

# Perl language evaluation

Marco Slot

**Abstract** - This report evaluates the programming language Perl.

## Introduction

From the original Perl manual page [1]:

*“Perl is a interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It's also a good language for many system management tasks. The language is intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal).”*

## History of Perl

In 1987 Larry Wall released Perl version 1.000. He was unsatisfied with the low productivity of many programming languages in terms of the time it took to make the program.

Perl was one of the first Open Source projects and was quickly picked up by the community. In the following years a new version of Perl was released nearly every year until 1994 when Perl 5 was released, the version that is still being used today (that is with over 10 years of updates and patches).

## Use of Perl

Perl is renowned for its text processing capabilities. The language has many facilities for parsing and manipulating strings. It's also popular for system administration tasks, not in the least bit because it is available on practically every Unix based system.

Today Perl also enjoys great popularity as a server-side scripting language for dynamic websites. Due to the ease at which it can process (and thus verify) user input it is often considered a safer than its competitors. Perl programmers

like to call their language ‘The Duct Tape of the Internet’. This because Perl can ‘easily’ deal with ugly problems ([3]).

Numerous additional libraries are available. Perl is also frequently used for graphical and networking applications.

## Hello World

To continue the tradition started by Brian Kernighan and Dennis Ritchie in ‘The C Programming Language’ we will start with the famous Hello World example:

```
1 print "Hello World";
```

output: Hello world

Line 1 calls the native print command to print the string (delimited by “) to the standard output. Lines are closed with ; in Perl.

## Variables & functions

Variables in Perl are in principle very similar to other imperative languages.

```
2 $variable = "Hello ";
3 $variable.= "World";
4 print $variable;
```

output: Hello World

Besides regular variables Perl also has special variables. The variable \$. always contains the current line number of the program. The variable \$\_ is used for numerous purposes, for example it can contain the current element when looping through an array. Special variables exist for file handling, string handling, error handling and even regular function calls. We will see a few of these variables in this report.

## Functions

Functions in Perl are called subroutines. A subroutine can be defined by using the `sub` keyword followed by the name and curly brackets to delimit the content of the subroutine.

All subroutines accept 1 parameter: an array. This array does not have a name but is accessed in 2 ways: `@_` to get entire array and `$_[n]` to get element `n` from the array.

```
5 sub hw {
6     print $_[0];
7     return $_[1];
8 }
9 print
  hw("Hello ", "World", "Ignored");
```

output: Hello World

## Scope

Perl strives to have as few restrictions as possible. All variables in a Perl script are global by default (even in subroutines!). Nonetheless the programmer is able to define the scope if necessary.

The `my` operator limits the scope of a variable to the current block, for example a function, a for loop or the entire file. It is a good programming habit to declare the variable as close to its first use as possible. This way the garbage collector can get to work as soon as possible.

```
10 sub scope1 {
11     my $local = "Hello";
12     $global = "World";
13 }
14 print $global;
```

output:

## Garbage collection

Perl does simple garbage collection by counting references to data. It is impossible to turn off garbage collection or to let the programmer manage memory.

When the program reaches the end of a block the interpreter will see which

variables were declared locally. These will be marked for cleaning and if the variable contained a reference the reference counter of the data pointed to is lowered.

## Error handling in Perl

Much like C the primary way of dealing with errors in Perl is the return value. If a function can go wrong the return value has to be checked. This (lots of repetitive code) doesn't sound very Perl like, but naturally there is a way of quickly catching these errors using `||` or `or`.

Error handling is also supported by a special variable. In this case `!`. When used in a numeric context `!` holds the value of the error number (like `errno` in C), in a string context it holds the associated error string (from [4]).

```
15 open(FILE, "helloworld.txt") ||
    die("$!");
```

This line of code attempts to open the file `helloworld.txt`, but if this fails for whatever reason the `die` command is executed. This prints the parameter to the `STDERR` (on Unix) and stops the program. A lighter alternative to `die` is `warn`, which does not stop the program. The code above is equivalent of:

```
16 open(FILE, "helloworld.txt")
17 if(!$!) {
18     warn "$!";
19     exit 1;
20 }
```

A special technique for catching errors is provided by the `eval` command in Perl. If you want to prevent a fatal error from stopping the program then `eval` can be used to catch this error. `eval` does this by evaluating the code that is passed to it and once the evaluated code tries to exit the `eval` function simply returns like a regular function.

```
21 eval {
```

```

22     open(FILE, "helloworld.txt")
      || die("$!");
23 }
24 if($?) print $?;

```

The `$_` special variable is specific to the `eval` function. This contains the error message that was passed to `die`, which would normally be printed to the `STDERR`. The `eval` mechanism is in many ways similar to the try-catch mechanism found in languages like Java.

## Regular expressions

Perl and regular expressions go hand in hand. Perl programs without any regular expressions are rarely seen. Regular expressions can be used for several operations.

Often strings need to be verified, especially when coming from potentially unreliable users. Perl has a simple syntax for conditional matching of strings with regular expressions.

```

25 $string = "' OR 1=1 OR '";
26 if($string =~ /[a-z]*/) {
27     print $string; }

```

output:

In Perl two or three lines of code can eliminate significant security threats.

An even more powerful function of regular expressions is replacement. In this case the expression is matched to a part or multiple parts of the string and those parts are then replaced by another string.

Another type of replacement is translation in which the matched string is used to determine what the replacement is.

The sample below replaces the first word including the `'` with `Hello` and makes the string lowercase.

```

28 $string = "Wayne's World";
29 $string =~ s/['a-zA-Z]* /Hello
      /g;
30 $string =~ tr/[A-Z]/[a-z]/;
31 print $string;

```

output: hello world

The third functionality of regular expressions is string selection. Regular expressions can be used to dissect a string into parts. For example in Perl it is relatively simple to get the number behind the third word with a capital letter.

```

32 $s="The Small cat Has 4 tails";
33 if($s =~
      /([A-Z][a-z ]*){3}(\d*)/) {
34     print $2; }

```

output: 4

## Types

Perl has weak typing. Types of variables, parameters, return variables, etc. are not declared but checked at run-time.

Perl has 3 basic structures: Scalars, arrays and hash tables. Each of these 3 structures has its own syntax for declaration and usage.

```

35 $scalar = 2006;
36 @array  = ("September",
37           "April",
38           "December");
39 %hash    = ("March" => 31,
40           "April"  => 30,
41           "May"    => 31);
42 print "$hash{$array[1]} ";
43 print "$array[1] ";
44 print "$scalar ";

```

output: 30 April 2006

Most primitive types such as numbers, booleans, strings, regular expression, references, etc. are all scalar types. There is some polymorphism for scalar types, but it is somewhat limited. For example both numbers and strings can be inserted in another string, but they cannot be concatenated. Partly because inserting and concatenating have the same effect while inserting has a shorter syntax. Perl also have some very unique forms of polymorphism. The `++` operator can not only increment numbers, it can also

increment strings. `$a="az"; $a++;` evaluates to "ba".

Basic Perl doesn't have any complex types like C-structs, but when more advanced programs are required Perl does have object-oriented facilities.

## Object-oriented Perl programming

For some programming languages, the interesting part doesn't start until object-oriented programming is covered. For Perl this is not really the case. Perl was not developed to be object oriented and its facilities have more or less been built into the existing syntax rather than on top of it (Objective C) or around it (PHP).

Object orientation in Perl is done in a way which is not commonly seen but is very practical and therefore perfectly fits into Perl. All objects are associative arrays containing name-function and name-attribute pairs. Like most object-oriented languages objects aren't spoken to directly, but through references. Internally the type of the object is a scalar containing a reference to an associative array.

Perl uses packages to organize code, a package which contains constructors and methods for an object could be called a class.

A basic 'Class' looks like this:

```
45 package Animal;
46
47 sub new {
48     my $self = {
49         name => $_[1],
50     };
51
52     bless $self, $_[0];
53     return $self;
54 }
55
56 sub hello {
57     my $name = $_[0]->{name};
58     print "Hello $name";
59 }
```

The new routine is a constructor for an animal object. The constructing-magic

happens on line 52 which tells the Perl interpreter to retain the associative array called `$self` in memory (instead of letting the garbage collector destroy it), assign a reference to the associative array to the variable `$self` and remember its 'type' which was passed as the first parameter. Note that 'type' only means that function references in the associative array point to the package `Animal`.

The `hello` method is not aware that it is part of an object, nor is the object aware that there is an associated method. `hello` is just a function to which an object reference is passed as a parameter.

In another Perl program we can do:

```
60 using Animal;
61 $animal = Animal->new("Bello");
62 $animal->hello();
```

output: Hello Bello

The `->` is just a shortcut, it translates into `Animal::hello($animal, ...)`. Perl knows `hello` of `$animal` is in package `Animal` because it was blessed with that type.

### Inheritance

Perl also supports single and multiple inheritance. The programmer can't do this completely by himself, other special commands were introduced to support inheritance. The following piece of code makes `Dog` the equivalent of `Animal`.

```
63 package Dog;
64 use Animal;
65 our @ISA = qw(Animal);
```

Line 65 causes functions and variables of package `Animal` to be accessible through this package unless they are overridden.

This means a new constructor can now be written.

```
66 sub new {
67     my $class = $_[0];
68     my $self =
69         SUPER::new($class, $_[1]);
69     $self->{color} = $_[2];
```

```

70     bless $self, $class;
71     return $self;
72 }
73
74 sub hello {
75     my $name = $_[0]->{name};
76     my $dog   = $_[0]->{color};
77     SUPER::hello($me);
78     print " the $color dog.";
79 }

```

We now have extended the `Animal` class and overridden the `hello` method. The code sample below shows the Java-style polymorphism that has now been enabled. The method `hello` can be called for both an instance of `Animal` and an instance of `Dog`.

```

80 using Dog;
81 $animal =
    Dog->new("Bello", "brown");
82 $animal->hello();

```

output: Hello Bello the brown dog

This Perl programming does practically nothing and already it is far more complex than its Java equivalent. With multiple inheritance the complexity goes up even further. Programming object-oriented in Perl is not very productive and thus it goes against the very nature of Perl.

## Perl Poetry

Perl has been known to be so loose in its syntax that it is possible to write stories that are valid Perl. Usually these stories do very little when executed (but don't produce errors).

The first Perl poem to appear is also the most famous and perhaps the most vicious: *Black Perl* [5]. The poet is the creator of Perl itself, Larry Wall. This poem no longer parses in recent versions of Perl, but it was the start of the art of Perl poetry.

Below is the winner of a Perl haiku contest [6].

```

83 no less can I say;
84 require strict, close attention
85 while you ... write haiku

```

output:

## Perl obfuscation

Not only does Perl allow artistic syntax. It also allows dreadful syntax. Besides poetry contests, the Perl community has also had its share of obfuscation contests.

Unlike Perl poems, obfuscated Perl really has to be functional. In a contest organized in 2000 the task was to print "The Perl Journal" using a Perl program of at most 256 characters. The winning code (from [7]) is shown below:

```

86 #:: :-| :-| .-. :||-:: 0-| .-
   | :||-| .:|- .:|
87 open(Q,$0);while(<Q>){if(/^#(.*)
   )$/){for(split('-
   ', $1)){ $q=0;for(split){s/\|
88 /:./xg;s/:./g;$Q=$_?length:$
   _;$q+=$q?$Q:$Q*20;}}print
   chr($q);}}print"\n";
89 #.: :||-| .||-| :||-| :||-|
   ||-:: :||-| .:|

```

output: The Perl Journal

The program makes excellent use of Perl regular expressions and applies a smart trick by encoding the text in the source comments and subsequently reading the source file.

## A real-life example

Finally an example that has really been used for a computer science course.

```

90 #! /bin/perl -w
91
92 sub trim;
93 sub println;
94 sub contains;
95
96 @null = ("NIET VAN TOEPASSING",
   "NVT", "NIETS");
97
98 while($line = <STDIN>) {
99     $line =~ s/<.+?>/g;
100     @lineitems = split(/;/, $line);
101
102     foreach $item (@lineitems) {

```

```

103     $item = trim($item);
104
105     if(!contains($item, @null)) {
106         println $item;
107     }
108 }
109 }
110
111 sub trim {
112     my $string = shift;
113     $string =~ s/^\s+//;
114     $string =~ s/\s+$//;
115     return $string;
116 }
117 sub println {
118     print "$_[0]\n";
119 }
120 sub contains {
121     $needle = shift;
122     for(@_) {
123         if($needle eq $_) {
124             return 1;
125         }
126     }
127     return 0;
128 }

```

This Perl script is a typical example of what one would do with Perl. This is script that extracts information from an XML file in the least elegant, yet very effective way. In every line that is passed to this script to the STDIN the script removes anything delimited by < and > or better yet, all XML tags. The remaining string can be list a of names delimited by ‘;’, this list is split into an array and the elements are printed if they are not ‘null values’.

This script enjoys no other use than extracting words from a specific type of text file, this is somewhat typical for Perl scripts. The next step in this particular example was to sort the list and remove doubles. One could make an elaborate Perl script for this, but in Unix we can simply pass the results through `sort -u`. This is how Perl programming works, duct-taping things together for practical solutions.

## Tools

The code samples were created using vim on Solaris and tested using the perl interpreter on Unix (v5.6 on Solaris and v5.8 on Minix 3).

## Bibliography

### General

Larry Wall,  
 “Programming Perl, 3rd Edition”  
 ISBN: 0-596-00027-8

### References

- [1] <http://history.perl.org>
- [2] <http://www.perl.com>
- [3] <http://www.stonehenge.com/merlyn/UnixReview/col45.html>
- [4] <http://www.cs.cf.ac.uk/Dave/PERL/>
- [5] [http://en.wikipedia.org/wiki/Black\\_Perl](http://en.wikipedia.org/wiki/Black_Perl)
- [6] <http://aspn.activestate.com/ASPN/Perl/Haiku/InPerl>
- [7] <http://mysite.verizon.net/les.peters/id2.html>
- [8] [http://www.codeproject.com/perl/camel\\_poop.asp](http://www.codeproject.com/perl/camel_poop.asp)
- [9] <http://www.troubleshooters.com/codecorn/littperl/perlreg.htm>