# Reimplementing the Wheel: Teaching Compilers with a Small Self-Contained One

Daniil Berezun Dmitry Boulytchev

**Saint-Petersburg State University**
**JetBrains Research**

**10th International Workshop on Trends in Functional Programming in Education**
February 16, 2021
Online

# Background

# Background

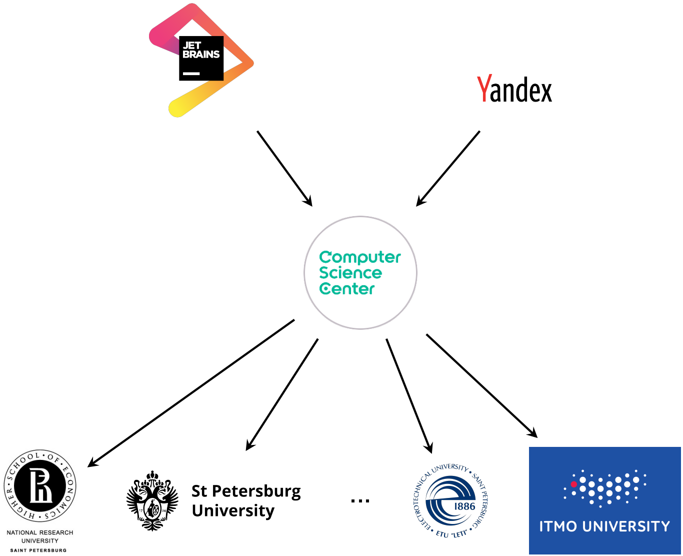
# Background

JET BRAINS

Yandex

Computer Science Center

# Background


JET BRAINS
Yandex
Computer Science Center
HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY
SAINT PETERSBURG
St Petersburg University
...
ELECTROTECHNICAL UNIVERSITY • SAINT PETERSBURG •
ETU "LETI"
1886
ITMO UNIVERSITY

# PL Program

An education track in programming languages & tools:

- Semantics of programming languages;
- Metacomputations;
- Logic & relational programming;
- ...

# PL Program

An education track in programming languages & tools:

- **Programming languages and compilers;**
- Semantics of programming languages;
- Metacomputations;
- Logic & relational programming;
- ...

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.
- No parser-centricity.

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
- Self-contained codegenerator (no ~~no animals are hurt~~ heavy infrastructure is involved).

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
- Self-contained codegenerator (no ~~no animals are hurt~~ heavy infrastructure is involved).
- Functional programming techniques.

# Compiler Construction as an Introductory Course

**Prerequisites (soft):**

- Functional programming.
- Formal grammars, languages and automata.

**Course outline:**

- Source language = implementation language.
- No parser-centricity.
- An evolving family of languages.
- Big-step operational semantics for each language.
- Self-contained codegenerator (no ~~no animals are hurt~~ heavy infrastructure is involved).
- Functional programming techniques.

TOGETHER

# The $\lambda^{a}_{\mathcal{M}^{a}}$ Programming Language

# The $\lambda^a\mathcal{M}^a$ Programming Language

$$\lambda^a\mathcal{M}^a = \lambda + \textsc{Algol}$$

# The $\lambda^a\mathcal{M}^a$ Programming Language

$$\lambda^a\mathcal{M}^a = \lambda + \text{A}\textsc{lgol}$$

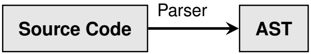
```
fun reverse (l) {
  (fix $ fun (rec) {
           fun (acc, l) {
             case l of
               {}       →  acc
             | x : xs   →  rec (x : acc, xs)
             esac
         }}) ({}, l)
}
```

# The Structure of the Compiler

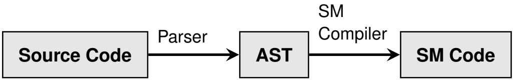
# The Structure of the Compiler

**Source Code**

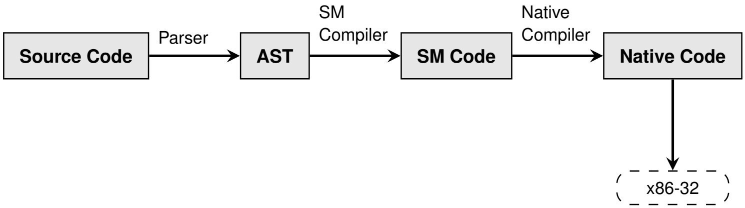
# The Structure of the Compiler



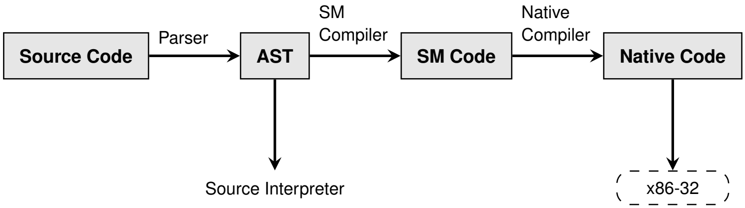
# The Structure of the Compiler



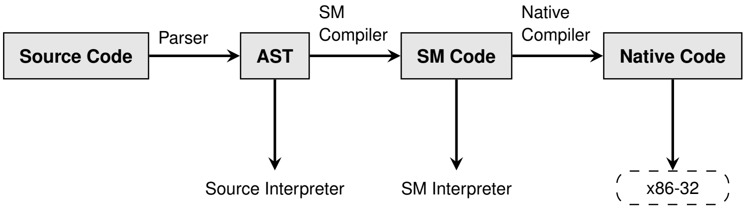
# The Structure of the Compiler



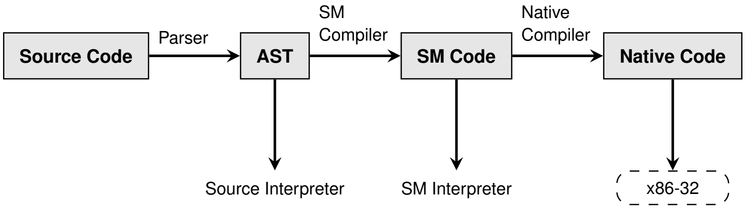
# The Structure of the Compiler



# The Structure of the Compiler



# The Structure of the Compiler




| Component | LOC |
|---|---|
| Compiler (OCAML) | 3000 |
| Runtime (C+GAS) | 1000 |
| Standard library ($\lambda^a\mathcal{M}^a$) | 900 |

# A Stack of Languages with "Vertical" Homework Assignments

# A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;

# A Stack of Languages with "Vertical" Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;

# A Stack of Languages with "Vertical" Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**

# A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**
4. + **local definitions, scopes and functions;**

# A Stack of Languages with "Vertical" Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**
4. + **local definitions, scopes and functions;**
5. + arrays and builtin functions;

# A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**
4. + **local definitions, scopes and functions;**
5. + arrays and builtin functions;
6. + fixednum arithmetics;

# A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**
4. + **local definitions, scopes and functions;**
5. + arrays and builtin functions;
6. + fixednum arithmetics;
7. + S-expressions;

# A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**
4. + **local definitions, scopes and functions;**
5. + arrays and builtin functions;
6. + fixednum arithmetics;
7. + S-expressions;
8. + pattern-matching;

# A Stack of Languages with “Vertical” Homework Assignments

1. Simple straight-line programs made of assignments and sequential composition;
2. + control flow statements: branching and looping;
3. + **all control constructs treated as expressions;**
4. + **local definitions, scopes and functions;**
5. + arrays and builtin functions;
6. + fixednum arithmetics;
7. + S-expressions;
8. + pattern-matching;
9. + first-class functions (this point actually have never been reached within one semester).

# Deep Embedding vs. Syntax Analyser

# Deep Embedding vs. Syntax Analyser

*No* syntax analyzer initially.

# Deep Embedding vs. Syntax Analyser

*No* syntax analyzer initially.

```
infix +  at +  (l, r) {
  Binop ("+",  opnd (l), opnd (r))
}
infix -  at -  (l, r) {
  Binop ("-",  opnd (l), opnd (r))
}
infix *  at *  (l, r) {
  Binop ("*",  opnd (l), opnd (r))
}
infix /  at /  (l, r) {
  Binop ("/",  opnd (l), opnd (r))
}
infix == at == (l, r) {
  Binop ("==", opnd (l), opnd (r))
}
...
```

# Deep Embedding vs. Syntax Analyser

*No* syntax analyzer initially.

```
infix +  at +  (l, r) {
  Binop ("+",  opnd (l), opnd (r))
}
infix -  at -  (l, r) {
  Binop ("-",  opnd (l), opnd (r))
}
infix *  at *  (l, r) {
  Binop ("*",  opnd (l), opnd (r))
}
infix /  at /  (l, r) {
  Binop ("/",  opnd (l), opnd (r))
}
infix == at == (l, r) {
  Binop ("==", opnd (l), opnd (r))
}
...
```

```
read ("x") >>
read ("y") >>
"z" ::= "x" < "y" >>
write ("z") >>
"z" ::= "x" <= "y" >>
write ("z") >>
"z" ::= "x" == "y" >>
write ("z") >>
"z" ::= "x" >= "y" >>
write ("z") >>
"z" ::= "x" > "y" >>
write ("z")
```

# Syntax Analysis with Parser Combinators

# Syntax Analysis with Parser Combinators

OSTAP — a library of monadic parser combinators is CPS and memoization [Johnson, 1995; Izmaylova, Afroozeh, van der Storm, 2015].

# Syntax Analysis with Parser Combinators

OSTAP — a library of monadic parser combinators is CPS and memoization [Johnson, 1995; Izmaylova, Afroozeh, van der Storm, 2015].

Embedded DSL for $\lambda^a\mathcal{M}^a$:

```
syntax (kSkip                                    {Skip}          |
        x=lident s[":="] e=exp                    {Assn (x, e)}   |
        kRead   x=inbr[s("("), lident, s(")")]    {Read (x)}      |
        kWrite  e=inbr[s("("), exp   , s(")")]    {Write (e)}     |
        kWhile  e=exp b=inbr[kDo, stmt, kOd]      {While (e, b)})
```

# Operational Semantics

# Operational Semantics

$$\frac{\sigma \xrightarrow{e}_{\mathscr{E}} n \neq 0 \quad \langle \sigma, w \rangle \xrightarrow{S} c' \quad c' \xrightarrow{\texttt{while}\ e\ \texttt{do}\ S} c''}{\langle \sigma, w \rangle \xrightarrow{\texttt{while}\ e\ \texttt{do}\ S} c''}$$

$$\frac{\sigma \xrightarrow{e}_{\mathscr{E}} 0}{\langle \sigma, w \rangle \xrightarrow{\texttt{while}\ e\ \texttt{do}\ S} \langle \sigma, w \rangle}$$

# Operational Semantics

$$\frac{\sigma \xrightarrow{e}_{\mathscr{E}} n \neq 0 \quad \langle \sigma, w \rangle \xrightarrow{S} c' \quad c' \xrightarrow{\texttt{while}\ e\ \textbf{do}\ S} c''}{\langle \sigma, w \rangle \xrightarrow{\texttt{while}\ e\ \textbf{do}\ S} c''}$$

$$\frac{\sigma \xrightarrow{e}_{\mathscr{E}} 0}{\langle \sigma, w \rangle \xrightarrow{\texttt{while}\ e\ \textbf{do}\ S} \langle \sigma, w \rangle}$$

```
fun eval (c@[s, w], stmt) {
  case stmt of
  ...
  | While (e, b)  →  if evalExpr (s, e)
                     then eval (eval (c, b), stmt)
                     else c
                     fi
  ...
}
```

# Operational Semantics (SM)

# Operational Semantics (SM)

$$\frac{\langle (x \oplus y)s, c\rangle \xrightarrow{p} c'}{\langle yxs, c\rangle \xrightarrow{[\mathtt{BINOP}\ \otimes]p} c'}$$

# Operational Semantics (SM)

$$\frac{\langle (x \oplus y)s, c\rangle \xrightarrow{p} c'}{\langle yxs, c\rangle \xrightarrow{[\texttt{BINOP}\ \otimes]p} c'}$$

```
fun eval (c@[st, s, w], insns) {
  case insns of
    {}          →  c
  | i : insns  →
    eval (
      case i of
      ...
      | BINOP (op)  →
        case st of
          x : y : st  →  [evalOp (op, y, x) : st, s, w]
        esac
      ...
}
```

# Operational Semantics (Static)

# Operational Semantics (Static)

$$\texttt{ref}\, x : \textbf{Ref} \qquad x : \textbf{Val} \qquad \texttt{ignore}\, x : \textbf{Void} \qquad x \in \mathscr{X}$$

# Operational Semantics (Static)

$$\texttt{ref}\, x : \textbf{Ref} \quad x : \textbf{Val} \quad \texttt{ignore}\, x : \textbf{Void} \quad x \in \mathscr{X}$$

```
syntax (x=lident  {fun (a) {
                     case a of
                       Ref   →  Ref (x)
                     | Void  →  Ignore (Var (x))
                     | Val   →  Var (x)
                     esac
                   }} |
       ...
```

# Codegeneration with Symbolic Interpreters

# Codegeneration with Symbolic Interpreters

**Idea:** symbolic interpreter which operates on *locations* instead of data values can be used for codegeneration.

# Codegeneration with Symbolic Interpreters

**Idea:** symbolic interpreter which operates on *locations* instead of data values can be used for codegeneration.

| Stack before | Stack machine instruction | Stack after | Machine instruction emitted |
|---|---|---|---|
| {} | `CONST` 1 | {%eax} | `movl $1, %eax` |
| {%eax} | `LD` *x* | {%eax, %ebx} | `movl $x, %ebx` |
| {%eax, %ebx} | `BINOP` + | {%eax} | `addl %ebx, %eax` |
| {%eax} | `ST` *y* | {} | `movl %eax, $y` |

# Codegeneration with Symbolic Interpreters

```
| CONST (n)  →
   [n : st, cst, s, w]

| LD (x)  →
   [lookup (s, x) : st, cst, s, w]

| ST (x)  →
  let n : _ = st in
   [st, cst, assign (s, x, n), w]
```

```
| CONST (n)  →
  let [s, env] = env.allocate in
   [env, code <+ Mov (L (box $ n), s)]
| LD (x)  →
  let [s, env] = env.allocate in
   [env, code <+> move (env.loc (x), s)]
| ST (x)  →
   [env, code <+>
         move (env.peek, env.loc (x))]
```

# Organization Trivia

- The course has been taught since 2016 in OCAML; since the spring of 2020 — in $\lambda^{a}\mathcal{M}^{a}$ itself.
- 80+ students each semester.
- Homework assignment each week.
- Continuous integration (TRAVISCI via GITHUB).
- “Lightning” division: a questionnaire of 100+ items for grade C (3/5), no homework.

## Students' Feedback

- The vast majority qualified the course material as *new* for them (42% — completely new, 58% — mostly new);
- 42% qualified the material as potentially *irrelevant* to their future professional activity; 25% as relevant, and the rest as partially relevant;
- An essential fraction complained about the lack of a type system in $\lambda^a\mathcal{M}^a$ (prior to the spring of 2020 — about the type system in OCaml).

- "Writing a compiler for $\lambda^a\mathcal{M}^a$ in $\lambda^a\mathcal{M}^a$ was a terrible thing when you had no experience with neither $\lambda^a\mathcal{M}^a$ nor its relative language OCaml."
- "A very pleasant thing was that $\lambda^a\mathcal{M}^a$ was developed specifically for the course and was truly convenient for compiler implementation, especially if one had no prior experience with OCaml".

# Conclusions and Future Work

- \- Not very mature, not very efficient.
- \+ Self-contained, small, good for introduction purposes.
- \+ With diversity of constructs.
- \+ A “tower” of sublanguages.
- \+ With compiler-oriented DSLs.

Future:

- Multiple backends (IA64? ARM? WebAssembly? JVM? LLVM?)
- Static semanatics (type system?)
- Better codegeneration (but still *within* symbolic interpreter model).