Jonathan Worthington

OH HAI

What Makes Perl Great

Get the job done.

Accepts that different problems need different solutions.

Procedural Object Oriented Functional

Easy things easy.

Hard things possible.

CPAN

Thousands of modules. Good documentation and testing culture.

The Perl 6 Project

Take all of the things that make Perl great.

Learn from the things that didn't work so well in Perl 5.

Be inspired by the latest and greatest ideas from other languages and language research.

Build a new Perl.

Perl 6 Language Specification + **Official Test Suite**

No official implementation.

Implementation projects:

Rakudo Pugs **SMOP/Mildew** Perlito **Sprixel** ...more...

Different implementations often have a different focus or strength

=>

Not a duplication problem; e.g. Rakudo has learned from other implementations

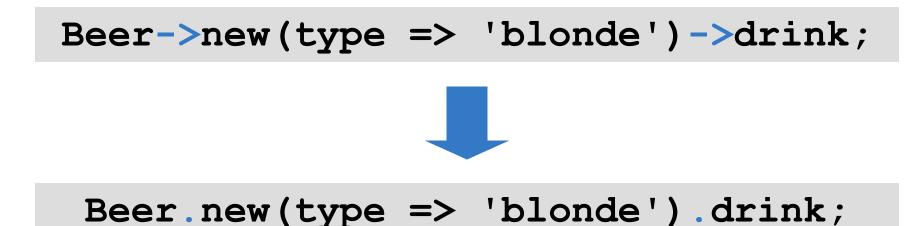
Fixing Mishuffmanizations

Things that you do frequently should be short

Things that you do rarely should be longer

In Perl 6, various language constructs have been re-evaluated so we can give them a more appropriate length.

Method Calling



say (also in Perl 5.10)

print "Привет!\n";
print "Пиво?\n";

say "Привет!"; say "Пиво?";

Less Parentheses

if (\$answer == 42) {
 for (1..10) { say "Correct!"; }
}



if \$answer == 42 { for 1..10 { say "Correct!"; } }

Easy Things Made Easier

Perl 5 makes many easy things easy.

In Perl 6, we've worked to make them even easier.

Hash Iteration

for my \$name (keys %ages) {
 say "\$name is \$ages{\$name}";
}



for %ages.kv -> \$name, \$age {
 say "\$name is \$age";
}

Subroutine Signatures

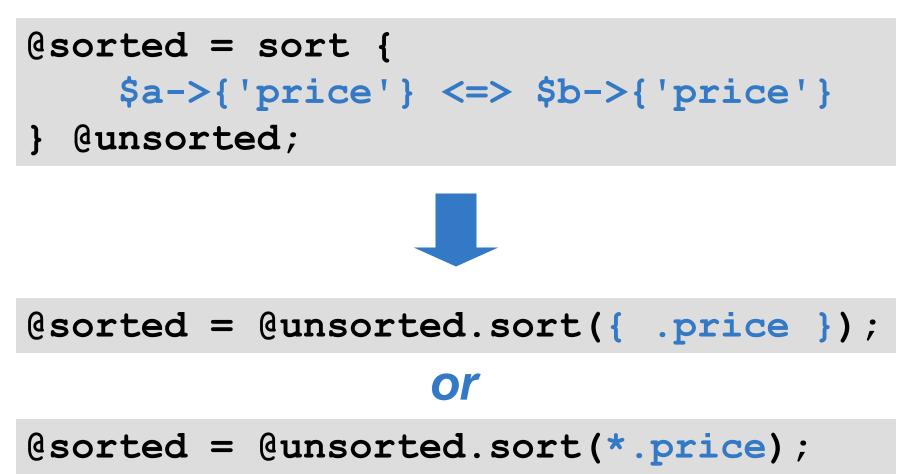
sub order { **my** (\$beer, \$pints) = @ ; print "\$pints pints of \$beer\n"; sub order(\$beer, \$pints) { say "\$pints pints of \$beer";

Sorting

@sorted = sort { \$a->{'price'} <=> \$b->{'price'} } @unsorted;

@sorted = @unsorted.sort({ .price });

Sorting



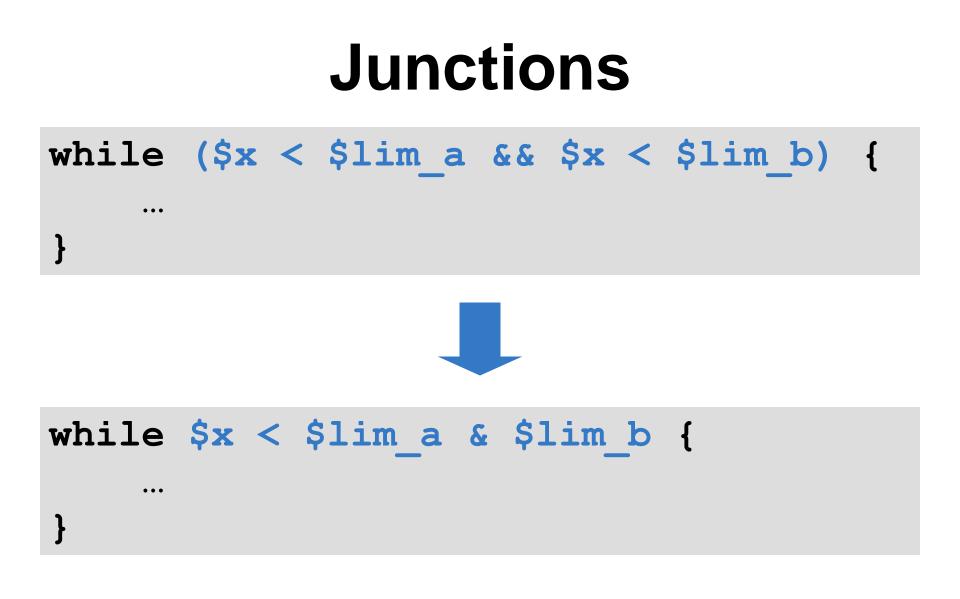
}

Chained Conditionals

if (\$age >= 18 && \$age <= 65) { say "Pay the adult price";</pre>



if 18 <= \$age <= 65 { say "Pay the adult price"; }</pre>



Junctions

if (\$drink eq 'Beer' ||
 \$drink eq 'Vodka') {

}

}

. . .



if \$drink eq 'Beer' 'Vodka' {

Reductions

my \$total = 0; \$total += \$_ for @values; say \$total;



say [+] @values;

Reductions

my \$factorial = 1; \$fact *= \$_ for 1..\$n; say \$factorial;



say [*] 1..\$n;

Better OO Programming

In Perl 6, you can treat everything as an object if you want to.

say	4.25.round # 4
say	"dam".flip; # mad
say	(1,2,3).join(' ') # 1 2 3
say	(110).grep({ \$_ !% 3 }); # 369

say	4.25.round # 4
say	"dam".flip; # mad
say	(1,2,3).join(' ') # 1 2 3
say	(110).grep({ \$_ !% 3 }); # 369
(Of course, the function forms still work too ⓒ)	

If you've used Moose, you will probably find the Perl 6 object model easy to start using.

Different syntax, but a lot of the same keywords and concepts.

Creating and using a class is quick and easy.

```
class Beer {
   has $!name;
   method describe() {
      say "I'm drinking $!name";
   }
}
```

```
my $pint =
   Beer.new(name => 'Baltika');
$pint.describe();
```

Attributes are private; declarative accessor syntax.

```
class Dog {
    has $.name is rw;
    has $.color;
}
```

```
my $pet = Dog.new(
    name => 'Spot', color => 'Black'
);
$pet.name = 'Fido';  # OK
$pet.color = 'White';  # Fails
```

Also provides...

Inheritance Delegation Constructors **Deferral to parents** Introspection **Meta-programming**

Real World Example:

A karma tracking IRC Bot

Written by Carlin Bingham

Perl 6 supports (multiple) inheritance.

However, multiple inheritance has issues (e.g. diamond problem), and single inheritance limits re-use.

As well as classes, the Perl 6 object model includes support for roles.

A role can have attributes and methods, but unlike a class is not intended to be used on its own.

Instead one or more roles are composed into a class.

}

```
role Logging {
    method log($message) {
        my \ fh = open('log', :a);
        $fh.say($message);
        $fh.close;
class MailSender does Logging {
    . .
```

Methods and attributes are "copied" into the class, as if they were declared there.

If two roles try to supply a method with the same name, you get a compile time error.

```
role Drinking {
    method go-to-bar() { ... }
}
role Gymnastics {
    method go-to-bar() { ... }
}
class DrunkGymnast {
    does Gymnastics;
    does Drinking;
}
```

Such conflicts can be resolved by:

Writing a method in the class that decides what to do or

Having a proto method in the class to make them multis

```
role Drinking {
    method go-to-bar() { ... }
}
role Gymnastics {
    method go-to-bar() { ... }
}
class DrunkGymnast {
    does Gymnastics;
    does Drinking;
    method go-to-bar() {
        self.Gymnastics::go-to-bar();
```

More Powerful Parsing

Perl has always been a leader in regexes.

In Perl 6, regex syntax has been radically updated and made much, much more powerful.

In Perl 6, regexes are not just strings.

They are a first class language within the Perl 6 language.

Many changes make regex syntax more consistent.

Any letter, number or the underscore (by unicode semantics) is a literal.

Everything else is considered to be syntax.

The /m modifier is gone

^ matches start of string ^^ matches start of line

\$ matches end of string \$\$ matches end of line

A range in a character class is written using .. – just like ranges in the rest of Perl!

/<[0..9A..F]>/

Use single quotes for matching a literal string – just like in the rest of Perl.

/'[''?\w+']'/

Perl 6 also tries to fix the problems with regex culture.

Just like you break the rest of your program up into reusable parts, you are encouraged to do the same with your regexes in Perl 6.

```
regex IntPhoneNumber {
    <CountryCode> \s+
    <AreaCode> \s+
    <LocalNumber>
}
regex CountryCode {
    '+' \d**1..3
}
regex AreaCode {
    '(' \d**{3..5} ')'
}
regex LocalNumber {
    d**{5..10}
}
```

Notice that whitespace in your regex does not match anything (other than in rule).

So you've no excuse not to space your regexes out and add comments so that others can read them! ©

Just as you collect methods together into a class, you can collect regexes together into a grammar.

So Perl 6 CPAN will contain ready-made grammars for parsing things. ③

Real World Example:

A grammar for JSON

Written by Moritz Lenz

Perl 6 itself is parsed using Perl 6 regexes.

This means that once you learn Perl 6 regexes, you can start to understand or even hack on the Perl 6 parser. ③

Multiple Dispatch

The idea of "DWIM" (Do What I Mean) has always been an important part of Perl.

In Perl 6, one feature that helps deliver this is multiple dispatch.

Write many subroutines or methods with the same name but with different signatures.

When you make a class, the runtime decides which one is best and calls it.

}

}

Example (from Test.pm): different number of parameters

multi sub todo(\$reason, \$count) is export {
 \$todo_upto_test_num = \$num_of_tests_run + \$count;
 \$todo_reason = '# TODO ' ~ \$reason;

multi sub todo(\$reason) is export {
 \$todo_upto_test_num = \$num_of_tests_run + 1;
 \$todo_reason = '# TODO ' ~ \$reason;

Example: different types of parameters

- class Paper { }
- class Scissor { }
- class Stone { }
- multi win(Paper, Stone) { "Win" }
 multi win(Caissen Demon) { "Win" }
- multi win(Scissor, Paper) { "Win" }
 multi win(Stand Caisson) { "Win" }
- multi win(Stone, Scissor) { "Win" }
 multi win(::T, T) { "Draw" }
 multi win(Any, Any) { "Lose" }
- say win(Paper, Paper); # Draw
 say win(Paper, Scissor); # Lose
 say win(Stone, Scissor); # Win

Perl 6 multiple dispatch can also consider values and the structure of a complex value.

This enables a lot of "write what you know" style solutions.



Factorial: fact(0) = 1

Factorial: fact(0) = 1fact(n) = n * fact(n - 1)

Factorial: fact(0) = 1fact(n) = n * fact(n - 1)

multi fact(0) { 1 }
multi fact(\$n) { \$n * fact(\$n - 1) }

Fibonacci Sequence: fib(0) = 0 fib(1) = 1fib(n) = fib(n - 1) + fib(n - 2)

Fibonacci Sequence: fib(0) = 0 fib(1) = 1fib(n) = fib(n - 1) + fib(n - 2)

<pre>multi fib(0)</pre>	{	0 }	
<pre>multi fib(1)</pre>	{	1 }	
<pre>multi fib(\$n</pre>) {	fib((n - 1) + fib((n - 2)))	

Quicksort

Quicksort

Empty list sorts to the empty list
multi quicksort([]) { () }

Quicksort

```
# Empty list sorts to the empty list
multi quicksort([]) { () }
```

}

```
# Otherwise, extract first item as pivot...
multi quicksort([$pivot, *@rest]) {
```

}

Quicksort

```
# Empty list sorts to the empty list
multi quicksort([]) { () }
# Otherwise, extract first item as pivot...
multi quicksort([$pivot, *@rest]) {
    # Partition.
    my @before = @rest.grep({ $_ < $pivot });
    my @after = @rest.grep({ $_ >= $pivot });
    ...
```

Quicksort

```
# Empty list sorts to the empty list
multi quicksort([]) { () }
# Otherwise, extract first item as pivot...
multi quicksort([$pivot, *@rest]) {
    # Partition.
    my @before = @rest.grep({ $_ < $pivot });
    my @after = @rest.grep({ $_ >= $pivot });
    # Sort the partitions.
```

(quicksort(@before), \$pivot, quicksort(@after))

Perl 6 operators are implemented as multi-subs.

Operator overloading means simply writing another multi sub for your type.

Designed To **Evolve – But** Sanely

The natural languages that we speak change over time

New words and structures to express new concepts

Adapt to the needs of users

adapt || die

Perl 5 is 15 years old and already wanting to evolve. Thus...

Source filters Devel::Declare

Perl 6 is designed to accept that language evolution is something that Just Happens.

Thus it provides clean ways to extend the language.

Languages changes are lexically scoped, not global.

Parser always knows what language it's parsing. Reader knows what dialect of Perl 6 they are reading.

Some things (new operators) are trivial.

multi postfix:<!>(0) { 1 }
multi postfix:<!>(\$n) { \$n * (\$n - 1)! }

say 10!; # 3628800

Some things (new operators) are trivial.

multi postfix:<!>(0) { 1 }
multi postfix:<!>(\$n) { \$n * (\$n - 1)! }

say 10!; # 3628800

More advanced things harder, but possible. ③

Conclusions

Perl 6 is a large and ambitious project being developed by a relatively small team.

We already have a compiler that can run all of today's examples, however. ©

Rakudo *

Distribution release taking place in late May or early-mid June 2010.

Rakudo *

Not all Perl 6, but a large and powerful subset of it with good coverage of OO, regexes and grammars, built-in operators and functions, multi-dispatch, and more.

Rakudo *

Will include not only the compiler, but also a selection of modules and a module installation/update tool.

Rakudo *

Also aiming to include support for using many Perl 5 **CPAN modules from Perl 6.** (This is an area of active development currently.)

And Perl 6.0.0?

Not setting a date.

When we have a viable implementation that is considered "good enough" (not perfect.)

Rakudo beyond Rakudo * Native types/structures Parallelism **Improved IO** Add other missing features More backends More speed, more stability

Get Involved / Learn More

Want to learn more?

- Get Rakudo Perl 6 from: http://www.rakudo.org/
- Lots of Perl 6 resources can be found at: http://www.perl6.org/
- Join the friendly IRC channel: #perl6 on irc.freenode.org
- Write modules, write applications, jump into the evolving Perl 6 community and make your mark on it ^(C)

Спасибо!

Questions?