Homework 6: Features of Perl
Programming Languages

Charlie Coleman

Contents

1 Overview

Perl is an interpreted language developed mainly for text manipulation. It has grown to be usable for almost any application. It's goal is ease of use & efficient code[1]. It claims to focus on these over "aesthetic" programming languages, which focus on tiny/elegant/minimal code.

Perl includes support for procedural & object-oriented programming. It has a strong focus on modules, which can be added for any number of specialized features. Overall it can be compared to Python, as both are high-level, interpreted programming languages that gain a lot of functionality through modules/imports. Both Perl and Python go without explicit variable typing. Here is an example of Perl code, a simple "Hello, World!".

Example 1: Hello, World!

```
1 print "Hello, World!\n";
```

2 Features

2.1 Functional? Imperative?

Or both? Perl is typically used as an imperative language, but it is also able to be used as a functional language, fulfilling all of the commonly required features of a functional language[2]. For example, Perl supports first class functions and higher order functions[3]. This means that functions can be passed as parameters, and that functions can take functions as inputs and/or return a function. Here is some example code for these features.

Example 2: Functional Programming

```
1  use strict;
2  use warnings;
3
4  my $first_class = sub {
5    print "hello\n";
6  };
7
8  sub higher_order {
9    my ($function) = @_;
10   $function->();
11   return $function;
12 }
13
14 my $should_print_hello = higher_order($first_class);
15 $should_print_hello->();
```

This is a fairly simple chunk of code that adequately displays some functional programming concepts. First, we create a first class function named `$first_class` that prints "hello". Next, we create a higher order function called `higher_order` which recieves a function as input, calls it, then returns it. Then we create a variable, `$should_print_hello`, which is set

equal to the higher order function that is passed the first class function. Then we call that variable. When run, this code matches the expected output, simply printing "hello" twice.

2.2 Lexical Scoping

Perl is a lexically scoped language[1], however, when a variable is created in Perl, its scope depends on how it was declared. Using `my` to declare a variable will create a lexically scoped variable, and this is the recommended approach. If the `my` is left out, however, the variable will be a global variable. This causes the variables to behave like they are dynamically-scoped in some situations. But as this is considered bad practice, we will focus on the lexical scoping with `my`. For example, in this code:

Example 3: Lexical Scoping

```
1  sub fcn1 {
2    my $x = 1; # x1
3
4    sub fcn2 {
5      $x = 2;
6    }
7
8    sub fcn3 {
9      my $x; # x2
10     fcn2();
11   }
12
13   fcn3();
14   print($x, "\n"); # prints "2\n"
15 }
16
17 fcn1();
```

Because Perl is lexically scoped, the output of this program is 2. This is because when `fcn2` is called, the `$x` it refers to is determined at compile time to be the innermost definition of `$x`, which is the variable definition labeled as x1. If this were a dynamically scoped programming language, we would expect this code to output a 1, as the `$x` marked as x2 would be pushed to the stack after x1, and the assignment in `fcn2` would be applied to this x2. After `fcn3` is completed, x2 would be removed from the stack and `print($x, "n")` would print 1.

2.3 Scopes & Modules

All scopes in Perl can nest within each other. This can be seen in Example 3, we can nest functions within each other. These scopes are open, as the functions nested inside of a function are accessible from the outside without calling the enclosing function. For example:

Example 4: Nested Scopes

```
1  sub f1 {
```

```
2     print "f1\n";
3     sub f2 {
4       print "f2\n";
5       sub f3 {
6         print "f3\n";
7       };
8     };
9   }
10
11  f1();
12  f2();
13  f3();
```

In this example, all of the function calls will be successfully executed, and we will see all of the print statements executed.

Perl also supports modules. Modules are imported using the `use` keyword, and are open scope as well. The functions within a module can be used without explicitly importing them, or accessing them from the module as an object. This is shown in the code below.

Example 5: Modules

```
1  use Math::Trig;
2
3  my $y = atan(1);
4  print $y, "\n";
```

This code uses the Math::Trig module from Perl, which includes the atan function. This code will correctly print out the inverse tangent of 1.

2.4 Recursion

Recursion is accomplished by calling the subroutine within the subroutine itself. This a very common way of doing recursion, so it should look fairly familiar to most people who have used Python/C/Java.

Example 6: Recursion

```
1  sub gcd {
2    my ($a, $b) = (@_);
3    if ($b == 0) {
4      return $a;
5    }
6    return gcd($b, $a % $b);
7  }
8
9  print gcd(8192, 384), "\n"; # should print 128
```

2.5 Type Checking

Whether Perl is a weakly & dynamically typed language. Perl only has 3 types: scalars, arrays, and hashes[1]. These are designated by the character before the variable name; $, @, and % correspondingly. The scalar data type encompasses everything like int, float, string, etc. Any variable that is a scalar can be set to any of these at any time, without complaint by the compiler. Arrays in Perl can contain any combination of scalar variables, and are variable in size.

Example 7: Type Checking

```
1  use strict;
2
3  my $a = 15;
4  print $a, "\n"; # 15
5  $a = "hello there\n";
6  print $a; # hello there
7
8  my @b = ("this ", "is ", "an ", "array.\n");
9  print @b; # this is an array.
10 @b = (1, " two ", 3, " four ", 5, " VI ", 7, "\n");
11 print @b; # 1 two 3 four 5 VI 7
12
13 $a = @b;
14 print $a, "\n"; # 8
```

In the code above, you can see various assignments to different variables. The results are mostly as expected when compared to the rules defined above, except for the last print statement. When a scalar data type is set equal to an array, it only gets the number of elements in the array and none of the elements.

2.6 Overloading, Coercion, and Polymorphism

Perl will coerce all scalar values into the "proper" format for its functions. Scalars do not differentiate between ints and floats, instead treating both as "numbers". It also lacks any support for booleans, instead using 0/null/undefined for FALSE, and anything else as TRUE. This means that in code like below, we can concatenate any combination of strings/numbers, we can add strings to numbers, etc. If a string is not in the form of a number, Perl resolves it to be 0 when treated like a number.

Example 8: Coercion

```
1  use strict;
2
3  my $a = "12";
4  my $b = 3;
5  my $c = " = 12";
6
7  print $a.$b, "\n"; # 123
8  print $a+$b, "\n"; # 15
```

```
print $a.$c, "\n"; # 12 = 12
print $b+$c, "\n"; # 3
print $b==$b, "\n"; # 1
```

Overloading in Perl is done differently than in a lot of Object Oriented programming languages. It relies on examining the parameters passed to a function and changing behavior based on them. This is because the language is weakly & dynamically typed, there is no restriction on what is passed to a function. Therefore, when calling a constructor or function, we can pass any number of different scalars, arrays, instances of classes, etc. All will be passed to the same function/constructor, and that will handle all the different supported options[4].

Polymorphism is supported by Perl. When a function is called on an object, Perl checks to see if that method is supplied by the object, and calls that function. This allows the same function name to be used in multiple classes without conflict. If the function is not supplied, it checks base classes for the object, then the universal set of functions.

2.7 Order of Operations

Operator precedence in Perl follows mathematics for the most part. Operators of the same precedence are evaluated from left to right. For example, `8 + 2 * 17 == 42`, not `170` and `3 - 2 - 1` → `1 - 1 == 0` and not `3 - 2 - 1` → `3 - 1 == 2`.

Perl also considers `&&` and `||` as 'short-circuit' operators. This can be seen in the following example.

Example 9: Short Circuiting

```
my $x = 1;
if (($x == 1) || (++$x == 2)) {
  print $x, "\n"; # if it short circuits, it prints 1, else prints 2
}
```

When run, this code prints a 1, which implies that Perl supports boolean short circuiting. In other words, when the `||` operator encounters a TRUE in the first term, or the `&&` operator encounters a FALSE, it does not need to check the second term, because the statement will evaluate to TRUE/FALSE no matter what. This can be useful when you want to check the value of a variable that may not exist. Checking its value may through an error if it is not initialized, but checking if the var exists and checking its value will solve this conundrum.

3 Conclusion

Perl is a very interesting language. It is the seeming opposite of a programming language like Java, which has strong, static typing, closed scope, etc. I feel that it focus a bit too much on offering every option under the sun to the programmer, instead of narrowing its focus to a few features. It does not constrict itself to Object Oriented/Functional/Imperative/Procedural/etc, instead allowing the programmer to utilize any of these paradigms depending on how the code is written. This could prove to be very useful in some cases, but provides a bit of a barrier to entry as the user can't look at a set of 'best practices' for the language.

References

[1] Robert, Kirrily. *Perl Programming Documentation*. `https://perldoc.perl.org/`

[2] Functional Programming, `http://wiki.c2.com/?FunctionalProgramming`

[3] Dominus, Mark Jason. *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann Publ., 2009.

[4] Overloading, `https://www.perl.com/pub/2003/07/22/overloading.html/`