
Table of Contents

Rindolf Specification Document	1
Version	1
Introduction	1
Author	1
Philosophy	1
The Assumptions behind Rindolf	2
Explanation	2
Analogy with the Perl 4/Perl 5 Conversion	3
Analogy with C++ and ANSI C	3
How to handle the Competition?	4
Object Oriented Programming Features	5
Package Level Namespace	5
Classes as First Order Objects	5
expand and extend	6
Defining New Operators	7
New Additions to the language	8
A Fundamental I/O Subsystem	8
A Better Hash Construct	8
Proper Tail Recursion	8
Examples	8
Nested Variable to Hold the Regex Search Results	9
LISP-like Features	10
Re-usable Virtual Machines Instances	10

Rindolf Specification Document

Version

0.3.5

Introduction

This document is meant to describe the Rindolf programming language which is a dialect of Perl. It describes the main changes between it and Perl version 5.6.x.

Good knowledge of Perl is assumed.

Author

Shlomi Fish

E-mail: shlomif@shlomifish.org

Homepage: <http://www.shlomifish.org/>

Philosophy

The purpose of this section is to explain the motivation behind Rindolf, what it is meant to accomplish, and what not. This section should be consulted before one wishes to add new functionality to Rindolf, in order to see if it corresponds with the philosophy.

The Assumptions behind Rindolf

1. Perl is a *family* of programming languages.
2. Perl 5 is a good language.
3. Perl 5 is not good enough for some purposes.
4. Perl 6, by not being backward compatible, does not provide a real answer to the Perl 5 deficiencies.
5. Rindolf aims to be an improved dialect of Perl compatible to Perl 5.

Explanation

OK, here's the deal: if Perl 6 would have been better designed, less bloated with superfluous features, and backward-compatible with Perl 5, I would not have started Rindolf. In fact, I consider Rindolf an anti-thesis to Perl 6. Thus, don't wonder if I criticize Perl 6 a lot, because it is the reason for starting Rindolf in the first place.

Perl 5 is a good language. Yet, it is facing a problem: competition. Perl faces competition from Python, PHP and to lesser extent Ruby, Tcl and other less dominant languages vowing for the so-called "scripting languages category". There's nothing wrong with this competition, and if someone is content programming in another language, there's no reason to tell him to use Perl instead. Still, it would not be good if Perl 5 lost ground due to the fact that it lacked features, which other languages possessed in a straightforward manner.

Now, Perl 6 is the wrong solution for the problem. As Perl 5 served as a handy scripting, glue and integration language in the past years, which saw an explosion of the Internet, of Linux, and of computing in general, there are now millions of lines of Perl 5 code written by hundreds of thousands of programmers. (perhaps even milliards LOCs written by millions of programmers - it's hard for me to approximate). These pieces of code do their job, work very nicely and are sometimes tweaked and modified as time goes by.

What is present in CPAN is just the tip of the iceberg. We can expect many modules in CPAN to have ad-hoc code that does the same job in many other places around the world. (I know I wrote a lot of code, that I later found out that there is a CPAN module that does exactly that). Furthermore, and equally importantly we can expect that there are now millions of Perl 5 programmers of varying skill and knowledge who constantly maintain such code or write new one for their own purposes.

Now, here comes a rhetorical question: exactly how are all of them going to convert to Perl 6, which is incredibly different and not compatible with Perl 5?

The answer is they are not going to. They are going to stick to Perl 5. Converters, you say? They are not going to work for several reasons. First of all, the programmers themselves are used to Perl 5. As we all know, everyone has his own Perl 5 style which is different than others.¹ Teaching them all Perl 6 is a recipe for failure. Secondly, a converter will probably convert your code to something crude and hard to manage. (if it will convert it correctly at all - eval "" anyone?). Lastly, people will be very reluctant to use the converter, if the code is already working in Perl 5. "It's working - don't touch". Perl 5 is probably going to be around for a while, so why bother converting a perfectly good code to the incompatible, new-fangled Perl 6?

And that's where Rindolf kicks in. Rindolf is a Perl dialect that is fully backwards compatible with Perl 5, yet offers some features not present there. Running "rindolf test.pl" will 99% of the time yield the same results as running "perl test.pl". A person can take a code written for perl5, use it as is with Rindolf, and gradually add Rindolf-specific features should he wish to.

¹And it is possible that a programmer writing the same script twice will not write the same script!

Rindolf maintains backwards-compatibility because Perl 5 is good enough as a basis for a language. More importantly, it is a language that is in wide use today, and people depend on.

Analogy with the Perl 4/Perl 5 Conversion

A few years ago, Larry Wall decided to dump Perl 4 for good and to create Perl 5, which was not backward compatible. Right now, the lack of backwards compatibility in Perl 6 is rationalized on similar notes. The situation, however, is completely different.

Perl 4 was a system and scripting language that still originated from the same Perl 1 codebase. It did not have half the features that Perl 5 has, and it was very hard to write maintainable code in it. I've seen Perl 4-like code written in Perl 5, and it was not a pretty sight. (I ended up Perl5-ing it). Perl 4 was an ad-hoc language used by a (relatively to nowadays small) number of UNIX sys-admins and users and was not much more than an awk with an attitude.

Furthermore, Perl 4 code could have been converted with some effort to Perl 5. Perl 5, in fact, still includes some operators only for backwards compatibility to Perl 4.

The situation today is very different. Perl 5 is widely deployed and used by many more users around the world. It is not only used for system scripts, but for automating web processes (in thousands of sites used by millions of online users), for complex text processing, database access, GUI programming (take the Mandrake system utilities and installer for example), games (Frozen-Bubble is a notable example), numerical processing (PerlDL) and almost everything else under the Sun that does not require very fast low-level processing.

Furthermore, it is very usable and supports most of the modern programming paradigms very well: Object-Oriented Programming, Functional Programming, Exceptions, text-processing, garbage collection, nested data structures, arbitrarily-sized data, etc. I'm not saying it's a perfect language, but inventing something entirely different, just to solve its deficiencies is sure to make many users (if not most) unhappy.

Now is the point for incremental renovation. Don't throw everything away. Start with what you have and build on there, without breaking everything. That's exactly what Rindolf aims to be.

Analogy with C++ and ANSI C

The analogy between Rindolf and Perl 5 and C++ and ANSI C is naturally something that needs to be addressed. C++ has seen wide use in the industry, while has almost been completely rejected by the open-source community who stucked to the bare ISO C 89'. Even some software houses maintain ANSI C codebases of various sizes, which they do not indeed to convert to C++. Is Rindolf going to have the same fate?

The answer is that it makes no difference this way or another. I don't mind people sticking to the bare Perl 5 subset because Perl 5 is fine as it is. In fact, considering the Parrot architecture, I may write the Rindolf compiler in Perl 5 and keep it in Perl 5. (assuming rindolf will be based on Parrot). I believe that C++ could have had much more success, if it supported a saner subset of functionality, and did not push the standard beyond the limits of functionality, that takes compilers a lot of time to support properly.

Single-inheritance is useful (it is nicer than the Gtk+-style implementation in software), as are templates, operator overloading and friend functions. But at the moment, Standard C++ has become a completely different language than ANSI C, which is still backwards compatible. Mozilla, for instance, restricts itself to only a very small subset of the functionality to maintain compatibility with different broken compilers. (I also know someone who used friend classes or functions in his code, which worked well in gcc, and later broken in a different Solaris proprietary compiler)

Rindolf will still essentially be Perl 5-like with a Perl 5 feel and behaviour. Moreover, C++ has proved of some use to many projects, and is still commonly used. As much as there is a hype about Java and .NET,

they are not compatible with ANSI C, and so aren't actually a replacement for C++. ² If Perl 5 is ANSI C, then Perl 6 is going to be something like Java. Rindolf will be C++, and the analogy is perfectly fine.

How to handle the Competition?

When one is facing a competition, one has to admit it, to understand it is a good thing, but also to try not to lose ground. Many times two competitors each gain new users. Many times users are gained by one on the expense of the other. At the moment, Perl faces competition from Python, Ruby, Tcl, O'Caml and some other minor players. Our most serious competitor is Python.

Now if you ask a common programmer who knows Perl and Python well enough what he thinks of both languages there are several possible answers. One is "I *love* perl and despise everything that is Python." (I fall into this camp) One is "Python rocks, while Perl is a hideous, ugly inconsistent language". (I know some people who think that way). And another common one is "Perl is OK. Python is OK. Perl is good for some things, python for others. I like them both". A less common one is "They are both stupid. C++/Java/LISP is the ultimate language!".

Now it's hard to tell which person will tell you which. It depends on the character of the Person, what he was used to before, and many other factors. My mother language was BASIC, and so I found Perl highly attractive. Later on when I learned other paradigms (The SICP stuff, Lambda Calculus, Object Oriented Programming, FP, nested data structures, etc.) I could easily find them in Perl 5. But then there may be other BASIC hackers who would prefer Python or Ruby.

There are dog people who prefer Perl, and there are dog people who prefer Python. There are cat people who prefer Perl (like me), and there are cat people who prefer Python. Now, one can notice that the fact that Perl resembles a natural language, and it is possible to write with it in various styles (even mixing several styles in one program) make it highly attractive to people who like think about *how they are going to write the code*, and translate it naturally into code. People who like a strict syntax and conventions, an "everything is an object" concept, lack of "inelegant" features, and the "There's One Way to Do it" dogma prefer Python. Some people accept both opposing camps and like both languages

Many of the members of the Python camp are loud, keep professing the supposed advantages of its language, how maintainable the code can become, and how everyone is capable of understanding it. While Python has many features Perl does not yet have, it also does not have many others (here documents, labeled loops with breaking and continuing, a continue block, the use strict pragma and so forth). It also overloads every operator to many things: + can be used for both adding numbers, concatenating strings and concatenating arrays, three uses that are three different operators in Perl. The Perl people on the other hand take the "if you like programming in Python/C/C++/Java/Ruby/whatever, then hack on and stay cool".

I have already met two relative very young newbies whom I told that they should start with learning Perl, and they said "Perl? Why not Python?" Apparently, some people are under the impression that Perl is an inferior language to Python. It's very easy to get it. Eric Raymond recommends in the [How to become a Hacker](#) HOWTO to start with learning Python and then learn C/C++, Perl, LISP, Shell and Java. (and afterwards become familiar with other languages that are of lesser significance to the world). I would recommend someone to start with Perl instead, but this is a difference of opinions between me and Raymond, which is not productive to delve on. (he took the time to write the HOWTO and so he gets to endorse his particular favourite)

My point in making Rindolf is to make sure it has most of the important features Python (and similar languages like Ruby, Tcl or Haskell) has and Perl does not. I'd hate to lose ground because someone tried to do something in Perl and did not. However, as opposed to Perl 5, I'm not going to be concerned with Java (enough to make the arrow operator a "."), C++ (which supports Object Oriented Programming roughly

² I did in fact convert an entire ANSI C codebase (MikMod) to Java through several stages of C++. This took a lot of time, and the resultant code was completely not compatible with the Java conventions. If I had to make MikMod play nice with C++, all I needed were a few extern "C" declarations.

as much as COBOL supports Functional Programming), C#, Visual Basic, Smalltalk or any language that is nowhere near Perl and does not compete for the same niche.

The Perl niche is high-level languages optimized for writing a lot of code quickly, commonly referred to as "scripting languages". These are the languages that most people like for their day to day hacking, that can be used as substitute for shell scripts (like I did when I first worked as a web-technician on UNIX), and can be used for GUI programming, numerical programming (with some extensions), simple games, text processing (naturally), web scripting, automating system tasks, and almost anything where you can afford to spend some overhead and don't need every single CPU cycle. Perl 6 from my impression aims to feel the niche for every possible use under the sun. But we cannot invent a language that will be good for anything, just as we cannot invent such a screw-driver.

Object Oriented Programming Features

Package Level Namespace

Rindolf will support the class `MyClass { ... }` construct to designate package and file level namespaces. Its advantage over package would be that it can be nested:

```
class Hello {
  class Good {
    sub my_func
    {
      print "my_func is called!\n";
    }
  };
  sub another_func
  {
    print "another_func is called!\n";
  }
};

Hello::Good::my_func();
Hello::another_func();
#
# prints:
# my_func is called!
# another_func is called!
```

This should make it more convenient that specifying package `Hello;` and package `Hello::Good;` and so forth.

Generally such classes are considered lightweight namespaces. They are visible to an outside package, however, may override other "global" packages. Moreover, an external package will first look for global packages before them.

Classes as First Order Objects

In similar to the `sub { ... }` syntax it would be possible to do a `$myclass = class { ... }`, construct and get a reusable class as a scalar. (=first order object). In Perl 5, namespaces are global and cannot be created on the fly without some namespace collision prevention Voodoo. In Rindolf, such classes can simply be assigned to variable. (`bless` and similar operators will accept such classes as their second argument):

```

my $myclass = class {
  sub new {
    my $class = shift;
    my $self = {};
    bless($self, $class);
    #
    $self->{'counter'} = 10;
    pt_return $self;
  }
  sub decrement
  {
    my $self = shift;
    sleep(1);
    my $val = --$self->{'counter'};
    print "$val\n";
    if ($val == 0)
    {
      die "Take off!";
    }
  }
};

my $countdown = $myclass->new();
for(;;)
{
  $countdown->decrement();
}

```

This class references will also be accepted by @ISA and all other operators requiring class names.

expand and extend

Rindolf will provide two operators named `expand` and `extend` that would enable manipulating such namespaces efficiently.

`expand` takes an entire class and dumps a cope of all of its package-scope symbols into the current namespace. `extend` takes a class reference and extends with another class. Together they can be used to manipulate classes.

Here is a similar example with some `expand` and `extend` magic:

```

my $class1 = class {
  sub hello {
    my $you = shift;
    print "Hello $you!\n";
  }
};

my $class2 = class {
  sub goodbye {
    my $you = shift;
    print "Goodbye!\n";
  }
};

```

```

    }
};

class You {
    expand($class1);
    expand($class2);
    sub message
    {
        my $you = shift;
        my $message = shift;
        hello($you);
        print "$message\n";
        goodbye($you);
    }
};

my $object_class = class {
    sub new
    {
        my $class = shift;
        my $name = shift;
        my $self = {};
        bless($self, $class);
        $self->{'name'} = $name;
        pt_return $self;
    }
    sub msg
    {
        my $self = shift;
        my $message = shift;

        pt_return message($self->{'name'}, $msg);
    }
};

extend $object_class, typeref{CLASS}{YOU};

my $person = $object_class->new("Muli");
$person->msg("Time to track system calls!");

```

Note the `typeref` operator which is a generic way of casting references, that is introduced due to the fact that we are running out of special characters in the keyboard (:-)). It will be available for hashes, arrays and others language primitives.

Defining New Operators

In Rindolf it would be possible to define completely new operator and modify the syntax and grammar of the language on the fly. Thus, the PerlDL guys can easily define the more Matlab like `.*` and `.*=` instead of perl's `*` and `dump x` in favour of `*`.

At the moment perl5 uses `lex` and `yacc` which I am almost certain will not allow to define new operators. However, I believe such change is something that I've always missed in C++ and Perl, and I really like about Haskell. So, it is going to be implemented.

New Additions to the language

A Fundamental I/O Subsystem

perl 5 depends on `typeglobs` to initialize filehandles, and on `local` to localize them. However, this does not fall very nicely with Perl's lexical scoping, which makes it necessary to use such modules as `FileHandle` which behave more lexically, but may not be well-supported.

Rindolf will have a fundamental I/O subsystem which I dub `aFile_*` after its function names. Generally, this subsystem is not meant to be used directly (albeit it can be) and will be available for those writing low-level file manipulation APIs.

This filesystem will have support for most basic facilities (File I/O, sockets, pipes, `ioctl`'s and `fcntl`'s, asynchronous I/O, etc) and should be flexible enough. `Typeglobs` will use it, as will all other modules (directly or indirectly) and it will behave in a completely Perl 5-ish manner.

A Better Hash Construct

Rindolf will supply an internally coded hash that will support most of the hash extensions (such as `StrictHash`, `Tie::RefHash`). I believe Perl 5 already has a pragma that can auto-magically initialize any hash declared with `my` with a `tie` statement, so this hash can be made to be used by default.

Proper Tail Recursion

The new statement `pt_pt_return` (short for "point `pt_return`") will behave in a similar manner to `pt_return` only will be proper tail recursive if possible.

The new statement `line_pt_return` will always be non-proper tail recursive. A pragma `use Recursion` will exist that will enable to decide how a plain `pt_return` will behave, and should be very flexible in doing that.

I give a choice between both types of tail recursion because proper tail recursion is faster and consumes $O(1)$ memory while non proper tail recursion is easier to debug.

Examples

Here's an example using `pt_return`:

```
sub gcd
{
    my ($a,$b) = @_;
    if ($b > $a)
    {
        pt_return gcd($b,$a);
    }
    if ($b == 0)
    {
        return $a;
    }
    else
    {
        pt_return gcd($b, $a % $b);
    }
}
```

```
}  
  
print gcd(@ARGV[0 .. 1]), "\n";
```

Here's the same example, using the Recursion module:

```
#!/usr/bin/perl -w  
  
use Recursion;  
  
BEGIN  
{  
    Recursion::set_default('pt');  
}  
  
sub gcd  
{  
    my ($a,$b) = @_;  
    if ($b > $a)  
    {  
        return gcd($b,$a);  
    }  
    if ($b == 0)  
    {  
        return $a;  
    }  
    else  
    {  
        return gcd($b, $a % $b);  
    }  
}  
  
print gcd(@ARGV[0 .. 1]), "\n";
```

Nested Variable to Hold the Regex Search Results

At the moment, querying the results of a regular expression pattern match using the \$0, \$1, \$2 variables leaves a lot to be desired. As another way of doing things, Rindolf will provide a \$^RR variable which would be a Perl 5 data-structure that is nested to a necessary depth to contain the results of the pattern and sub-pattern matches.

Here's an example:

```
#!/usr/bin/perl  
  
use string;  
  
my $string = "[ (5,6), (7,3), (8,9)], [(5,7), (2,10)], [(100,92),(14,8)]";  
  
# Define a pattern for (x,y)  
my $pt_pattern = "\s*\(\s*(\d+)\s*,\s*(\d+)\s*\)\s*";
```

```
# Define a pattern for an array of points [pt1,pt2,pt3,pt4]
my $array_pattern = "\[(?:${pt_pattern}\s*,)*${pt_pattern}\]";
# Define a pattern for an array of arrays of points arr1,arr2,arr3
my $whole_pattern = "(?:${array_pattern}\s*,)*${array_pattern}"
$string =~ /$whole_pattern/;

my $results = $^RR;

print "a[0][2].x = ", $results->[0]->[2]->[0], "\n"; # prints 8
print "a[1][1].y = ", $results->[1]->[1]->[1], "\n"; # prints 10
```

LISP-like Features

Re-usable Virtual Machines Instances