Implementing Perl 6

Jonathan Worthington

Dutch Perl Workshop 2008

I didn't know I was giving this talk until yesterday.

I could have written my slides last night, but...

Implementing Perl 6

Guess what will be released at Christmas?*

Guess what will be released at Christmas?*

* Which Christmas not specified.

Perl 6!

Introducing Rakudo

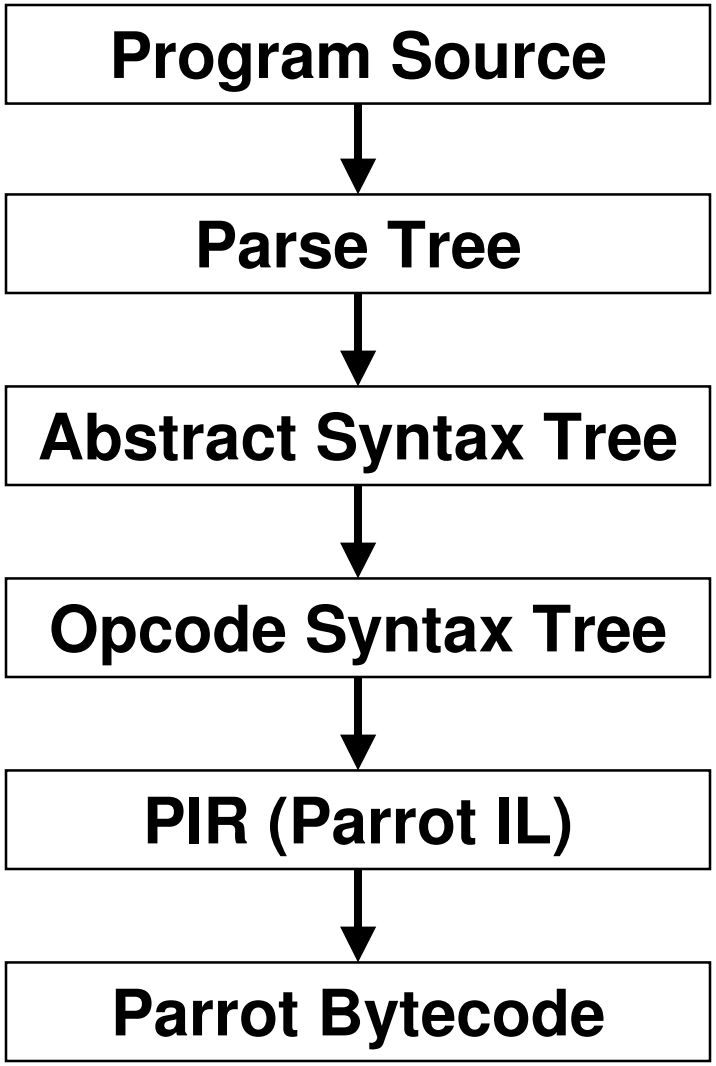
- Name of the Perl 6 compiler targeting the Parrot Virtual Machine
- Parts written in Perl 6
  - Parser written using Perl 6 regexes (now known as rules)
  - Parser actions (more later) written in subset of Perl 6 called NQP
- Other bits in Parrot Intermediate Repr.

Compiler Architecture

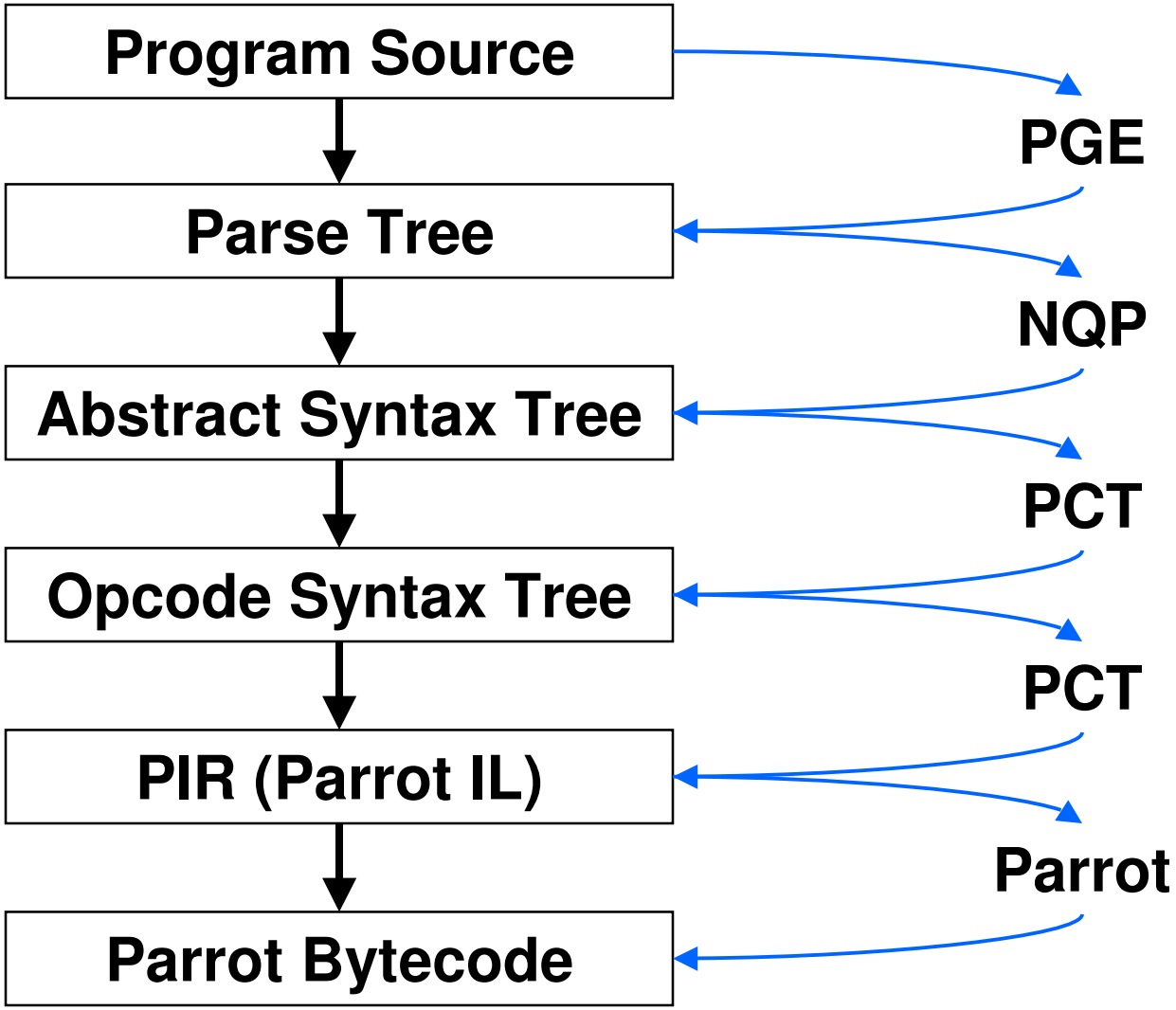
Parrot Compiler Tools

- PCT is a tool chain for building compilers
- You write the "front end":
  - Grammar, which specifies syntax
  - Actions, which produce an Abstract Syntax Tree from the Parse Tree
- The backend (from the AST down to Parrot bytecode) is done for you

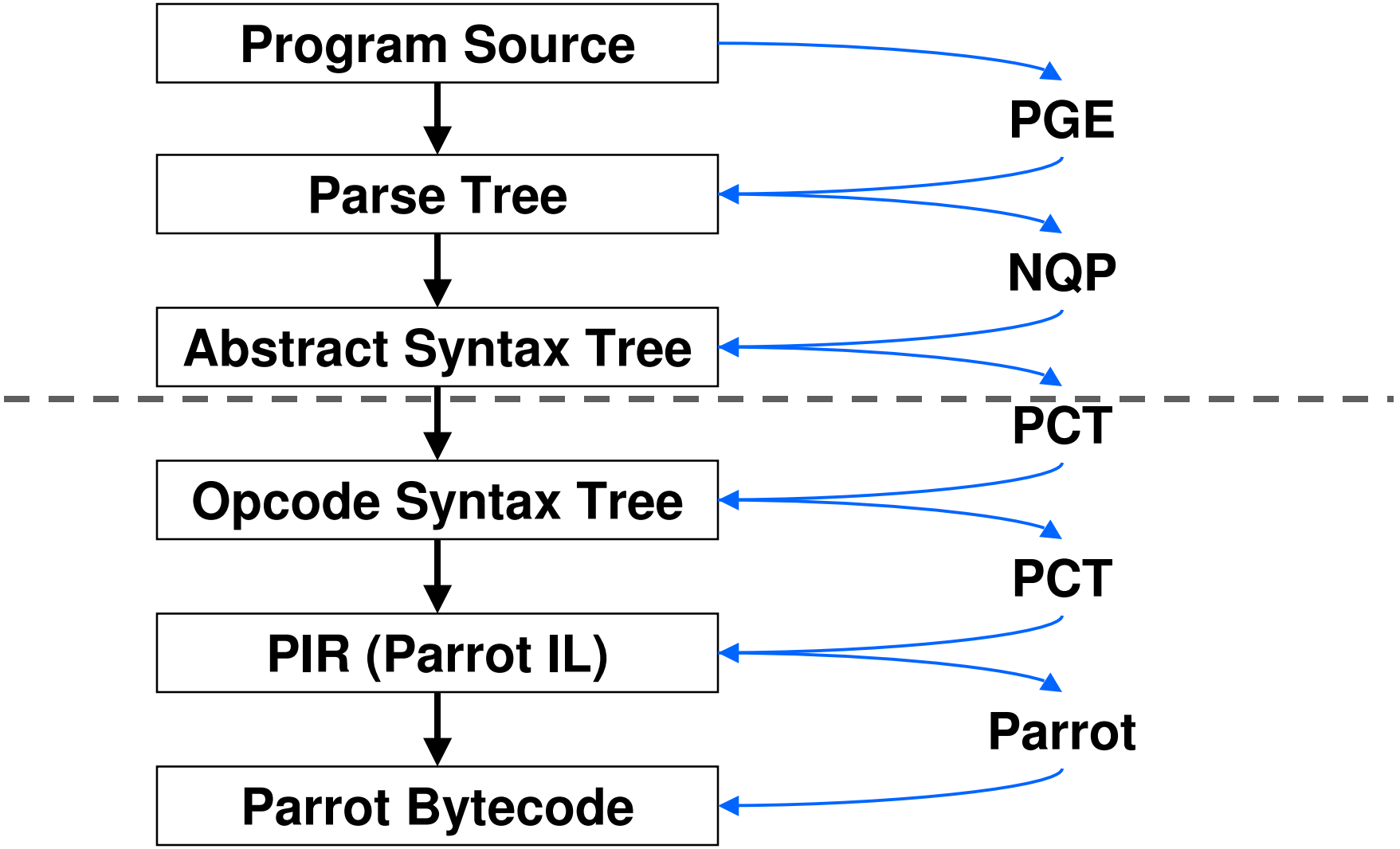
Compilation Process



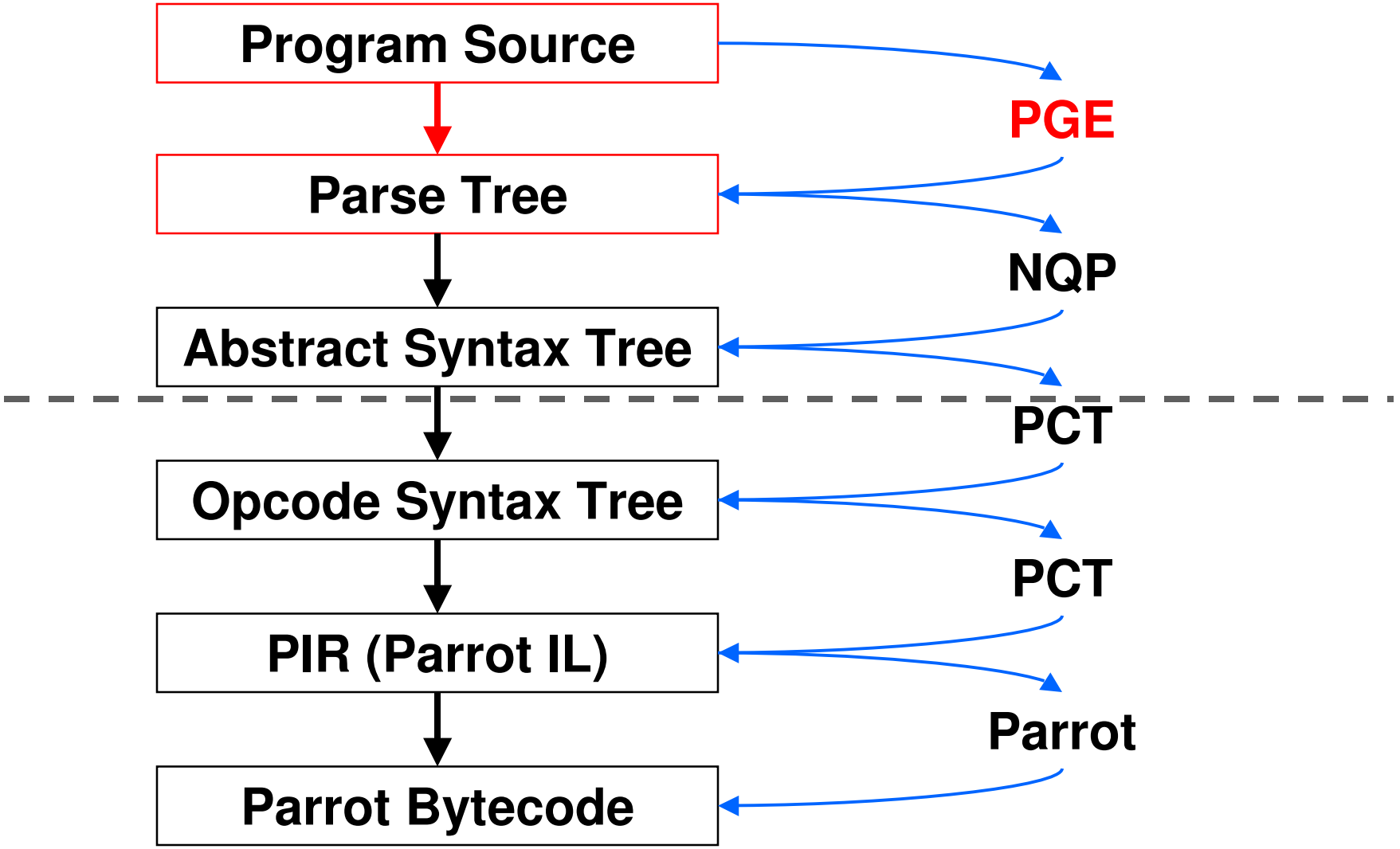
Compilation Process



Compilation Process



Compilation Process



PGE = Parrot Grammar Engine

- Implementation of Perl 6 regexes
- Can name regexes and call them from each other (recursively too)

```
regex Year { \d**4 };
regex Place { Ukrainian | Dutch | German };
regex Workshop {
    <Place> \s Perl \s Workshop \s <Year>
};
regex YAPC {
    'YAPC::' ['EU'|'NA'|'Asia'] \s <Year>
};
regex PerlEvent { <Workshop> | <YAPC> };
```

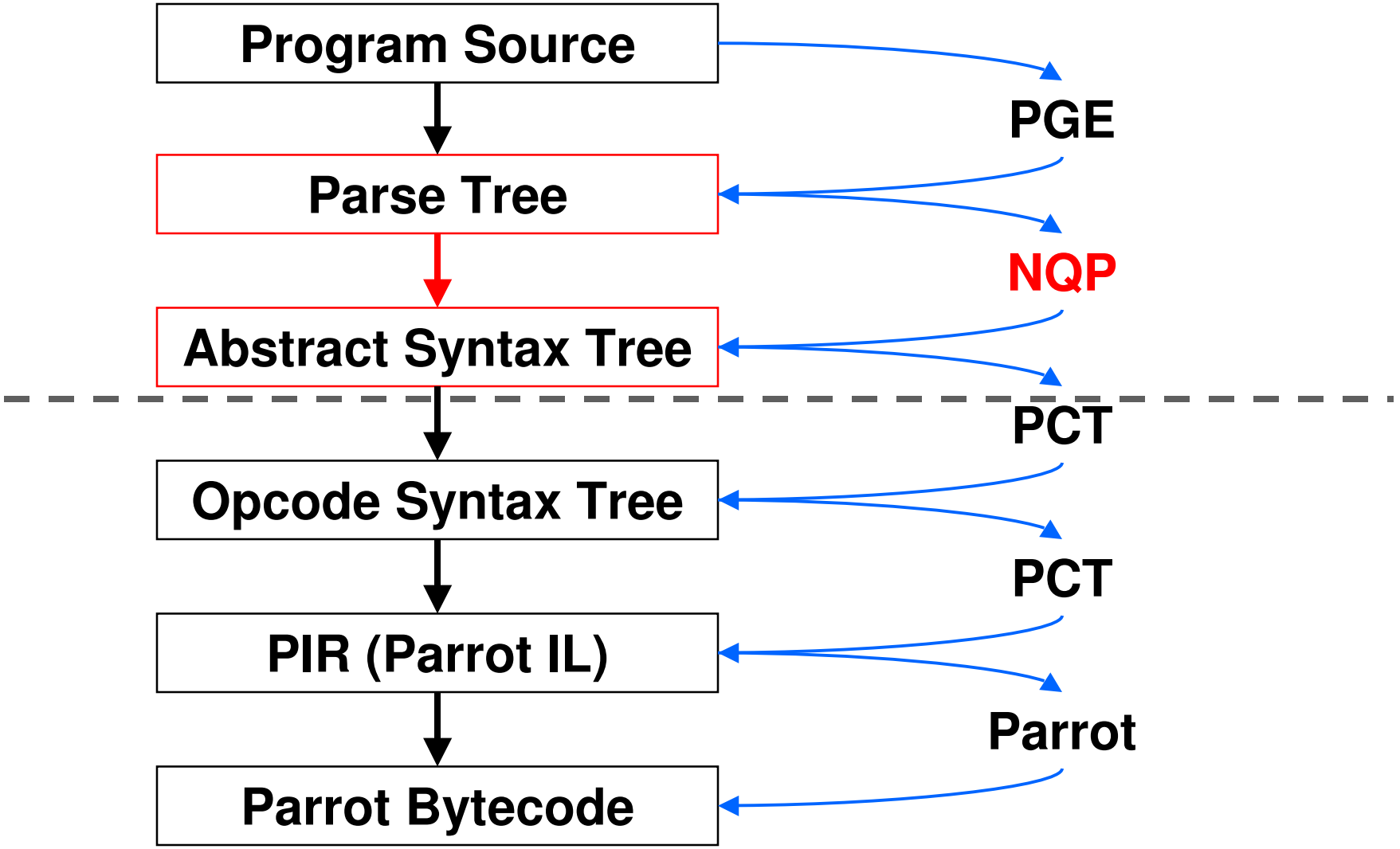
PGE = Parrot Grammar Engine

- You use PGE to write the grammar for your language

```
rule unless_statement {
     'unless'
     <EXPR> <block>
     {*}
}
```

- You put a {*} in place to indicate that we should run an action

Compilation Process



NQP = Not Quite Perl 6

- A subset of Perl 6
- Contains just enough to allow you to produce an Abstract Syntax Tree from the parse tree
  - Variables and literals
  - Binding (but not assignment)
  - Conditionals and loops
  - Object instantiation and method calls

NQP = Not Quite Perl 6

- This method is called when the parser encounters the {*} in the grammar

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- We are passed $/, the match object, which describes what was parsed

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- Named captures ($<….>) give you the match object for the sub rules

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- Writing $($<…>) gets you the AST for that match object

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- We instantiate a new AST node of type Op

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- This node has two children: the condition and the block to run

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- Also specify the type of operation; PCT will then generate the appropriate code

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- Also specify the match object that we made this from, for line numbers etc.

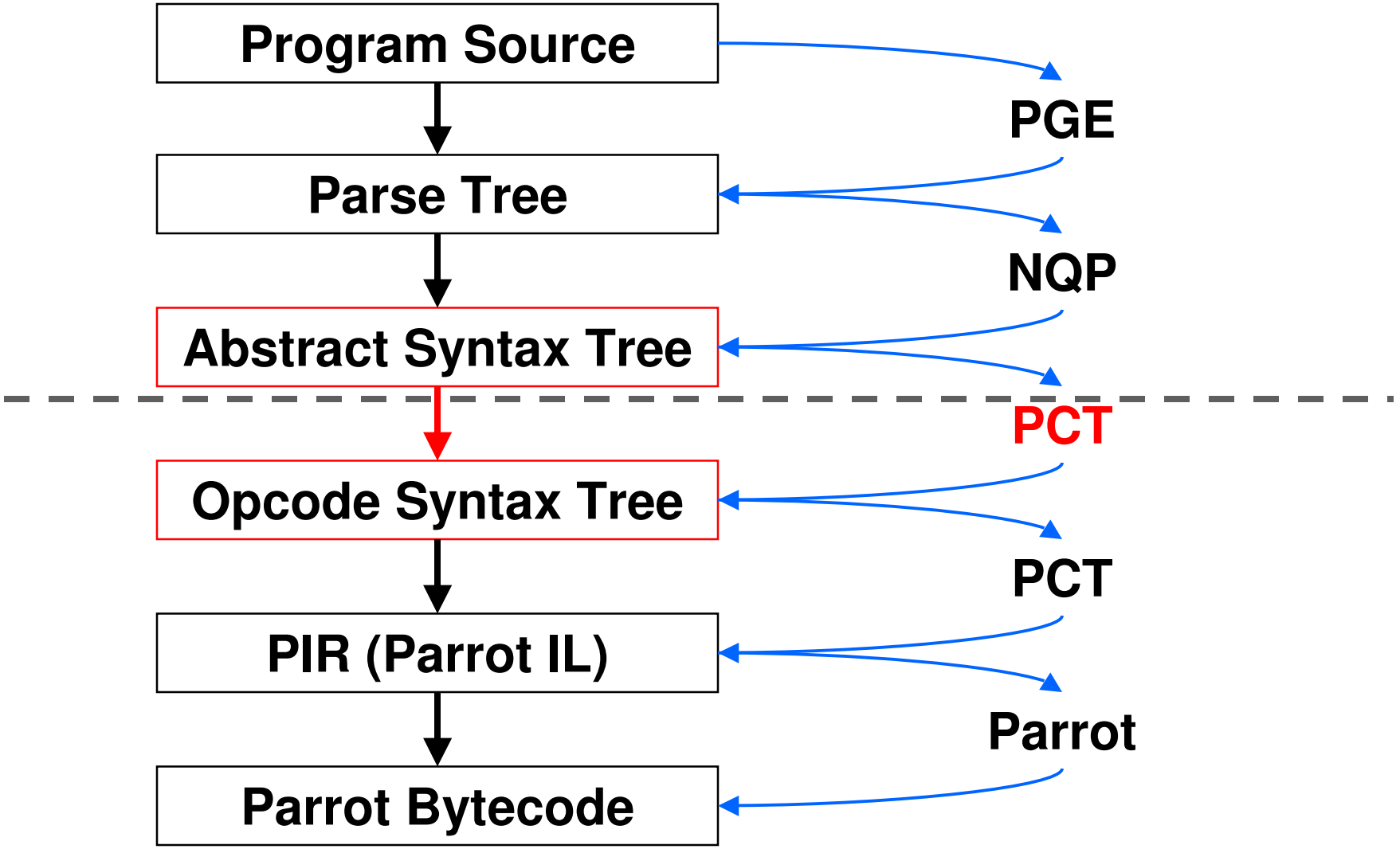
```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

NQP = Not Quite Perl 6

- The "make" statement specifies the tree node we have made

```
method unless_statement($/) {
    my $then := $( $<block> );
    $then.blocktype('immediate');
    my $past := PAST::Op.new(
        $( $<EXPR> ), $then,
        :pasttype('unless'),
        :node( $/ )
    );
    make $past;
}
```

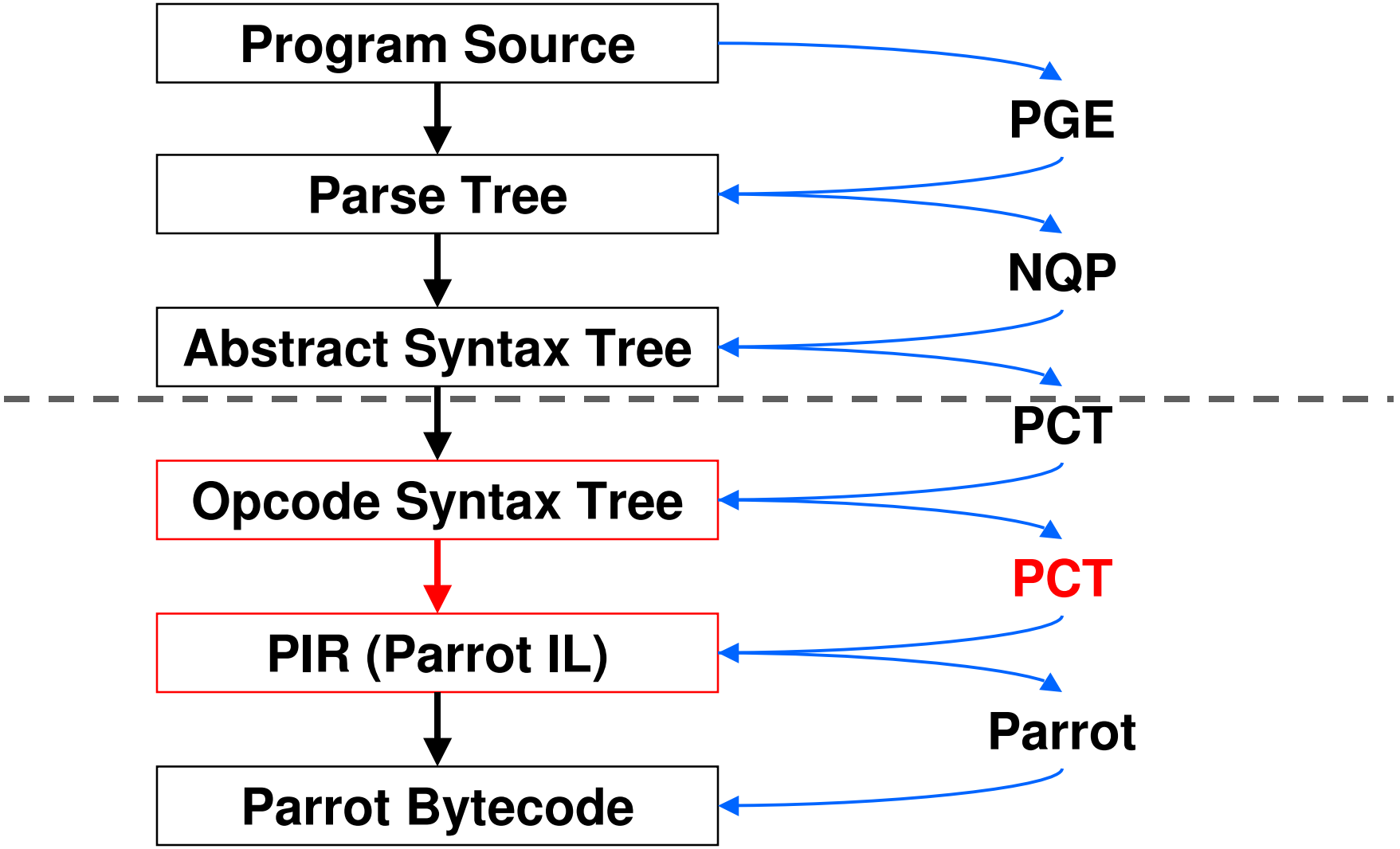
Compilation Process



PAST to POST

- POST is the Parrot Opcode Syntax Tree
  - Tree representation of Parrot assembly program
  - Often one node = one instruction
- The PAST compiler, part of PCT, transforms a PAST node into (usually many) POST nodes

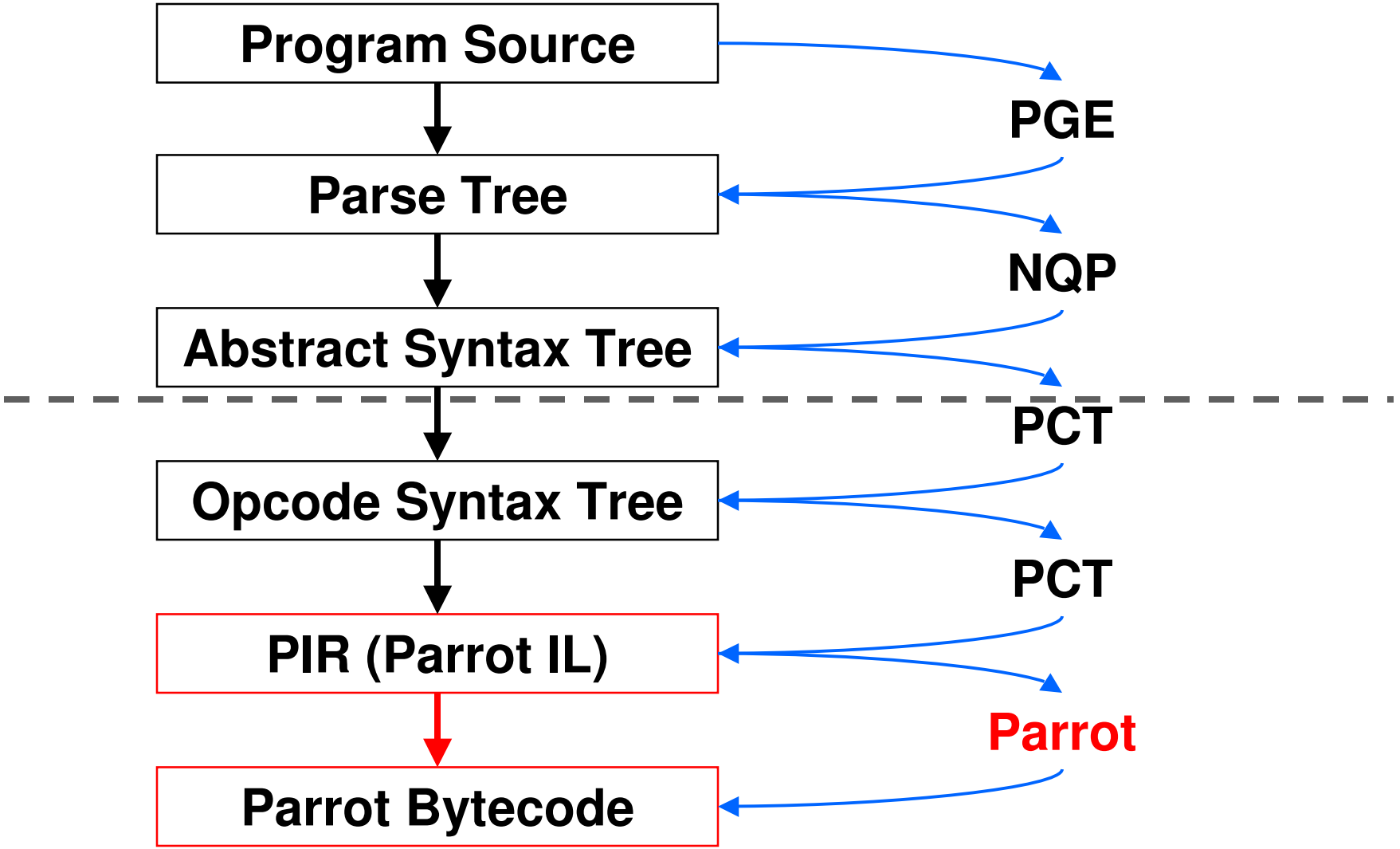
Compilation Process



POST to PIR

- PIR = Parrot Intermediate Representation
- Text based rather than tree based
- The Parrot VM itself understands PIR, so for now we have to turn the POST tree into PIR
- One day, we may be able to go direct from the tree to the bytecode

Compilation Process



PIR to Parrot Bytecode

- The Parrot VM actually executes bytecode – a binary representation of the program
- It contains a compiler that turns PIR into Parrot Bytecode
- We can write the bytecode to disk so we can load it again in the future => don't need to compile our program every time => performance!

The Perl 6 Grammar

STD.pm

- STD.pm is the standard Perl 6 grammar, written in Perl 6 rules
- Mostly complete, though we find missing things occasionally
- PGE doesn't support all of the syntax it uses yet, so we don't use it as is; instead, import it bit by bit and tweak it
- End goal is that they will converge

Two Parsers In One

- Use bottom-up parsing for expressions and top-down parsing for the rest
- Have to call between them
  - When top-down parser needs an expression, uses <EXPR> to call into bottom-up parser to get one
  - If it needs a term, uses <term> to call into the top-down parser to get one

Top-down Parser

- Defined using token, rule and regex

	Backtracking	Sigspace
`regex`	yes	no
`token`	no	no
`rule`	no	yes

- sigspace means replace any whitespace in the pattern with <.ws>, which is the current language's whitespace rule

Bottom-up Parser

- We specify the operators in the expression grammar, for bottom up parsing

```
## multiplicative operators
proto infix:<*> is precedence('u=') { ... }
proto infix:</> is equiv(infix:<*>) { ... }
proto infix:<%> is equiv(infix:<*>) { ... }

## additive operators
proto infix:<+> is precedence('t=') { ... }
proto infix:<-> is equiv(infix:<+>) { ... }
```

Implementing Built-ins

Implementing Operators

- In Perl 6, an operator is just a (multi-dispatch) sub called with special syntax
- Operator implemented in PIR

```
.sub 'infix:+' :multi(_,_)
    .param pmc a
    .param pmc b
    $P0 = n_add a, b
    .return ($P0)
.end
```

Random Aside: Operator Overloading

- Note that because they are just multi-dispatch subs, operator overloading is just an extra sub.
- This is one of the overloads for junctions

```
.sub 'infix:+' :multi('Junction',_)
    .param pmc x
    .param pmc j
    $P0 = find_global 'infix:+'
    .return infix_junc_helper($P0, j, x, 1)
.end
```

Implementing Built-ins

- For now, writing a lot of these in PIR too, because quite a few of them map to Parrot opcodes
- Here is the built-in to compute the co-tangent

```
.sub 'cotan'
    .param num a
    $N0 = tan a
    $N0 = 1 / $N0
    .return ($N0)
.end
```

Implementing Built-ins

- Recently someone submitted a patch to allow writing of built-ins in Perl 6
- Has needed a few tweaks, but folks are working on that and it will be applied probably within a week or so
- Will write what we can in Perl 6 rather than PIR, but some things will always just be easier to do in PIR

What's Implemented

Never do live demos…

- Because it WILL go wrong
- Because somebody will probably have checked in something that broke what you are about to demonstrate
- Because when things don't work everyone will think…
  - I didn't learn Perl 6 yet
  - That Rakudo sucks, not me

How To Play And Help

How To Build Rakudo

- Check out the source from SVN https://svn.perl.org/parrot/trunk/
- Build it:

```
perl Configure.pl
make perl6
```

- Run it on the command line, with a script or in interactive mode

```
perl6 -e "say 'Hello, world!'"
perl6 script.p6
perl6
```

How To Explore The Source

- Go into the rakudo directory

```
cd languages/perl6
```

- In here you should run the PBC file, not the executable

```
../../parrot perl6.pbc
```

- Most exciting stuff in the src directory, especially under classes, builtins and parser

Ways To Help

- Try to use it and report problems that you encounter
- Contribute to the test suite
- Write a built-in (some fairly easy stuff here; anyone up for implementing `sort`?)
- Contribute to the grammar and actions

Implementing Perl 6

rakudo.org

parrotcode.org

dev.perl.org/perl6/

Thank You

Questions?