Semantic Engineering

Tesla Ice Zhang

RubyConf China 2020

I go by Tesla Ice Zhang, an undergraduate junior at Penn State.

https://github.com/ice1000 – where I code
https://ice1000.org – where I compose
https://personal.psu.edu/yqz5714

Slides will be available soon at:
https://github.com/ice1000/Books

My apology. My only experience with Ruby is a few weeks reading some articles.

Unselected topics: the Dark language, the Unison language, the Arend language.

In this talk, you'll hear about Semantic Engineering, which I want to collaborate with @thautwarm on.

Embedded DSLs

Ruby is good at making embedded DSLs.

- Active Records – ORM, mapping language-level actions to real-world actions
- RSpec – Test framework, mapping code structure to test data's structure

Facts

Many languages are good at making embedded DSLs, including Haskell, Raku (Perl 6), Racket, Groovy.

Constructs from the meta-language can be used in the object language.

Why are these languages good at making embedded DSLs?

Well, because they are good at meta-programming. They have either powerful macro systems, runtime introspections, `method_missing`s, etc.

Problems of Embedded DSLs

—

IDE doesn’t work well with macros and runtime generated definitions.

Compiler won’t be able to optimize your program aggressively, as your language is complicated.

Furthermore, you cannot break the limit of your meta language’s syntax, unless you have reader macro (which totally prevents nice IDE support).

External DSLs

So, maybe we should look for external DSLs instead?

Fact

—

An external DSL, is almost the same as a standalone programming language.

Still, fact

—

It’s too expensive to make a full-blown programming language just for some domain-specific needs.

You’re going to make:

- Parsers and lexers
- Type checking and inference
- Optimization passes
- Code generation passes
- Basic IDE support

Ooops

Your meta-language already has almost all of these. That’s why people tend to choose creating embedded DSLs.

Inspiration

—

Parsers can be generated.

Therefore, we can focus on the grammar, instead of being distracted from the parser implementation.

The rest of the list:

- Type checking and inference
- Optimization passes (part of codegen)
- Code generation passes
- Basic IDE support

Semantic Engineering

… aims at generating type checkers, and potentially compilers and IDEs as well as the parser.

What to input?

—

Parsers are generated from (E)BNF declarations.

What do we generate everything from?

Basic IDE support:

We can derive syntax highlighting and syntactical completions from the (E)BNF declarations as well.

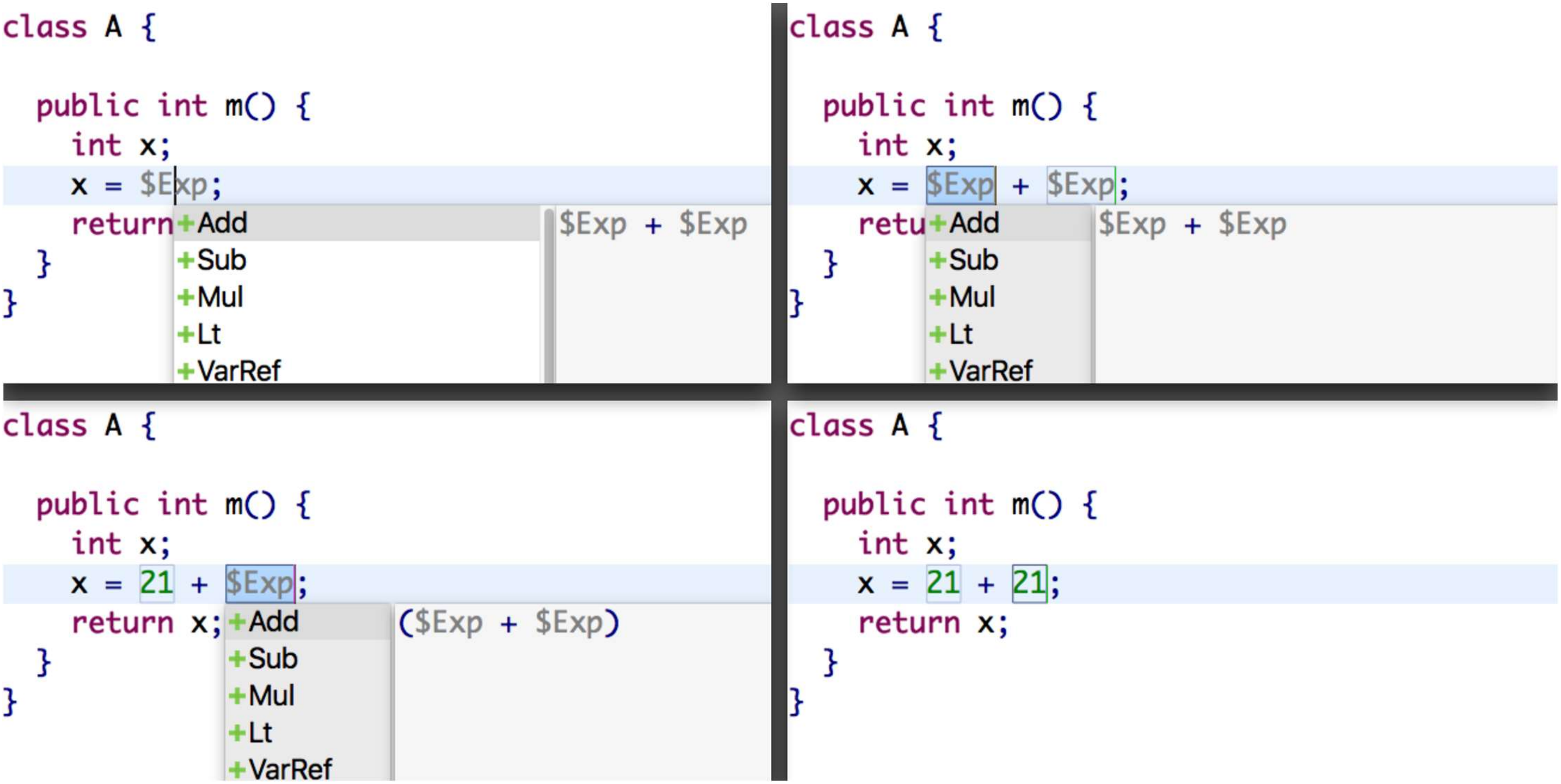
Done in Spoofax (SDF3), as well as @thautwarm’s POC toy, as well as my intellij-pest.

```
module structure

imports Common

context-free start-symbols Exp

context-free syntax

  Exp.Var = ID

  Exp.Int = INT

  Exp.Add = Exp "+" Exp

  Exp.Fun = "function" "(" {ID ","}* ")" "{" Exp "}"

  Exp.App = Exp "(" {Exp ","}* ")"

  Exp.Let = "let" Bnd* "in" Exp "end"

  Bnd.Bnd = ID "=" Exp
```

```
module Common

lexical syntax

  ID  = [a-zA-Z] [a-zA-Z0-9]*

  INT = [\-]? [0-9]+
```

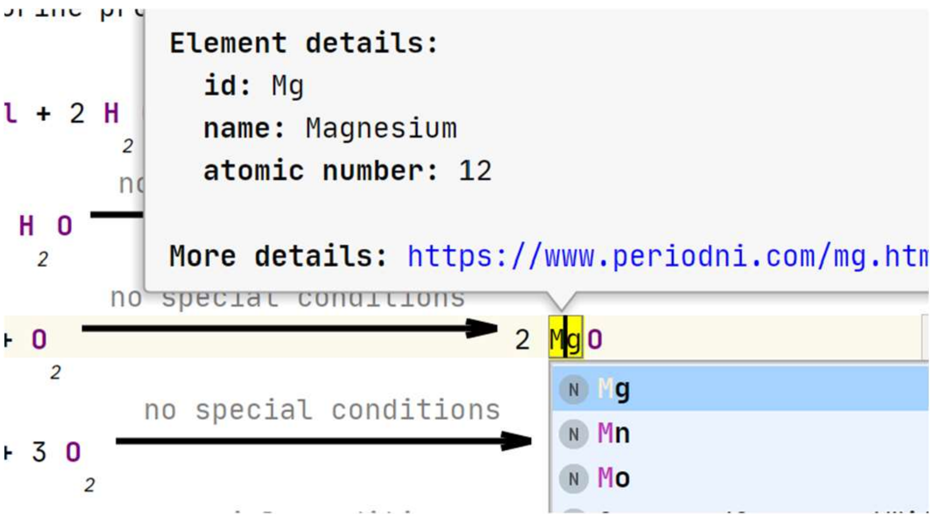
```
let
  inc = function(x) { x + 1 }
 in
  inc(3)
end
```

```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), [Int("3")])
)
```


class A {
public int m() {
int x;
x = $Exp;
return
+Add
+Sub
+Mul
+Lt
+VarRef
$Exp + $Exp
}
}
class A {
public int m() {
int x;
x = $Exp + $Exp;
retu
+Add
+Sub
+Mul
+Lt
+VarRef
$Exp + $Exp
}
}
class A {
public int m() {
int x;
x = 21 + $Exp;
return x;
+Add
+Sub
+Mul
+Lt
+VarRef
($Exp + $Exp)
}
}
class A {
public int m() {
int x;
x = 21 + 21;
return x;
}
}

JetBrains MPS:

Rich IDE support. Simple operations like completions are automatically derived.



DivExpressionDivToFraction

```
intention DivExpressionDivToFraction for concept DivExpression {
  error intention : false
  available in child nodes : false

  description(node, editorContext)->string {
    "Use Fraction Notation for Division Operation";
  }

  <isApplicable = true>

  execute(node, editorContext)->void {
                      %( node.leftExpression )%
    node.replace with(<--------------------------->);
                      %( node.rightExpression )%
  }
}
```

Code generation passes:

Specify how program are translated or executed.

Translation is done in MPS, while interpretation is done in k-framework.

JetBrains MPS: write code generation rules.

```
root template
input Calculator

public class $[CalculatorImpl] extends JFrame {
  private DocumentListener listener = new DocumentListener() {
    public void insertUpdate(DocumentEvent p0) {
      update();
    }
    public void removeUpdate(DocumentEvent p0) {
      update();
    }
    public void changedUpdate(DocumentEvent p0) {
      update();
    }
  };
  $LOOP$InputFieldDeclaration[ private JTextField $[inputField] = new JTextField(); ]
  $LOOP$OutputFieldDeclaration[ private JTextField $[outputField] = new JTextField(); ]
  public CalculatorImpl() {
    setTitle("$[Calculator]");
    setLayout(new GridLayout(0, 2));
    $LOOP$[{
            ->$[inputField].getDocument().addDocumentListener(this.listener);
            add(new JLabel("$[Title]"));
            add(->$[inputField]);
          }]
```

K framework: an interpreter of operational semantics.

```
rule (lambda X:Id . E:Exp) V:Val => E[V / X]

syntax Val ::= Int | Bool
syntax Exp ::= Exp "*" Exp          [strict, left]
             | Exp "/" Exp          [strict]
             > Exp "+" Exp          [strict, left]
             > Exp "<=" Exp         [strict]
rule I1:Int * I2:Int => I1 *Int I2
rule I1:Int / I2:Int => I1 /Int I2
rule I1:Int + I2:Int => I1 +Int I2
rule I1:Int <= I2:Int => I1 <=Int I2

syntax Exp ::= "if" Exp "then" Exp "else" Exp  [strict(1)]
rule if true  then E else _ => E
rule if false then _ else E => E

syntax Exp ::= "let" Id "=" Exp "in" Exp
rule let X = E in E':Exp => (lambda X . E') E
  [structural]
```

Type checking and inference:

The most interesting part. Normally we write inference rules in papers to express typing rules, and we can transform them into a programming language, and compile them as a type checker.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : A \to B}{\Gamma \vdash f(a) : B}$$

Done in Spoofax, JetBrains MPS, etc.

$$\frac{\Gamma \vdash \mathbf{valid}}{\Gamma \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (x : A) \times B : \mathsf{Set}_i}$$

$$\frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (x : A) \to B : \mathsf{Set}_i} \qquad \frac{\Gamma \vdash \mathbf{valid}}{\Gamma \vdash \mathsf{1} : \mathsf{Set}_0} \qquad \frac{\Gamma \vdash \mathbf{valid} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B[x := s]}{\Gamma \vdash \langle s, t \rangle : (x : A) \times B} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash \pi_1\ t : A} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash \pi_2\ t : B[x := \pi_1\ t]}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : (x : A) \to B} \qquad \frac{\Gamma \vdash s : (x : A) \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B[x := t]} \qquad \frac{\Gamma \vdash \mathbf{valid}}{\Gamma \vdash \langle\rangle : \mathsf{1}}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash t : B}$$

Spoofax (statix): encode scoping rules as typing rules and write typing rules as unification rules.

```
module lang/base/statics

imports signatures/lang/base/syntax-sig

rules // type of ...

  typeOfType : scope * Type → TYPE
  typeOfExp  : scope * Exp  → TYPE

rules // well-typedness of ...

  declOk : scope * Decl
  declsOk maps declOk(*, list(*))

  bindOk : scope * scope * Bind
  bindsOk maps bindOk(*, *, list(*))
```

```
typeOfExp(s, Add(e1, e2)) = INT() :-
  typeOfExp(s, e1) == INT(),
  typeOfExp(s, e2) == INT()
```

Then, you describe typing rules for each AST type.

```
module lang/arithmetic/statics

imports lang/base/statics
imports signatures/lang/arithmetic/syntax-sig

signature
  constructors
    INT : TYPE

rules
  typeOfType(s, IntT()) = INT().

rules
  typeOfExp(s, Int(i)) = INT().

  typeOfExp(s, Min(e)) = INT() :-
    typeOfExp(s, e) == INT().

  typeOfExp(s, Add(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Sub(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().

  typeOfExp(s, Mul(e1, e2)) = INT() :-
    typeOfExp(s, e1) == INT(),
    typeOfExp(s, e2) == INT().
```

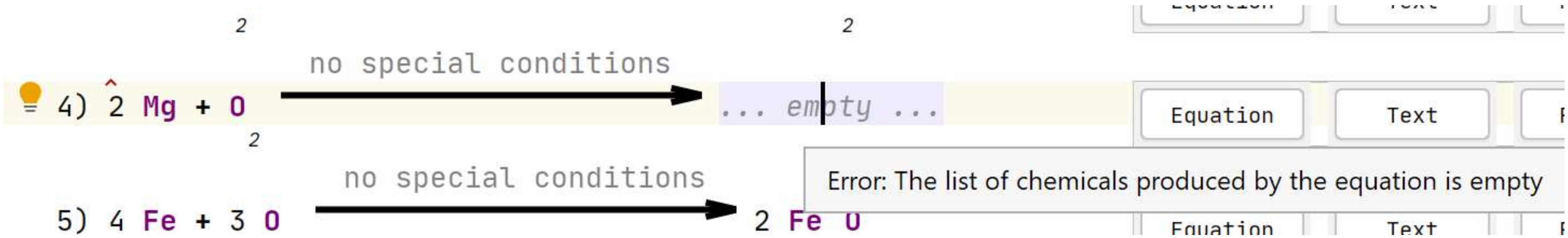
```
module signatures/lang/arithmetic/syntax-sig

imports signatures/lang/base/syntax-sig

signature
  constructors
    Int : INT -> Exp
    Min : Exp -> Exp
    Add : Exp * Exp -> Exp
    Sub : Exp * Exp -> Exp
    Mul : Exp * Exp -> Exp
    IntT : Type
```

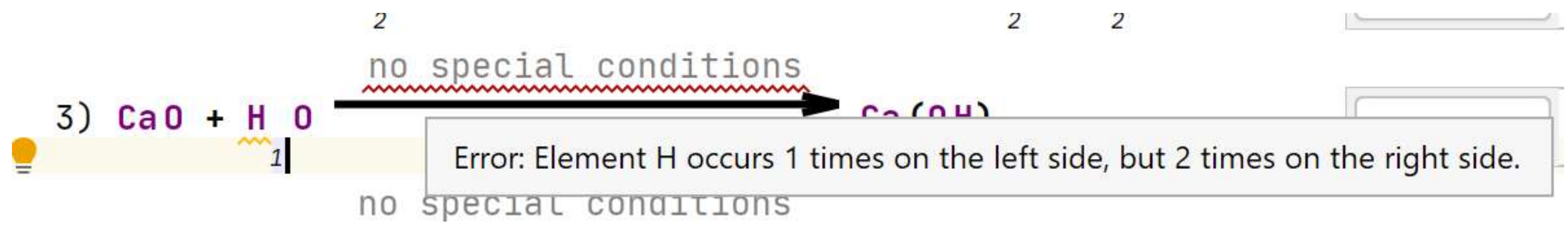
JetBrains MPS: write inference rules in Java.

```
concept ChemEquation extends    BaseConcept
                     implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  condition : string

  children:
  left  : EquationComponent[0..n]
  right : EquationComponent[0..n]
```

```
checking rule check_ChemEquation {
  applicable for concept = ChemEquation as chemEquation
  overrides <none>

  do {
    if (chemEquation.left.isEmpty) {
      error "The list of chemicals entering the equation is empty" -> chemEquation;
    }
    if (chemEquation.right.isEmpty) {
      error "The list of chemicals produced by the equation is empty" -> chemEquation;
    }
    if (chemEquation.left.isNotEmpty && chemEquation.right.isNotEmpty) {
      ElementSummary sumL = new ElementSummary(chemEquation.left.ofConcept<Compound>);
      ElementSummary sumR = new ElementSummary(chemEquation.right.ofConcept<Compound>);
      if (!sumL.isSameAs(sumR)) {
        error sumL.comparisonReport(sumR) -> chemEquation;
      }
    }
  }
}
```

Typing rules are checked automatically.





If we can compile this, we can make something that takes syntax + typing rule + code generation as input, and produces a compiler, a pretty printer, an IDE, and maybe formatters, etc.

Thank you
for your attention

jetbrains.com