Write You a Compiler for Great Fun!

(Tunight Talk)

Paul MashPlant

October 19, 2019

Tonight

1. Compiler Construction
2. New Decaf Compilers
   - Java v.s. Scala Versions
   - Rust Version
3. Decaf's Road Ahead

Contents

1. Compiler Construction
2. New Decaf Compilers
   - Java v.s. Scala Versions
   - Rust Version
3. Decaf's Road Ahead

What can Compilers Do?

- Generate machine code
- Run your programs
- Optimize your programs
- Analyze your programs
- etc.

World is Changing

Compilers are ...

NOT JUST compilers, BUT ALSO

- optimizers
- shells (Read-Evaluate-Print Loop)
- analyzers (lint)
- databases
- verifiers
- synthesizers

How to Construct a Compiler?

For some:

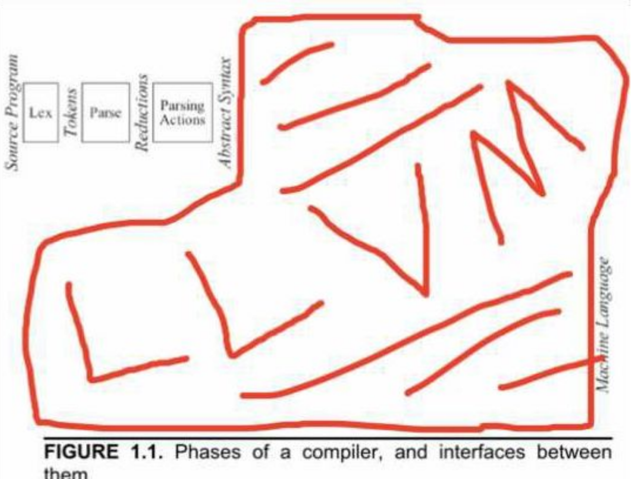


FIGURE 1.1. Phases of a compiler, and interfaces between them

How to Construct a Compiler?

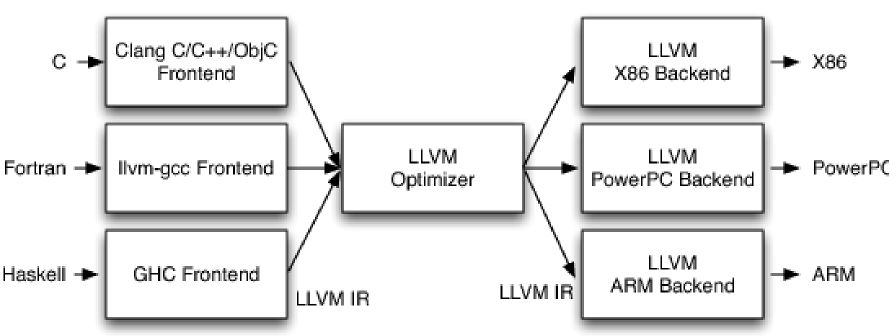
For Martin Odersky and Scala team:

```
scala -Xshow-phases
```

Guess: how may phases?

How to Construct a Compiler?

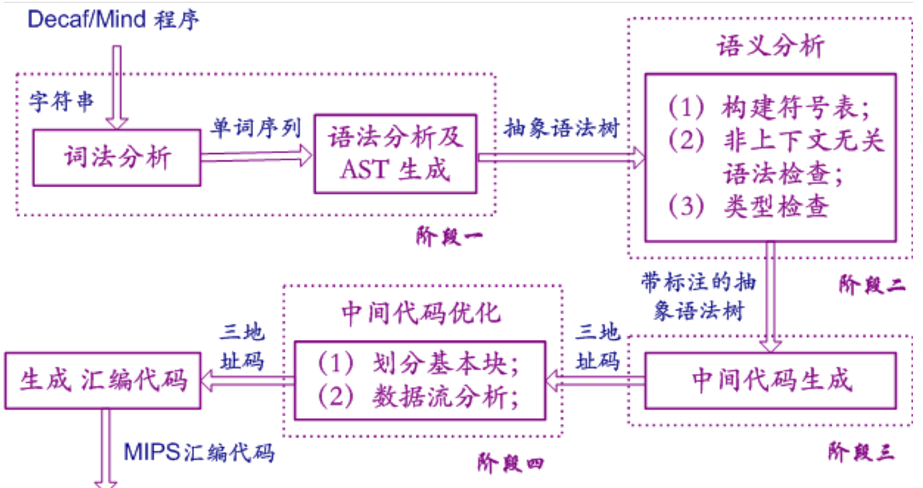
For LLVM fans:



Intermediate Representation bridges everything

How to Construct a Compiler?

For students who enrolls the Decaf project:



Multiple Phases/Passes

parser	⟵ PL[1]/TCS[2]
type checker/synthesizer	⟵ PL/logic
desugar/transformation	⟵ PL
IR generation	⟵ PL
IR optimization	⟵ SE[3]/system
instruction optimization	⟵ system/architecture
assembler	⟵ architecture

Compiler is cross-disciplinary!

[1] Programming Languges
[2] Theoretical Computer Science
[3] Software Engineering

Contents

1 Compiler Construction

2 New Decaf Compilers
- Java v.s. Scala Versions
- Rust Version

3 Decaf's Road Ahead

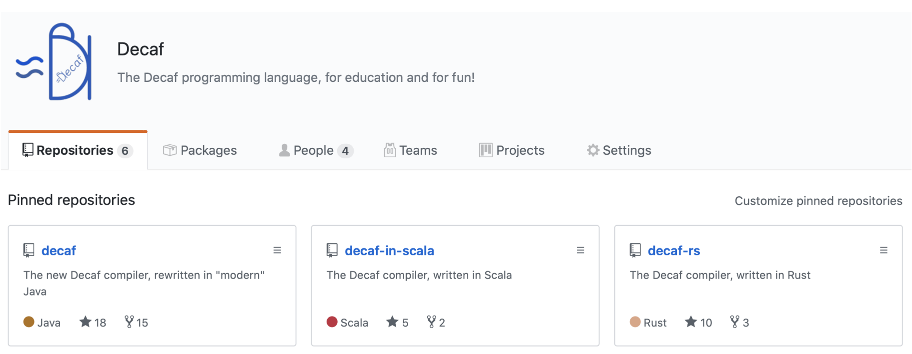
Decaf?

> *Decaffeination is the removal of caffeine from coffee beans, cocoa, tea leaves, and other caffeine-containing materials. Decaffeinated products are commonly termed decaf.*
>
> – https://en.wikipedia.org/wiki/Decaffeination

Logo

https://git.tsinghua.edu.cn/decaf-lang/decaf-logo

Star Our Repos!



https://github.com/decaf-lang

Lines of Code[4]

- Java: 10335 (`*.java`)
- Scala: 5176 (`*.scala`) + 5260 (`*.java`) = 10436
- **Rust: 4862** (`*.rs`)

[4] Excluding empty lines.

Contents

1. Compiler Construction
2. New Decaf Compilers
   - Java v.s. Scala Versions
   - Rust Version
3. Decaf's Road Ahead

jflex + jacc v.s. antlr4

Lines of grammar specification[5]:

- jflex + jacc: 126
- **antlr4: 80**

Example:

```
/* jacc */
TopLevel       : ClassList;
ClassList      : ClassList ClassDef | ClassDef;
ClassDef       : CLASS Id ExtendsClause '{' FieldList '}';
ExtendsClause : EXTENDS Id | ;
FieldList      : FieldList Var ';' | FieldList MethodDef | ;
/* antlr4 */
topLevel       : classDef*;
classDef       : CLASS id extendsClause? '{' field* '}';
extendsClause : EXTENDS id;
field          : varDef | methodDef;
```

[5] Only grammar lines are computed. Data comes from PA1-A doc.

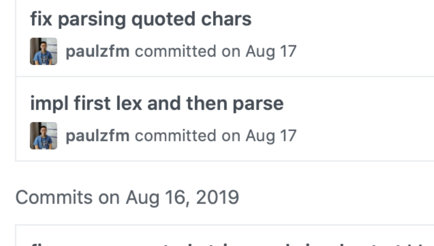
Parser Combinators?

In a very early commit of the Scala version:

```
class StmtParsers extends ExprParsers {
  def typed = positioned(typ ~ id ^^ ???)
  def localVarDef = positioned(typed <~! SEMI ^^ ???)
  def block = positioned(LBRACE ~>! stmt.* <~ RBRACE ^^ ???)
  def ifStmt = positioned { IF ~>!
    (LPAREN ~> expr <~ RPAREN) ~ stmt ~ (ELSE ~>! stmt).? ^^ ??? }
  def whileStmt = positioned { WHILE ~>! expr ~ stmt ^^ ??? }
  def breakStmt = positioned { BREAK ~! SEMI ^^^ ??? }
  def returnStmt = positioned { RETURN ~>! expr.? <~ SEMI ^^ ??? }
  /* ... */
  def controlStmt = ifStmt | whileStmt | breakStmt | ...
  def stmt = block | localVarDef | controlStmt
}
```

What's Wrong with Combinators?

- Manually tweak lexing:



- The "evil" left-recursion and left-factor.
- Why not try generalized LL parser combinators?

Don't Trust Your IDE!

False negative:

```
override def transform(in: Reader): Tree =
  parser.parseAll(parser.topLevel, in) match {
    case parser.Success(result, _) => result : Any
    case f: parser.NoSuccess =>
      issue(new SyntaxError(f.next.pos))
      println(s"${ f.next.pos }:${ f.msg }:\n${ f.next.pos.longString }")
      TopLevel(Nil)
  }
```

False positive: code that can pass the linter may NOT type check due to implicit conversions, variances (e.g. `java.util.List` is not covariant), etc.

ASTs with Annotations

```
/* template */
trait TreeTmpl {
  type ExprAnnot <: Annot
  trait Expr extends Node with Annotated[ExprAnnot]
  case class Binary(op: BinaryOp, lhs: Expr, rhs: Expr)
                   (implicit val annot: ExprAnnot) extends Expr
}
/* syntax tree */
implicit object NoAnnot extends Annot
object SyntaxTree extends TreeTmpl {
  type ExprAnnot = NoAnnot.type
}
/* typed tree */
object TypedTree extends TreeTmpl {
  type ExprAnnot = Type
}
```

Pattern Matching ...

To implement an expression evaluator, using pattern matching:

```
sealed abstract class Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr
case class Number(value: Int) extends Expr

def eval(expr: Expr): Int = expr match {
  case Add(l, r) => eval(l) + eval(r)
  case Sub(l, r) => eval(l) - eval(r)
  case Number(v) => v
}
```

... v.s. Visitors I

To do the same in Java, using visitors:

```
interface ExprVisitor<T> {
  T visitAdd(Add e);
  T visitSub(Sub e);
  T visitNumber(Number e);
}

abstract class Expr {
  abstract <T> T accept(ExprVisitor<T> v);
}

class Add extends Expr {
  Expr lhs; Expr rhs;
  @Override <T> T accept(ExprVisitor<T> v) {
    return v.visitAdd(this); }
}
/* Similar for Sub and Number */
```

… v.s. Visitors II

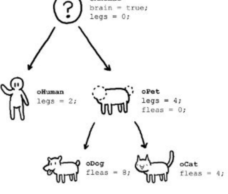
```
class EvalVisitor implements ExprVisitor<Integer> {
  @Override int visitAdd(Add e) {
      var l = e.lhs.accept(this);
      var r = e.rhs.accept(this);
      return l + r;
  } /* Similar for visitSub */

  @Override int visitNumber(Number e) {
      return e.value;
  }
}

int eval(Expr expr) {
  var v = new EvalVisitor();
  return expr.accept(v);
}
```

Why Visitors?

- Because Java does NOT support pattern matching!
- If a language supports pattern matching, then visitors are NOT necessary!
- Good news: Rust version is 100% visitors free. In scala version, visitors are ONLY used to call Java code.

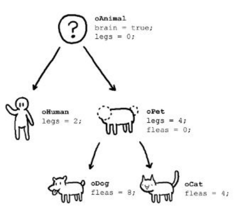
What's Wrong with Java?

What OOP users claim

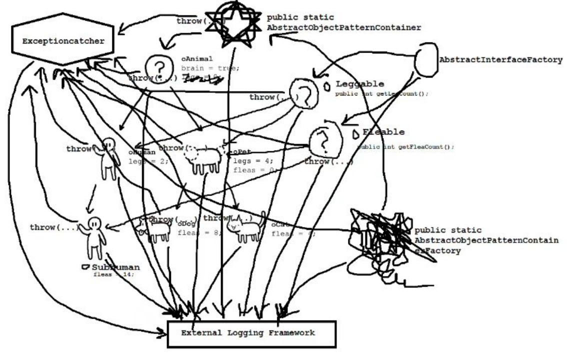


What’s Wrong with Java?

What OOP users claim



What actually happens



On Design Pattern

Examples:

- visitor pattern
- (abstract) factory pattern
- builder pattern
- monad pattern

My opinions:

- Design patterns are “bad” design.
- Design patterns are NOT needed if the language feature already include this.

Case Classes ...

Express algebraic data types via case classes:

```
trait BasicBlock[I <: PseudoInstr] extends Iterable[Loc[I]]
case class ContinuousBasicBlock[I <: PseudoInstr](...)
  extends BasicBlock[I]
case class EndByJumpBasicBlock[I <: PseudoInstr](...)
  extends BasicBlock[I]
case class EndByCondJumpBasicBlock[I <: PseudoInstr](...)
  extends BasicBlock[I]
case class EndByReturnBasicBlock[I <: PseudoInstr](...)
  extends BasicBlock[I]
```

… v.s. Kind Enum

Express algebraic data types via enum and kind:

```
public class BasicBlock<I extends PseudoInstr> implements
  Iterable<Loc<I>> {
  public enum Kind {
    CONTINUOUS, END_BY_JUMP, END_BY_COND_JUMP, END_BY_RETURN
  }
  public final Kind kind;
  /* ... */
}
```

“Be Pure or Impure?”

Even in Java we can be pure:

```
public static class LoadVTbl extends TacInstr {
  public final Temp dst;
  public final VTable vtbl;
  /* ... */
}
```

Even in a functional language like Scala we can be impure:

```
class Context {
  val global: GlobalScope = new GlobalScope
  val classes: mutable.Map[String, ClassDef] = new mutable.TreeMap
}
```

Running Time

Table: Total execution time on old test sets.

Phase	Java	Scala
PA1-A	**3.88 s**	12.99 s
PA2	**8.84 s**	27.34 s
PA3	**5.94 s**	18.20 s
PA5	**6.99 s**	19.36 s

Scala is much slower than Java!

Core of CS?

We've just discussed:

- Parser generator v.s. combinator?
- Design patterns v.s. language features?
- Pure v.s. impure?
- Efficient v.s. expressive?

Tradeoff!

Core of PL?

Neither programming nor language, but design!

For Fun: Java v.s. Scala

Which one has a "better" design?

In my opinion:

- Java is compiler-friendly, but Scala is programmer-friendly.
- Scala teaches your more OO than Java.
- Java is for the past, but Scala is for the future.

Contents

1. Compiler Construction
2. New Decaf Compilers
   - Java v.s. Scala Versions
   - Rust Version
3. Decaf's Road Ahead

Parser Generator: re2dfa + lalr1 I

Thanks to Rust's procedural macro, we can write a parser without any external config file, via MashPlant's toolchains:

```
#[lalr1(Expr)]
#[lex(r#"
priority = [
  { assoc = 'left', terms = ['Add'] },
  { assoc = 'left', terms = ['Mul'] }
]

[lexical]
'\+' = 'Add'
'\*' = 'Mul'
'\d+' = 'IntLit'
'\s+' = '_Eps'
"#)]

impl Parser {
```

Parser Generator: re2dfa + lalr1 II

```
  #[rule(Expr -> Expr Add Expr)]
  fn expr_add(l: i32, _op: Token, r: i32) -> i32 { l + r }

  #[rule(Expr -> Expr Mul Expr)]
  fn expr_mul(l: i32, _op: Token, r: i32) -> i32 { l * r }

  #[rule(Expr -> IntLit)]
  fn expr_int(i: Token) -> i32 { str::from_utf8(i.piece)
    .unwrap().parse().unwrap() }
}
```

jflex + jacc v.s. antlr4 v.s. re2dfa + lalr1

All codes related to lexer & parser[6]:

- Java: 390 (`parsing/*.java`) + 481 (`Decaf.jacc` + `Decaf.jflex`) = 871
- Scala: 363 (`parsing/*.scala`) + 260 (`antlr4/`) = 623
- **Rust: 373** (`syntax/src/lib.rs` + `syntax/src/parser.rs`)

...maybe this is somewhat biased, because I (MashPlant) like to compress lines of code when coding.

[6] Not including generated code & comment & blank lines.

Parser Generator: re2dfa + lalr1

Other Features:

- strongly-typed parser
- zero-copy
- IDE support (limited but still quite useful)
- dump all kinds of tables for debugging

Case Classes v.s. Tagged Union

- Rust's enum is essentially a tagged union.
- Efficient v.s. expressive? It is easy to have both in Rust:
  - to match a Scala's case class: `instanceof` + `checkcast` + `athrow` (JVM instructions)
  - to match a Rust's enum: jump table
- Knowing this can dispel my concern about performance when writing high-level code.

Fight against Borrow Checker

How to write a linked list in Rust?

```rust
pub struct Tac<'a> {
  pub payload: RefCell<TacPayload>,
  pub prev: Cell<Option<&'a Tac<'a>>>,
  pub next: Cell<Option<&'a Tac<'a>>>,
}
// still need the help of Arena memory allocator
```

Running Time

Table: Total execution time on old test sets.

Phase	Java	Scala	Rust
PA1-A	3.88 s	12.99 s	**0.11 s**
PA2	8.84 s	27.34 s	**0.11 s**
PA3	5.94 s	18.20 s	**0.12 s**
PA5	6.99 s	19.36 s	**0.27 s**

For Fun: Java v.s. Scala v.s. Rust

- Java is compiler-friendly.
- Scala is programmer-friendly.
- Rust is neither, but ONLY WHEN you fail to get your code compiled.

Table: Build time.

Version	Command	Seconds
Java	`gradle build`	**5.852**
Scala	`sbt compile`	24.049 (compile 15)
Rust	`cargo build`	261.29

Contents

1 Compiler Construction

2 New Decaf Compilers
- Java v.s. Scala Versions
- Rust Version

3 Decaf's Road Ahead

Missing Features of Decaf

- method override (18' optional)
- exception handler (18' optional)
- local type inference (18' & 19' mandatory)
- first-class functions (18' optional & 19' mandatory)
- package and module
- toolchains

Call for Toolchains!

- syntax highlighter
- REPL
- linter
- debugger
- language server
- IDE plugins

TAC[7] Virtual Machine

- Already have: built-in simulator
- Hope to do:
  - TAC language standard (semantics, bytecode, etc.)
  - TAC program debugger
  - a “real” virtual machine (with garbage collection)

[7] Three-Address Code

Call for Backends!

- RISC-V
- x86
- etc.
- LLVM to rule 'em all!

- vecaf (verified decaf): constraint solving-based program verifier
- secaf (sketched decaf): sketch-based program synthesizer
- etc.

Weaknesses of Decaf

- package and module
- type system
- OO system
- redundant grammar

Difficult to overcome these based upon the current version!

In Future: Faced

- A new language for education, for research and for fun!
- statically-typed with a rich type system
- functional & OO
- adaptive syntax
- JVM, CLR and native (LLVM)

Goals:

- write less, synthesize more
- productive & type-safe

Beginning: like Scala, but more independent from JVM

Thanks!

Q & A