An Inside Look at Perl 6

A New Star is Born

Perl 6 was announced in 2000, but it is still under development with no shipping date in sight. And people keep wondering: what's the deal with Perl6? BY MICHAEL SCHILLI

The truth is that hardly anyone is busy waiting for Perl 6. The humongous module collection on CPAN keeps thriving; an average of three dozen new submissions and updates reach the repository on any given day, and they're all written in Perl 5, not Perl 6.

Perl 6 has a very agile community behind it, though. The design is fairly mature already, no radical changes are expected. Development of the virtual machine named "Parrot" has shown first results. The Perl 6 compiler, which transforms Perl 6 into bytecode for the VM, is currently under development. The lack of publically announced deadlines doesn't mean anybody is slacking. The mailing lists are active, and the team does have internal milestones, although they do not publish these milestones in order to avoid creating unproductive hype around the project. Perl 6 will be released when it's ready, and that's that.

Two O'Reilly books on Perl 6 have been published already. Writing a book on a language that hasn't been released is a curious undertaking, if you think about it, knowing that the newly proposed Perl 6 syntax could still be changing and parts of the books will be outdated by the time they are published.

Explore the Future Today

For those who can't wait to get their fingers on Perl 6, a number of crafty people have provided a bundle of Perl 5 modules. These modules simulate the future behavior of Perl 6 by warping the way Perl 5 interprets source code.

The entire collection is available on CPAN as a module bundle called *Bundle::Perl6*. When installed, 27 different modules get downloaded from CPAN. Their scope reaches from defining new variable syntax in *Perl6::Variables* to providing majorly advanced regular expressions with *Perl6::Rules* (see Table 1).

One of the most significant syntactical changes is the sigils in front of variable names: In Perl 5, to retrieve the value of a Hash *%hash* associated with the key "*key*", you have to say *$hash{key}*. In Perl 6, this will be *%hash{key}*. While the reasoning in Perl 5 was: "The sigil indicates what you get" (a scalar in this case, since the stored value is a scalar), in Perl 6 the mantra goes "The sigil shows the variable type."

Listing 1, *hash.pl*, shows this by pulling in *Perl6::Variables* for Perl6's new variable syntax and *Perl6::Say*. The latter exports the *say()* function, a fancy new way of printing a message with a newline at the end.

Quantum Leap

Damian Conway's quantum superpositions also make it into Perl 6 as a standard feature. The fact that a single variable can hold several values as a superposition might look disturbing at first, but it allows for breathtaking constructs.

Junctions, as they are called in Perl 6, superimpose one or more values onto a variable. Imagine, a scalar *$age* could be 7 and 42 at the same time, both *$age == 7* and *$age == 42* are true! This feature is available today in the *Quantum::Superpositions* module on CPAN. In [3] earlier this year, you've already seen a practical example of what junctions can be used for.

Perl 6 will not only accept calls to *any()*, *all()*, *one()*, *none()* to obtain superpositions, but it will provide operators (&, | and ^) entirely devoted to this new feature. Now, instead of using *any()* for having *$hand* hold all possible Blackjack counts of four aces, as in

```
$hand = any(4, 14, 24, 34, 44);
```

you can simply say

```
$hand = 4 | 14 | 24 | 34 | 44;
```

This of course means that the Perl 5 meanings of (&, | and ^), used for bitwise operations, will go away. Since these are rarely used in high-level Perl, they are going to be replaced by admittedly awkward looking two-byte combinations: +&, +| and +^ are the Perl 6 bitlevel operators. The goal with this move was Huffman coding: rarely used syntax can be more complicated, and popular constructs should be as short as possible to save typing and make reading the code easier.

Listing 1: hash.pl

```
use Perl6::Variables;
use Perl6::Say;

my %ages = (
  Huey  => 4,
  Dewey => 5,
  Louie => 6 );

for(keys %ages) {
    say "$_ is %ages{$_} years old.";
}
```

Listing 2: hashref.pl

```
my $agesref = \%ages;
say "Huey is $agesref{Huey} years old.";
```

Named Parameter Lists

Perl 5 functions accepting long parameter lists require developers to remember the exact ordering:

```
buy($count, $price, $item);
```

But who can remember the function signature without looking up a (hopefully) available API manual? What comes first, *$price* or *$item*? To free the programmer of this burden, *named parameter lists* are often used in Perl 5:

```
buy(count     => $count,
    item      => $item,
    price     => $price);
```

This way, arguments can be passed in random order. In Perl 5, programmers need to manually add code to allow for this syntax: Functions typically feed all incoming arguments as name-value pairs into a hash and then make sure that all required ones are present.

Perl 6, however, lets you specify the argument list as part of the function definition, like in

```
    # Perl 6
sub buy($count, $price, $item) { ... }
```

which eliminates the typical Perl 5 workaround with <\@>_ as in

```
    # Perl 5
sub buy {
    my ($count, $price, $item) = @_;
    # ...
}
```

In Perl 6, all parameters in the function's signature are automatically aliased to the corresponding lexically scoped variables named *$count*, *$price*, and *$item*.

This syntax not only handles ordered parameter lists, but also named lists:

```
    # Define it
sub buy($count, $price, $item) {
     print "Buying $count items at $price<\>n";
     # ...
}

    # Call it
buy(count => 1,
    item  => "TV",
    price => 999);
```

and again the order in which arguments are provided to the function does not matter. Perl 6 parameter list handling for subroutines is available for Perl 5 today with *Perl6::Parameters* from CPAN. However it does not handle named parameter passing yet.

The OO Syntax

Perl 5 suffers from a patched-on object orientation: Clearly not part of the original language design, the object-oriented

Listing 3: class.pl

```
use Perl6::Classes;

class Car {

# Only with Perl6::Classes,
# Perl 6 takes care of it
  submethod BUILD {
    my %hash = @_;
    $.model = $hash{model};
    }

  has $.model;

  method drive() {
    print("$.model drives\n");
  }
}

my $car = Car->new(
  model  => "Acura Integra");

$car->drive();
```

(OO) syntax was added later on with very little language modification. This resulted in an OO implementation in "user space": packages as classes, objects most often implemented as blessed hashes and an inheritance model that requires a lot of manual twiddling.

Perl 6 fixes all this, embedding the *class* and *method* keywords, attributes (Perl 6 lingo for instance variables), automatic accessors, and inheritance directly into the language.

Listing 3, *class.pl,* shows how *Perl6::Classes* somewhat simulates the new behavior. As of right now, early adopters have to jump through some hoops, though, using a *BUILD* sub-method for the constructor, while Perl 6 will initialize new objects automatically. Inheritance is requested by the *is* keyword, *class Car::Electric is Car* indicates that *Car* is the base class of the class modeling electric cars.

But inheritance isn't the only class relationship in Perl6. *Roles* define class-like behavior. Using *role* instead of the *class* keyword, roles can provide methods, but no attributes. Classes can assume roles by using the *does* keyword, but subclasses don't inherit them. *Car does FuelConsumer* in a class definition indicates that *Car* utilizes *FuelConsumer*'s methods, but that's not an inherent feature of *Car*, but rather an add-on that shouldn't be part of the inheritance tree. Looking at Listing 3, if we were truly running Perl6, then the constructor would look different:

```
my $car = Car.new(
    model  => "Acura Integra");
```

Listing 4: regex.pl

```
use Perl6::Rules;

grammar Calc {
    rule number    { \d+ }
    rule op        { <[+-]> }
    rule term :w   {
      <term> <op> <number> |
      <number> }
}

my $text = "1 + 2 - 3 + 4";

if($text =~ m/<Calc.term>/) {
    print "Well-formed!\n";
}
```

Also, the dot has replaced the arrow in method calls:

```
$car.drive();
```

Within a class definition, what used to be *$self* is now just *topicalized* in *$_*, so *$_.drive()* or, even shorter, *.drive()* will work just fine.

Regex Overhauled

Probably the biggest change in Perl6 is the complete overhaul of the regular expression syntax. Perl 5's Regexes are already the leading edge, but Perl 6's grammars and rules go far beyond that.

Table 1: Modules in *Perl6::Bundle*

Module	Description
Attribute::Handlers	Simpler definition of attribute handlers
Attribute::Types	Attributes that confer type on variables
Attribute::Overload	Attribute that makes overloading easier
Attribute::TieClasses	Attribute wrappers for CPAN Tie classes
Attribute::Util	A selection of general-utility attributes
Attribute::Deprecated	Mark deprecated methods
CLASS	Alias for __PACKAGE__
Class::Object	Each object is its own class
Coro	create and manage coroutines
Exporter::Simple	Easier set-up of module exports with attributes
NEXT	Provide a pseudo-class NEXT for method redispatch
Scalar::Properties	Run-time properties on scalar variables
Switch	A switch statement for Perl
Perl6::Binding	Implement Perl6 aliasing features
Perl6::Classes	First class classes in Perl 5
Perl6::Currying	Perl 6 subroutine currying for Perl 5
Perl6::Export	Implements the Perl 6 'is export(...)' trait
Perl6::Form	Implements the Perl 6 'form' built-in
Perl6::Gather	Implements the Perl 6 'gather/take' control structure in Perl 5
Perl6::Interpolators	Use Perl 6 function-interpolation syntax
Perl6::Parameters	Perl 6-style prototypes with named parameters
Perl6::Placeholders	Perl 6 implicitly declared parameters for Perl 5
Perl6::Say	Implements the Perl 6 say (print-with-newline) function
Perl6::Tokener	A Perl 6 tokener. It tokenizes Perl 6.
Perl6::Variables	Perl 6 variable syntax for Perl 5
UNIVERSAL::exports	Lightweight, universal exporting of variables
Want	Implement the want() command

Grammars are like classes, containing rules like methods. Listing *regex.pl* shows an example.

Reminiscent of a throwback to lex/yacc grammars, Perl 6's grammar serves exactly the same purpose as these ancient dinosaurs. That makes it possible to parse even the most complex syntactic structures – maybe even Perl 6 some day! Unfortunately, *Perl6::Rules* pushes Perl 5's regular expression engine to the limit, *regex.pl* will currently just crash the program and cause a segfault. Hopefully, this will be fixed soon.

Perl 6 regexes are easier to read than their Perl 5 counterparts. "/x modifier mode," in which whitespace is ignored and comments are allowed, is now the default. And there's the incredibly useful *words modifier*, which is activated via the leading *:w* as in

```
rule fraction :w  ↩
{ <\>d+ / <\>d+ }
```

and interprets regex whitespace smartly. Inbetween two words in the regex (think *<\>w*), it inserts *<\>s+*, and inbetween words and non-words (like between *<\>d+* and */*, and also between */* and *<\>d+* above), it inserts *<\>s** to allow zero or more spaces. That's most likely what you'll want.

There's lots more to Perl 6. Subroutines and variables can have attributes, Variables can be strictly typed. No more typeglobbing. Formats reworked from the ground up. True exception handling. The modules in *Perl6::Bundle* as listed in Table 1 allow for trying out many new features – but be careful, some of them represent Perl 6 in a state it is no longer in. It's a moving target, after all.

All I can give you at this time of writing is a snapshot of some of the most intriguing new features of Perl 6. It will certainly be exciting to play with Perl 6 once it hits the shelves! ■

INFO

[1] Perl6 Development Page: *http://dev.perl.org/perl6*

[2] "Perl 6 and Parrot Essentials", Allison Randal, Dan Sugalski, Leopold Tötsch, O'Reilly, 2004.

[3] "Quantum Casino", Michael Schilli, Linux-Magazine, Issue #38 / January 2004, p61