

# Once Upon A Perl

Willie Stevenson  
Computer Science  
University of Maine  
COS 301

December 14, 2014

## **Abstract**

This paper will discuss the design and formulation of the Perl Programming Language in relation to practical programming functions and concepts. In discussing the design of Perl, a brief history of the language and its founder are given. At the core of this paper, Perl syntax, grammar, and various data structures are evaluated and sometimes contrasted with the languages that bore Perl (C, sed, and awk) or languages of today such as Java. Analyzing some real world applications as well, we see when, where, and how to program without surprise in Perl.

## Contents

<b>1</b>	<b>Overview And History</b>	<b>1</b>
1.1	Simple Intro . . . . .	1
1.2	Implementation and Design . . . . .	2
1.3	Data Structures and Features of Perl . . . . .	2
1.3.1	Scalars . . . . .	3
1.3.2	Arrays . . . . .	3
1.3.3	Hashes . . . . .	4
1.3.4	References . . . . .	4
1.3.5	Special Variables . . . . .	5
1.3.6	File Handling . . . . .	6
1.3.7	Conditionals . . . . .	7
1.4	Applications of Perl . . . . .	9
1.5	Perl's strengths and weaknesses . . . . .	9
1.6	Perl Ports (Distributions and Availability) . . . . .	10
1.7	Acceptance of Perl by the developer community . . . . .	10
<b>2</b>	<b>Syntax, Operators, Scope, and Primitives</b>	<b>11</b>
2.1	Perl Syntactic Structure . . . . .	11
2.1.1	Basics: Statements . . . . .	11
2.1.2	Code Block Structure . . . . .	11
2.1.3	Peculiarities . . . . .	12
2.2	Scope . . . . .	13
2.3	Private Data Types . . . . .	14
2.3.1	Variable Names . . . . .	14
2.3.2	Flexibility of Scalar Variables . . . . .	14
2.3.3	Flexibility of Arrays . . . . .	15
2.4	Operators: Precedence and Associativity . . . . .	16
<b>3</b>	<b>Data Types, Expression Evaluation, and Assignment Statements</b>	<b>18</b>
3.1	Data Types . . . . .	18
3.1.1	Constants . . . . .	18
3.1.2	Typeglobs . . . . .	18
3.1.3	Moose . . . . .	19
3.1.4	Strings . . . . .	20
3.2	Expression Evaluation . . . . .	20

3.3	Assignment Statements . . . . .	21
4	<b>Control Flow Constructs</b>	<b>23</b>
5	<b>Subroutines and Recursion</b>	<b>23</b>
6	<b>Conclusion</b>	<b>25</b>

## List of Figures

1	The foundations of Perl . . . . .	2
2	An array in a heap . . . . .	5
3	Print Blocks [11] . . . . .	13
4	Operator precedence from lowest to highest in Perl. [9] . . . .	16
5	Complications with operators [9] . . . . .	17
6	Complications with operators [9] . . . . .	17
7	Symbol Table and typeglobs . . . . .	19
8	Obfuscated Perl . . . . .	25

# Once Upon A Perl

Willie Stevenson  
Computer Science  
University of Maine  
COS 301

## 1 Overview And History

### 1.1 Simple Intro

Larry Wall, founder of the Perl Programming Language, intended on becoming a missionary [1]. Using his background in linguistics, he planned to learn a new language and produce a writing system that had not yet been developed. However, Wall developed some health issues and instead of doing some backpacking, he developed Perl. The first iteration of Perl was released in 1987, with versions two and three quickly after. Version 4 was released in 1994. Perl soon rose to success to become a powerhouse for development due to its easy learning curve and extensibility. Version 5 became publicly available in 2012.

Wall's background in linguistics and knowledge of how natural languages work greatly effected the design of Perl. Ideas were taken from other existing programming languages, and, combined with Wall's own, and were all pulled together in a random way similar to that of natural languages do to create Perl (languages are aggregate and transform over time). Perl was developed to solve problems that programming languages at the time could not accomplish or did not solve in an efficient fashion.

Perl was developed around what most programmers are [2]

- lazy
- impatient
- hubris (arrogant, show excessive pride)

Accordingly, Perl has the following motto: TIMTOWTDI (There is more than one way to do it). According to an interview by Erick Davis of Larry Wall, Perl stands for "Practical Extraction and Report Language" or if Wall

is in the mood "Pathologically Eclectic Rubbish Lister" [1].

## 1.2 Implementation and Design

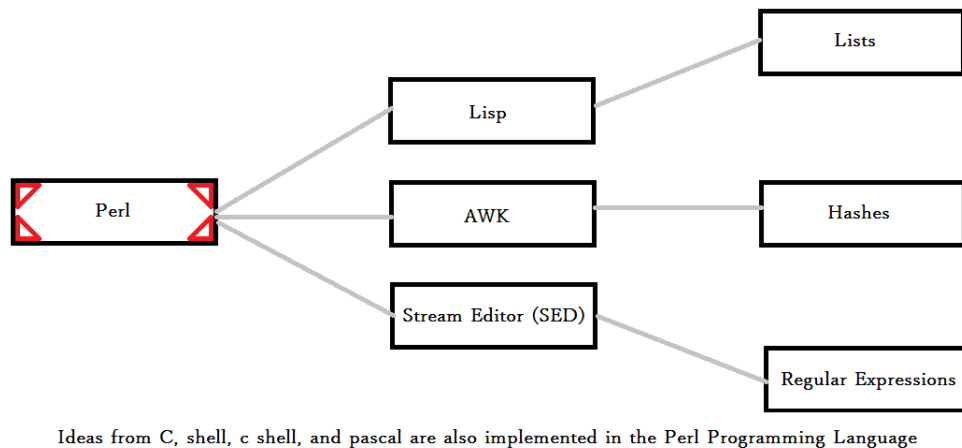


Figure 1: The foundations of Perl  
[3]

Perl is an aggregate language largely based in C. It is implemented with a C interpreter and compiled at run-time. Perl is a procedural language with statements, functions, conditional loops, and subroutines. Features found in Perl were taken from other languages deemed worthy for Perl. Perl takes the prefixes that get added on to the three main data types, scalar, arrays, and hashes from Shell Programming. Likewise, like Shell, Perl is able to work closely with the operating system (traversing directories, searching for or over files). The ability to use lists comes from lisp, hashes from AWK, and Regular expressions from SED. This increases Perl's ability to parse and manipulate text.

## 1.3 Data Structures and Features of Perl

The Perl Programming Language is a loosely typed language. This means that Perl tries to do the right thing, i.e., numbers are interpreted as numbers and strings are interpreted as strings.

Variables in Perl do not specifically need to be assigned a value at declaration time. They can be assigned later [4]. A variable is set up in the following way: The symbol, representing the type of data, is typed first and immediately following is the variable name. The = operator is used for assigning data to a variable. The data that will be assigned is found to the right side of the operator and where that data will be stored is found to the left of the operator. The basic data structures that exist in Perl are listed below and are covered in this section.

- Scalars
- Arrays
- Hashes

References and special variables are also covered briefly.

### 1.3.1 Scalars

Simply put, scalar variables hold one unit of data. The unit of data could be anything from a string, an integer, to a memory reference. But strictly holds only one unit of data. When a scalar variable is declared, a \$ must be put in front of the variable name. Some examples are given below.

```
$i = 100;    # an integer called i is declared with the value 100
$i = "where is timbucktoo?";
    # a string i is declared with the value "where is timbucktoo?"
$i = 'a';   # a character called i is declared with the value 'a'
$i = "100";
    # a string i is declared with the value "100"
```

\$ represents the *s* in scalar, hence it is used when declaring scalar variables.

### 1.3.2 Arrays

An array is comprised of multiple scalar values. When an array is declared, a @ must be put in front of the variable name. Since an array is a list, ( ) are used when declaring elements in the array followed by a coma after each element.



```
@i = ('a', 'b', 'c');  
@i = (1, 2);  
@i = ("The", "Perl", "Programming", "Language");  
@i = (1, "Perl", 'A', 5.5);
```

@ represents the *a* in array, hence it is used when declaring array variables. The variable name, *i* immediately follows the @ symbol.

### 1.3.3 Hashes

A hash is similar to an array in the sense that it is set up the same way. A hash is, however, structurally different. For every key in a hash, there is a corresponding value.

```
%i = ('John', 10, 'Bob', 28);  
%i = ('key1', 'value1', 'key2', 'value2', 'key3', 'value3');
```

% represents a hash. The variable name *i* immediately follows the % symbol.

### 1.3.4 References

In Perl, all data instances are stored in the heap and are of type scalar, holding only one type of data at a time. References are meant to hold a value that contains the location to another value. Value types that references point to could be anything from floating point numbers to arrays.

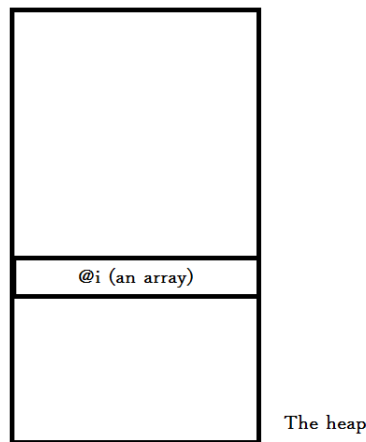


Figure 2: An array in a heap  
[3]

For example,  $i$  is an array in the heap. Let's assume  $@i = (100, 200, 300, 400)$ . If you put a backslash in front of the structure name, in this case  $@$ , you get an address.

```
\ @i = ARRAY(0x131d387);
```

This memory address is a scalar! And therefore the following is correct  
`$ref = \ @i; , \ ref->[ 1] ... this returns 200.`

### 1.3.5 Special Variables

Some variables have special meanings in Perl. To illustrate a common special variable let's take the scalar symbol  $\$$ . If we append an underscore  $\$_$  we obtain the special variable used to refer to the current or default input string. This is explained in the code snippet below.

```
foreach @a{ print $_, "\n";}
```

This code would print out every element in array  $a$  on a newline.

```
foreach @a{    print, "\n";}
```

This piece of code produces the same output even without the explicit use of `$_`. The program automatically assumes the value in `$_` and prints it for each pass in the loop.

### 1.3.6 File Handling

Perl is a language that generally deals with many files, therefore there has to be some way that Perl manipulates those files or gains access to them. This is accomplished by using filehandles.

A filehandle is a temporary name given to a file, incoming text, pipe, or socket so that input or output is efficiently dealt with. Perl comes with a few filehandles out of the box, `STDIN` and `STDOUT` (These commands are built into the UNIX system which Perl then utilizes). Let's look at the following code snippet that makes use of filehandles.

```
print "Enter your name : ";
$response = <STDIN>;
print "Your name is : ", $response, ", am I right?";
```

The code above reads in a statement from the user (their name). And then prints it back at them on the next line. In addition to reading in text from the user or from files, Perl also reads files. This means that Perl can access file names, and process them (the files). For this, a special filehandler called `ARGV` may be used. `$ARGV` refers to the current element or file and `@ARGV` represents the special array variable that is traversed that holds these filenames [17]. to work through any given files. When the `ARGV` filehandler is used, a scalar variable can be instantiated to hold the current filename which can be put collectively into an array.

Filehandles can also be utilized in while loops, like the following construction.

```
while ($_ = <ARGV>) {  
    print $_;  
}
```

Because the filehandle (`< ARGV >`) actually holds a value (whatever exists in `@ARGV`), until it doesn't, the while loop condition prints out the contents of `@ARGV` [4].

### 1.3.7 Conditionals

Similarly, like other languages, Perl has its share of while loops, for loops, and if - else statements. Let's look at the following for loop below.

```
$count = 5;  
while (count != 0){  
    print $count, "\n";  
    -$count;  
}
```

What this function prints here is obvious. It prints numbers 5 through 1, each on a newline, decrementing on each pass of the loop. When a compare is carried out in Perl, such as `'=='` (comparing if numbers are equal) or `'!=='`, the compare returns true when the values match, returning a 1. All other times, these operators return a 0, indicating false. This is Perl's built in boolean system. Perl does not have built in words such as `'True'` or `'False'`. Even the string `"0"` evaluates to false. It is converted to the integer 0 at the time of comparison because Perl is smart. However, `"00"` does not get converted to 0! Anything that has value can be used as the condition for a while loop.

However, there is more than one way to do this Perl. Sometimes it is easier to say "do something until something is true." For this reason, the "until" loop is also implemented in Perl. Below is pseudo-code illustrating the until loop.

```
until (some expression) {  
    do this;  
    do this too;  
}
```

Like many other programming languages, Perl also utilizes for and for each loops.

If statements are also included in Perl.

```
if (something) {  
    do this;  
    and this too;  
}
```

Perhaps one exciting tidbit is that Perl has unless statements in which else statements can also follow. An unless statement is like saying "do this if statement is false" compared to a similar statement in its regular context "do that if not statement is true". An example unless statement is illustrated below.

```
print "Are you 21?  :  ";  
$response = <STDIN>;  
  
unless ($response > 21) {  
    print "Great!  Go buy me a beer.";  
}  
else {  
    print "Another few years heh?";  
}
```

In the next section we will look at the real world applications of Perl.

## **1.4 Applications of Perl**

As Perl was primarily developed as a text processing and file manipulator/processing language, it has many real world uses.

The first place where Perl is used is system administration. Small Perl scripts can be used to automate system tasks.

Perl is used for parsing text. Perl even sometimes stands for "Practical Extraction and Report Language."

Many games and demos can be written in Perl. SDL Perl contains libraries that Perl utilizes to build scripts for games. The graphical user interface for Padre, an IDE for Perl, was written in wxPerl, an interface to the open source code of Perl.

Perl code can be found running in databases. Perl developers have developed an API to interface with the language of SQL. This provides a standard way to initiate queries across databases.

## **1.5 Perl's strengths and weaknesses**

[5] Perl is efficient in the following

- efficient string manipulation
- Perl is forgiving of incomplete data
- Perl is component oriented, encouraging the creation of software in small modules
- Perl is easy to write and fast to develop in
- Perl is a good prototyping language
- Perl is a good language for web CGI scripting

- Perl utilizes many special characters and operators that may take getting used to (\$,@,
- Perl Syntax is somewhat illegible and confusing to read [18].

## **1.6 Perl Ports (Distributions and Availability)**

Perl is widely supported by a wide array of systems. Most Unix/Linux systems and Macs have Perl already installed [6]. Perl is not built into Windows and therefore requires a download. Perl supports over 100 platforms and is licensed under both the Artistic License and the GNU General Public License. On the date of the publishing of this report, 5.20.0 was the latest stable variant of Perl. A complete list of supported systems can be found on the CPAN.org website.

CPAN.org is a repository for Perl code that was released in 1995. Since its release, CPAN.org has allowed a place for developers to share and produce well written, highly debugged code. During the past 27 years, CPAN.org has assembled a large developer and code base (a reason why Perl code is so stable)

## **1.7 Acceptance of Perl by the developer community**

The first few iterations of Perl were released during the advent of the World Wide Web [7]. Perl's community steadily developed during this time, as many services became available on the internet. Even 27 years after the first public release of Perl, many developer forums, Perl tutorials, and Perl code databases exist to this day showing the support and continued interest in this programming language. Here are a few :

<http://www.perlmonks.org/>  
<http://www.perl-tutorial.org/>  
<http://www.cpan.org/>

This concludes the overview and history of the Perl programming language.

## 2 Syntax, Operators, Scope, and Primitives

### 2.1 Perl Syntactic Structure

#### 2.1.1 Basics: Statements

There is more than one way to do it in Perl. This notable property may allow programmers to write Perl code without changing their current programming style. However, Perl hopes to make you do the opposite and write Perl code like Perl programmers (write beautiful, efficient code using the language's provided syntax). Let's take a look at how the famed "hello, world!" statement can be written in Perl.

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

As we can see Perl implements the simplistic version of the program with a print statement. Other keywords offer unique ways to carry out the statement.

```
say "Hello, world!"
```

You may notice that such a statement does not require a semicolon to terminate, as there are no statement terminators in the Perl Programming Language. Although this is the case, Perl does use, but not require, statement separators. They come in the form of a semicolon. Perl code may be perfectly readable without them, however, a statement separator (the semicolon) should always be added for good measure: It's usually in the coder's best interest that the compiler fully understand the code as well.

#### 2.1.2 Code Block Structure

As with languages such as C, Java, and many others, Perl allows the programmer to group and compound simple statements, such as the above examples, into blocks of code. All blocks of code must be enclosed in curly brackets ({}). Regular blocks of code are controlled by an expression and look like the following

```
if (something) {
```



```

    do this;
    and this too;
}

```

The above example of a block of code will be simplified for the purpose of explanation, where BLOCK is everything enclosed in parentheses.

```
if (something) BLOCK
```

Here are some other examples

```

until (something) BLOCK
while (something) BLOCK
foreach VAR (ARRAY) BLOCK

```

Lastly, there are goto statements governed by LABELS where the LABEL identifier provides the information as to where in the program to move to.

```

goto LABEL;
until (something) BLOCK
LABEL: while (something) BLOCK # goes here.
foreach VAR (ARRAY) BLOCK

```

### 2.1.3 Peculiarities

Rather than referring to a handful of features of Perl syntax as peculiar, the programmer should note that the following may be of compelling importance, like the following example.

```

print "Hello, world\n";
print 'Hello, world\n';

```

The result when run

```

Hello, world
Hello, world\n

```

Notice that only double quotes interpolates variables and special character such as newlines `\n` (meta-characters). The use of single quotes do not [8]. Although perplexing, following the template given by Perl's block structure, the following example is entirely legal.

```

{
    print "1\n"; {
        print "2\n"; {
            print "3\n";
        }
    }
}

```

Figure 3: Print Blocks [11]

Perl code is usually written as small scripts or modules read and executed line-by-line until completed. No `main()` functions are utilized in this process. In fact, `main()` functions are non-existent in Perl. It is implied that "main" code is the code not inside of any block structure.

Number punctuation can be represented rather oddly in Perl. The number 10000 can be represented as 10\_000, where the underscore implies a comma. Just one more element that exemplifies Perl's motto, "There is more than one way to do it."

## 2.2 Scope

The scope of a variable exists in its code block [10] . The example below creates a global variable.

```
$var = "value";
```

The next example creates a lexically scoped variable.

```
my $foo = "foo foo";
```

The use of the keyword *my* creates a variable that is scoped to the code block where they are defined. Common programming errors are caught by adding by the *use strict*; statement at the top of your program.

```

my $foo = "Little rabbit foo foo";
if (1 < 2) {
    my $foo1 = " running through the forest ";
    my $foo2 = " scooping up the field mice and bopping them on them head.";
}

```

```
    print $foo;  
    print $foo1;  
}  
print $foo2
```

The use of `strict` throws a compile-time error because `foo2` does not live in the block of code it is asking to be printed in.

## 2.3 Private Data Types

Languages such as C, C++, and Java require that variable type is declared in advance (more so, the compilers require it) [11]. The variable types do not change. Conversely, Perl does not care about the type of data that is stored in a variable. Anything you store (or organize), will try to be handled duly by Perl.

There are three data types. They are predominately based on the first.

- Scalars
- Arrays
- Hashes

A scalar variable holds a single value. It could be a string, a character, an address, anything. Arrays contain multiple scalar values. Hashes (Dictionaries) in Perl contain pairs of scalars. Variables are identified by the symbol that precedes their name. All scalar variables employ a `$`, arrays an `@` symbol, and hashes a `%` symbol.

### 2.3.1 Variable Names

Variable names in Perl may consist of any number of letters, numbers, and underscores. However, starting a variable name with a number is prohibited. A variable's name is also case sensitive.

### 2.3.2 Flexibility of Scalar Variables

Mentioned above, a scalar variable is not confined to one type of data. Along these same lines, Perl interestingly interprets the contents of a scalar

variable when operations are executed upon it. Below is a key example.

```
$var = "5";  
print var * var
```

First, note that multiplication is trying to be carried out on a string. Secondly, the result of the operation yields the integer 25. Perl tries to understand the programmers' intentions and automatically converts the stored values and prints something that is hopefully desirable. In the case of a letter following the number in the above string (undergoing the same operation), the print statement would still produce 25. The compiler looks at all digits up until the first letter, ignores it, and executes the operation.

Although still treated as type scalar, placing a 0 in front of a series of numbers (0455 as opposed to 455) assumes an octal number (base 8). Analyze the following code fragment below.

```
$number = 0455; # Assigns the decimal value 301 to number
```

If a variable is assigned a sequence that begins with 0x or 0X, it is treated as a hexadecimal number. Here is another code fragment.

```
print 0x455; # prints the decimal value 1109
```

### 2.3.3 Flexibility of Arrays

An array in Perl is declared with a set of parentheses and each element (of type scalar) in an array is separated by a comma. To fill an array with values from x to y use the range operator (..).

```
$numbers = (5 .. 55); # Fills the array with the elements 5 through 55
```

Arrays can only contain scalars. The result of inserting an array into an array is a "flattened" [11] array.

## 2.4 Operators: Precedence and Associativity

Although there are operators that are unique to Perl, the majority that are in current use are chiefly borrowed from C.

Dealing with Perl's picky operators may be vexing at times, however it

Associativity	Operators
nonassoc	The list operators
left	,
right	= += -= *= etc.
right	?:
nonassoc	..
left	
left	&&
left	^
left	
left	&
nonassoc	== != <=> eq ne cmp
nonassoc	< > <= >= lt gt le ge
nonassoc	The named unary operators
nonassoc	-r -w -x etc.
left	<< >>
left	+ - .
left	* / % x
left	=~ !~
right	**
right	! ~ and unary minus
nonassoc	++ --
left	'( '

Figure 4: Operator precedence from lowest to highest in Perl. [9]

is all meant for the greater good. Therefore particular attention to precedence and parentheses may be needed. For example

What you wrote	What compiler sees
<code>rand 5    10</code>	<code>(rand 5)    10</code>

Figure 5: Complications with operators [9]

With this functional statement, what is actually executed first is the function `rand` on the value `5`. `|| 10` is then considered. This is because the operator `||` actually has lower precedence than the function `rand`. The example below preforms the opposite way.

What you wrote	What compiler sees
<code>rand 5 * 10</code>	<code>rand (5 * 10)</code>

Figure 6: Complications with operators [9]

The `*` operator has higher precedence than the function and therefore the multiplication operation is carried out first.

There is a multitude of information related to each type of operator and the precedence it takes in a given statement. For further more detailed information please refer to Wall's camel book, Programming Perl.

This consummates the operators, syntax, variable scope and primitives that are present in Perl.

## 3 Data Types, Expression Evaluation, and Assignment Statements

### 3.1 Data Types

In Perl, a scalar variable, denoted by the prefix \$, is the constituent that defines all variables. A scalar variable contains one unit of data, whether it be a string, integer, etc,. Rather than calling higher level variables (non-scalar variables) non-primitive data types in Perl, it is rather preferred to describe non-primitive variables in the context of type scalar (the only primitive data type) [11]. Thus, describing an aggregation of scalar variables is referred to by the way they are ordered. An un-ordered aggregation (collection) of scalar variables is called a hash. Inversely, an ordered collection of scalar variables is referred to as an array or list.

At a higher level, there are also data types that impart structure to Perl files.

#### 3.1.1 Constants

A constant, although not exactly a variable (a constant is a function that takes no arguments but is defined like a variable), defines a value that stays constant throughout the life of the program. Constants in Perl are weighed in at compile time and are accounted for even if the block of code they live in does not execute. A constant, unlike other Perl variables, does not include a prefix symbol \$, @, etc,.

```
use constant VARIABLE_NAME => SOME_VALUE;
```

Note that the use namespace must be used when defining constants.

#### 3.1.2 Typeglobs

Typeglobs, another data type, are used to eliminate issues that may arise during the writing of a program. Comparable to passing by reference (used in Java), Typeglobs may be aliased to ordinary references in such a way that you don't have to use de-referencing syntax [13]. They may also create

aliases of symbols (variable prefixes). In other words, in Perl, you can refer to all of the items of a particular name by prefixing the name with "\*", as in `*thing`. The diagram below, taken from the first edition of *Advanced Perl Programming*, illustrates the idea of typeglobs referring to and representing every item, whether it be scalar values, filehandles, or other data types that live in a program.

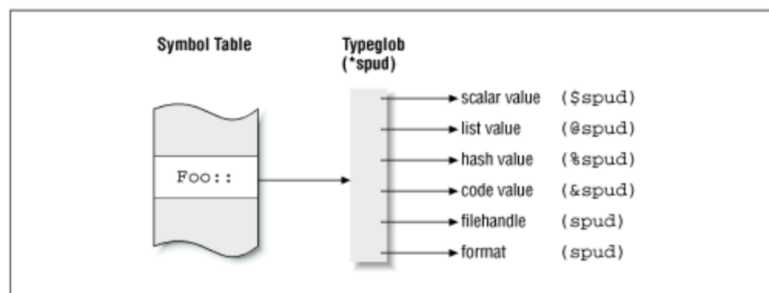


Figure 7: Symbol Table and typeglobs  
[13]

With this example, it may also be worth while to mention that Perl provides explicit namespaces for each data type. Thus, as Figure 1 shows, `$spud`, `@spud`, `%spud`, etc., are valid and are all independent of each other.

### 3.1.3 Moose

As oddly as it is called, Perl has an object system called Moose (what was Larry thinking?) [15].

```
package Animals {
    use Moose;
}
```

Even though this appears to do nothing, this is all that is needed to instantiate a class. A class may have zero or more attributes, methods and superclasses (parent classes). Classes also have constructors and destructors.



To create objects of the Animals class, the following is used:

```
my $animal = Animals->new;
```

### 3.1.4 Strings

A string in Perl utilizes no formatting rules. A string could be one word, the code an image is made up of, or even the contents of an entire dictionary. Representing a literal string in Perl requires that you surround them with quotes. String delimiters include single and double quotes. Although a string is considered a primitive data type in Perl (a string is always contained in a scalar variable, the most primitive of data types), it is worth mentioning how they are interpolated with relevance to expression evaluation. Let's look at the examples below and then move on to the next section in which expression evaluation is discussed in detail.

```
my $myString = "Bob's your uncle.\n"
prints Bob's your uncle (does the new line and interpolation)
my $myString = 'Bob's your uncle.\n'
prints Bob's your uncle.\n
(does not do the new line and does not do interpolation)
```

When backslashes are often used to insert quotes and other symbols, the operator `q` may be used in their place. Let's take a look at an example.

```
my $myString = qq{ "Bob", what are you doing? "Nothing." } ;
```

## 3.2 Expression Evaluation

To program without surprise in Perl requires that one know all cases in which their variables and their corresponding values change [12]. Perl heavily relies on variable context to evaluate expressions, yet expressions sometimes also determine this context. Defined earlier, it is known that scalar variables contain exactly one "piece" of data. Therefore care must be taken when we use such a variable in an expression.

```
my $val = 0 + $val1;
```

This assumes dealing with integer values, append 0 to force the evaluation to your liking.

```
my $bool = !! $val2;
```

Dealing with booleans, double negate val2 to force a boolean value out.

```
my $str = `` . $val3;
```

To obtain a value in the form of a string, concatenate val3 with a null string.

Above: Unary coercion in Perl [14]

To make absolutely sure that an expression evaluates to what you require, append the base null value relating to the type of variable you know you will be dealing with. However, this may be totally unnecessary, because Perl is smart. For example, when Perl sees an expression with `'.'` (string concatenation operator) it treats the operands, whatever they may be, as strings. Likewise, when Perl comes across the `'+'` operator it treats the operands in the expression as numbers.

### 3.3 Assignment Statements

The semantics of a language define the grammar of how symbols are allowed to be arranged. Perl uses semantics when assessing assignment statements similar to those of the languages C and Java.

An assignment statement in Perl does the job in two steps [16]. First the calculation on the right of the `'='` sign is evaluated. If there is nothing to evaluate, the value on the right is used. The next step would be to replace the contents of the variable on the left with the newly obtained value (if any) on the right. A variable in Perl can be used on both the left and the right side of the `'='` in the same assignment statement. When used on the right side of the `'='` sign, it is used as a number to calculate a value. When

it is used on the left it says where to store it in the confines of the program. When the same variable is found on both the right and the left side of the '=' sign, the value on the right side is re-stored in the left. Here are a few examples of valid statements in Perl.

```
my $val = $val1;
```

Here, the value in val1 is stored in val.

```
$val = $val1 + $val;
```

Here, the current value in val and the value in contained in val1 gets stored in val.

```
$val += $val1;
```

Here, the += shortens the statement `term = term1 + term`. The evaluation is the same.

```
my $val = $val1 = 1;
```

The value 1, is assigned to both variables val1 and val.

Many other examples exists, such as list assignment statements. Knowing the way these statements are interpreted, given their various operators, is always important and should be studied by the programmer. This way, we may learn to program without surprise in Perl.

## 4 Control Flow Constructs

As this paper has pointed out, particularly in the code block and conditional sections, Perl has an array of control flow constructs, allowing a program to operate based on current values that exist at a given time. Please refer to those sections for more information about them.

## 5 Subroutines and Recursion

The definition of a subroutine in Perl consists of the keyword `sub`, followed by the name of the subroutine, followed by a block of code in curly braces [19]. A subroutine may be put anywhere in a program, it does not matter where. A subroutine may be called by using the `&` symbol in front of the name of the subroutine. Subroutines are used as a form of abstraction, hiding away details that may not need to be immediately seen.

Arguments that are passed to a subroutine are known as actual arguments [19]. These parameters are placed in a LIST after the name of the subroutine, in parenthesis, like the following:

```
&run(item1, item2);
```

Perl is an exception, compared to languages such as Java; you are not forced to declare the names and types of the arguments that you pass. Since Perl already provides arguments as elements of the special `@_` array, there is no need to define types or names for these arguments. However, if this is not practiced, it is customary to copy the arguments immediately into a named list, which has this effect (naming the arguments) [19]. Here is an example:

```
sub run {  
    ($string1, $string2) = @_;  
    print $string1 . ", " . $string2;  
}
```

However, we must be careful when pulling variables from a program into the subroutine. We may want to modify them only inside of the subrou-

tine. To get around this, Perl supplies the programmer with a special function called `local()`. The `local()` function acts just like a LIST of variables in parentheses, except that each variable you mention is "localized" for the remainder of the current block statement. A localized variable can be modified, but the modification is temporary (the life of the subroutine) [19]. This has the effect of turning call-by-reference into call-by-value, since the assignment copies the values. The correct way to use the local function is as follows:

```
sub run {  
    local($string1, $string2) = @_;  
    print $string1 . " , " . $string2;  
}
```

This way, only the localized copy of a variable is worked upon and the main program's variable is left untouched.

It should also be noted that a return statement may be used to specify the returned value and exit the subroutine. The returned value of a subroutine is the value of the last expression evaluated and can either be of scalar or array value.

Regarding whether subroutines may be passed to other subroutines, the Camel book, Programming Perl, does not contain any mention of this. This leads me to believe that there is no support for such a function.

Perl is a language that is often used for processing file structures and system directories. Perl therefore has an efficient way of doing so by means of recursion. Recursion is simply done by calling a subroutine inside of that same subroutine. Many interesting examples may be found in Larry Wall's book, Programming Perl.

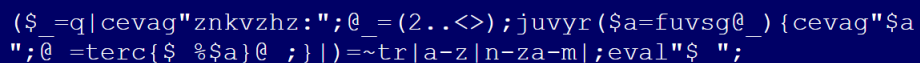
## 6 Conclusion

Perl has many one liners. This is arguably the best in all of Perl.

```
perl -pi -e 's/text/text1/g' *.file_extension
```

This short one liner is meant to be run from the system terminal. If you have Perl installed, only then can you take advantage of it. Sometimes called Perl pi, this statement is means to search through all text in a file an replace it. In this example, wherever the word 'text' exists in all of the files in the current directory is replaced with the string 'text1'. This has implications beyond programming itself. For example, if you were running a website that contained 100 or more different web pages whose links you wanted to change, running this command could batch rename every single link in a matter of moments (in today's computing not even a moment). Instead of having to go into each file and having to change each link name manually, this proves to be exponentially more efficient. I also have personal experience in this, as I work on and manage a website. I find it incredibly handy and it saves time. This to me showcases the beauty of Perl.

However, Perl can be rather difficult to understand at times. Here is an excerpt from the Perl Obfuscated Code Challenge that was held at MIT.



```
($_=q|cevag"znkvzhz:";@_=(2..<>);juvyr($a=fuvsg@_){cevag"$a";@_=terc{$_%$a}@_;})=~tr|a-z|n-za-m|;eval"$_"
```

Figure 8: Obfuscated Perl  
[20]

I use difficult only as a nice word.

As a whole, I view Perl as such a language. It is a mash of beautiful code and code that is hard on your eyes and makes your head spin. For this reason, it is not my go to language. But, there are times where Perl is the most appropriate language to solve the problem at hand. For this reason, to me, Perl is just another weapon in my arsenal.

## References

- [1] Erik Davis. (1999, February 10). Divine Invention: An interview with Larry Wall (edition 1)[Online]. Available: <http://www.techgnosis.com/wall1.html>

This reference provided handy interview information from the creator of Perl, Larry Wall, himself.

- [2] BigThink.com. Big Think Interview with Larry Wall, [Online video]. Available: <http://bigthink.com/videos/big-think-interview-with-larry-wall>

This was along the same lines as the first reference. It provided interview information from Larry Wall.

- [3] Dr. Robert Edwards of SDSU. (2013, October 10). Perl Data Structures [Online video]. Available: <http://www.youtube.com/watch?v=r8aWZFXHG8>

This reference was a helpful lecture video found on youtube that was given by Professor Dr. Robert Edwards at San Diego State University.

- [4] Larry Wall and Randal L. Schwartz, "An Overview of Perl", Programming Perl, Edition 2. Sebastopol, CA, O'Reilly, 1991, ch. 1, pp. 7-13

The cornerstone of all Perl Programmers and the first book in the series to discuss Perl. It was partly co-authored by Larry Wall himself.

- [5] Lincoln Stein. (1996, February). How Perl saved human genome (Reprinted ed.)[Online]. Available: [http://genetics.stanford.edu/gene211/handouts/How\\_Perl\\_HGP.html](http://genetics.stanford.edu/gene211/handouts/How_Perl_HGP.html)

A resource describing a use of Perl in the real world - Relating to Bio-Genetics.

- [6] CPAN.org. (2013). Perl Ports (Binary Distributions)[Online]. Available: <http://www.cpan.org/ports/>

This resource provided the current available versions of Perl and which systems they were available for.

- [7] Stackoverflow. (2014). About Perl [Online]. Available: <http://stackoverflow.com/tags/perl/info>

This resource provided web-sites the link to Perl tutorials and Perl discussion boards. This is helpful in that many people are always willing to offer help when it comes to programming in Perl.

- [8] tutorialspoint. (2014). Perl Syntax Overview [Online]. Available: [http://www.tutorialspoint.com/perl/perl\\_syntax.htm](http://www.tutorialspoint.com/perl/perl_syntax.htm)

A very helpful resource that covers all things syntax related in Perl. Code snippets and sample demo programs were found here and proved to be useful for understanding some oddities in Perl grammar and syntax.

- [9] Larry Wall and Randal L. Schwartz, "An Overview of Perl", Programming Perl, Edition 2. Sebastopol, CA, O'Reilly, 1991, ch. 3, pp. 65-

Again, the cornerstone of Perl Programmers. This was a reference that I continually went back to over and over again.

- [10] Kirrily "Skud" Robert. (2014). perlintro [Online]. Available: <http://perldoc.perl.org/perlintro.html>

Similar to the tutorialspoint reference, this helped considerably in understanding the syntax of Perl especially key identifier words.

- [11] Reuven M. Lerner, "Getting Started", Core Perl, Edition 1. Upper Saddle River, NJ, Prentice-Hall, Inc, 2002, ch. 2, pp. 21-

This resource points out some basics of Perl that were useful in understanding the language.



- [12] Matthew Might. A guide to Perl: By experiment (edition 1)[Online]. Available: <http://matt.might.net/articles/perl-by-example/>

Similar to other resources in that it points out the syntax and grammar of Perl.

- [13] Sriram Srinivasan, "Typeglobs and Symbol Tables", Advanced Perl Programming, Edition 1. Upper Saddle River, NJ, O'Reilly, 1997, ch. 3, pp. 68

A resource I used to solely understand what typeglobs were.

- [14] Chromatic. (2011-2012) Modern Perl: Unary Coercions (edition 2011-2012)[Online/Book]. Available: [http://modernperlbooks.com/books/modern\\_perl/chapter\\_10.html](http://modernperlbooks.com/books/modern_perl/chapter_10.html)

Somewhat of a modern textbook for those who want to learn Perl. It analysis' of the many practical programming concepts in Perl are limited, but sometimes are explanatory enough and useful.

- [15] Chromatic. (2014) Modern Perl: Moose (edition 2014)[Online/Book]. Available: [http://modernperlbooks.com/books/modern\\_perl\\_2014/07-object-oriented-perl.html](http://modernperlbooks.com/books/modern_perl_2014/07-object-oriented-perl.html)

The updated version of the reference below. Each online edition seems to contain slightly different information in less or more detail.

- [16] Chromatic. (2011-2012) Modern Perl: Undefined (edition 2011-2012)[Online/Book]. Available: [http://modernperlbooks.com/books/modern\\_perl/chapter\\_03.html](http://modernperlbooks.com/books/modern_perl/chapter_03.html)

Somewhat of a modern textbook for those who want to learn Perl. It analysis' of the many practical programming concepts in Perl are limited, but sometimes are explanatory enough and useful.

- [17] Larry Wall and Randal L. Schwartz, "An Overview of Perl", Programming Perl, Edition 2. Sebastopol, CA, O'Reilly, 1991, ch. 1, pp. 2-3

The cornerstone of all Perl Programmers.

- [18] Lars Marius Garshol (Date Unavailable) What's wrong with Perl (edition 1)[Online]. Available: <http://www.garshol.priv.no/download/text/perl.html>

This text discusses the writers opinion of weaknesses that exist in Perl. And I must admit that I agree with him and his reasoning.

- [19] Larry Wall and Randal L. Schwartz, "An Overview of Perl", Programming Perl, Edition 2. Sebastopol, CA, O'Reilly, 1991, ch. 2 & 3 pp. 50-53, 99-101

The cornerstone of anybody who studies Perl.

- [20] MIT. (-) Perl Programming (edition 1)[Online]. Available: <http://stuff.mit.edu/iap/perl/>

Some really really obfuscated Perl code can be found at this reference.