The Beauty of Perl

Jordan Marshall Matthew Ghigliotti

1 Introduction

Perl is an amazing programming language. Perl is amazing because it is both simple and expressive. You can open a file and search through it in three lines. Perl is amazing because it is versatile. You can set up hashes on the fly, and if you want to index with both strings and numbers, feel free. Perl is amazing because you can accomplish a wide variety large tasks easily.

While being amazing certainly is a fair accomplishment, there are other important aspects of Perl. It is easy to learn the basic constructs and write effective programs. Perl is so easy to learn because it is consistent with the familiar notions about natural languages. Perl programs are very portable – they can be run it on almost any system in existence. Perl even has module functionality; it supports programming in the large.

Perl might be amazing, but it also has drawbacks. It has no formal written specification, and as a consequence, it does not always behave in a way the programmer expects. Perl has multiple evaluation contexts that are often defined solely by the operation, and mistakes are very easy to make. Perl has considerable flexibility, allowing the same task to be written multiple ways, but this can overwhelm beginning programmers. And while Perl has excellent module support, its lack of definition and uniformity can make large projects a chore.

First, a brief history. Perl is an interpreted multi-functional language that was designed by Larry Wall and released in 1987. Mr. Wall studied linguistics while in graduate school and then applied principles garnered from linguistics in developing Perl. Perl is widely based off of Unix scripting languages *awk*, *grep*, and *sed*, and it also incorporates features from other languages such as objects and subroutines. It was built for ease of use and simplicity, in a time when hardware costs were falling and labor costs were rising. It is capable of efficiently taking care of many tasks from text processing to web development, giving way for its label as a "glue" language.

As a discussion and exploration of Perl, the next several sections will examine the implementation of Perl, as well as focus on several topics common to most programming languages. These topics include types, scope, objects, and programming paradigms. Once this foundation is in place, some of the highlights of Perl's power will be demonstrated, as well as a brief summary outlining the properties of Perl in relation to what was covered overall.

2 The Perl Interpreter

There can be many implementations of a compiler or a runtime environment, given a language specification. Interestingly, this is not so much the case for Perl. It has been said that "only *perl* can parse Perl;" that is, that only the standard Perl interpreter can properly handle the Perl language and syntax. There is one primary reason for this: *perl* (the official interpreter) is the de facto specification for the Perl language. As of the latest version of Perl, there is no written specification. In this way, the language and the interpreter can be considered one of the same, as well as that aspects of the interpreter can be discussed without considering different implementations.

Perl is commonly seen as an interpreted language, but this is not entirely so. Much like Java, Perl has a front-end compiler which compiles the program into a parse tree; and a back-end interpreter, which starts executing it immediately. However, unlike Java, both of these phases occur during the execution of the *perl* program. The bytecode is not explicitly saved, but it can be viewed with command-line options. More recent versions of Perl support standalone compiling.

Garbage collection within a language is often very useful, as it frees programmers from the the painstaking tasks of memory management. Most modern languages support automated memory management, including Java, Ruby, and Python. In this way, the Perl interpreter is no different; it too supports automated memory management and garbage collection. The Perl interpreter uses reference counting for its garbage collection scheme. One likely reason for this is ease of implementation: producing a garbage collector which uses a more advanced scheme is decidedly more complicated. This scheme, however, has some pitfalls with it. In particular, if an object references itself directly or indirectly, it will not be considered garbage by the garbage collector engine. To address the issue of circular references, Perl has a solution: the programmer can free variables and structures manually, through use of the `undef` command. Although the scheme of garbage collection could change in a future version of Perl, it is unlikely because the downsides to the current scheme have already been addressed.

3 Properties of Perl

3.1 Types

Perl contains three main primitive types: scalars (denoted by `$`), arrays (denoted by `@`), and hashes (denoted by `%`). File handles (denoted by a name in capitals) and subroutines (denoted with `&`) can also be considered types. Scalars represent the fundamental type from which the other two primary types are built. Within a scalar can be stored any single value – a string, number, or reference. An array is an ordered list of scalars that can be accessed via indexing. A hash is an unordered list of data, accessed by a key specified when the data is stored.

These basic types essentially define the Perl typing system. Each type is equivalent and compatible with members of the same type. Since both are scalars, a programmer can add a string to an integer or an integer to a float. In both cases, the Perl interpreter will determine what the resulting values should be. However, in most cases, it is possible to overload the operator in question and and use objects of differing types. This important detail is known as context.

In Perl, an operation is evaluated according to its context. The two major contexts are scalar context and list context, and the determination of which is used depends upon the operation and arguments. Consider an instance, where an assignment operation is occurring. If the left hand value was a scalar, the operation would be evaluated in a scalar context. Alternately, if the left hand value was an array, the operation would be evaluated in a list context. Thus, it is possible to assign an array to a string and vice versa, although unexpected results are likely to occur. This leads to some interesting quirks, as shown in Example 1.

The scalar context can be further broken down into several subgroups, including string, numeric, and boolean contexts. Unlike the distinction between list and scalar contexts, an operation does not differentiate between string and numeric context. Rather, it simply returns a scalar and lets Perl evaluate the result. Numeric operators, such as increment (`++`) and arithmetic (`+`, `-`) put a scalar in numeric context, while string operators such as concatenation (`.`) and repetition (`x`) put a scalar in string context. A boolean context occurs when an expression is being evaluated as true or false. A scalar is false whenever it is the empty string, any representation of '0', or if it is undefined;

Example 1 Playing with types.

```
$string = "jordan";
@array = ("great","awesome","charming");
$string = $string.@array;   # concatenates the array length onto the string
                            # $string now becomes "jordan3".
@array = $string;   # the string becomes the only entry in the array
                    # the array has only one entry, "jordan3"
```

any other values for scalars evaluate to true. In Example 2, `$joe` evaluates to false in all of the cases.

Example 2 Context evaluation.

```
$jordan = "great";
$matt = "5.7";

print $jordan + $matt;    # prints "5.7", as $jordan evaluates to 0
                           # in a numeric context.
print $jordan.$matt;      # prints "great5.7", as $matt evaluates to 5.7
                           # in string context.

$joe = ""; $joe = 0; $joe = "0";
@jordan = (); $joe = @jordan;  # @jordan in scalar context returns
                               # the length of the array: 0.
```

Arrays and hashes are made up of scalars, and since scalars can be strings or numerical values, it is perfectly legal to declare an array with an integers, doubles, and strings. Furthermore, lists can also be placed inside of lists, so it is legal to make the declaration shown in Example 3. In this example, `@moreQualities` will be interpreted in a list context and added to the array; thus, `@jordan` would be `("cool","awesome","intelligent","savvy")` after the assignment. If a programmer wanted to put an array or hash inside of an array without losing the structure of the former, they would need to insert a reference (or, as C programmers are familiar with, a "pointer") to the added array or hash instead.

Example 3 An array inside of an array.

```
@moreQualities = ("intelligent","savvy");
@jordan = ("cool","awesome", @moreQualities);
```

References are technically scalars, but these are worthy of special attention. References can be created by assigning a new scalar to reference an existing variable (`$ref = \@jordan;`), or by declaring a reference to a set of data; this latter example is known as an anonymous reference. References can be dereferenced by using the appropriate sigil in front of the reference name. So, for example, `@$ref[0]` would return the first value in the `@jordan` array mentioned earlier. It is important to note that, while Perl supports dynamic structures in memory, it does not allow for pointer arithmetic.

Since Perl is especially effective at file manipulation, another type that should be mentioned is file handles. File handles do not have to be predeclared and are created with the open command; the statement `open(NAME, "filename");` opens the file filename and gives it handle `NAME`. This handle can be used throughout the program to access the file. A programmer could, for instance, store the file to an array with the statement `@myfile = NAME;`, or loop through each line with `while(<NAME>) {}`.

3.2 Scope and Modules

In Perl, the notion of scope and modularity are closely tied together. Modules, or packages as they are known in Perl, each have their own namespace which can be referenced within or outside of the package. The current package can be changed at any time, though doing so is not necessary to access the data.

Example 4 Basic package use.

```
package jordan;
$name = "jordan";

package matt;   # Here we are switching to the package "matt".
$name = "matt";      # We are now in the "matt" namespace.

package jordan
print $name       # prints "jordan" as we are now in the jordan namespace.
print $matt::name  # prints "matt".
```

Packages can also include subroutines which, like variables, can be called outside of the namespace (this is important for implementing objects). Packages can be attached to programs with either the `require` or `use` declarations. Both are similar in that they work like a C `import` command; the difference between the two being that `require` is implemented at run time whereas `use` is implemented at compile time.

Without a specific package declaration all variables are assumed to be in package main. So the statements `$x = 10` and `$main::x = 10` refer to the same variable. Local, or more commonly called "lexical" variables, can be obtained through use of the `my` declaration. This creates a variable that cannot be accessed outside of the block it was declared in. This can be seen in Example 5.

Another scope related concept worth exploring has to do with a Perl construct known as typeglobs. In Perl, it is possible to declare variables of the same name if they have a different type, so `$jordan` and `@jordan` refer to different variables. The Perl symbol table is really a large hash table, and because of that, one name can only point to one value. The uniqueness of different types is accomplished by having each name in the symbol table point to something called a typeglob. A typeglob is essentially a pointer to to one of each different types of variable. Thus, `$jordan` refers to the scalar value of the typeglob `jordan`, and `@jordan` refers to the array value of the typeglob `jordan`. Thus, when a variable is initialized, it creates a typeglob that only has a value for one type, but creating different types with the same name will add more values. It is also possible to

Example 5 Lexical and Global Variables.

```
my $x = "main";
package jordan;
        $x = "jordan";
package matt;
        $x = "matt";
package main;
print $x;            # prints "main", the lexical $x never went out of scope!
print $jordan::x;    # prints nothing, it was never initialized!
```

access the typeglobs and even assign one to point to another, as Example 6 demonstrates. Similarly, changing the value of `$y` would also change the value of `$x`.

Example 6 Typeglobs.

```
$x = 10;
@x = ("jordan"," is great");
*y = *x                          # Assign typeglob y to points to x;
print $y;                        # prints 10
print "@y";                      # prints "jordan is great" (with a newline).
```

3.3 Object Orientation

Perl supports objects, but it is not an object oriented language. That is, it is possible to create and use custom data types that have methods and instance variables, but not in a way that meets the definition of object oriented. The requirements of classification, polymorphism, and data encapsulation are all properties of true object oriented languages that are not strictly satisfied by Perl.

Objects in Perl are created through the use of packages and references. When a reference points to an object, it is given a special type value of whatever it is pointing to. This can be accessed with the `ref($pointer)` command. Consider the output from Example 7.

Example 7 Blessing and reference types.

```
$jordan = "the man";
$great = \$jordan;
print ref($great)."\n";    # prints "SCALAR"
```

The `bless` command allows the programmer to change the type of a reference to an arbitrary type. Therefore, a pointer can be assigned to an array or hash and given any type. The reference then becomes the “object” and the list that it points to are the equivalent of the object’s instance variables. The methods come from a package with a name that (hopefully) matches the type that was just given as the reference. Consider, then, a simple package definition, such as the one given in Example 8.

Example 8 A basic package declaration.

```
package boring

sub display {
print "You have just printed out some text.\n";
}
```

The standard way to call the subroutine would be `boring::display`, followed by any arguments. However, if a reference were blessed with type `boring`, the display subroutine could be invoked like a method with `$reference->display();`. Further, this invocation would pass the reference as an argument, allowing use of the data it points to. In essence, this encapsulates all that an object is: a reference to data that is given a package name as a type. Typically, the package in question will contain a constructor that sets up the data and returns the reference. A more realistic object is given in Example 9.

Example 9 A simple object.

```
package shape;

sub new {
$color=shift; $age = shift;    #shift works on the array of input parameters
$pointer = [ $color, $age ];   # setting up an "anonymous" array.
bless $pointer, "shape";     # changing the type of our pointer.
return $pointer;
}

sub describe {
$shape = shift;
$color =  $shape->[0]; $age = $shape->[1];
print "This shape is a pretty shade of $color and is $age years old.\n";
}

$new_shape = shape::new("blue","100");
$new_shape->describe();
                # "This shape is a pretty shade of blue and is 100 years old."
```

In relation to other object oriented languages, the classification scheme that Perl presents is different. Objects are not defined by stating the instance variables they have; rather, they are defined through simply giving them those variables. If another variable were added to `$new_shape`, it would still be a shape. Similarly, changing the two scalars it references to does not cause `$new_shape` to reference data that is not a shape. Further, if the parameters match, any method can be called on any object. The notions of “has-a” and data encapsulation are almost non-existent.

Building off the shape package from Example 9, Example 10 calls the `describe` subroutine in two different ways, though the output in both cases is the same.

Example 10 Method calls.

```
$new_shape->describe();
shape::describe($new_shape);
        # Output: "This shape is a pretty shade of blue and is 100 years old."
```

Inheritance exists in Perl, but as was demonstrated with the notion of ownership above, it is also not the same as traditional object oriented languages. Parent packages can be specified by adding them to the array `@ISA` at the top of a package. So, for a package "square" that extended shape, the line `@ISA = ("shape");` would be added right after the package declaration. The child package then inherits all of the methods of the parent. Perl supports multiple inheritance, a package can inherit any package except itself or any of its children.

Because the notion of encapsulation is so loose in Perl, a child need not have any of the same instance variables as its parent. Only the methods are inherited by the child, and because of this, it is possible that the parent's methods will not work for the child. Of course, a careful programmer would make sure to give the child the correct instance variables, but this is not a requirement.

Without encapsulation or ownership, and considering Perl's already weak typing system, the notion of polymorphism is out of the question. A carefully written method can manually check that objects are of a specific type, but without this explicit checking, any object with the correct parameters could be passed in. The "objects" in Perl are nothing more than simulated objects, used more for organization than anything else. Certainly, classes can be written to mimic the appropriate checks and encapsulation, but this is not built in.

3.4 Programming Paradigms

Depending on the problem which a program is being written to solve, sometimes the problem is better addressed using one style of programming as compared to another. These styles, known as programming paradigms, each have their own advantages and disadvantages, and are each best suited for certain kinds of problems. Perl, interestingly, has adopted the philosophy that "there's more than one way to do it." That is, depending on the particular problem being solved, Perl is flexible enough to allow such a paradigm to be used, as compared to switching to an alternative programming language to better address the problem, or trying to force the answer out of a less efficient paradigm.

The most commonly seen programming paradigm is the imperative one. Imperative programming consists of statements, each of which can affect a larger, more global state. This type of programming is probably what most modern programmers are familiar with, as languages such as C and Java utilize these to express their instructions. Not surprisingly, Perl programs can easily be written to make use of imperative programming, as many of the examples demonstrate.

That is not to say, however, that imperative programming is the only paradigm that Perl supports. Perl makes use of subroutines and being able to execute them at points in the program. This style of programming, known as procedural programming, stands in contrast to an older and less common style, which makes use of `goto` instructions to execute various segments of code. Many programmers take this paradigm for granted, especially with the advent of object-oriented programming.

Although seemingly related to procedural programming, functional programming has a decidedly different feel for the style of code. In contrast to imperative programming, there is no global state that is updated, and operations that are performed are done through functions operating only on the data which they are passed; this behavior is similar to that of a mathematical function. Although Perl clearly does have a global state to be modified, the programmer can still write programs in a functional manner and ignore the overall global state of the program.

Example 11 Functional programming in action.

```
sub quicksort {
  !@_ ? () : (quicksort(grep {$_ < $_[0]}
                @_[1..$#_]),
              $_[0],
              quicksort(grep {$_ >= $_[0]}
                @_[1..$#_]));
}

print join ' ', quicksort(1,9,7,6,5,3);
# The numbers are sorted, and then printed to the screen
```

Clearly, Perl does support a number of programming paradigms, though by no means are all such paradigms supported internally by Perl itself. Fortunately, though, Perl can be extended through various modules, some of which actually add such functionality to Perl. One such instance is using the module `AI::Prolog` to allow a Perl programmer to utilize the logical paradigm. The syntax is not as clean as native Perl code nor (in this case) native Prolog code. Rather, an instance of a Prolog simulation is created in Perl, and facts can be added to the simulation, as well as that queries can be submitted to it. An example of this can be seen in Example 12. Here, a set of initial facts are stored in `$input`, and this is used to initialize a new Prolog object. With this new object, queries can be submitted, in much the same way as they could be submitted in normal Prolog.

Example 12 Logical programming within Perl.

```
use AI::Prolog;

my $input = <<'PROLOG';
   color(tree, green).
   color(sun, yellow).
   plant(X) :- color(X, green).
PROLOG

my $pro_inst = AI::Prolog->new($input);
$pro_inst->query("plant(sun).");

while (my $results = $pro_inst->results) {
  print $results->[2], "\n";
}
```

Overall, Perl has a considerable range of support for various programming paradigms. Although most programmers would find themselves at home by using the procedural and imperative programming paradigms, Perl is not strictly limited to these. Some programs are more concisely written by using a functional style, or perhaps it is more ideal to use a functional style when the overall state of the program could potentially affect the algorithm. And although logical programming appears to be relatively foreign in comparison to these other paradigms, it can still be useful in sorting facts and extracting information.

4 Text Processing and Shell Integration

Although Perl may have many properties and capabilities similar to other languages, such as C or Java, Perl is best known for its text processing power. There are two major reasons for this. Arguably most important in the eyes of a programmer is simply the lightweight aspect of a Perl program. Unlike C and Java, a Perl program can be executed directly from its sourcecode; both C and Java, like many other programming languages, have a separate compilation phase, separating the sourcecode from the actual executable. A change in a C or Java sourcecode requires recompiling the program in full, in order for the execution to reflect the changes; Perl's execution directly from the sourcecode eliminates this middleman operation. The second reason is that Perl has built-in constructs in the language which makes text processing considerably easier than a similar operation in C or Java.

One of the first differences between Perl and other languages is through their means of accessing information passed to the program via "standard input." Standard input is a concept Unix programmers are very much familiar with; it is used by programs to receive input from the keyboard during execution. In C, such a stream is identified as a special type of file pointer which the program then reads from. In Perl, however, reading from standard input is very much simplified, and can be done in one line and without referring to a file: `while (<>) {}`. This is clearly loop, but the diamond as the argument is a special shorthand for reading from standard input. In this way, reading typed in input or redirected outputs from other programs can be performed easily.

Example 13 Reading standard input with Perl's special syntax.

```
while (<>) {
  if (length($_) > 4) {
     print substr($_, 3);
  }
}
# It checks if the line typed is at least 4 letters long including the newline
# and if it is, it prints out the line, minus the first 3 letters.
```

Other forms of integration between the underlying operating system shell and Perl can be performed as well. If a Perl programmer wishes to execute some external program from within Perl, then make use of the results, this can easily be done by placing the exact external command within a set of backquotes. This says to Perl that the statement contained within the backquotes is an external command, and that it should be executed by the shell. Any output the command produces is sent back to Perl for processing. As such, consider the line of Perl, `@listing = 'ls';`. This line sends the Unix command '`ls`' to the shell, which gets a listing of the contents of the current directory. This listing is then assigned to the array '`@listing`', and due to Perl's context conversions, each line becomes its own element in the array.

Of course, getting various data into a Perl program is one point of contention; another is how to manipulate this data in a manner which is straightforward for a programmer. In order to address this, Perl introduced regular expression handling. The particular syntax used for regular expressions in Perl is actually derived from the Unix command *sed*, in which they were used for much the same purpose. By using regular expressions, a programmer can determine if a line of text has a particular syntax, parse it into pieces, and use those individual pieces to construct a new product. As an example, assume that `$line` contains a line of text. Then the Perl command `$line =~ /^.....a$/` attempts to match `$line` with the structure of the regular expression. The result of such regular expression matching can either be a replacement for `$line`, or it can be used as the condition of an if/else block. The difference simply depends on the particular regular expression used.

Example 14 Using regular expressions and shell commands.

```
@listing = `ls`;

foreach $name (@listing)
{
   if ($name =~ /^[aeiouAEIOU].*[aeiouAEIOU]$/)
   {
      print "Begins/ends with a vowel: ".$name;
   }
}
```

Overall, each of these points mentioned, about getting data into Perl and manipulating it, are ways in which Perl has been able to secure its name as a text manipulation powerhouse. As it turns out, some of these aspects of textual manipulation and shell interactions have been incorporated into other languages. One such instance is Ruby, which allows for execution of shell commands by surrounding the command with backquotes, reading the standard input stream by using a similar syntax, and uses regular expressions in much the same way as Perl does. In this sense, Perl has set a standard which other languages attempt to match.

5 Conclusion

Having now explored the major features of Perl, its strengths and weaknesses can now be examined more clearly. Perl's simplicity flows from its basic types and their ease of use. Memory does not need to be managed, variables can be created on the fly, and most natural operations do what one would expect. Perl's expressiveness is a result of its simplicity. Complex data structures, from arrays to files, are easy to work with and manipulate, and the many built-in functions make this even easier. Perl's compiler/interpreter was written in C, making it portable to almost every operating system and architecture. Finally, Perl's package system makes large scale programming projects possible.

This ease and simplicity also contributes to the dangers within the language. Most natural operations do work, though not always as intended. As was seen with the concatenation example, virtually anything can be done in the language and not produce an error. However, the behavior might not be what was expected. This is compounded by Perl's lack of uniformity; depending on the context, different operators might function differently. Perl supports objects and packages, which should be useful in large scale projects. However, lack of data encapsulation and the aforementioned safety issues make large scale programs challenging.

Perl is an amazing language to program in. It has many powerful features, such as ease of text processing, shell functionality, and module support. Programs are easy to write and at the same time effective, what more could you want?

References

[1] Christiansen, Tom. ”Compilation vs Interpretation in Perl.” Perl.com: The Source for Perl. 01 June 1996. Perl Consulting and Training. 28 April 2008 <http://www.perl.com/doc/FMTEYEWTK/comp-vs-interp.html>.

[2] ”Functional Programming.” *Perl Design Patterns Wiki.* 30 April 2008. http://perldesignpatterns.com/?FunctionalProgramming

[3] Lehmann, Marc. ”Coro - coroutine process abstraction.” The CPAN Search Site. 10 May 2008. Comprehensive Perl Archive Network. 13 May 2008 <http://search.cpan.org/dist/Coro/Coro.pm>.

[4] Marshall, A. Dave. ”Bless the Hash and Pass the Reference.” Practical Perl Programming. 2005. Cardiff School of Computer Science. 14 April 2008 <http://www.cs.cf.ac.uk/Dave/PERL/node124.html>.

[5] ”Perl.” *Wikipedia, The Free Encyclopedia.* 13 May 2008, 09:47 UTC. Wikimedia Foundation, Inc. 14 May 2008 <http://en.wikipedia.org/w/index.php?title=Perl&oldid=212074559>.

[6] Ragle, Dan. ”Perl Subroutine Primer.” Mother of Perl: Free Perl Tutorials, Tips, Lessons, Tricks, and Code. 20 Feburary 2006. Jupitermedia Corp.. 23 April 2008 <http://www.webreference.com/programming/perl/subroutines/>.

[7] Srinivasan, Sriram. Advanced Perl Programming. O’Reilly & Associates, Inc., 1997.

[8] Sugalski, Dan et al. ”perlthrtut.” Perl version 5.10.0 documentation. 21 March 2008. 14 May 2008 <http://perldoc.perl.org/perlthrtut.html>.

[9] ”The Perl CD Bookshelf.” 1999. O’Reilly & Associates. 1 May 2008 <http://www.unix.org.ua/orelly/perl/index.htm>.

[10] Wall, Larry, and Randal Schwartz. Programming Perl. 1st ed. Sebastopol, CA: O’Reilly & Associates, Inc., 1991.

[11] Wall, Larry, Tom Christiansen & Jon Orwant. ”Chapter 18: Compiling.” Programming Perl, 3rd Edition. July 2000. O’Reilly & Associates, Inc.. 20 April 2008 <http://www.oreilly.com/catalog/pperl3/chapter/ch18.html>.