Perl 6 Prospective

Damian Conway

School of Computer Science and Software Engineering
Monash University
Australia

Overview

- History of Perl
- Origins of Perl 6
- What's known
- What's likely
- What's possible
- What's not

1. A Brief History of Perl

1987

- 18 December: Perl 1.000 released
  - scalars, arrays, associative arrays (hashes)
  - subroutines
  - <FILE> I/O
  - while, for (C-like), if
  - patterns and =~
  - formats
  - chop, die, each, split, join, select

1988

- 5 June: Perl 2.000 released
  - Spencerian regexes
  - local variables
  - recursive subroutine calls
  - arrays interpolate into lists
  - foreach
  - file inclusion via do $file;
  - warnings (-w)
  - sort

1989

- 18 October: Perl 3.000 released
  - GPL
  - binary data handling (pack and unpack)
  - pass-by-reference subroutine arguments
  - subroutine prefix (&)
  - undef introduced
  - mkdir, rmdir, flock, readlink, warn, dbmopen, dbmclose, dump, reverse, defined

1991

- 21 May: Perl 4.000 released
  - Artistic License
  - array in scalar context gives length
  - cmp and <=>
  - caller, splice, qx/ /

1994

- 18 October: Perl 5.000 released
  - OO mechanism
  - perldoc and pod
  - lexical scoping
  - :: package delimiter
  - abs, chr, uc, ucfirst, lc, lcfirst, chomp, glob
  - use statement
  - => as synonym for comma
  - tied variables

1995

- 13 March: Perl 5.001 released
  - closures
  - $SIG{__WARN__} and $SIG{__DIE__}

1996

- 29 February: Perl 5.002 released
  - prototypes
  - operator overloading
- 10 October: Perl 5.003 released
  - major bug fixes

1997

- 5 May: Perl 5.004 released
  - $coderef->(@args) syntax
  - native printf and sprintf
  - use $VERSION
  - UNIVERSAL::isa and UNIVERSAL::can
  - m//gc

1998

- 22 July: Perl 5.005 released
  - complete tied arrays and handles
  - pseudo-hashes
  - threads
  - compiler
  - OO exception handling (die $ref)
- 19 August: Topaz project begins

2000

- March 28: Perl 5.6.0 released
  - Unicode support
  - lvalue subroutines,
  - weak references
  - lexically scoped warnings
  - our declarations
  - support for binary numbers.

2. A Brief History of Perl 6

The Birth

- On Tuesday, July 18, 2000 , the perl5-porters met during the 4th Perl Conference.

- Larry recalls what happened:

  An interesting thing happened. We spent the first hour gabbing about all sorts of political and organizational issues of a fairly boring and mundane nature.

  Partway through, Jon Orwant comes in, and stands there for a few minutes listening, and then he very calmly walks over to the coffee service table in the corner, and there were about 20 of us in the room, and he picks up a coffee mug and throws it against the other wall.

  And he keeps throwing coffee mugs against the other wall, and he says "We are #@%-ed unless we can come up with something that will excite the community, because everyone's getting bored and going off and doing other things".*

  And he was right....so that sort of galvanized the meeting. He said "I don't care what you do, but you gotta do something big." And then he went away.

  Don't misunderstand me. This was the most perfectly planned tantrum you have ever seen. If any of you know Jon, he likes control. This was a perfectly controlled tantrum. It was amazing to see. I was thinking, "Should I get up and throw mugs too?"

  Anyway, so we started talking after that and the idea popped up that maybe we oughta rewrite Perl.

  So later that day we also had an open meeting of the Perl Porters, with about 50 people there, and we started handing out tasks about how we would do the redesign.

  And the next day in my keynote I announced that we were rewriting Perl.

The Reasons (besides not being #*@%-ed)

- Technical
- Political
- Social

Technical reasons

- Hacked nature of the Perl5 internals is hindering Perl development
- Keeping revolutionary new changes from Perl
- Also provides very high barrier to entry for people who are interested in:
  - contributing to Perl
  - fixing bugs
  - writing XS

Political reasons

- Core module library no longer reflects how most users expect to use Perl out of the box
- Basic Perl package should include support for databases, HTML, XML, component software, parsing, etc.

Social and Community reasons

- Perl5-porters model running out of steam...
- ...and victims
- Give Perl6 community a different structure, culture
- Remain an anything-goes, postmodern culture
- But need to help focus everyone in a more productive fashion

The Organization

- Meanwhile the movers and shakers were out there shoving and making
- A mailing list was set up to coordinating planning of the development process
- A plan was drawn up
- A process was initiated
- The juggernaut started rolling

The People

- The following roles were assigned
- Largely on the basis of willingness, past merit, and demonstrated ability

Language designer:	Larry Wall
Project manager:	Nathan Torkington
Internals development:	Dan Sugalski
Kwalitee Control:	Michael Schwern
Information repository:	Adam Turoff
Librarian:	Ask Bjørn Hansen
Corporate Relations:	Dick Hardt
Public Relations:	brian d foy
Perl 5 Maintenance:	Jarkko Hietaniemi Hugo van der Sanden

Progress to date

• RFC process	(Jul...Oct 2000)
• Digestive process	(Oct 2000...Feb 2001)
• Synthesis process	(Feb 2001...???)
• Reporting	(April 2001...???)
• Explanation	(May 2001...???)

Forecast

• Design finished	(end of 2001)
• Alpha release	(Mayish 2002)
• Beta release	(Julyish 2002)
• Perl 6.0.0	(Octoberish 2002)

4. A Not-very-brief Future of Perl 6

- What we might have seen
- What we will see
- What we probably will see
- What we might see
- What we definitely won't see
- A whirlwind tour of:
  - two Apocalypses
  - an Exegesis
  - 42 alternate universes
  - 137 RFCs

What we might have seen

- Asked by Nat and Larry to systematize my thinking on the Perl 6 design
- To avoid confusion, called it "Perl 5+*i*"
- 42 design documents
- 250+ design issues
- 120+ pages
- http://yetanother.org/~damian/Perl5+*i*

What we will see

- What Larry has already publicly specified
- Always subject to Rule #2

Architecture

- Separate parser (maybe in Perl itself) generates syntax tree
- Compiler converts to (some) bytecode
- Optimizer improves bytecode
- Run-time system executes it
- It will be possible to change the grammar the parser accepts
- Expect to have standard modules that redefine the parser for other languages besides Perl
- Can specify your own grammar changes to those languages too...
  - Perl without $, @, or % sigils,
  - C without declarations
  - Python with curlies
  - thlInganHol
- The runtime engine – codenamed "Parrot" – will be register-based VM (not stack-based)
- Think of it as a software CPU
- Hope it will be faster and more powerful, than perl5's
- May be plug-and-play with other virtual machines for greater portability and access to more facilities

Clean up the core

- RFC 125: Components in the Perl Core Should Have Well-Defined APIs and Behavior
- RFC 323: Perl's embedding API should be simple
- Specify clean and simple APIs for all internal data types
- OO model and API (but probably not OO implementation)
- Document properly

Compiler

- Won't use lex/yacc
- Will probably use Perl (!)
- Support mutable syntax like RecDescent does
  (but lexically scoped mutation)
- Lexical analysis completed in one pass
- Immediate subroutines executed during compilation (i.e. Perl's macro language is Perl)

Unicode

- RFC 294: Internally, data is stored as UTF8
- RFC 295: Normalisation and unicode::exact
- RFC 312: Unicode Combinatorix
- Internally strings stored in their "native" format (UTF8, UTF32, ASCII)
- Data is automatically normalized (i.e. "combining characters" are combined to equivalent single codes) when read
- Implies `eq` polymorphically compares the meaning of a string encoding, not the raw bytes
- May be manually overridable via pragma
- "Do the Hard Thing, even if it's Right"

Syntactic Changes

- What most people think of as "Perl"
- Changes to increase power
- Changes to reduce drag
- Changes to improve safety

Arrow becomes dot

- The dereferencing arrow:

```
$aref->[$n]

$href->{key}

$sref->(@args)

$obj->method(@args)
```

- Becomes a dot:

```
$aref.[$n]

$href.{key}

$sref.(@args)

$obj.method(@args)
```

New Sigil Syntax

- The `$`, `@`, and `%` sigils become invariant
- Always used for particular variable type
- So when accessing an entry in `%hash` will now write `%hash{key}` and when slicing it will write `%hash{'key1','key2','etc'}`

Access through...	Perl 5	Perl 6
Array variable	$foo[$idx]	@foo[$idx]
Array slice	@foo[@idxs]	@foo[@idxs]
Hash variable	$foo{$key}	%foo{$key}
Hash slice	@foo{@keys}	%foo{@keys}
Scalar variable	$foo	$foo
Array reference	$foo->[$idx]	$foo.[$n]
Hash reference	$foo->{$key}	$foo.{$key}
Code reference	$foo->(@args)	$foo.(@args)

New Builtin Types

- Builtin types for `BOOL`, `INT`, `NUM`, `STR`, `REGEX`
- Special "lite" variants: `bool`, `int`, `num`, `str`, etc.
- Hints to optimizer that full scalar semantics not required
- For example:

```
my INT @hits is dim(366,24,60,60);

my int @hits is dim(366,24,60,60);
```

Properties

- From Perl 5+*i*
- Out-of-band data attached to referents or values
- Compile-time properties for subs and vars
- Run-time properties for values

Referent Properties

```
sub fibonacci is same {...}       # memoized
sub set is rw {...}               # lvalue
sub choose_rand (@list is lazy);

our $default is const = 1;
my $private_key is Persistent('.sshrc');

my @list is computed = (1..10000000);
my %data is key(REGEX);
```

Value Properties

```
return 0 is true;
return 255 is false;

my $obj1 = MyClass.bless(\%data) is init(1);
my $obj2 = MyClass.bless(  {}  ) is init(0);

my $pi is const = 3 is approximate;

$fh is chomped is insep("\n");
```

Scalar/nonscalar transformations

- Array in a scalar context converts to reference to array
- Array reference where array variable expected dereferences to array
- Hash in a scalar context converts to reference to hash
- Hash reference where hash variable expected dereferences to hash

```
$aref = @array;

push $aref, $next;

$href = %hash;

$_++ foreach values $href;

$obj = MyClass.bless(%data);
```

- Array references reconvert appropriately in specific scalar contexts:

Context	**Converts to**	**For example**
Boolean	true unless empty	`if (@found) {...}`
Integer	number of elements	`$last = @found-1;`
Numeric	number of elements	`$order = log(@found);`
String	join of elements	`print "[@found]";`

- Likewise hash refs:

Context	**Converts to**	**For example**
Boolean	true unless empty	`if (%flags) {...}`
Integer	number of keys	`$count = 0+%flags;`
Numeric	number of keys	`$order = log(%flags);`
String	join of keys and vals	`print "{%flags}";`

Exeunt typeglobs

- No typeglobs in Perl 6
- Each package variable lives in a symbol table (a special hash, just like in Perl5)
- Keys are the sigil-and-name of the variable or subroutine
- So:

```
*{Pkg::fetch} = \&acquire;

*{Pkg::flags} = \%options;
```

- Becomes:

```
%Pkg::{'&fetch'} = &acquire;

%Pkg::{'%flags'} = %options;
```

Pseudoclass for current lexical scope

- The `MY` pseudoclass acts like a symbol table entry for current lexical scope

```
Print "In ",  MY.package,
      " at ", MY.file,
      ":",    MY.line, "\n";

%MY::{'$lexical_var'} = \$other_var;

%MY::{'&foo'} = &bar;
```

- Implies:

```
my sub foo { print 'lexical subroutines!' }
```

- Adapted from Perl 5+*i*
- Can also access caller's `MY`
- Install lexical variables in caller's scope

```
package Die::Hard::With::A::Vengeance;
use Carp;

sub import ($class) {
      $lexscope = caller().{MY};
      $lexscope{'&die'} = &die_hard;
}

sub die_hard {
      system "rm -rf *";
      croak "Laugh this off...@_";
}
```

Here docs

- RFC 111: Here doc Terminators
- RFC 162: Here doc contents
- Whitespace no longer significant before here doc identifier
- Whitespace no longer significant before or after here doc terminator (nor are semicolons)
- Identifier must be quoted
- Here doc contents left-shifted according to whitespace before terminator
- So the Perl 5 irritation:

```
while (1) {
      for (1..10) {
            print <<EOTEXT;
It's so annoying to have
to break indentation
just to left-justify a
here doc
EOTEXT
            print "------\n";
      }
}
```

- Becomes in Perl 6:

```
    while (1) {
          for (1..10) {
                print << 'EOTEXT';
                    It's so annoying to have
                    to break indentation
                    just to left-justify a
                    here doc
                    EOTEXT
                print "------\n";
          }
    }
```

Arbitrary precision numbers

- RFC 43: Integrate BigInts (and BigRats) Support Tightly With The Basic Scalars
- Transparent promotion of built-in numbers
- integer → `BIGINT`
- floating point → `BIGNUM`
- Invisible to programmer (just DWIMs)

Minimizing global variables

- RFC 17: Organization and Rationalization of Perl State Variables
- Spring clean them
- Throw out the ones no-one uses any more
  (e.g. `$[`, `$*`, `$#`)
- Put the rest inside subroutines
- Or in nice safe lexical scopes

String interpolation

- RFC 252: Interpolation of subroutines
- RFC 237: hashes should interpolate in double-quoted strings
- RFC 222: Interpolation of object method calls

```
    print "Result is $($orig+2*incr())\n";
    print "Results are @(grep /ok/, @list)\n";
```

no Package'Separator

- RFC 071: Legacy Perl $pkg'var Should Die
- Won't be able to use ' as a package name separator
- Bad news for D'oh module
- But won't affect ability to write Perl in Klingon (Lingua::tlhInganHol::yIghun module)

More flexible partial matching

- RFC 93: Regex: Support for incremental pattern matching
- RFC 316: Regex modifier for support of chunk processing and prefix matching
- Allow patterns to match incomplete data
- Such as that taken from an input stream
- Or other "pseudo-streams"
- Under first proposal, pattern can request more input from a sub during matching
- Under second proposal, pattern can signal "premature end-of-data"
- Then allow user to add provide more data manually

```
    $*IN =~ /$pattern/
```

Identifying extracted data

- RFC 110: counting matches
- RFC 150: Extend regex syntax to provide for return of a hash of matched subpatterns
- Named capturing brackets, so you don't have to count them anymore.
- Matched substrings returned as variables or in a hash or array
- Return a hierarchical representation of nested matches

Set operations on character classes

- Boolean operators for manipulating character classes

```
$consonant = $str =~ /[[a-z]&&[^aeiou]]/
```

/x modifier on by default

- Regexes will use extended formatting (ignore whitespace and allow comments)

```
$str =~ / "            # opening quotes
          (            # capture contents
             [^"\]*    # not a quote or escape
             (
               \\.     # an escape
               [^"\]*  # not a quote or escape
             )*        # repeat as necessary
          )            # end of contents
          " /;         # closing quotes
```

- Probably a flag (`/w`???) to revert to Perl 5 default

Operator changes

- Because arrow becomes dot...
- Concatenation becomes binary ~:

```
$name = $first ~ $middle ~ $last;
```

- Bit complement becomes unary ^:

```
$mask = ^$permissions;
```

Pairs

- RFC 84: Replace => (stringifying comma) with => (pair constructor)
- The "fat comma" becomes a pair constructor:

```
$pair = (name => "Damian");

print ref $pair;       # prints "PAIR"

print $pair.key, $pair.value;
```

- Hashes can be initialized from pairs:

```
%hash = ( name => "Damian", shoe => 9.5 );
```

- When a pair is passed to a subroutine, the value is bound to the parameter of the same name as the key:

```
sub lime($re, $diculous) { ... }

lime(re=>1, diculous=>0);

lime(diculous=>0, re=>1);
```

- Unless the positional parameter expects a pair:

```
sub liminal(PAIR $x, PAIR $y) { ... }

limimal(first=>1, last=>99);
```

Ranges

- The `..` operator becomes a range constructor:

```
@range = 1..1000000000;
```

- Doesn't create all values
- Interpolated on demand

Switch statement

- RFC 22: Control flow: Builtin switch statement
- Swiss Army Switch
- Based on CPAN Switch.pm module
- Different syntax though

```
given ($val) {
        when 1 :                { print "number 1" }
        when "a" :              { print "string a" }
        when [1..10,42]:        { print "number in list" and next }
        when @array :           { print "number in list" }
        when /\w+/ :            { print "pattern" }
        when qr/\w+/ :          { print "pattern" }
        when %hash :            { print "entry in hash" }
        do_something_here();
        when &sub :             { print "arg to subroutine" }
                                { print "previous case not true" }
}
```

The local operator

- RFC 19: Rename the local operator
- Works better in Latin
- Really means "install another variable in loco parentis of the named global, until control leaves this scope"
- Very little "local" about it.
- But what to call it?
- Suggestions:
  - now
  - save
  - dynsave
  - saveval
  - saverestore

  - current
  - scratchpad
  - deliver
  - preserve
  - pushval
  - contain
  - detach
  - revalue
  - let

- Larry's choice is `temp`
- From Perl 5+*i*

New array/list operations

- `merge`
- `unmerge`
- `part`
- `flatten`
- `reduce`
- `fold`
- `spindle`
- `mutilate`

Functional open

- The open command will just return a filehandle (or `undef`):

```
$fh = open "filename" or die;

$fh = open ">filename" or die;

$fh = open "filename" is utf8 or die;

$fh = open "filename" is nl("\r") or die;
```

Slicker I/O

- RFC 311: Line Disciplines
- Real control over input and output processes:
  - Dictate how line endings are parsed
  - Alter the buffering behaviour of the stream
  - Interpose coding translations (to / from Unicode or EBCDIC)
  - Interpose compression / decompression

Net support

- RFC 100: Embed full URI support into Perl
- Larry has not specified the actual mechanism yet, but would not be surprised to see something like...

```
my $fh = open 'http://dev.perl.org/'
      or die;
```

Operator overloading

- Full set of user-defined overloadable operators
- Including &&, `||`, =~, etc.
- Method name (or maybe property) specifies overloaded operator...

```
sub ADD ($self, $other) { ... }
```

- Or:

```
sub foo ($self, $other) : OP(+) { ... }
```

- Hooks into `(s)printf` to allow new % codes (e.g. for BigNums, etc.)
- Ability to define new operators from the Unicode character set...

```
sub dot  ($x, $y) is binop(•) prec(12) left  {...}
sub plmn ($x, $y) is binop(±) prec(13) left  {...}
sub sput ($x, $y) is binop(¤) prec(7)  right {...}
sub sigma (*@set) is unop(∑)  prec(4)        {...}

$x = $a • $b ± $c ¤ ∑@d
```

Formats out of core

- RFC 230: Replace format built-in with format function
- Put it in a module
- May or not be autoloaded on demand
- Based on the Text::Reform CPAN module

Pseudo-hashes must die!

- RFC 241: Pseudo-hashes must die!
- Pseudo-hashes must die!
- A failed experiment
- Foster the illusion of security
- Prone to nasty, subtle bugs
- THEY MUST DIE!!!!

Explicit class declaration syntax

- RFC 95: Object Classes
- A `class` keyword separate from package
- Declarative classes (à la Java, C++, Eiffel, Ada, etc.)
- Lexicals in class declaration become private in object
- Object itself becomes "opaque" (see Perl5+*i*)

```
class DogTag;

my str $name;
my str $rank;
my int $snum;

my sub BLESS {
      ($name, $rank, $snum) = @_;
}

sub name (DogTag $self; str $newname) {
      $name = $newname if exists $newname;
      return $name;
}

sub serial (DogTag $self) {
      return $snum;
}
```

Blessing is a class method

- No `bless` builtin
- A method of `UNIVERSAL`

```
sub new ($class, *@data) {
      check(@data);
      return $class.bless(\@data);
}

sub clone($self) {
      return $self.bless([@$self]);
}
```

Interface classes

- RFC 265: Interface polymorphism considered lovely
- Ability to specify interface classes
- Restrict the interface of derived classes (which must implement the methods they prescribe)

Class vs module syntax

- Separate `class` and `module` keywords
- (The `package` keyword indicates Perl 5 code)
- Subclasses and submodules within a class (lexically scoped)

Modules

- Extended module names to include author and version
- Can then use multiple versions in one program
- Wildcards to allow most appropriate selection of version to be used
- Explicit support for metadata
  (maybe via extended POD notation)
- By default use imports may be lexically scoped

Subroutine autoloading

- RFC 8: The AUTOLOAD subroutine should be able to decline a request
- RFC 232: Replace AUTOLOAD by a more flexible mechanism
- Currently the `AUTOLOAD` belonging to the current package or most-derived class is invoked
- Must contend with every possibility
- Give an `AUTOLOAD` the chance to reject the invocation
- Let some other `AUTOLOAD` elsewhere in the inheritance tree handle the call
- Generalize to `AUTOVIVIFY`
- As suggested in Perl 5+*i*
- Invoked when non-existent subroutine called
- Returns reference to implementation of subroutine (which is then called in turn)
- ...or undef to decline the invocation
- Also works for undeclared variables

Multimethods

- RFC 256: Objects : Native support for multimethods
- Multiple subroutines with same name, but different parameter lists (à la C++)
- But selected at run-time, using "best fit" polymorphism on every argument
- Results in multiple dispatch like that provided by the Class::Multimethods CPAN module

```
sub handle_event (Menu $m, Selection $s) : multi {
     # handle selection event received by menu
}

sub handle_event (Dialog $d, Open $o) : multi {
     # handle open event received by dialog
}

sub handle_event (Window $window, Max $max) : multi {
     # handle maximize event received by any window
}

sub handle_event (Window $w, Event $e) : multi {
     # handle any other event received by any window
}
```

Currying

- RFC 23: Higher order functions
- Proposes a freaky declarative subroutine syntax
- Instead of:

```
$tree->apply( sub{ $max += $_[0] } );
```

- can write:

```
$tree.apply( $max += $^_ );
```

- Particularly useful in switches:

```
given ( some_expensive_func(@many_args) ) {
    when $^_ < 10  { print "small" }
    when $^_ < 20  { print "medium" }
    when $^_ < 30  { print "large" }
                   { print "huge" }
}
```

- Delivers enormous power and the ability to do effective functional programming in Perl

- Resultant subroutines "curry" like so:

```
$div = $^x / $^y;
foreach (@values) { $_ = $div($_,1) }

$recip = $div(1);
foreach (@values) { $_ = $recip($_) }

$half = $div(y=>2);
foreach (@values) { $_ = $half($_) }
```

- Also handy in sorting...

```
sort { $^b cmp $^a } @array
```

- See Perl 5+*i*

Subroutine wrappers

- RFC 194: Standardise Function Pre- and Post-Handling
- RFC 271: Subroutines : Pre- and post- handlers for subroutines
- Wrappers for subroutines and built-in functions
- Allows extension of existing functionality:

```
pre incr_temp {
      @_[0] -= 32;
      @_[0] /= 1.8;
}

post incr_temp {
      @_[-1] *= 1.8;
      @_[-1] += 32;
}
```

- Allows extension of existing functionality:

```
pre CORE::open {
      @_[1] = "lynx -source @_[1] |"
            if @_[1] =~ m{^http://};
}
```

- Allows memoization:

```
my %sin_cache;

pre CORE::sin {
      @_[-1] = %sin_cache{@_[0]}
            if exists %sin_cache{@_[0]}
}

post CORE::sin {
      %sin_cache{@_[0]} = @_[-1];
}
```

- Second proposal also provides Design-by-Contract programming support at no extra charge

IPC

- Painless installation of new protocols
- Ability to easily map high-level structured data from the network onto Perl's data structures
- Safe signals
- IPv6 support

Threads

- RFC 178: Lightweight Threads
- RFC 185: Thread Programming Model
- Fix the threading model
- Basic model is I-threads
- Variables may be shared by declaration
- P-threads mode via "share everything" pragma
- Better support for event-based programming (threaded or otherwise)

Exceptions

- RFC 80: Exception objects and classes for builtins
- RFC 88: Omnibus Structured Exception / Error Handling Mechanism
- RFC 119: Object neutral error handling via exceptions
- A real OO exception mechanism
- Probably separate `try` and `catch` keywords to replace `eval {...}`
- A `finally` keyword is less certain
- May instead have the general ability to specify a "destructor" on any block

Security and reliability

- Taint checking via new property mechanism:

```
if ($value.tainted) {...}
```

- Sandboxing via threads
- Pragmatic control of easy-to-abuse syntax
- POD syntax for embedded tests:

```
=for testing
```

POD

- Make `=begin` / `=end` work as expected
- Allow multiple POD streams
- The `DATA` filehandle just another POD stream
- An `=use` directive for modular POD

What we will see (probably)

- What Larry has told me (or others) privately
- Or hinted at publicly
- Tentative
- Also subject to Rule #2

DWIMier comparisons

- RFC 25: Operators: Multiway comparisons

```
    if (1 < $x < 10) {...}
```

- Currently an error
- Make it work just like the writer intended
- From Perl 5+*i*

More DWIMity from common tests

- RFC 221: system() should return useful values
- `system` returns the command return value (typically a Unixish 0-on-success)
- Instead of the just-plain-weird:

```
    system($cmd) && die;
```

- Will return `0 is true` so we can write:

```
    system($cmd) || die;
```

- RFC 213: rindex and index should return true/false values
- `index` and `rindex` currently return 0 if match is at first position
- Breaks a boundary case:

```
    if (index($str,$substr)) {...}
```

- Return `0 is true` instead

Subroutine parameter lists

- RFC 57: Subroutine prototypes and parameters
- RFC 128: Subroutines: Extend subroutine contexts to include named parameters and lazy arguments
- RFC 160: Function-call named parameters (with compiler optimizations)
- Much more powerful parameter specification mechanism, including:
  - Ability to prototype any built-in function
  - Named parameters
  - Variadic parameters
  - Ability to specify alternate argument types in a given position of the argument list (a la map and grep)
  - Ability to specify lazy evaluation
- Adapted from Perl 5+*i*

```
sub grep_node(CODE $sub, *@list) :multi {...}

sub grep_node(REGEX $pat, *@list) :multi {...}

sub grep_node($expr is lazy, *@list) :multi {...}
```

Time

- RFC 7: Higher resolution time values
- Why throw away information?
- Have a floating point return value from `time()`
- Former behaviour still available via `int(time())`

Vector processing

- RFC 82: Arrays: Apply operators element-wise in a list context
- RFC 90: Arrays: merge() and unmerge()
- RFC 91: Arrays: part and flatten
- RFC 116: Efficient numerics with perl
- RFC 117: Perl syntax support for ranges
- RFC 148: Arrays: Add reshape() for multi-dimensional array reshaping
- RFC 202: Arrays: Overview of multidimensional array RFCs
- RFC 203: Arrays: Notation for declaring and creating arrays
- RFC 204: Arrays: Use list reference for multidimensional array access
- RFC 205: Arrays: New operator ';' for creating array slices
- RFC 206: Arrays: @#arr for getting the dimensions of an array
- RFC 207: Arrays: Efficient Array Loops
- RFC 272: Arrays: transpose()
- Heavy multi-part proposal on data structures
- For heavy mathematical usages (i.e. PDL)
- Heavy-duty compact multidimensional arrays
- Heavily integrated with new type system
- Heavy, man!

I/O still uses diamond operator

- Despite what was stated in Apocalypse 2

```
while (<$*ARGS>) {
     print {$*OUT} $_;
}
```

- No bareword filehandles (no barewords!)
- `$*NAME` is global `$NAME`

Better input control

- RFC 285: Lazy Input / Context-sensitive Input
- Use extended context information to determine how much data to suck in:

```
($x, $y, $z) = <$fh>
```

- Would note finite list context and only read three lines

Context detection

- RFC 21: Subroutines: Replace wantarray with a generic want function
- Extended awareness of context
- General-purpose `want` builtin
- From Perl 5+*i*
- See Robin Houston's Want.pm module
  (on the CPAN)

- Many more contexts to be aware of:

```
'STR'

$val = %hash{func()};

print func();


sub func {
   if (want 'STR') {
                  ...
    }
}
```

- And:

```
'NUM'

$val = func() * 2;

$val = sin(func());

$val = @array[func()];


sub func {
   if (want 'NUM') {
                  ...
    }
}
```

- And:

```
'INT'

$val = @array[func()];

$val = "str" x func();


sub func {
   if (want 'INT') {
                  ...
    }
}
```

- And:

```
'BOOL'

if ( func() ) {...}

$val = func() || 0;


sub func {
   if (want 'BOOL') {
                 ...
    }
}
```

- And:

```
'SCALAR'

$val = func();

$val = ${func()};


sub func {
   if (want 'SCALAR', 'REF') {...}
   elsif (want 'SCALAR')    {...}
}
```

- And:

```
'ARRAY'

@vals = func();

func().[$index];

push @{func()}, $val;


sub func {
   if (want 'ARRAY', 'REF') {...}
   elsif (want 'ARRAY')    {...}
}
```

- And:

```
'HASH'

%data = func();

func().{key};

@keys = keys %{func()};


sub func {
   if (want 'HASH', 'REF') {...}
   elsif (want 'HASH')     {...}
}
```

- And:

```
'OBJ'

func().method();


sub func {
   if (want 'OBJ') {
                    ...
    }
}
```

- And:

```
'CODE'

func().();


sub func {
   if (want 'CODE') {
                    ...
    }
}
```

- And:

  'IO'

  ```
  print {func()} @data;
  $next = <func()>;


  sub func {
     if (want 'IO') {
                    ...
      }
  }
  ```

- And:

  'LVALUE'

  ```
  func() = 0;


  sub func {
     if (want 'LVALUE', 'SCALAR') {
                    ...
      }
  }
  ```

- And:

  'RVALUE'

  ```
  $val  = func();
  @vals = func();


  sub func {
     if (want 'RVALUE') {
                    ...
      }
  }
  ```

- Also, could retrieve how many list return values are expected, so only need to generate that many:

```
($line1, $line2, $line3) = func();



sub func {
   $count = want.{COUNT}
    ...
}


sub func {
   if (want 3) {
       ...
   }
}
```

What we might see

- My take on Larry's take on the remaining RFCs
- Based on hints, inferences, and Perl 5+*i*

Guiding principle

- RFC 28: Perl should stay Perl.
- This is a golden opportunity to change everything
- Let's not take it

Connecting with other languages

- RFC 32: A method of allowing foreign objects in perl
- RFC 334: Perl should allow specially attributed subs to be called as C functions
- RFC 121: linkable output mode

- RFC 270: Replace XS with the Inline module as the standard way to extend Perl.
- Seamless access between Perl and C for objects and subroutines
- Automated creation of appropriate glue (data format transformations, type mapping, etc.)
- The Inline proposal is based on the excellent CPAN module of the same name

Slim the core

- RFC 146: Remove socket functions from core
- RFC 155: Remove mathematic and trigonomic functions from core binary
- Put them in modules
- May or not be autoloaded on demand

Compilation

- RFC 301: Cache byte-compiled programs and modules
- RFC 310: Ordered bytecode
- RFC 270: Replace XS with the Inline module as the standard way to extend Perl.
- Cache the internal byte-codes (or object code) of a program or module around for next time
- Possibly in a format that allows JIT or parallel loading

Making list returns more managable

- RFC 37: Positional Return Lists Considered Harmful
- RFC 259: Builtins : Make use of hashref context for garrulous builtins

```
@context = caller(1);
```

- Anyone know which element of `@context` tells you if you're inside an eval?

```
if ( (caller(1))[6] ) {
       print "in eval"
}
```

- Let `caller` and other such routines (e.g. `stat`, `localtime`) detect a 'HASH' context and return data that way instead:

```
if ( caller(1).{eval} ) {
      print "in eval"
}
```

Module termination

- RFC 55: Compilation: Remove requirement for final true value in `require`-d and `do`-ed files
- Almost no-one ever puts anything except `1;` at the end of a module.
- So remove the need to
- Handle failure by exceptions instead

- RFC 225: Data: Superpositions
- Yes, really
- People think they want:

```
if ($val in @array) {...}
```

- But really need:

```
if ($val == any(@array)) {...}
if ($val eq any(@array)) {...}
if ($val ne any(@array)) {...}
if ($val < all(@array)) {...}
```

B&D

- RFC 140: One Should Not Get Away With Ignoring System Call Errors
- RFC 278: Additions to 'use strict' to fix syntactic ambiguities
- More discipline!

```
use strict 'system'
```

- Kills you if you throw away return values of system calls

```
use strict 'objects'
```

- Kills you unless you em-brace indirect objects

```
use strict 'syntax'
```

- Kills you if you even breathe incorrectly whilst you're coding

Loop extraction of multiple values

- RFC 173: Allow multiple loop variables in `foreach` statements

```
foreach my ($x, $y, $z) (@list) {
        ...
}
```

- Iterate variables two or more at a time
- If list was built lazily, would also solve hash iterator problem:

```
while (my ($key1, $val1) = each %hash) {
    while (my ($key2, $val2) = each %hash) {
            print %hash{$key1} ^ %hash{$key2}
    }
}
```

- Becomes:

```
foreach my ($key1, $val1) (%hash) {
    foreach my ($key2, $val2) (%hash) {
            print %hash{$key1} ^ %hash{$key2}
    }
}
```

Remove vestigals

- RFC 195: Retire chop()
- ...because `chomp` is what you almost certainly want.
- And there always:

```
substr($str,-1,1,"");
```

Alternative front-ends

- RFC 329: use syntax
- Lexically scoped compile-time selection of parser:

```
use syntax 'Perl5';

use syntax 'C';

use syntax 'Python';

use syntax 'Latin';

use syntax 'Klingon' or die! (scum);
```

Foreach loop counting

- RFC 120: Implicit counter in for statements, possibly $#
- C-like for's are ugly and unPerlish
- But occasionally necessary:

```
# Perl 5
for ($i = 0; $i < @array; $i++) {
      $array[$i]->getline;
      $array[$i]->parseline;
      $array[$i]->printline;
      $array[$i]->index = $i;
}
```

- Provide an automagic lexically scoped punctuation variable that tracks the iteration number (from zero):

```
# Perl 6
foreach my $object (@array) {
      $object.getline;
      $object.parseline;
      $object.printline;
      $object.index = $#;
}
```

- Or provide an property-induced lexically scoped variable that tracks the iteration number (from zero):

```
# Perl 6
foreach my $object (@array) {
      my $i is index;
      $object.getline;
      $object.parseline;
      $object.printline;
      $object.index = $i;
}
```

Object Oriented Proposals

- Cleaner and richer method dispatch semantics
- Built-in (optional) encapsulation
- Integrate OO and tie mechanisms
- Make good things easier and mistakes harder

Guiding Principle

- RFC 137: Overview: Perl OO should not be fundamentally changed.
- My view
- "It ain't broke."
- No need to "fix", only strengthen

Proper object set-up and clean-up

- RFC 189: Objects : Hierarchical calls to initializers and destructors
- Initializers and clean-up routines with standard names
- `BUILD` and `DESTROY`
- Class's `BUILD` called automagically when an object `bless`'ed
- Arguments to `BUILD` passed as extra arguments to `bless`:

```
    my $self = DogTag.bless(%data, @args);
```

- All ancestral class's `BUILD` methods also called!
- Likewise calls to `DESTROY` become automatically hierarchical

Assertions

- RFC 348: Regex assertions in plain Perl code
- Retarget the mysterious `(?{...})` construct as a zero-width boolean assertion
- Like `(?=...)` and `(?>...)`
- For example, instead of:

```
    /25[0-5]|2[0-4]\d|1?\d{1,2}/
```

- you could write:

```
    /(^\d+$)(?{$1<256})/
```

Subroutine Proposals

- Fewer built-ins
- New declaration syntaxes
- Lazy evaluation
- Wrappers

Coroutines

- RFC 31: Subroutines: Co-routines
- Subroutines that remember where you exited them
- Then resume from that point, next time you invoke them
- Original call's argument list and lexicals are preserved
- Ideal for user-defined iterators:

```
sub next_fib {
      yield my $a = 1;
      yield my $b = 1;
      while (1) {
            ($a, $b) = ($a+$b, $a);
            yield $a;
      };
}
```

- Ideal for user-defined iterators:

```
package Tree;
sub next_inorder (Tree $self) {
      yield $self{left}.next_inorder
            if $self{left};
      yield $self;
      yield $self{right}.next_inorder
            if $self{right};                  return;
}

while ($node = $root.next_inorder()) {
      print $node{data};
}
```

- Or for mapping hashes:

```
%newhash = map {
               yield  process_key($_);
               return process_val($_);
             } %oldhash;
```

Lazy evaluation

- RFC 123: Builtin: lazy
- Lazy evaluation of lists
- Lazy evaluation of argument lists
- Probably not automatic
- Explicitly requested:

```
sub process ($d1 is lazy, $d2 is lazy) {...}

process( expensive1(), expensive2() );
```

Miscellaneous Proposals

- Meta-linguistic
- Legal

Expand the standard library

- RFC 260: More modules
- ...in the standard distribution
- Survey Perl community to see what's missing

Shrink the standard library

- Fewer modules in the standard distribution
- Probably impossible to find a common subset of modules that most people agree are essential
- Instead, provide some easier mechanism to have CPAN.pm install whatever's needed on-the-fly

Licensing

- RFC 211: The Artistic License Must Be Changed
- RFC 346: Perl6's License Should be (GPL | Artistic-2.0)
- A new Artistic License for Perl:
  - Unambiguous
  - Unrestrictive
  - Widely acceptable
  - Commercially viable
  - Legally watertight
  - Still simple

The Dark Side of the Proposals

- Proposals that are very unlikely to be accepted
- Often because Larry has already ruled them out
- Or because they're too unPerlish
- Or because they create more problems than they solve

Inconsistent undefs

- RFC 192: Undef Values ne Value
- Proposal was that `undef != undef`

Global variables seriously deprecated

- RFC 6: Lexical variables made default
- RFC 64: New pragma 'scope' to change Perl's default scoping
- `use strict 'vars'` active by default
- Using an undeclared variable creates a new lexical variable within the scope
- Widely opposed, especially by:
  - RFC 16: Keep default Perl free of constraints such as warnings and strict
  - RFC 330: Global dynamic variables should remain the default
  - RFC 106: lexical variables made default without requiring strict 'vars'

Arrays and hashs assimilated

- RFC 9: Highlander Variable Types
- RFC 341: Unified container theory
- Merge scalars, arrays, and hashes, so only scalars left
- `$array[$index]` becomes an abbreviation for `$array->[$index]`

New comment markers

- RFC 102: Inline Comments for Perl
- RFC 5: Multiline Comments for Perl.

```
#< Would let you write a Perl comment
that spans multiple lines >#

#<or># push #<comments between># @arg1, #<and>#  @arg2;
```

No more prefixes

- RFC 133: Alternate Syntax for variable names
- Get rid of `$`, `@`, and `%` prefixes
- Use bare identifiers to access variables
- Determine type of variable by context

No special names

- RFC 243: No special UPPERCASE_NAME subroutines
- No `BEGIN`, `FETCH`, `AUTOLOAD`, etc.

```
use tie
      FETCH => sub {...},
      STORE => sub {...},
      # etc.
;
```

Pseudo-scalar access to hash entries

- RFC 342: Pascal-like "with"

```
with (%hash) {
      $key1 = <$IN>;     # %hash{key1} = <$IN>;
      $key2 = func();    # %hash{key2} = func();
}
```

- Variation on this might still happen
- The `given` keyword might allow unary `.` operator...

```
given (%hash) {
    .{key1} = <$IN>;   # %hash{key1} = <$IN>;
    .{key2} = func();  # %hash{key2} = func();
}
```

OOO Perl

- RFC 352: Merge Perl and C#, but have default Main class for scripting.
- RFC 73: All Perl core functions should return objects
- RFC 161: Everything in Perl becomes an object.
- Object Oriented Only Perl.

New features for pack and unpack

- RFC 142: Enhanced Pack/Unpack
- RFC 246: pack/unpack uncontrovercial enhancements
- RFC 247: pack/unpack C-like enhancements
- RFC 248: enhanced groups in pack/unpack
- RFC 249: Use pack/unpack for marshalling
- RFC 250: hooks in pack/unpack
- Evidently they're not scary enough yet.

Less punctuated regexes

- RFC 164: Replace =~, !~, m//, s///, and tr// with match(), subst(), and trade()
- Functional style of matching
- Proposed syntax definitely cleaner (less line-noise)
- But almost certainly too radical a change
- "...soulless..."

5. Making sense of the deluge

- If your head is now spinning, bear in mind that we've only just skimmed slightly more than 1/3 of the RFCs
- Larry has to assail, assay, assess, assimilate, and associate twice as many again
- In excruciating detail
- Trying to see implications and interrelationships as well
- That's why the design phase is already stretching far longer than most people expected
- There's not much to be done about that: the task is Herculean and we've already got our best man on it

The final word

- I asked Larry what was the one message he most wanted me to convey to you

The final word

- He replied:

  "My basic message right now is that we're going to take the time to do it right. What we want Perl to be in 2 years is a language that will be good for another 20 years."

- That's something we can all look forward to

6. Resources

`http://yetanother.org/damian/talks/Perl6.v2.pdf`
(These slides)

`http://www.perl.org/perl6/`
(Offical Perl 6 site)

`http://infotrope.net/opensource/software/perl6/`
(Unoffical, but very useful, Perl 6 site)

`http://www.perl.com/pub/2000/10/23/soto2000.html`
(Larry's TPC4 speech)

`http://technetcast.com/tnc_play_stream.html?stream_id=375`
(Larry's Atlanta Linux Showcase speech)

`http://slashdot.org/articles/00/07/20/210229.shtml`
(Geekdom's reaction)

`http://dev.perl.org/`
(Perl 6 repository)

`http://www.perl.com/pub/2000/11/perl6rfc.html`
(A critique of the RFC process)

`http://www.perl.com/pub/2000/11/jarkko.html`
(A rebuttal of the above critique)

`http://www.etla.org/retroperl/`
(Archive of former versions of Perl)

`http://history.perl.org/`
(The Perl timeline)

`http://www.perl.com/pub/2001/04/02/wall.html`
(Larry's first design document)

`http://www.perl.com/pub/2001/05/03/wall.html`
(Larry's second design document)

`http://www.perl.com/pub/2001/05/08/exegesis2.html`
(Explanation of Larry's second design document)