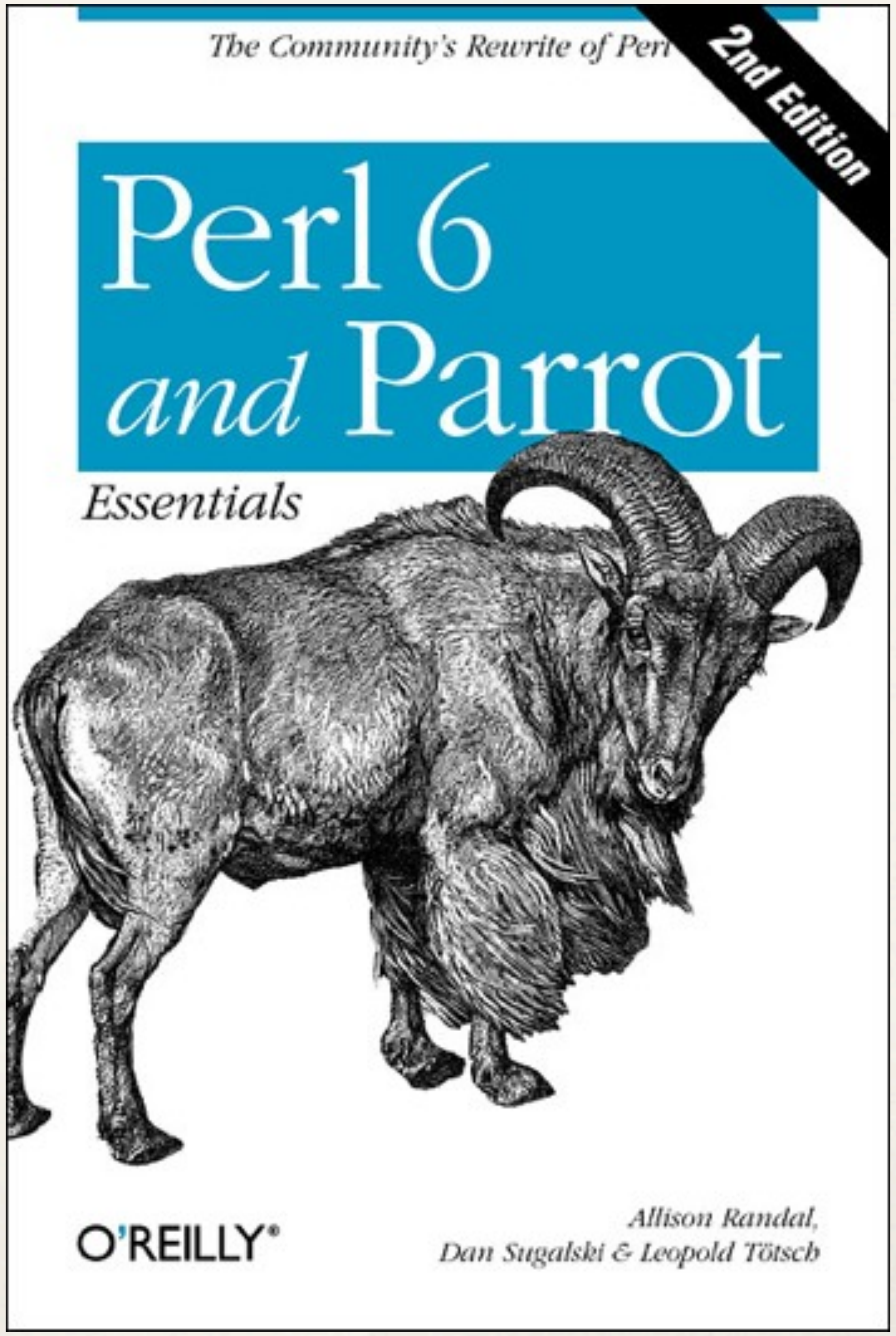
Perl 6 / Parrot


The Community's Rewrite of Perl

2nd Edition

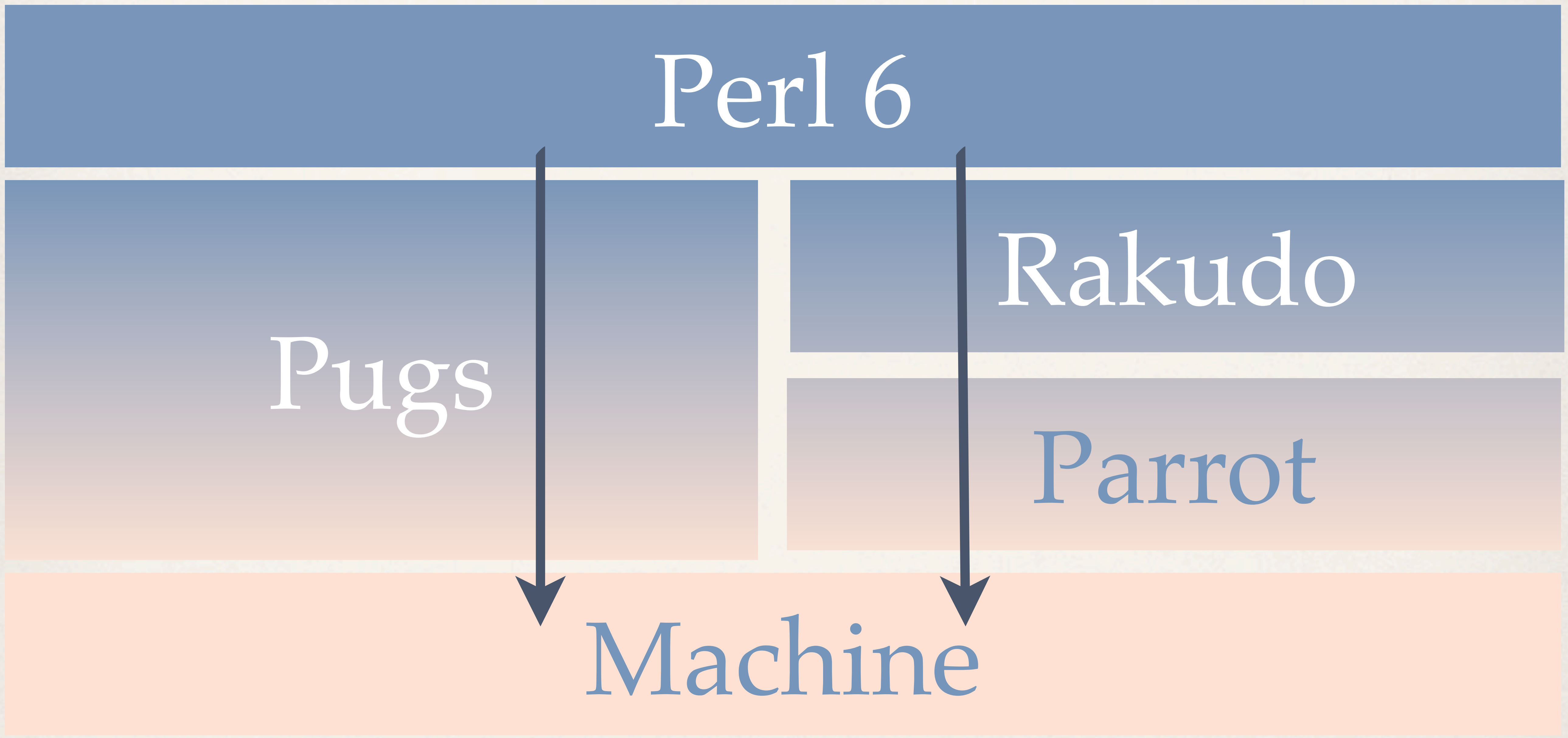
Perl 6 and Parrot Essentials

O'REILLY®

Allison Randal, Dan Sugalski & Leopold Tötsch

April 2009

Pieces and Parts



April 2009

Parrot is a virtual machine designed to efficiently compile and execute bytecode for dynamic languages.

Parrot is designed with the needs of dynamically typed languages (such as Perl and Python) in mind, and should be able to run programs written in these languages more efficiently than VMs developed with static languages in mind (JVM, .NET).

Parrot is also designed to provide interoperability between languages that compile to it. In theory, you will be able to write a class in Perl, subclass it in Python and then instantiate and use that subclass in a Tcl program.

April 2009

Parrot

Submitted by allison on Tue, 03/17/2009 - 21:54.

...

On behalf of the Parrot team, I'm proud to announce Parrot 1.0.0 "Haru Tatsu."

Four instruction formats

- PIR (Parrot Intermediate Representation)
- PASM (Parrot Assembly)
- PAST (Parrot Abstract Syntax Tree) enables Parrot to accept an abstract syntax tree style input - useful for those writing compilers.
- PBC (Parrot Bytecode) The above forms are automatically converted inside Parrot to PBC. Like machine code but understood by the Parrot interpreter. It is not intended to be human-readable or human-writable, but unlike the other forms execution can start immediately, without the need for an assembly phase. PBC is platform independent.

April 2009

http://docs.parrot.org/parrot/latest/html/

Parrot PIR

Parrot Intermediate Representation is designed to be written by people and generated by compilers. It hides away some low-level details, such as the way parameters are passed to functions.

Parrot PIR Example 1/4

```
.sub main
    print "hello, world\n"
.end
```

Parrot PIR Example 2/4

Use a string register:

```
.sub main
    set S0, "hello, world\n"
    print S0
.end
```

Parrot PIR Example 3/4

* replacing S0 with $S0 delegates the choice of which register to use to Parrot.
* use an = notation instead of writing the set instruction.

```
.sub main
    $S0 = "hello, world\n"
    print $S0
.end
```

Parrot PIR Example 4/4

- use named registers (later mapped to real numbered registers)
- .local indicates the register is only needed inside the current subroutine
- The type can be int (for I registers), float (for N registers), string (for S registers), pmc (for P registers) or the name of a PMC type

```
.sub main
    .local string hello
    hello = "hello, world\n"
    print hello
.end
```

PIR Example 5/4

```
sub factorial
    # Get input parameter.
    .param int n

    # return (n > 1 ? n * factorial(n - 1) : 1)
    .local int result

    if n > 1 goto recurse
    result = 1
    goto return

  recurse:
    $I0 = n - 1
    result = factorial($I0)
    result *= n

  return:
    .return (result)
  .end
```

```
.sub main :main
    .local int f, i

    # We'll do factorial 0 to 10.
    i = 0
 loop:
    f = factorial(i)

    print "Factorial of "
    print i
    print " is "
    print f
    print ".\n"

    inc i
    if i <= 10 goto loop
 .end
```

Parrot PASM

Parrot Assembly is a level below PIR - it is still human readable/writable and can be generated by a compiler, but the author has to take care of details such as calling conventions and register allocation.

PASM example

```
main:
 set   I1,0
 ## P9 is used as a stack for temporaries.
 new     P9, 'ResizableIntegerArray'
loop:
 print "fact of "
 print I1
 print " is: "
   new P0, 'Integer'
 set P0,I1
 bsr fact
 print P0
 print "\n"
 inc I1
 eq  I1,31,done
 branch  loop
done:
 end
```

```
### P0 is the number to compute, and also the return value.
fact:
 lt  P0,2,is_one
 ## save I2, because we're gonna trash it.
 push  P9,I2
 set I2,P0
 dec P0
 bsr fact
 mul P0,P0,I2
 pop     I2,P9
 ret
is_one:
 set P0,1
 ret
```

Instruction Set

The Parrot instruction set includes arithmetic and logical operators, compare and branch/jump (for implementing loops, if...then constructs, etc), finding and storing global and lexical variables, working with classes and objects, calling subroutines and methods along with their parameters, I/O, threads and more.

Full list of opcodes at:
http://docs.parrot.org/parrot/latest/html/ops.html

Registers and Data Types

The Parrot VM is register based. This means that, like a hardware CPU, it has a number of fast-access units of storage called registers. There are 4 types of register in Parrot: integers (I), numbers (N), strings (S) and PMCs (P). There are N of each of these, named I0,I1,..N0.., etc. Integer registers are the same size as a word on the machine Parrot is running on and number registers also map to a native floating point type. The amount of registers needed is determined per subroutine at compile-time.

Polymorphic Container (PMC)

PMC stands for Polymorphic Container. PMCs represent any complex data structure or type, including aggregate data types (arrays, hash tables, etc). A PMC can implement its own behavior for arithmetic, logical and string operations performed on it, allowing for language-specific behavior to be introduced. PMCs can be built in to the Parrot executable or dynamically loaded when they are needed.

- src/pmc for Parrot 1.0 contains 81 .pmc files
- perldoc <name.pmc> to see documentation

Garbage Collection

Parrot provides garbage collection, meaning that Parrot programs do not need to free memory explicitly; it will be freed when it is no longer in use (that is, no longer referenced) whenever the garbage collector runs.

Current Active Language Efforts

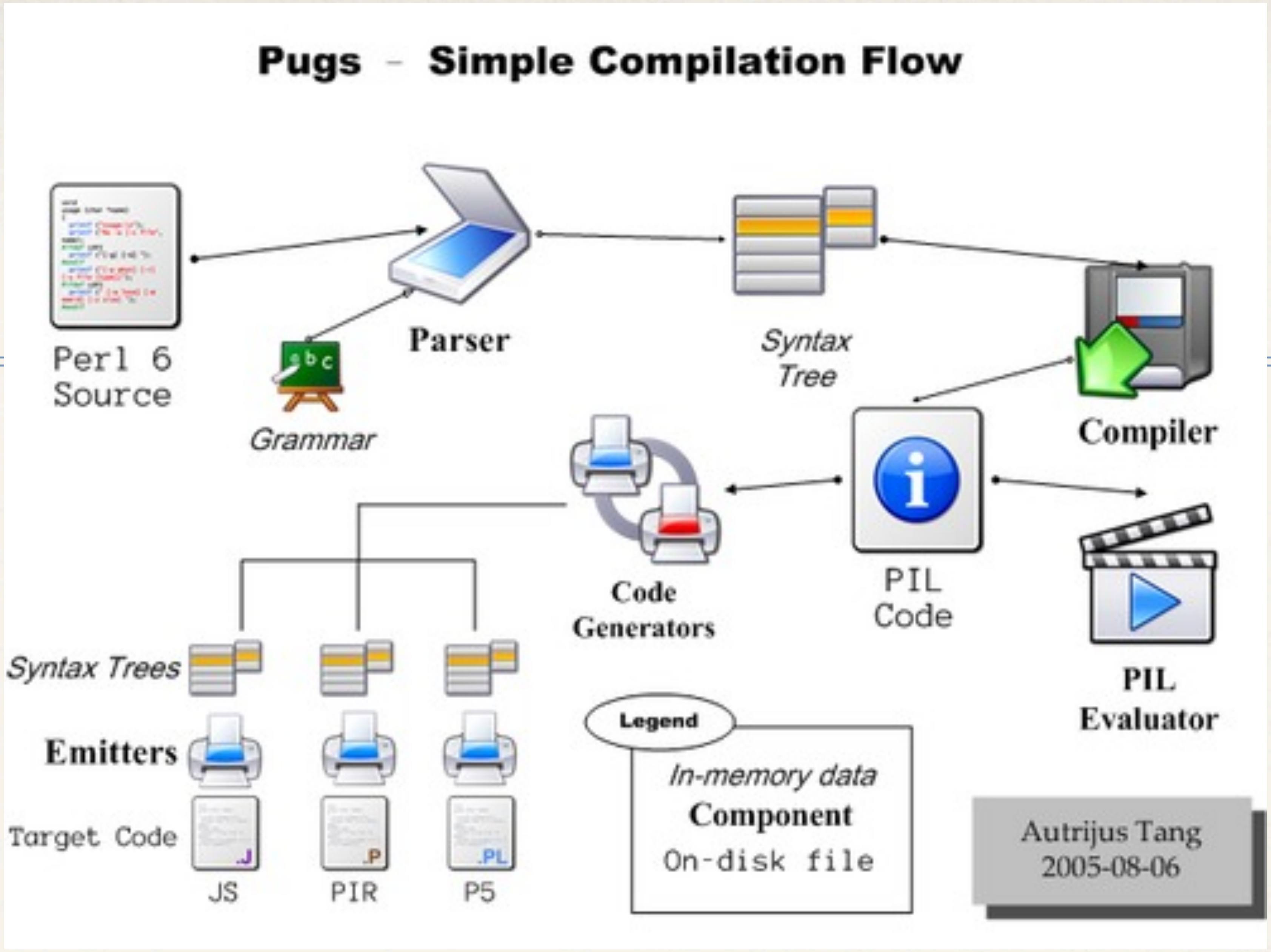
https://trac.parrot.org/parrot/wiki/Languages

cardinal	Ruby 1.9
ecmascript	ECMAScript
fun	An even happier Joy
jvm	Java VM bytecode translator
lua	Lua 5.1
matrixy	Octave
Perk	Java
Pipp	PHP
Porcupine	Pascal
Rakudo Perl 6	Perl 6
WMLScript	WMLScript

Pugs

- Pugs is an implementation of Perl 6, written in Haskell. It aims to implement the full Perl6 specification, as detailed in the Synopses
- Autrijus (Audrey) Tang, February 1st, 2005
- #perl6 on irc.freenode.net
- http://www.pugscode.org and http://dev.pugscode.org/wiki/
- download at: http://www.perlfoundation.org/perl6/index.cgi?download_pugs

Pugs



* pugs is the interpreter with an interactive shell.

* pugscc can compile Perl 6 programs into Haskell code, Perl 5, JavaScript, or Parrot virtual machine's PIR assembly.

Rakudo

- The compiler formerly known as 'perl6'
- An implementation of the Perl 6 specification that will run on the Parrot virtual machine.
- Some years ago, Con Wei Sensei introduced a new martial art: "The Way Of The Camel". Or, in Japanese: "Rakuda-do". This name quickly became abbreviated to "Rakudo", which happens to mean "paradise" in Japanese. (from http://use.perl.org/~pmichaud/journal/35400)
- http://www.rakudo.org
- ***Very*** easy way to get parrot.

Perl 6

http://www.perlfoundation.org/perl6

What is this thing?

- "Perl 5 was my rewrite of Perl. I want Perl 6 to be the community's rewrite of Perl and of the community."
  --Larry Wall, State of the Onion speech
  TPC4 -- July 19, 2000

- Any language which passes the Perl 6 specification tests is an implementation of Perl 6. -- http://www.perlfoundation.org/perl6

⭐ Perl 6 FAQ by Jonathan Worthington

Perl 6

- RFCs -- called for starting in Jul, 2000
- http://perlcabal.org/syn/
  - Apocalypses -- analyses and justifications of the RFCs
  - Exegeses -- explanations of the Apocalypses
  - Synopses: The ***real*** specifications

    Roughly correspond to book chapters of Programming Perl http://www.perlfoundation.org/perl6/index.cgi?the_long_perl_6_super_feature_list

Subroutines, Parameters, and Typing

* optional static type annotations (gradual typing)
* proper parameter lists
* user-defined operators
* multi dispatch
* named arguments
* generics

OO

* declarative classes with strong encapsulation
* full OO exception handling
* multi-dispatched methods (aka method overloading)
* hierarchical construction and destruction
* distributive method dispatch
* method delegation
* many widely useful objects/types
* custom meta classes, meta object programming

Regexes (now Grammars)

* LL and LR grammars (including a built-in grammar for Perl 6 itself, which is an overridable and reusable grammar)

* named regexes

* overlapping and exhaustive regex matches within a string

* named captures

* parse-tree pruning

* incremental regex matching against input streams

Power Features from Functional Languages and Elsewhere

* hypothetical variables
* hyperoperators (i.e. vector processing)
* function currying
* junctions (i.e. superpositional values, subroutines, and types)
* coroutines
* lazy evaluation (including virtual infinite lists)

Things we have in Perl 5 which will just be better in Perl 6

* better threading
* better garbage collection
* much better foreign function interface
    (cross-language support)
* full Unicode processing support
* string processing on various Unicode levels,
    including grapheme level
* a built-in switch statement

λ *April 2009*

Other

* macros (that are implemented in Perl itself)
* user-definable operators (from the full Unicode set)
* active metadata on values, variables, subroutines, and types
* support for the concurrent use of multiple versions of a module
* extensive and powerful introspection facilities (including of POD)
* chained comparisons
* a universally accessible aliasing mechanism
* lexical exporting (via a cleaner, declarative syntax)
* multimorphic equality tests
* state variables
* invariant sigils, plus twigils (minimalist symbolic "Hungarian")

Concurrency

- No user accessible locks
- Software Transactional Memory
- contend blocks
- maybe / defer functions

“contend” means that code executed inside that scope is guaranteed not to be interrupted in any way.

```
my ($x, $y);
        sub c {
            $x -= 3; $y += 3;
            $x < 10 or defer;
        }
        sub d {
            $x += 3; $y -= 3;
            $y < 10 or defer;
        }

        contend {
            # ...
            maybe { c() } maybe { d() };
            # ...
        }
```

"defer" restores the state of the thread at the last checkpoint and will wait there until an external event allows it to potentially run that atomic "contend" section of code again without having to defer again.

"maybe" causes a checkpoint to be made for "defer" for each block in the "maybe" chain, creating an alternate execution path to be followed when a "defer" is done.

Classes in Perl 6

- Two ways to declare classes:
  - Full file is the class:

```
class FullFileClassName;   # rest of file is class
```

  - Block is the class:

```
class BlockClassName {
                # class definition
    }
```

This and following slides flagrantly lifted from Moritz Lenz:
http://perlgeek.de/blog-en/perl-5-to-6/05-objects-and-classes.writeback

```
class HighClass {
        # these two methods do nothing but return the invocant
        method foo {
            return self;
        }
        method bar($s: ) {
            return $s;
        }
}

my HighClass $x .= new;    # same as $x = HighClass.new;
$x.foo.bar                 # same as $x
```

```
class SomeClass {
        has $!a;              # private
        has $.b;              # public
        has $.c is rw;        # public can modify

        method do_stuff {
            # self can use private name instead of public
            # $!b and $.b are the same thing for self
            return $!a + $!b + $!c;
        }
}

my $x = SomeClass.new;

say $x.a;       # ERROR!
say $x.b;       # ok

$x.b = 2;       # ERROR!
$x.c = 3;       # ok
```

```
class Bar { }
class Foo is Bar { }

my Bar $x = Foo.new();   # every Foo is a Bar

class ArrayHash is Hash is Array {
        ...
}
```

```
role Paintable {
        has $.colour is rw;
        method paint { ... }
}

class Shape {
        method area { ... }
}

class Rectangle is Shape does Paintable {
        has $.width;
        has $.height;
        method area {
            $!width * $!height;
        }
}
```

Perl 6 and Perl 5 differences

http://perlcabal.org/syn/Differences.html

Perl 6 example

```
my $i;

loop ($i = 0; $i < 31; $i++) {
     my $fac10 = [*] 1 .. $i;
     say $fac10;
 }
```