Perl in Ten Minutes

Spencer Tipping

October 26, 2016

Contents

1 Perl is not a good language

Python, Ruby, and even Javascript were designed to be good languages – and just as importantly, to *feel* like good languages. Each embraces the politically correct notion that values are objects by default, distances itself from UNIX-as-a-ground-truth, and has a short history that it's willing to revise or forget. These languages are convenient and inoffensive by principle because that was the currency that made them viable.

Perl is different.

In today's world it's a neo-noir character dropped into a Superman comic; but that's only true because it changed our collective notion of what an accessible scripting language should look like.[1] People often accuse Perl of having no design principles; it's "line noise," pragmatic over consistent. This is superficially true, but at a deeper level Perl is uncompromisingly principled in ways that most other languages aren't. Perl isn't good; it's complicated, and if you don't know it yet, it will probably change your idea of what a good language should be.[2]

[1] To get a sense of the historical context it came from, consider that it and TCL were contemporaries and that both Python and Ruby came late enough to inherit a strong OO influence from Java. TCL was a brilliant language in its own way, but even more misunderstood than Perl – and it didn't go on to shape the future the way Perl did.

[2] Along these lines, I continue to insist that Perl 6 never happened and never will. In the unlikely event that Perl 5 dies out to Perl 6 I'm jumping ship.

2 uncons

Perl is one of the very few languages that gets this right, and, I think, the only such mainstream language that isn't stack-based.[3]

Here's the idea. We've got a cons function and we want to write its inverse, uncons. How would we do it? Let's start in Ruby, where we can get close.

```
def cons(h, t)
  Cons.new(h, t)
end
def uncons(c)
  [c.head, c.tail]
end

c = cons 3, 4
x, y = uncons c
```

But uncons isn't a true inverse:

```
c2 = cons(uncons(c))            # this dies
c2 = cons(*uncons(c))           # we have to do this instead
```

And this illustrates a fundamental asymmetry in Ruby and in most applicative languages: a function can take multiple arguments, but can return only one.[4] If you think about how function calling works, of course, there's no particularly good reason it needs to work this way: functions are just transformations against the call stack, and you could easily define a calling convention that allowed a function to push multiple return values. And that's exactly what Perl does.

```
sub cons {
  {head => $_[0], tail => $_[1]}
}
sub uncons {
  my ($c) = @_;
  ($$c{head}, $$c{tail});
}
$c = cons 3, 4;
($x, $y) = uncons $c;
$c2 = cons uncons $c;           # this works
```

[3] That is, it supports applicative syntax – though you can also use Perl as a stack language if you're determined to. If you have no idea what any of this is, you should check out Joy and FORTH (in that order unless you like assembly code) because they will change your world.

[4] This is even true in most Scheme implementations because arity-checking happens before CPS-conversion:

```
(define uncons (lambda (c) (call/cc (lambda (k) (k (head c) (tail c))))))
(define c2 (cons (uncons c))) ; scheme dies here despite its validity in CPS
```

We have a problem, though: suppose we actually want a function that returns an array, not multiple values. Perl seems willing to blur the line between those two concepts. In practice that's because in Perl there's no difference: arrays aren't things, they really are multiple values:

```
$c = cons 3, 4;
@returned = uncons $c;
$c2 = cons @returned;           # this works
```

So Perl promotes arrays and hashes into language-level constructs because these things have language-level semantics that their counterparts in Ruby and Python don't. Having these semantics wouldn't be possible if Perl modeled these things as objects.[5]

3 Sigils

If `@xs` is an array, you refer to its first element as `$xs[0]`. To explain why this works and why it's a feature, I first need to describe how Perl looks at these two expressions; and before I get to that, I want to talk about Perl's symbol table.

Perl lets you define multiple variables with the same name: `@x` and `%x` are two completely different values. Because they have the same name, though, they're stored in the same location in the symbol table; both are accessible using `${main::}{x}` (that is, the `x` element of the `main::` hash). That hashtable value is a *typeglob*, which can be dereferenced as a scalar, an array, a hash, or a function:

```
$xs = 5;
@xs = 1..10;
%xs = (foo => 1, bar => 2);
*xs = sub {"hi"};

my $v = ${main::}{xs};          # this is a typeglob
print $$v;                      # the scalar slot: prints 5
print @$v;                      # the array slot: prints 12345678910
print %$v;                      # the hash slot: prints foo1bar2
print &$v;                      # the code slot: prints hi
```

(This insane detour has a purpose, I promise. Hang in there.)

Ok, now let's talk about what Perl looks at when it's figuring out which of these typeglob entries you're working with. Let's say you've got a mystery character `?` that can be any of `$`, `@`, or `%`, but Perl doesn't know which. Writing `?xs` is obviously ambiguous, but writing `?xs[0]` isn't; the reason is that square

[5] The obvious question of arrays-within-arrays is handled by *references*, which are single values that refer to complex data structures. There's no syntactic ambiguity around them because they must be dereferenced.

brackets only work on arrays. `?xs{foo}` also isn't ambiguous because curly braces only work on hashes.

So Perl is in an interesting position: it needs *some* leading character, but it doesn't matter which one. Rather than having you repeat information Perl already knows, it instead lets you specify something it doesn't know: the type of return value you want.

```
$ perl -de1
perl> @xs = 'a'..'j';
perl> p @xs
abcdefghij
perl> p $xs[0]                  # get a scalar
a
perl> p @xs[1..5]               # get an array
bcdef
perl> p %xs[1..4]               # get a hash of key/value pairs
1b2c3d4e
```

4 Regular expressions