PERL: Principles and Internals

Matti Koskimies

19th April 2004

Principles of Programming Languages
Spring 2004
Group 1 (Pietu Pohjalainen)
Department of Computer Science
University of Helsinki

Contents

Part I
Design principles

1 PERL philosophy

PERL has a very distinct history of evolution in that it was created not so much out of a vision of a novel methodology to tackling a problem domain, but rather as a swiss-army knife, combining many existing tools into a single, straightforward package. Therefore it combines features not only from other programming languages like C, FORTRAN and PASCAL, but also from many core Unix tools such as `sed`, `awk` and `grep`. It is unlikely that its creator, Larry Wall, ever had a singular objective when deciding on the tool's features, but instead just borrowed and combined features he saw as contributing something beneficial to the language as he went along, letting the end result form itself. This process was largely due to a key principle, coined by Wall in [Wal00] as "easy things should be easy, and hard things should be possible."

Partly due to its design process, PERL is notorious for disregarding such traditionally respected goals as orthogonality and explicitness in favour of an approach that in some respects attempts the complete opposite. This can clearly be deduced from the philosophical slogan perhaps most commonly attributed to PERL: *"There is more than one way to do it"* [Wal00]. Such a design principle is typically shunned by both traditional and modern language designers who tend to avoid providing overlapping models of accomplishing the same functionality at all cost. Unorthogonal features are seen as impure and degenerative; PERL embraces such features.

Not only is PERL unorthogonal by nature, it would seem to make things worse by adding an additional obfuscation layer: the so-called *default input and pattern-matching space*, more commonly addressed as `$_`. This global variable is automatically assigned values depending on the context; most typically it takes on the individual values of a list which is being iterated. The (mis)use of such implicit assignments can cause code to be illegible to even experienced PERL coders.

There is, however, an underlying logic to these seemingly unwanted features. It stems not from the design of programming languages, but from that of natural languages which provide an immense variety of expressions as a result of centuries or millennia of evolution. Similarly, PERL syntax is intended to be as flexible and expressive as possible, allowing a program flow to follow the programmer's mindset more smoothly than languages with a less forgiving syntax.

2 Basic syntax

Much of PERL syntax also has analogies to the grammar of natural languages. Variables are seen as nouns of which there are singular (scalar) and plural (array, hash) occurrences. Subroutines are little more than verbs specifying actions. The syntax of the available variables is shown in table 1.

Type	Character	Example	Use
scalar	`$`	`$age`	An individual value (number or string)
array	`@`	`@weekdays`	A list of values, keyed or numbered
hash	`%`	`%users`	A group of values, keyed by string
subroutine	`&`	`&convert`	A callable chunk of PERL code
typeglob	`*`	`*current`	Everything named `current`

Table 1: Variable syntax [Wal00]

Another linguistic feature of PERL is its strive to be context-sensitive whenever possible. Scalars may contain strings or numbers, but it is in the context of an expression that their type is determined. Consider the example provided in *Programming Perl* [Wal00]:

```
$camels = '123';
print $camels + 1, "\n";
```

As a result of the operations, 124 is printed out. This is because even though the value of `$camels` is originally a string, it will adapt its form depending on the context it is used in. In the above example, the + operation is applied to the value. As the + operation is numeric by nature, any values passed to it are interpreted as numbers. However, print uses a string context so the value of `$camels` is converted back to a string. The appending newline sign is already in the right context and needs no further conversion.

The strive for context-sensitivity has very far-reaching implications in PERL. Subroutines can return different values depending on whether they're called in scalar or list (array or hash) context. Similarly, list variables themselves may be evaluated in a scalar context, yielding the size of the list as a result.

On the other hand, PERL is also a mathematical language which includes a large amount of functions and operators, most of which have been incorporated from other languages. For instance, the arithmetic operators include ** and % as used in C, whereas numeric functions include hex() and oct() as used in PASCAL.

Another core characteristic of PERL is its tight system integration. As the languages origins are in the UNIX environment, most of the system interoperability bears a close resemblance to similar features especially in UNIX shells such as `bash` and `csh`. For instance, launching an external executable can be done in any of the following methods, the two latter of which have their origins in shell scripts and the standard C library:

```
$exitcode = system("/bin/foo");
$result = `/bin/foo param1 param2`;
exec("/bin/foo","param1","param2);
```

Another typical example of powerful system integration is PERL's I/O functionality. Consider the amount of lines required to read a file into an array of strings in JAVA or any other compiled language. In PERL, this can be done in three lines:

```
open(INPUTFILE,"textfile.txt");
my @filecontents = <INPUTFILE>;
close(INPUTFILE);
```

There is plenty more easily recogniseable system level interaction, some of which – such as the fork() call – has made porting PERL into other systems challenging. However, for almost all of it there exists portable library equivalents, especially in the `POSIX` library package.

3 Strings and pattern matching

The origins of PERL as a combinatory replacement for `sed` and `awk` are perhaps most obvious in its strong string manipulation and pattern matching features. Especially the latter have become a standard which other languages duplicate and are measured by.

3.1 Scalar operations

PERL provides a variety of operations that can be performed on any scalar content, although most of them will interpret the content as a string. Compiled languages, JAVA included, seem very cumbersome in comparison as they will often offer no option but to build the manipulation algorithm from scratch, often involving going through strings one character at a time. PERL programmers would not need to resort to such low-level handling of strings: not only does PERL provide a comprehensive set of functions for manipulating strings, they are combined with the power of regular expressions to ensure that the programmer will never need to come by his own solutions to extend the available toolset. Table 2 lists some of these functions along with example usage.

3.2 Regular expressions

Pattern matching provides a method for efficiently filtering out specific information from a potentially large amount of data using, using a set of formally defined rules specified in a *regular expression* (or regex for short). These rules are used by PERL's regular expression engine to determine whether the provided pattern matches the data.

Matching and substituting matches are such key features of PERL that they have been given their own `=~` operator. By default, the operator performs matching based on the succeeding regex, normally given between two forward

Function	Purpose	Usage
chomp	Purge trailing newline	`$fileline = <STDIN>; chomp $fileline`
chop	Purge last character	`if ($line =~ /\..{4}$/) { chop $line; }`
grep	Grep for string or pattern in array	`my @matches = grep "John" @names;`
join	Join array into scalar according to rule	`my $flat = join("\n",@lines);`
lc	Convert to lowercase	`my $user = substr($firstname, 0,1) .  lc $lastname;`
length	Length of string	`if (length $name > 30) { ...  }`
pack	Pack string according to rule	`$out = pack "H8", "5065726c";`
reverse	Reverse string	`my $devilspeak = reverse $monroelyrics;`
split	Split scalar into array according to rule	`my @tsvline = split("\t",$tsv);`
sprintf	Print formatted string	`sprintf("%.3d",$decnum);`
substr	Get a substring of this string	`$firstchar = substr($a,0,1);`
uc	Convert to uppercase	`my $shout = uc($whisper);`

Table 2: Some scalar manipulation functions in PERL

slashes. For substitution, an additional s is inserted in front of the first slash; for the sake of completeness an m can also be used to specify a matching operation but it can rarely be seen in use. The below expressions will produce a printout consisting of the string "anything and anything else":

```
my $match = "something and something else";
if ($match =~ /something\welse/) {
  $match =~ s/some/any/g;
  print $match;
}
```

4 Objects

PERL was not originally an object-oriented language. It is no surprise, then, that the approach it takes to object-oriented features – one which appears to attempt a minimal impact on existing syntax and keywords – has not won much acclaim; in fact it has been criticised extensively. By no means is PERL a match to PYTHON or other more essentially object-focused languages in regard to syntax intuitiveness or the level of abstraction.

Regardless of form, the functional aspects of object-oriented programming in PERL are rich. Polymorphism, multiple inheritance, dynamic feature replacement, operator overloading and introspection are all available, most of them without any additional fuss. The more exotic of these features are not discussed here; see [Sri97, Wal00] for further reference.

Algorithm 1 shows a trivial hierarchy of two objects, `Vehicle` and `Car`, each with certain properties and functionality. As the example shows, inheriting features from other classes is simply a matter of adding them to the `@ISA` list.

While the end result may seem messy at best, the foundations of the PERL object system implementation are very pragmatic by nature [Wal00]:

Algorithm 1 Simple object hierarchy in PERL

```
package Vehicle;
sub new {
  my $invocant = shift;
  my $class = ref($invocant) || $invocant;
  my $self = {
    color => "blank",
    passengers => -1,
    owner => "John Doe",
    @_,            # Override previous attributes
  }
  return bless $self, $class;
}

package Car;
our @ISA = "Vehicle";
sub new {
  my $invocant = shift;
  my $class = ref($invocant) || $invocant;
  my $self = SUPER::new(@_);
  $self->{passengers} = 4;
}
sub honk {
  print "Beepbeep!";
}
```

1. Objects are referents.[1]
2. Classes are packages.
3. Methods are subroutines.

The strategy seems very convenient in that the core object-oriented concepts map well into these existing language constructs, introduction of new syntax is (to a large extent) avoided and existing syntax is preserved. However, these principles do not yet shed light on how objects come to existence and where their data is stored.

As shown in algorithm 1, constructing an object does require one additional concept: *blessing*. Conceptually, the act of blessing binds an object to its class; in practice, an ordinary reference is turned into an object reference. The type of the reference doesn't matter; typically it would be a hash, but it could just as well be an array. Hashes are normally used because the contents of the reference are in essence the instance-specific variables. As such, the key-value structure of a hash provides a way to give each variable a name. Instead of actually passing around objects, then, just the internal data of the object is stored in the reference. Each time a class's method is called through an object reference,

[1]The choice to not use the word reference here is deliberate, as the object really is what's behind the reference, not the reference itself. Another way of expressing the issue is that objects are always used through references.

the first parameter to the method is always the reference itself. The method can then inspect by itself whether it was called in class or instance context and act accordingly. The object reference passed to it will provide access to the object's variable fields when needed.

Part II
Internals

As a scripting language, what happens behind the curtains in PERL is decidedly different from what is typical in compiled languages. This part of the seminar work is a superficial review of the methods and solutions used to accomplish the features of PERL.

5 Architecture

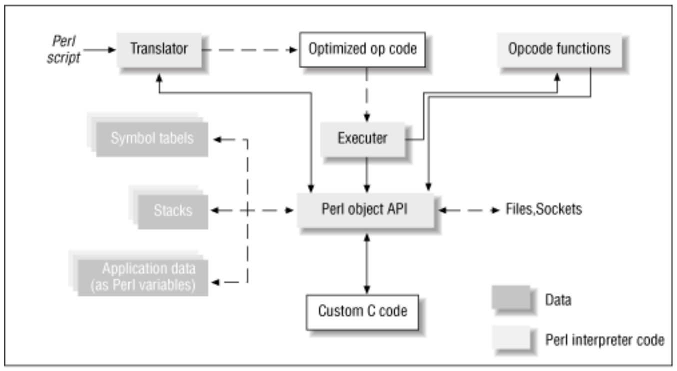

Figure 1: Snapshot of a running system

5.1 PERL data structures

The PERL Object API, shown in Figure 1, is a common interface for any actions taken on internal data structures such as variables, symbol tables and files.

Value type	Purpose
SV	Scalar value
AV	Array value
HV	Hash value
CV	Code value
GV	Glob value (typeglob)
RV	Reference value

Table 3: PERL values

PERL distinguishes between *global* and *lexical* (or local) variables. Global variables in the inner workings of PERL are seen as name-value pairs and can be categorised according to Table 3. The symbol table is nothing more than an HV (hash value) mapping identifiers (variable names) to their values. As can be seen from figure 2, the identifiers appear as typeglobs (the `*` notation) in the symbol table, with pointers to different value types. This two-level hierarchy is necessary because PERL allows different types of variables to use the same identifier even though they are completely independent; e.g. the values of a scalar named `$bag` and an array named `@bag` have nothing to do with eachother.

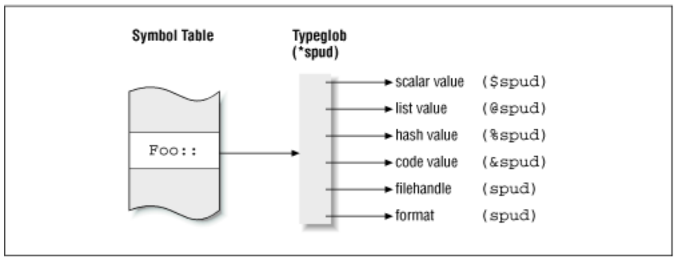

Figure 2: Structure of the symbol table

Lexical variables are always prepended by a `my` keyword and do **not** appear in the symbol table. Instead, blocks and subroutines use what is known as a *scratchpad*, an array where each variable declared within the block gets its own slot; in contrast to global variables, variables of different type using the same identifier all get their own slot.

Global variables can be made to act like local ones using the `local` keyword. In essence, the variable's value is localised for the remainder of the block, after which the original value is restored. However, the temporary value is not only visible to the block in which it was created but also to any subroutines called from that block. Such a feature is called *dynamic scoping* and can cause some unwanted effects; use of it in place of my is strongly discouraged. However, the features may be useful in temporarily replacing the values of built-in variables. As an example, the diamond operator expects to find its arguments in the global @ARGV variable. The local operator could be practical when iterating through other files than those supplied at the command prompt: when the block ends, the original command-line parameters are restored.

The PERL argument stack is simply an AV (array value) onto which subroutine arguments are inserted by the caller and from which they are subsequently picked up by the callee. Once the subroutine exits, whatever return value it has is also pushed to the argument stack. Contrary to the C stack though, there are several stacks in PERL for different purposes such as temporary variables, loop iterators, opcodes etc.

All I/O operations in PERL are handled by the PerlIO abstraction interface, which is little more than a wrapper for the stdio and sfio libraries.

5.2 Translator

The translator component is perhaps the most complex of the Perl subsystems. It translates script code into a tree structure consisting of *opcodes*, similar to Java's bytecode in that they're executed by a virtual machine. However, the opcodes are of a much higher abstraction level than the product of the Java compiler which is more akin to the instructions of a RISC machine. Many of the opcodes – pattern matching, push, pop to name a few – have direct counterparts in script code. Opcodes, on the other hand, are a far cry from machine language:

> Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. [Sri97]

The translator consists of a hand-coded lexer (`toke.c`), a `yacc` -based parser (`perly.y`) and a code generator (`op.c`). Furthermore, regular expressions are converted into an internal format using `regcomp.c`.

5.3 Executor

The executor is Perl's virtual machine: it iterates through the execution chain laid out in the syntax tree, calling the corresponding opcodes sequentially. The dynamic nature of Perl shows in that the final sequence can not be predetermined: each opcode returns to the executor a link to the one to be executed next, which may or may not be the one originally specified as the successor at compile-time.

References

[Sco00] Scott, Michael Lee: *Programming Language Pragmatics.* First edition, 2000. ISBN 1-55860-578-9.

[Sri97] Srinivasan, Sriram: *Advanced Perl Programming.* First Edition, August 1997. ISBN 1-56592-220-4

[Wal00] Wall, Larry, Christiansen, Tom, Orwant, Jon: *Programming Perl.* Third edition, July 2000. ISBN 0-596-00027-8.