leanovate
beyond product

PHP OGO NFN MEM LDL KCK JBJ

Bring PHP to the Java-World
(well … actually it is Scala)

by Bodo Junglas

What we would like to cover today

leanovate
beyond product

Agenda

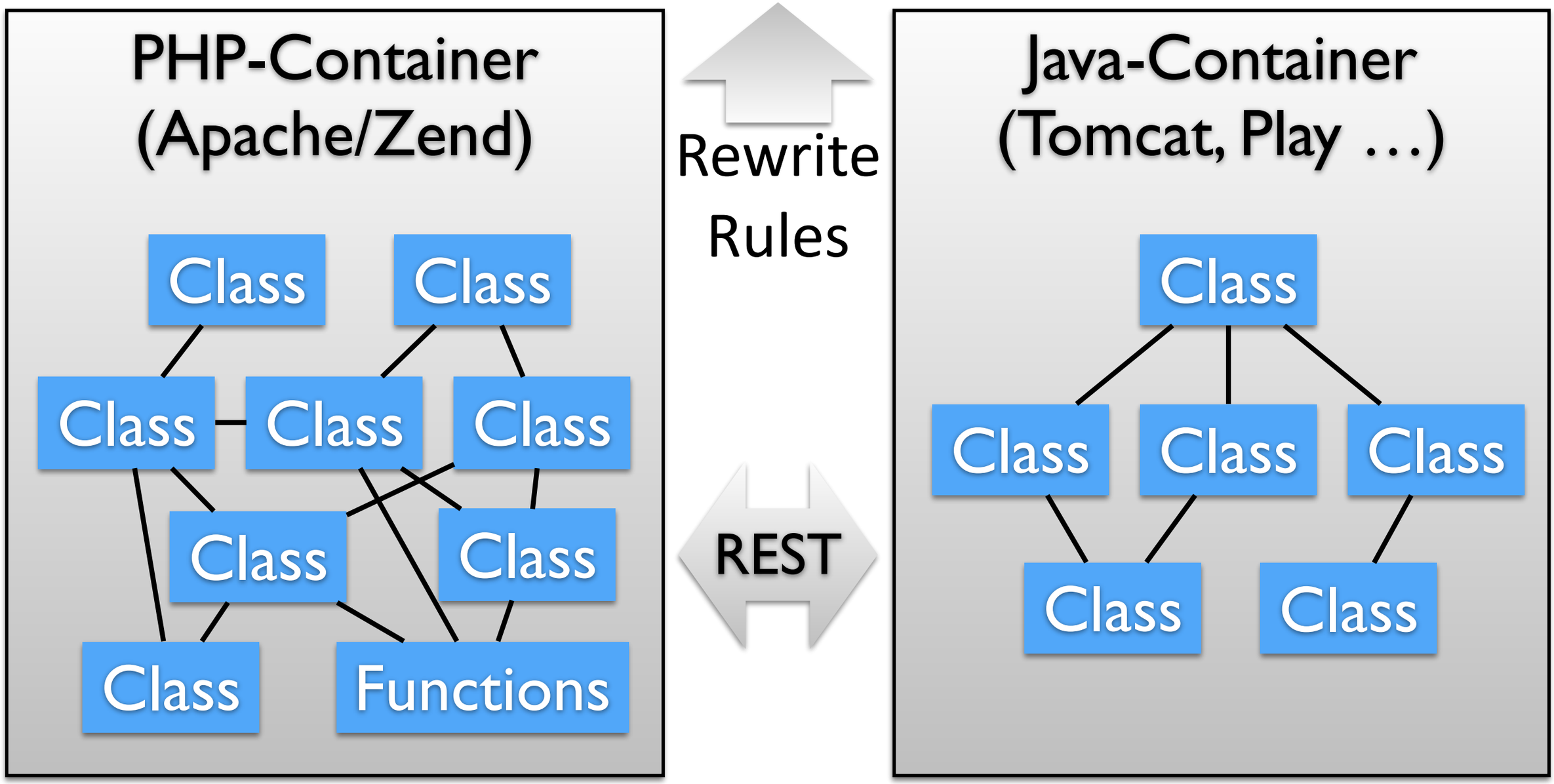
- Motivation and goals
- Is converted code still readable?
- Compatibility and test suite
- Ugly features of PHP (Why is this so complicated)
- Overall project layout
- How to write an interpreter in Scala

Why do we even want to do this?

leanovate
beyond product

1001

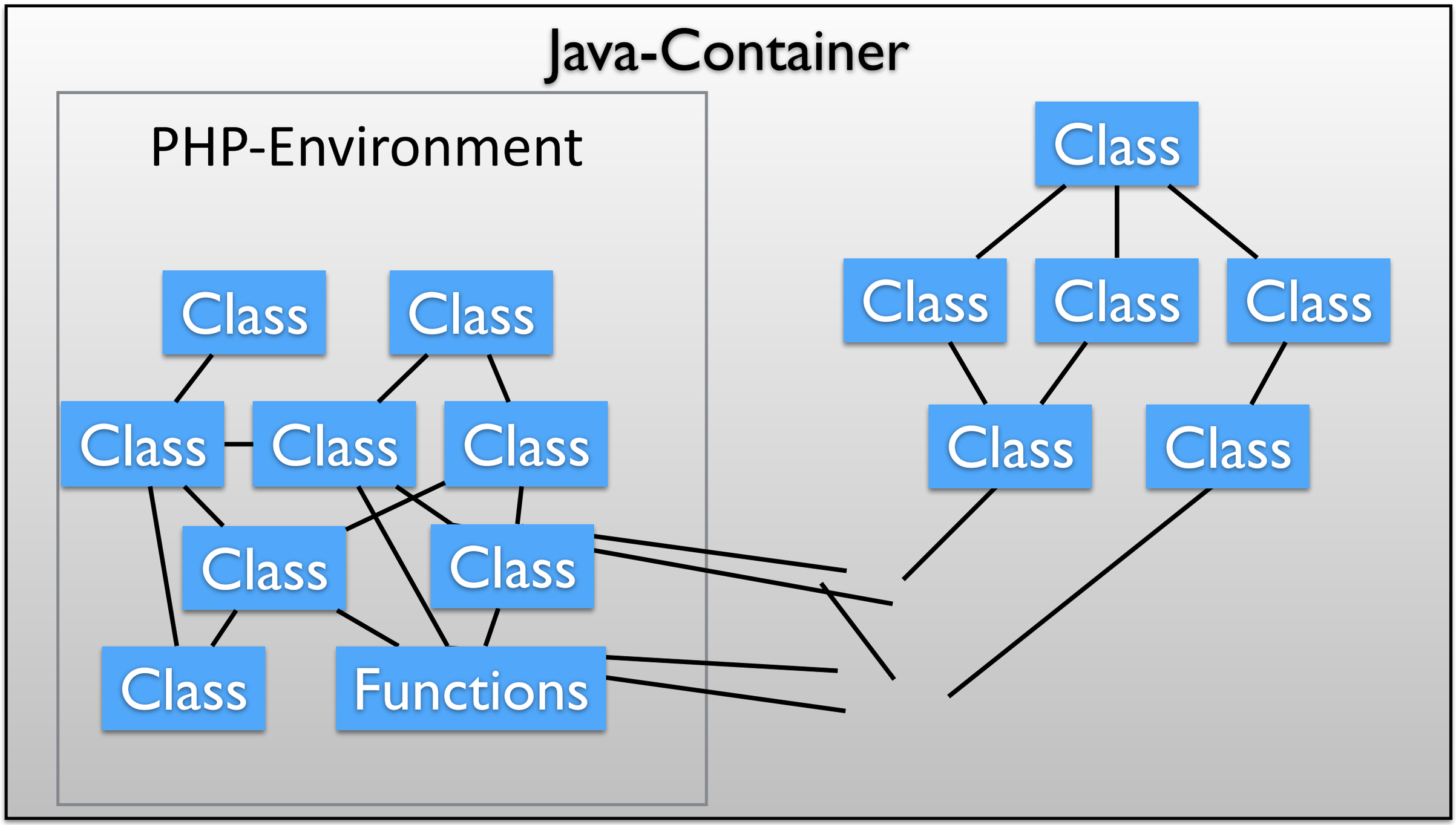
Real world example: Consider a large project with lots of legacy PHP code that wants to migrate to Java



Wouldn't it be much nicer, if ...

leanovate
beyond product

... we could use refactoring tools



Where do we see ourselves?

leanovate
beyond product

1001

The defining goals of JBJ

- Offer a way to run existing PHP code inside a Java VM (i.e. a PHP interpreter inside the Java-VM)
- Allow interaction between PHP and Java
- Automatic conversion of PHP code that …
  - … runs transparently with the remaining PHP code
  - … gives developers a starting point where to begin structured refactoring
  - … is still readable

… and pigs might just fly …

leanovate
beyond product

1001

Other projects with a similar direction:

- Quercus
  - Nearly complete PHP interpreter in Java
  - Part of Caucho/Resin, GPL license
  - Does not seem to be community driven
- JPHP
  - Compiles PHP Java-VM byte-code
  - Github project / Apache 2 license
- Project Zero/WebSphere sMesh
  - Probably dead by now

… and the other way round:

- PJP - PHP/Java Bridge
  - Tries to integrate the Java-VM into the PHP interpreter

… but non of them offers a real conversion.

What is next

leanovate
beyond product

1001

Agenda

- Motivation and goals
- Is converted code still readable?
- Compatibility and test suite
- Ugly features of PHP (Why is this so complicated)
- Overall project layout
- How to write an interpreter in Scala

Very first roundtrip.

leanovate
beyond product

1001

```
1 This is before
2 <?php
3     print "Hello" . " " . "world";
4 ?>
5 This is after
```

March 7 2014 on a Train Berlin->Dortmund

»After gaining consciousness its first intent was to kill its creator.«

```
 1 package testunits
 2
 3 import de.leanovate.jbj.runtime.context.Context
 4 import de.leanovate.jbj.runtime.value._
 5 import de.leanovate.jbj.runtime.JbjCodeUnit
 6
 7 object hello_world extends JbjCodeUnit {
 8
 9   def exec(implicit ctx: Context) {
10
11     ctx.out.print("""This is before
12       |""".stripMargin)
13     ctx.out.print("")
14     ctx.out.print(((StringVal("""Hello""") !! StringVal(""" """)) !!
   StringVal("""world""")).toOutput)
15     ctx.out.print("""This is after
16       |""".stripMargin)
17   }
18 }
```

More recent examples (Hello world)

leanovate beyond product

1001

```
1 This is before
2 <?php
3     print "Hello" . " " . "world";
4 ?>
5 This is after
```

```
1 trait hello_world extends JbjCodeUnit {
2
3   def exec(implicit ctx: Context) {
4
5     inline("This is before\n")
6     print(p("Hello") !! p(" ") !! p("world"))
7     inline("This is after\n")
8   }
9 }
```

- p(…) converts a scala Int, String, … to its PHP-counterpart (might become an implicit conversion)
- inline(…) encapsulates everything outside <?php ?>
- »!!« is a replacement for PHP's ».«

More recent examples (Variables)

leanovate
beyond product

1001

```php
<?php
$a = "Hello";
$b = "world";
$c = $a . " " . $b;

echo $c;

$d = $c + 42;

echo $d;
?>
```

```
trait hello_world2 extends JbjCodeUnit {

  def exec(implicit ctx: Context) {
    val a = lvar("a")
    val b = lvar("b")
    val c = lvar("c")
    val d = lvar("d")

    a := p("Hello")
    b := p("world")
    c := a !! p(" ") !! b
    echo(c)
    d := c + p(42L)
    echo(d)
  }
}
```

- Variables have to be declared with lvar(…) helper
- Assignment is done with »:=«

More recent examples (Arrays, Loops)


```php
<?php
 $a = array("Hello", "World", 42);

 for($i=0; $i<count($a); $i++) {
   echo $a[$i];
   $a[$i] = ($i + 2) * $i + 1;
   echo "\n";
 }

 for($i=0; $i<count($a); $i++) {
   echo $a[$i];
   echo "\n";
 }
?>
```

```
trait hello_world3 extends JbjCodeUnit {
 def exec(implicit ctx: Context) {
  val a = lvar("a")
  val i = lvar("i")

  a := array(p("Hello"), p("World"), p(42L))
  pFor(i := p(0L), i < p(count(a)), i.++) {
   echo(a.dim(i))
   a.dim(i) := (i + p(2L)) * i + p(1L)
   echo(p("\n"))
  }
  pFor(i := p(0L), i < p(count(a)), i.++) {
   echo(a.dim(i))
   echo(p("\n"))
  }
 }
}
```

- array(…) helper to create PHP-style arrays
- pFor(…,…,…) helper to create PHP-style for-loops

What is next

leanovate
beyond product

Agenda

- Motivation and goals
- Is converted code still readable?
- **Compatibility and test suite**
- Ugly features of PHP (Why is this so complicated)
- Overall project layout
- How to write an interpreter in Scala

How to test an interpreter?

leanovate
beyond product

1001

How to ensure compatibility

- Run lots of PHP scripts focussing on different aspects of the language
- See that all of them run smoothly (i.e. without any unexpected runtime exceptions)
- Compare the output with the expected output generated by the »real« PHP interpreter

… luckily there already is a test suite.

1001

The test suite of the PHP interpreter itself operates just like this. Look out for "*.phpt" files

lang/008.phpt

```
--TEST--
Testing recursive function
--FILE--
<?php

function Test()
{
        static $a=1;
        echo "$a ";
        $a++;
        if($a<10): Test(); endif;
}

Test();

?>
--EXPECT--
1 2 3 4 5 6 7 8 9
```

… which can be easily reused.

de.leanovate.jbj.core.tests.lang.Lang1Spec.scala

```
1 "Testing recursive function" in {
2   // lang/008
3   script(
4     """<?php
5       |
6       |function Test()
7       |{
8       | static $a=1;
9       | echo "$a ";
10      | $a++;
11      | if($a<10): Test(); endif;
12      |}
13      |
14      |Test();
15      |
16      |?>""".stripMargin
17  ).result must haveOutput(
18    """1 2 3 4 5 6 7 8 9 """.stripMargin
19  )
20 }
```

Where we are at the moment

leanovate
beyond product

1001

Raw test count:

- PHP's tests are split up:
  - 761 legacy tests
  - 1414 Zend engine tests
  - I.e. 2175 core interpreter tests
- JBJ: >820 core tests

But:

- This is just the core interpreter
- Every PHP extension has its own set of tests
  - Total sum: 12729

What is next

leanovate
beyond product

Agenda

- Motivation and goals
- Compatibility and test suite
- Is converted code still readable?
- Ugly features of PHP (Why is this so complicated)
- Overall project layout
- How to write an interpreter in Scala

Why not just Copy&Paste

leanovate
beyond product

1001

PHP is not easily converted:

- PHP is around since 1995 and has been influenced by several languages and concepts. Some of its features do not translate well to the Scala-world
- Even though some features could be considered »legacy« now, only developers can decide if a certain feature is relevant for some existing code or not

Hurdle 1: Implicit type conversions

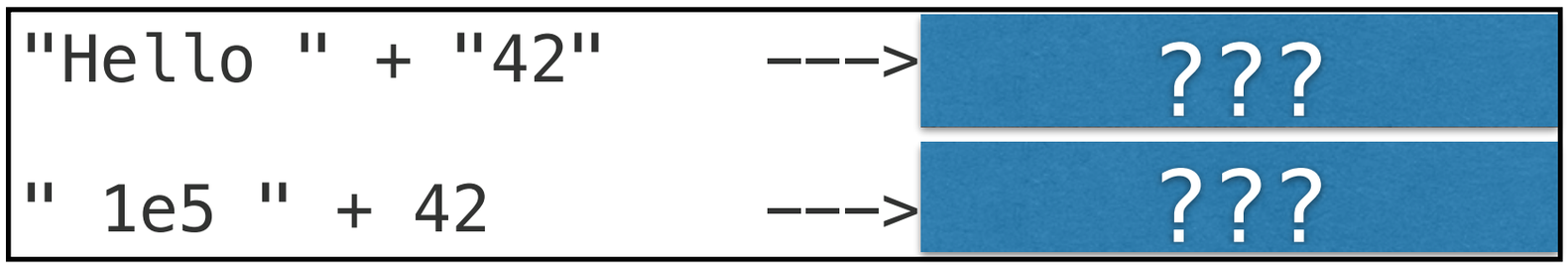
1001

Concatenation operator

```
"Hello " . "42"     ---> (string) "Hello 42"

"Hello " . 42       ---> (string) "Hello 42"
```

Arithmetic operators



Hurdle 1: Implicit type conversions

leanovate
beyond product

1001

Logical operators

```
"Hello " && true        --->  ???

"false" && true         --->  ???

"" && true              --->  ???

0 && true               --->  ???
```

Bitwise operators

```
"Hello" | "abcde"       --->  ???

"Hello" | 10            --->  ???

"13" | 10               --->  ???
```

Hurdle 1: Implicit type conversions

leanovate
beyond product

1001

Comparison operator

```
"42" < "10000"          --->  ???

"42a" < "10000"         --->  ???

42 < "10000"            --->  ???

42 < "10000a"           --->  ???

42 < "a10000"           --->  ???
```

Hurdle 1: Implicit type conversions

leanovate
beyond product

1001

pre/post-fix operators

```
$a = 1
$a++                    ---> (int) 2

$a = 10
$a--                    ---> (int) 9

$a = "Hello"
$a++                    --->  ???

$a = "Hello"
$a--                    --->  ???
```

Hurdle 2: By-Reference

By-reference parameters

```php
<?php

function squareIt(&$x) {
    $x = $x * $x;
}

$a = 2;
squareIt($a);
print "Result: $a\n";
?>
```

```
Result: 4
```

Hurdle 2: By-Reference

By-reference variables

```php
<?php
$a = 4;
$b = array(1, 2, 3, &$a);
$c = &$a;

echo "1. b[3] = ${b[3]}  a = $a\n";
$c = 1;
echo "2. b[3] = ${b[3]}  a = $a\n";
$b[3] = 8;
echo "3. c = $c   a = $a\n";
?>
```

```
1. b[3] = 4  a = 4
2. b[3] = 1  a = 1
3. c = 8   a = 8
```

Hurdle 3: Generators

leanovate
beyond product

1001

A hint of Python

```php
<?php
function generateNums() {
    for ( $i = 1; $i < 5; $i++ ) {
        yield $i;
    }
};
$generator = generateNums(); // this is a Generator class
                             // implementing the Iterator
                             // interface
foreach ($generator as $value) {
    print "Value: $value\n";
}
?>
```

Since PHP 5.5

Hurdle 4: Classes, interfaces … traits

Some Java, some C++

```php
<?php

class A {
    function __construct() {
        print "constructor\n";
    }

    function __destruct() {
        print "destructor\n";
    }
}

print "start\n";
$a = new A();
print "middle\n";
$a = NULL;
print "end\n";
?>
```

```
start
constructor
middle
destructor
end
```

Hurdle 5: Lambda expressions

A hint of Javascript

```php
<?php
$result = 0;

$one = function()
{ var_dump($result); };

$two = function() use ($result)
{ var_dump($result); };

$three = function() use (&$result)
{ var_dump($result); };

$result++;

$one();
$two();
$three();
?>
```

```
PHP Notice:  Undefined variable

NULL
int(0)
int(1)
```

Since PHP 5.3

What is next

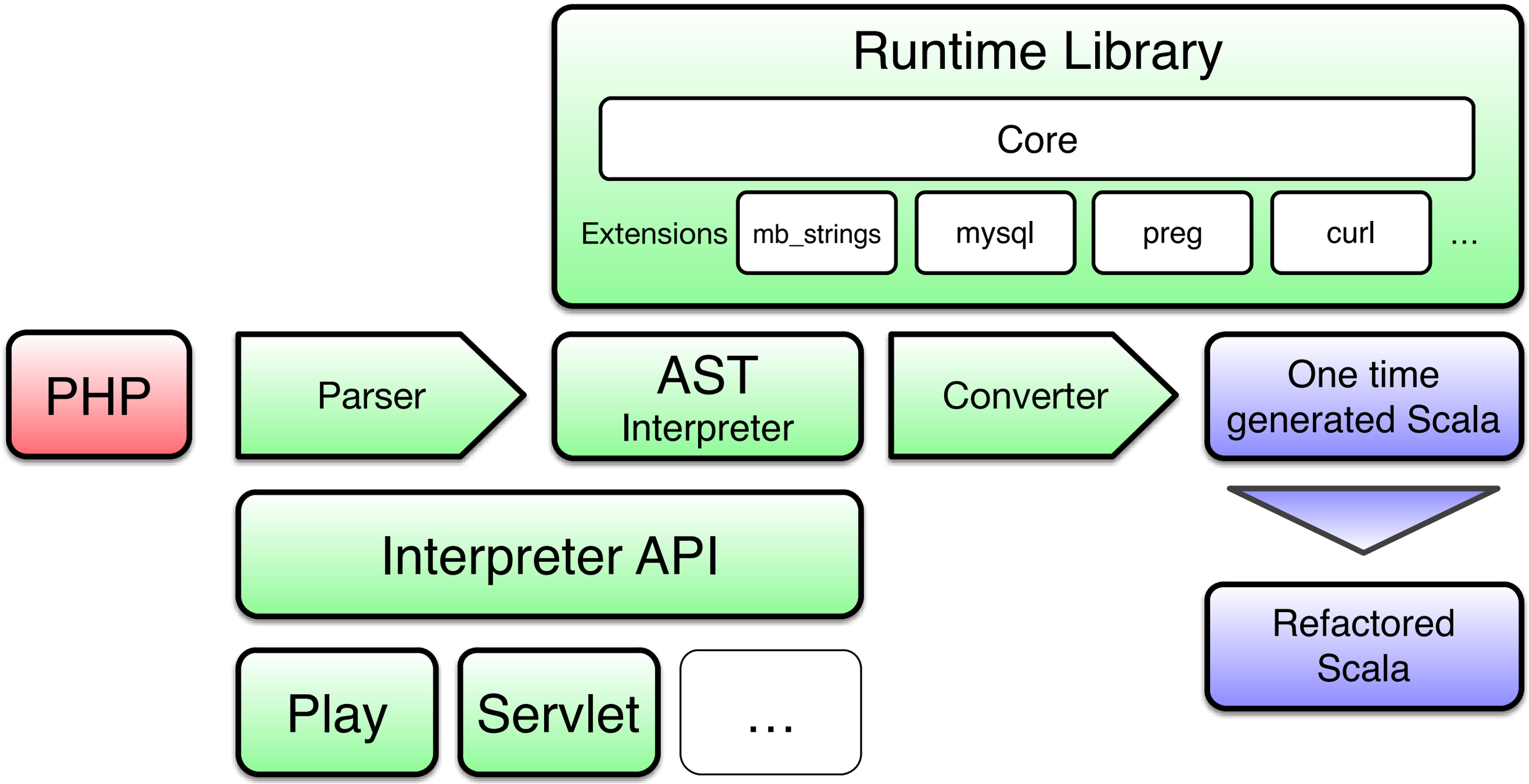
leanovate
beyond product

Agenda

- Motivation and goals
- Is converted code still readable?
- Compatibility and test suite
- Ugly features of PHP (Why is this so complicated)
- Overall project layout
- How to write an interpreter in Scala

Its good to have a plan

Project structure



How PHP is running it

Classical lexer/parser

- Lexical analyzer generated by »flex« (traditionally by »lex« as part of the POSIX standard)
- Parser generated by »bison« (traditionally by »yacc« as part of the POSIX standard)
- PHP’s parser compiles the source-code to a sequence of Op-Codes that a run by the Zend-Engine (i.e. Zend-Engine is the VM of PHP)

Its nice to have a choice

leanovate
beyond product

Alternatives in Java

- JavaCC: lexer + parser
- AntLR: lexer + parser
- JLex/JFlex: lexer
- CUP: parser
- byacc/J: parser
- jay: parser
- …

Many of these generate codes that exceeds the 64kb method size limit of Java

What is next

leanovate
beyond product

Agenda

- Motivation and goals
- Is converted code still readable?
- Compatibility and test suite
- Ugly features of PHP (Why is this so complicated)
- Overall project layout
- How to write an interpreter in Scala

Scala has its own toolset for parsers

leanovate beyond product

Scala combinators: Parsers for free

```
1 package scala.util.parsing.combinator
2
3 trait Parsers {
4     type Elem
5
6     trait Parser {
7         def apply(input: Reader[Elem]) : ParseResult[T]
8     ...
9     }
10
11    sealed abstract class ParseResult[+T]
12
13    case class Success[+T](...) extends ParseResult[T]
14    case class Failure(...) extends ParseResult[Nothing]
15    case class Error(...) extends ParseResult[Nothing]
16 ...
17 }
```

- There is no distinction between lexer and parser

Getting started with combinators

»Hello World« for parsers: Calculator

```
1 class Calculator1 extends Parsers {
2   type Elem = Char
3
4   def expr: Parser[Int] = addition | subtraction | number
5
6   def addition: Parser[Int] =
7     number ~ '+' ~ number ^^ { case left ~ _ ~ right => left + right }
8
9   def subtraction: Parser[Int] =
10    number ~ '-' ~ number ^^ { case left ~ _ ~ right => left - right }
11
12  def number: Parser[Int] =
13    digit.+ ^^ { digits => digits.mkString("").toInt }
14
15  def digit: Parser[Char] = elem("digit", ch => ch.isDigit)
16
17  def parse(str: String):Int = expr(new CharSequenceReader(str)) match {
18    case Success(result, remain) if remain.atEnd => result
19 ... error handling
20  }
21 }
```

How to combine

Combinator operators

```
15    def digit: Parser[Char] = elem("digit", ch => ch.isDigit)
```

»elem(kind: String, condition: Elem => Boolean«
creates a parser that consumes a single element if a condition is met

```
12    def number: Parser[Int] =
13      digit.+ ^^ { digits => digits.mkString("").toInt }
```

»rep1(p: => Parser[T]): Parser[List[T]]« (or »+« postfix) creates a parser by repeating a given parser at least once.

»^^« maps the result of a parser

How to combine

Combinator operators

```
 6   def addition: Parser[Int] =
 7     number ~ '+' ~ number ^^ { case left ~ _ ~ right => left + right }
 8
 9   def subtraction: Parser[Int] =
10     number ~ '-' ~ number ^^ { case left ~ _ ~ right => left - right }
```

»~« combines two parsers to a new one that is only successful if both parsers are successful in sequence.

```
4   def expr: Parser[Int] = addition | subtraction | number
```

»|« combines two parsers to a new one that is successful if one of the given ones is successful

How to parse

Using the parser

```
14   def parse(str: String):Int = expr(new CharSequenceReader(str)) match {
15     case Success(result, remain) if remain.atEnd => result
16     case Success(_, remain) =>
17       throw new RuntimeException(s"Unparsed input at ${remain.pos}")
18     case NoSuccess(msg, remain) =>
19       throw new RuntimeException(s"Parse error $msg at ${remain.pos}")
20   }
```

Examples

```
"42"          ---> 42
"42+54"       ---> 96
"42-54"       ---> -12
"42-54+12"    ---> "Unparsed input" exception
```

Pitfall 1: Longest match selection

1001

Order of combinations is important

What if

```
4   def expr: Parser[Int] = addition | subtraction | number
```

… is this

```
4   def expr: Parser[Int] = number | addition | subtraction
```

```
"42"          ---> 42
"42+54"       ---> "Unparsed input" exception
"42-54"       ---> "Unparsed input" exception
"42-54+12"    ---> "Unparsed input" exception
```

May be solved by

```
4   def expr: Parser[Int] = number ||| addition ||| subtraction
```

»|||« combines two parsers to a new one that is successful if one of the given ones is successful. If both are successful, the one which consumes more wins.

Pitfall 2: Recursion is not your friend

Do not repeat the yacc-way

```
 5   def expr: Parser[Int] = addition | subtraction | number
 5
 6   def addition: Parser[Int] =
 7     number ~ '+' ~ number ^^ { case left ~ _ ~ right => left + right }
 8
 9   def subtraction: Parser[Int] =
10     number ~ '-' ~ number ^^ { case left ~ _ ~ right => left - right }
```

```
 5   def expr: Parser[Int] = addition | subtraction | number
 5
 6   def addition: Parser[Int] =
 7     expr ~ '+' ~ expr ^^ { case left ~ _ ~ right => left + right }
 8
 9   def subtraction: Parser[Int] =
10     expr ~ '-' ~ expr ^^ { case left ~ _ ~ right => left - right }
```

Fails with Stack-overflow.

The correct way to do it

Parse elements delimited by operators

```
 4    def expr = addSub
 5
 6    def addSub: Parser[Int] = mulDiv * (
 7        '+' ^^^ { (left: Int, right: Int) => left + right}
 8      | '-' ^^^ { (left: Int, right: Int) => left - right} )
 9
10    def mulDiv = number * (
11        '*' ^^^ { (left: Int, right: Int) => left * right }
12      | '/' ^^^ { (left: Int, right: Int) => left / right } )
```

»*« repeats the left parser by using the right parser to parse the delimiters. The result of the right parser has to be a function to combine the results of the left parser.

»^^^« simply replaces the result of a parser

Pitfall 3: Pollution of the grammar

Potential way to handle whitespaces

```
13   def number: Parser[Int] =
14     whitespace.* ~> digit.+ <~ whitespace.* ^^ { digits =>
14                                                  digits.mkString("").toInt }
15
16   def digit: Parser[Char] = elem("digit", ch => ch.isDigit)
17
18   def whitespace: Parser[Char] =
19                                    elem("ws", ch => ch == ' ' || ch == '\t')
```

»<~« and »~>« are just like »~« but ignore the results of the parser to the left resp. right.

»*« postfix is just like the »+« postfix but succeeds even if there is no match at all.

Separate code into lexer and parser


```
 1 class Calculator3 extends StdTokenParsers {
 2   override type Tokens = StdLexical
 3
 4   override val lexical = new StdLexical
 5
 6   lexical.delimiters ++= List("(", ")", "+", "-", "*", "/")
 7
 8   def expr: Parser[Int] = addSub
 9
10   def addSub: Parser[Int] = mulDiv * (
11       '+' ^^^ { (left: Int, right: Int) => left + right}
12     | '-' ^^^ { (left: Int, right: Int) => left - right} )
13
14   def mulDiv = number * (
15       '*' ^^^ { (left: Int, right: Int) => left * right }
16     | '/' ^^^ { (left: Int, right: Int) => left / right } )
17
18   def term: Parser[Int] = "(" ~> expr <~ ")" | numericLit ^^ (_.toInt)
19
20   def parse(str: String) = expr(new lexical.Scanner(str)) match {
21 ...
24   }
25 }
```

Separate code into lexer and parser

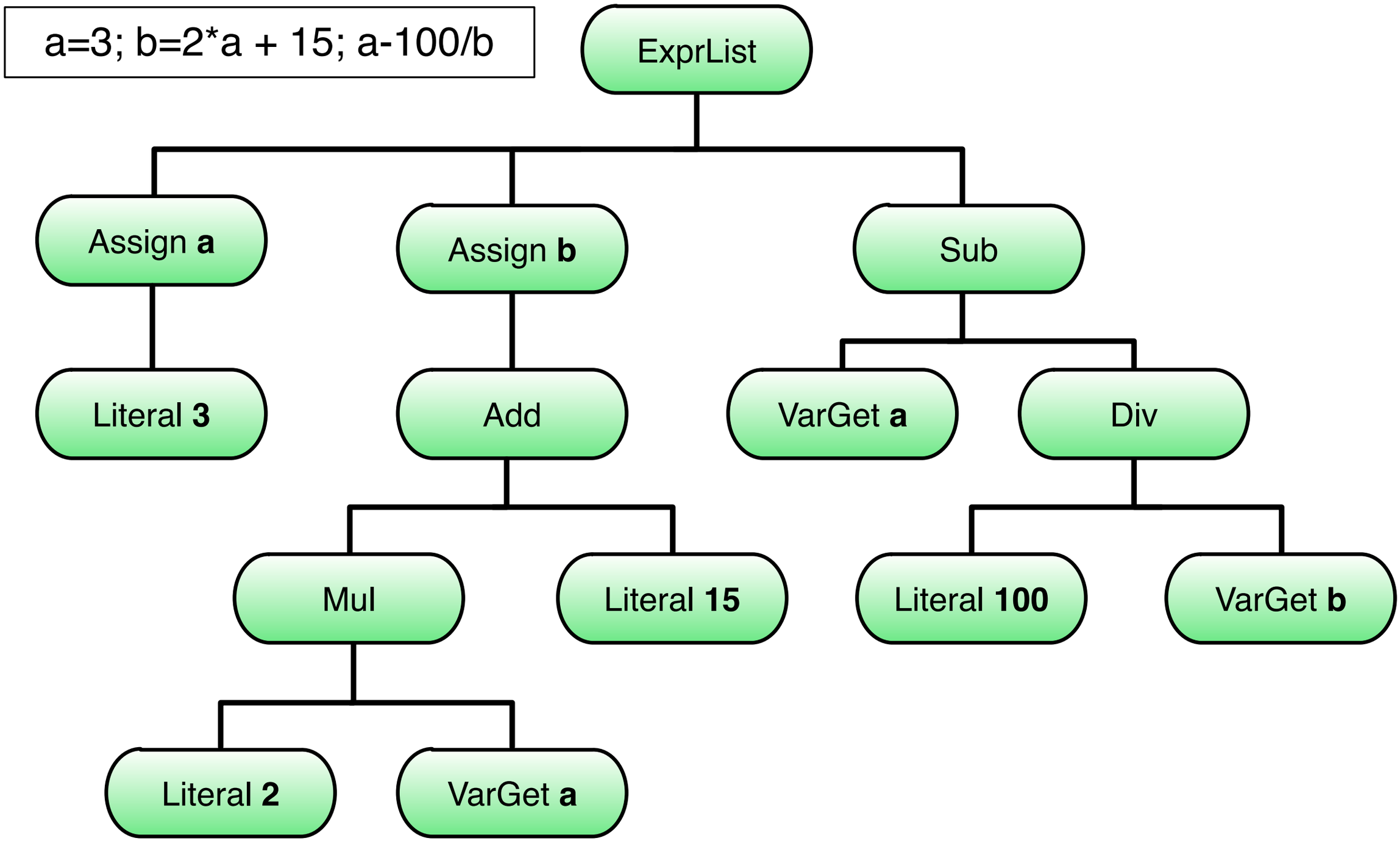
Examples

```
"42 - 3*3*3*2 + 24/2"                                  ---> 0
"2 * (3 /* blah blah */ +4) / 2 + 7 * 5"    ---> 42
```

- Even though this works quite nicely, all the »work« is done by the parser itself.
- Parser rules might become quickly polluted for more complex functionality: Type-conversion, variables, functions …

Go abstract … tree-like

Abstract syntax tree



Just a bunch of one-liners

Define an AST in Scala

```
1 trait Node { }
2
3 trait Expr extends Node { }
4
5 case class ExprListExpr(exprs: List[Expr]) extends Expr
6
7 case class LiteralExpr(value: Int) extends Expr
8
9 case class AddExpr(left: Expr, right: Expr) extends Expr
10
11 case class SubExpr(left: Expr, right: Expr) extends Expr
12
13 case class MulExpr(left: Expr, right: Expr) extends Expr
14
15 case class DivExpr(left: Expr, right: Expr) extends Expr
16
17 case class VarGetExpr(name: String) extends Expr
18
19 case class AssignExpr(name: String, expr: Expr) extends Expr
```

Case classes are their own factory

Parse to an AST

```
1 object ASTParser extends StdTokenParsers {
2   override type Tokens = StdLexical
3
4   override val lexical = new StdLexical
5
6   lexical.delimiters ++= List("(", ")", "+", "-", "*", "/", "=", ";")
7
8   def exprs: Parser[Expr] = repsep(expr, ";") ^^ ExprListExpr
9
10  def expr: Parser[Expr] = assign | addSub
11
12  def assign: Parser[Expr] = ident ~ "=" ~ addSub ^^ {
13                case name ~ _ ~ valueExpr => AssignExpr(name, valueExpr) }
14
15  def addSub: Parser[Expr] = mulDiv * ("+" ^^^ AddExpr | "-" ^^^ SubExpr)
16
17  def mulDiv: Parser[Expr] = term * ("*" ^^^ MulExpr | "/" ^^^ DivExpr)
18
19  def term: Parser[Expr] =
20    "(" ~> expr <~ ")" | ident ^^ VarGetExpr |
21    numericLit ^^ (str => LiteralExpr(str.toInt))
22 ...
23 }
```

Interpret: Just some more one-liners

1001

Add interpreter

```
1 trait Expr extends Node {
2   def eval(implicit context: CalculatorContext): Int
3 }
```

```
 1 case class LiteralExpr(value: Int) extends Expr {
 2   def eval(implicit context: CalculatorContext) = value
 3 }
 4
 5 case class AddExpr(left: Expr, right: Expr) extends Expr {
 6   def eval(implicit ctx: CalculatorContext) = left.eval + right.eval
 7 }
 8
 9 case class VarGetExpr(name: String) extends Expr {
10   def eval(implicit ctx: CalculatorContext) =
11     context.getVariable(name).getOrElse {
12       throw new RuntimeException(s"Variable $name not defined")
13     }
14 }
15 ...
```

… and just go on from here

leanovate
beyond product

1001

Outlook

- JBJ's parser and interpreter is just a calculator in its n-th iteration
- So far focus was purely on compatibility, not performance. Lots of internal refactoring will be required
- Many extensions are still missing (~12000 tests are still open)
- … any kind of input/contribution is welcome

Where to read on

leanovate
beyond product

Links

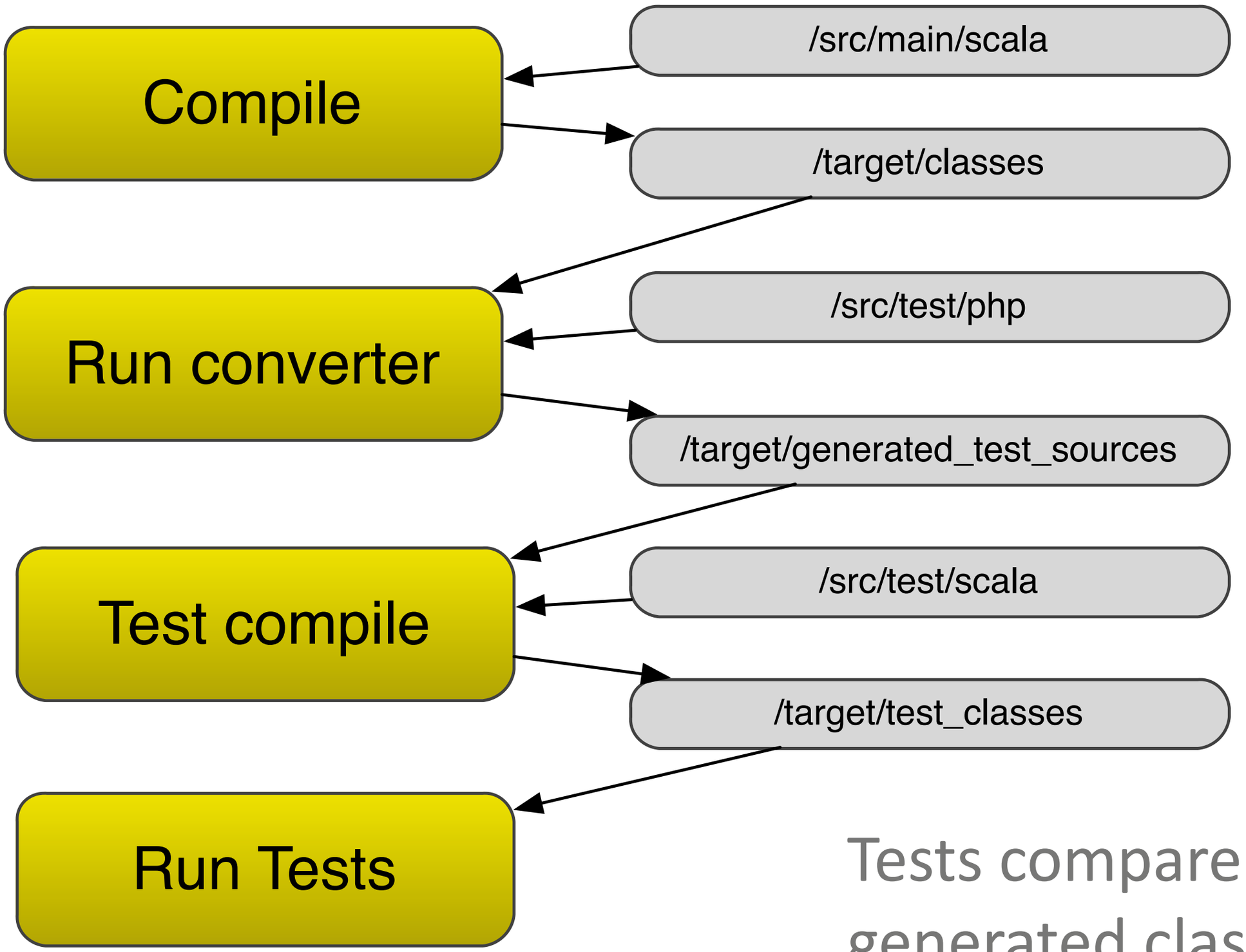
http://bedcon2014.leanovate.de

https://github.com/leanovate/jbj

Unused pages

How to test a converter

1001



Tests compare output of generated classes.

Hurdle 5: Implicit array/class creation

Assignments may create arrays/classes

```php
<?php

$a[][][] = 3;

var_dump($a);

$b[1][2]->bla = "Hello";

var_dump($b);
?>
```

```
array(1) {
  [0] =>
  array(1) {
    [0] =>
    array(1) {
      [0] =>
      int(3)
    }
  }
}
PHP Strict standards:
   Creating default object from empty value
array(1) {
  [1] =>
  array(1) {
    [2] =>
    class stdClass#1 (1) {
      public $bla =>
      string(5) "Hello"
    }
  }
}
```

Hurdle 7: Lots of extensions

leanovate
beyond product

1001

PHP ships with lots of buildin functionality

- mb_string: Basic multi-byte string support
- iconv: Deeper charset/unicode support
- curl: HTTP/FTP client
- preg: Regular expressions
- bcmath: Arbitrary length arithmetics
- mysql: MySql database driver
- gd: »libgd« wrapper to create images (e.g. CAPCHA)
- …