "$", and should not be a keyword of Perl language. For example, "$*var*" is a correct name for a variable instead, "$*do*" is not proper as variable name because can be confused with Perl's *do* keyword. The $ symbol before the name makes it possible to know that the $*var* is a *scalar* specifying that the variable can store a single value at the time. Whereas, variables starting with the symbol "@" can contain multiple values and are called *array* or *list*. A scalar variable can contain numbers that in Perl are classified in integer and floating-point numbers. Integers are whole numbers such as: "5, −9, 78" (without decimal part). Instead, floating points numbers has a decimal part, for example: "0.1, −0.12344" and so on. To put data into a variable, programmers have to use the assignment operator "=". An *array* or *list* is a variable that may holds zero or more primitive values. The elements stored in arrays and lists are numbered starting with zero, ranging from zero to the number of elements minus one. Code 3.1 conveys a simple example of using the array variable.

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
$@age=(25, 30, 44);
@names=("Joseph", "AnnaMary", "Bob");
$firstElem=$age[0];
print("$firstElem");
}
```

Code 3.1: An simple example to assign the first element of an array to a variable.

To modify the content of a variable is mandatory to use the assignment operator. For example, $*num*=12345, we defined a variable called "num" and assigned to it the value "12345", the content of num will successively modify assigning new value $*num*=1.3; for example. In addition to the numbers, Perl allows to variables to contain *strings*. A string is a series of characters surrounded by quotation marks such as "Hello World". Strings contain ASCII characters and escape sequences such as the \n of the example, and there is no limitation on the maximum number of characters composing a Perl string. Perl provides programmers mechanisms called 'escape sequences' as an alternative way of getting all the *UTF8* characters as well as the *ASCII* characters that are not on the keyboard. A short list of escape sequence is presented in **Table 1**.

There is another type of string obtained by using single-quotes: ''. The difference between single and double quotes is that no processing is done within single quoted strings, that is variable names inside double-quoted strings are replaced by their contents, whereas single-quoted strings treat them as ordinary text. To better explain the differences between single and double quotes consider Code 3.2 as example:

Table 1 Escape characters

Escape sequence	*Function*
\t	Tab
\n	New line
\b	Backspace
\a	Alarm

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
print '\t This is a single quoted string.\n';
print "\t This is a double quoted string.\n";
}
```

Code 3.2: Difference between the use of single and double quotes.

The differences between the double-quoted and the single-quoted string are that: the first one has its escape sequences processed, and the second one not. The output obtained is depicted in Code 3.3:

```
#Console Output
\t This is a single quoted string.\n
This is a double quoted string.
```

Code 3.3: The difference of output due to the use of single and double-quotes.

This operation is called escaping, or more commonly, *backwhacking* allows programmers to put special character such as backslash into a string as conveyed in Code 3.4, printing it on the screen in the correct format, let see Code 3.5.

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
print 'C:\WINNT\Profiles\'."\n";
print "C:\\WINNT\\Profiles\\\n";
}
```

Code 3.4: Combination of escaping character to print in output the special character \.

```
#Console Output
C:\WINNT\Profiles\
C:\WINNT\Profiles\
```

Code 3.5: Output result using single and double-quotes respectively.

Perl besides to allow users to define numbers and strings, it provides operators and functions to deal with numbers and strings.

Arithmetic Operators

The *arithmetic operators* comprise the basic mathematics operators like adding, subtracting, multiplying, dividing, exponentiation and so on. As appear for mathematics each operator comes with a precedence, which establishes the order in which Perl performs operations. Multiply and divide have a higher precedence than adding and subtract, and so they get performed first. To coerce Perl to perform operations with low priority first, it is mandatory to use brackets, for example, the following operation $1 + 2*3$ will

produce as a result 7. To obtain 9, as a result, it is necessary to rewrite the expression by using brackets in this way $(1 + 2)*3$. Other arithmetics operators are *exponentiation operator* ** and *module* %. Where module operator has the same precedence as multiple and divide, whereas exponentiation operator has higher precedence that multiple and divide, but lower precedence than minus operator.

Bitwise Operators

Bitwise operators work on bits since computer represent the complete information using bits. Bitwise operators perform bit by bit operations, from right to left. Where the rightmost bit is called the 'least significant bit,' and the leftmost is called the 'most significant bit'. Given two numbers 9 and 5 that in binary using 4 bits are expressed as $9 = 1001$, and $5 = 0101$ let see, which are the bitwise operators available in Perl. Bitwise operators include: the *and* operator and is written "&" in Perl. The "&" operator compares pairs of bits, as follows: if bits are both 1 the & gives 1 as a result, otherwise if one of the bits or both is equal to 0 the & gives 0 as a result. For example, the result of 9&5 is: $\$a\&\$b = 0001$. The *or* operator in Perl is "|", where 0|0 is always 0 whereas, 1|1 and 1|0 is always 1 (independently of the left-right operator values). The result of $9|5 = 1101$. To know if one bit or both bits are equals to 1 it is possible to use the *exclusive-or* operator $\wedge$, the result of $5 \wedge 9$ is 1100. Finally, by using the *not operator* "$\sim$" it is possible to replace the value from 1 to 0 and vice versa, for example, ~ 5 is 1010.

Equality Operators

Perl provides users operators able to compare equality of numbers and strings.

Comparing numbers

The equality operator $==$ checks if the value of two numerical operands are equal or not, if are equal the condition gives *true* as result *false* otherwise. In Perl *true* is represented as 1 and *false* as 0. The Inequality operator, $!=$, verifies if two operands are different, if left value and right value are different the condition becomes true ($5!=9$ gives as result *true*). The compare operator $<=>$ checks if two operands are equal or not providing as result -1, 0 or 1 if the left operand is numerically less than, equal to, or greater than the second operand. Finally, the operators $<$, $<=$, $>$ and $>=$. The $<$ operator give *true* if the left operand is less than the right operand (e.g., $5<9$ gives as a result true). The $<=$operator gives *true* if the left operand is less or equal than the right operand (e.g., $5<=5$ gives as result true). The $>$ operator give *true* if the left operand is greater than the right operand (e.g., $5>9$ gives as a result false). Finally, The $>=$ operator give *true* if the left operand is greater or equal than the right operand (e.g., $9>=5$ gives as a result true).

Comparing strings

To compare two strings in Perl is necessary to use the comparison operators "*cmp*". "*cmp*" compares the strings alphabetically. "*cmp*" returns -1, 0, or 1 depending on whether the left argument is less, equal to, or greater than the right argument. For example, "Bravo" comes after "Add", thus ("*Bravo*"*cmp*"*Add*") gives as result -1. To test whether one string is less than another, use "*lt* ". Greater than becomes "*gt* ", equal to is "*eq* ", and not equal becomes "*ne*". There are also the operators *greater than or equal to* referred to as "*ge*" and *less than or equal to* referred to as "*le*".

Logical operators

Logical operators make possible to evaluate the truth or falsehood of some statements at the time. The logical operators supported by Perl are *and* referred to as "&&". The "&&" operator evaluates the condition and if both the operands are *true* returns *true* as result, *false* otherwise. The "or" referred to as "$\|$" evaluates the condition, returning *true* if at least one of the operands is *true*, *false* otherwise. Not operand "!" is used to negate the logical state of the condition. As an alternative it is possible to use logical operators through the easier to read versions, *and*, *or*, and *not*.

Other operators

Other useful operators available in Perl are: string concatenation "." given two string $a = "abc" and $b = "def" $a.$b gives as result "abcdef". Repetition operator "x" gives in output the left operand repeated x-times, for example (print "ciao"x3), will print "ciaociaociao". Range operator "..", return a list of value starting from the left value to the right value included. For example (2.6), will return the following values (2,3,4,5,6). Finally, the auto-increment "++" and auto-decrement "$--$" operator, that increases and decreases integer value by one respectively.

Conditional Statement

The *if-else* statement is the fundamental control statement that allows Perl to make decisions executing statements conditionally. The simplest conditional statement in Perl has the form:

$$\texttt{if}(<condition>)\{<Statement\,1>;<Statement\ 2>;\ldots\};$$

The if-else statement has an associated expression and statement. If the expression evaluates to true, the interpreter executes the statement. If the expression evaluates to false the interpreter skips the statement. An example of using if statement is presented in Code 3.6.

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
if($a >= $b){
print ('The values of $a and $b is:').("$a");
}
```

Code 3.6: An example illustrating a basic use of the if statement.

An if statement can include an optional else keyword. In this form of the statement, the expression is evaluated, and, if it is true, the first statement is executed. Otherwise, the second statement (else) is executed. The more general conditional in Perl presents the form:

`if(<`*condition*`>){`*StatementsBlock*1`}; else{`*StatementsBlock*2`};`

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
if($a>$b){
print ('$a is greater than $b').("$a > $b");
 }else {
       print ($b is greater than $a').("$b > $a");
}
```

Code 3.7: An example of if else statement.

When you use nested if/else statements, some caution is required to ensure that the else clause goes with the appropriate if statement. Nesting more than one condition could be difficult to read. Thus Perl provides to programmers the if elsif statement, which presents an easier to read form:

if (< *condition*1 >) < *action* > **elsif** (< *condition*2 >) < *secondaction* >
elsif (< *condition*3 >) ... **else** <if all **elsif** fails>

Code 3.8: An example of if elsif statement:

Loops

The loops statement are the basic statement that allows Perl to perform repetitive actions. Perl programming language provides specific types of loop.

while loop that only executes the statement or group of statements only if the given condition is *true*. The general form of *while* loop is:

`while (<condition>){<block of statements>}`

Code 3.9 illustrates the of while loop.

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
use strict;
#initialisation of $a to 0
$a = 0;
# while loop execution
while($a<=9){
    printf "Value of a: $a \n";
    $a = $a + 1;
}
```

Code 3.9: An example of while loop that prints the numbers from 0 to 9.

until loop execute a statement or a group of statements till the given condition not becomes true. The general form of until loop is:

`until(<condition>){<block of statements>}`

Code 3.10 presents a simple use case of until loop.

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
use strict;
#initialisation of $a to 0
$a = 0;
# until loop execution
until($a==10){
    printf "Value of a: \$a \$\backslash n$";\newline
    $a++;
}
```

Code 3.10: An example of until loop that prints the numbers from 0 to 9.

do loop is very similar to the while loop, except that the loop expression is tested at the bottom of the loop rather than at the top. do ensures that the body of the loop is executed at least once. The syntax of do loop looks like:

`do{<block of statements>}while(<condition>)`

The for loop executes the statements in a block a determinate number of times. The for in is more general form is:

`for` (*init*; *condition*; *increment*){Block of statements; }

The for iterates on each element in an array as well as in a list as conveyed. As Code 3.11 shows the use of for loop to print the elements of an array.

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
use strict;
my @array = (1,2,4,3);
my $i;
# for loop
for $i (@array){
   printf "\n $i";
}
```

Code 3.11: An example of for loop that prints the values of an array.

It is possible to use foreach and for loop indistinctly on any type of list. It is worthy to note that the both loops create an alias, rather than a value. Thus, any changes made to the iterator variable, whether it be $ or one you supply, will be reflected in the original array. For instance:

```
#!/usr/bin/perl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM
use warnings;
use strict;
# create an array of ten elements
my @array = (1..10);
foreach (@array) {$_++;}
print "Array is now: @array\n";
```

Code 3.12: foreach loop that prints and modify the values of the array:

The code in Example 3.12 will change the contents of the array, as follows:

```
Array is now: 2 3 4 5 6 7 8 9 10 11
```

Fibonacci Sequence Example

In this section, will be introduced the algorithm wrote in Perl to compute the Fibonacci sequence. The Fibonacci sequence is a recursive formulation where each element is equal to the sum of the first two. This sequence owes its name to the Italian mathematician Fibonacci. The purpose of the sequence was to identify a mathematical law to describe the growth of a population of rabbits. The Code 3.13 shows the computation of the Fibonacci's sequence through Perl programming language.

```
#!/usr/bin/perl
# @File Fibonacci.pl
# @Author Perl
# @Created Mar 29, 2017 3:30:48 PM

use strict;
use warnings;

my $f1 = 0;
my $f2 = 1;

printf "\n How many numbers of the Fibonacci's sequence
 would you display ? \n";
#We use <STDIN> library to get the data from stdin here
my $n = <STDIN>;
   for (my $i=0;$i<$n;$i++){
      printf "%d\n", $f1;
    my $sum = $f1 + $f2;
      $f1 = $f2;
      $f2 = $sum;
   }
```

Code 3.13: A simple script example to compute the Fibonacci's Sequence.

Closing Remarks

Perl first appeared in 1987 as a scripting language for system administration, but thanks to it's very active community became a very powerful, flexibility and versatility programming language. The main strength of Perl's popularity is the Comprehensive Perl Archive Network (CPAN) library, that is very extensive and exhaustive collection of open source Perl code, ranging from Oracle to iPod to CSV and Excel file reader as well as thousands of pages of Perl's core documentation. Thus, Perl exploiting the knowledge and experience of the global Perl community, it provides help to everyone to write code, bug resolution and code maintenance. In summary, the key points of Perl are (i) management of Regular Expressions are natively handled through the regular expression engine. Regular expression engine is a built-in text process that, interpreting patterns and applying them to match or modify text, without requiring any additional module. (ii) The flexibility, Perl provides programmers only three basic variable types: Scalars, Arrays, and Hashes. That's it. Perl independently figures it out what kind of data developers are using (int, byte, string) avoiding memory leaks. Finally, (iii) the portability. Perl works well on several operating systems such as UNIX, Windows, Linux OSX, as well as on the web.

References

Arakawa, K., Mori, K., Ikeda, K., *et al.*, 2003. G-language genome analysis environment: A workbench for nucleotide sequence data mining. Bioinformatics 19 (2), 305–306.
Hall, J.N., Schwartz, R.L., 1998. Effective Perl Programming: Writing Better Programs With Perl. Addison-Wesley Longman Publishing Co., Inc.
Letunic, I., Bork, P., 2011. Interactive tree of life v2: Online annotation and display of phylogenetic trees made easy. Nucleic Acids Research. gkr201.

Lim, A., Zhang, L., 1999. Webphylip: A web interface to phylip. Bioinformatics 15 (12), 1068–1069.
Newville, M., 2001. Ifeffit: Interactive xafs analysis and feff fitting. Journal of Synchrotron Radiation 8 (2), 322–324.
Ousterhout, J.K., 1998. Scripting: Higher level programming for the 21st century. Computer 31 (3), 23–30.
Ravel, B., 2001. Atoms: Crystallography for the x-ray absorption spectroscopist. Journal of Synchrotron Radiation 8 (2), 314–316.
Stein, L., 1996. How perl saved the human genome project. Dr Dobb's Journal (July 2001).
Stein, L.D., Mungall, C., Shu, S., *et al*., 2002. The generic genome browser: A building block for a model organism system database. Genome Research 12 (10), 1599–1610.
Wall, L., Christiansen, T., Orwant, J., 2000. Programming Perl. O'Reilly Media, Inc.
Wall, L., *et al*., 1994. The perl programming language.
Warren, D.L., Glor, R.E., Turelli, M., 2010. Enmtools: A toolbox for comparative studies of environmental niche models. Ecography 33 (3), 607–611.
Wishart, D.S., Tzur, D., Knox, C., *et al*., 2007. Hmdb: The human metabolome database. Nucleic Acids Research 35 (suppl 1), D521–D526.