Model-Driven Development of DSL Interpreters

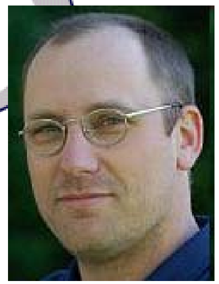
Using Scala and oAW

Markus Völter
voelter@acm.org
www.voelter.de

About me

- Independent Consultant
- Based out of Goeppingen, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Product Line Engineering

völter.de
ingenieurbüro für softwaretechnologie



Markus Völter
voelter@acm.org
www.voelter.de

Founder and Editor of

Software Engineering Radio
the Podcast for Professional Developers
http://se-radio.net

CONTENTS

CONTENTS

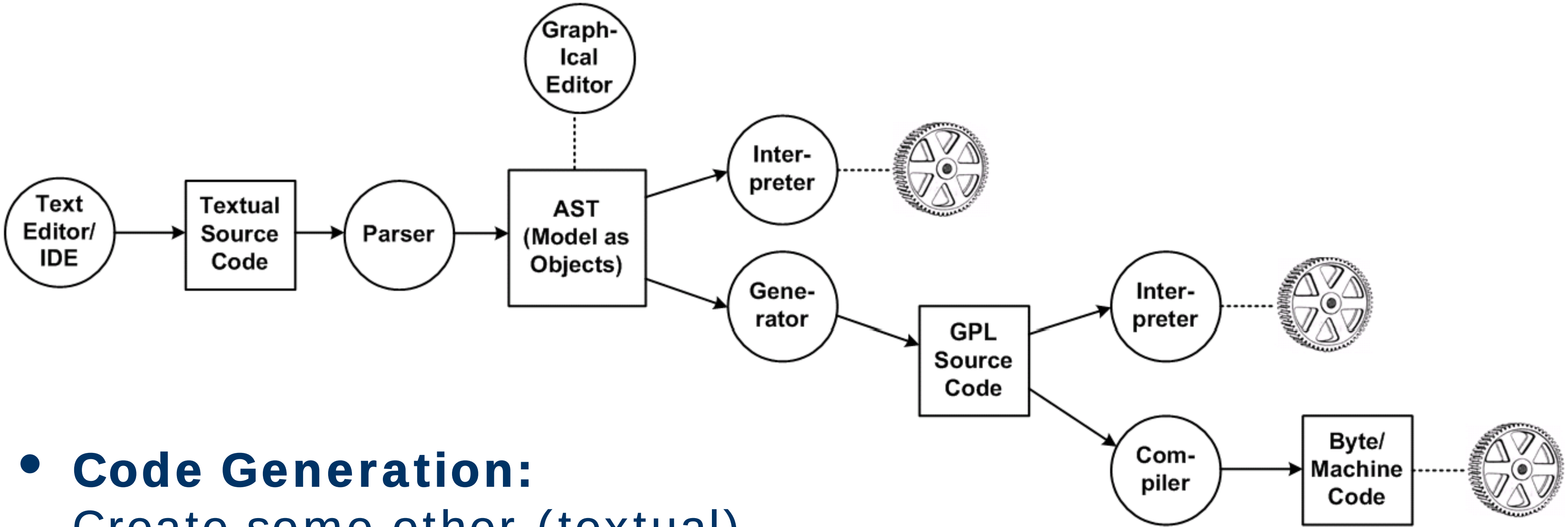
Domain-Specific Languages

A DSL is a **focused**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- **Processable** means that the „DSL Program“ is to be processed by some tool
  - Analysis/Simulation
  - Code Generation
  - Interpretation

CONTENTS

Interpretation vs. Generation: Basics



- **Code Generation:**
  Create some other (textual) representation of the model, typically in a 3GL, interpret or compile it in order to execute.

- **Interpretation:**
  Write a program (the interpreter) that directly executes the AST, i.e. walks it and creates sideeffects.

Code Generation vs. Interpretation: Pros and Cons

- Code Generation is the **mainstream** for external DSLs.
  - Maybe it is **perceived to be simpler**
  - It certainly has **better tool support** (template languages such as Xpand, MofScript or JET.

- And **code generation** does have a number of **advantages:**
  - Can generate **artifacts necessary for certain platforms** (config files, satisfy existing APIs)
  - Can produce **small**, **fast** or **optimized** code if necessary
  - Can be done „secretly“ – **nothing** special required at **runtime** of the final system
  - Generated code is one meta level down **– semantic gap is reduced**, simplifies understanding and debugging
  - Templates can be **derived** from **existing** (manually written) code

Code Generation vs. Interpretation: Pros and Cons II

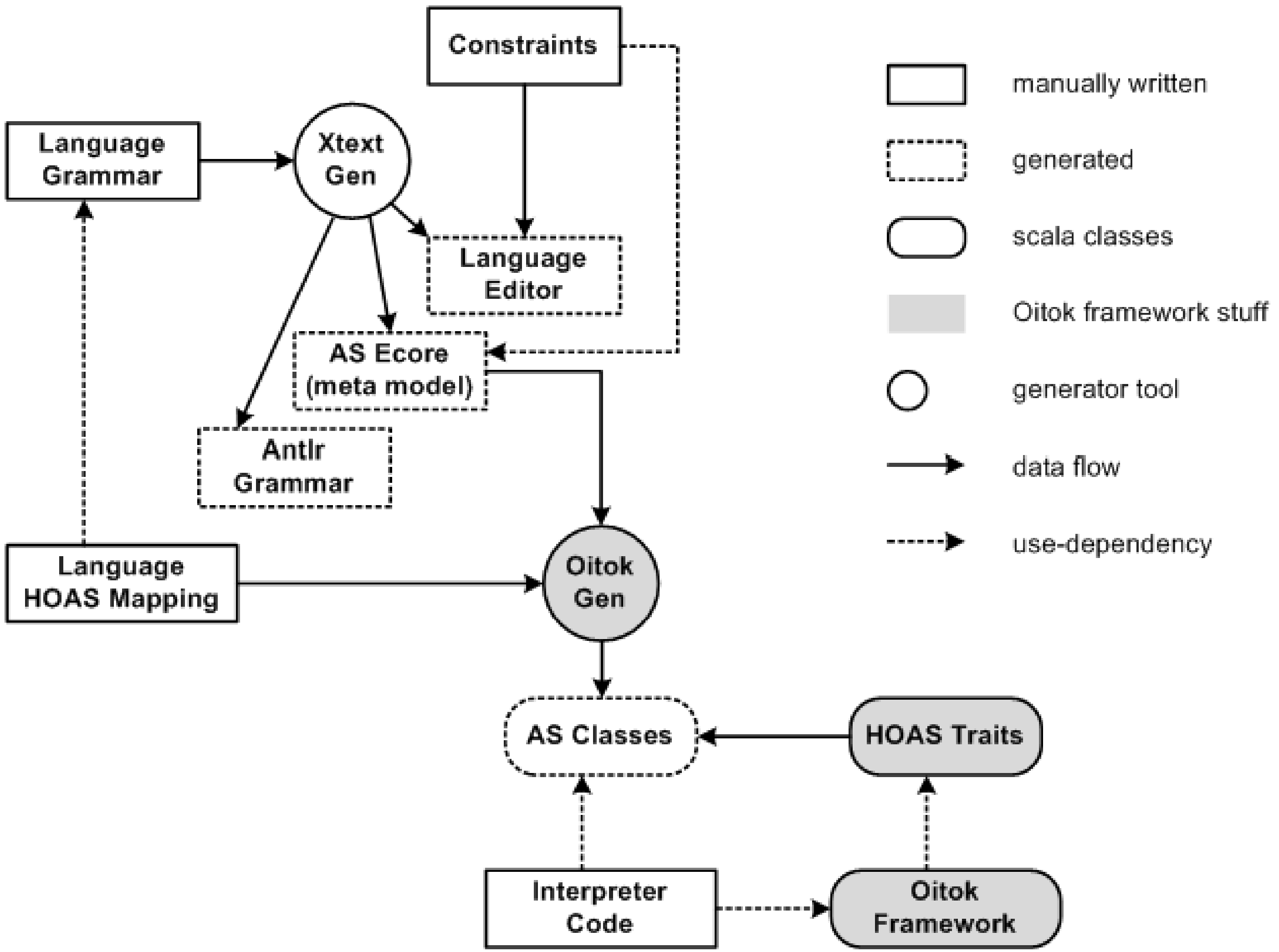
- There are some **advantages** of an an **interpreter**, however:
  - **Faster Turnaround**, because no regeneration necessary
  - **Embeddable**, scripts can be edited and rerun in the deployed application
- There are also **combinations** of the two approaches where
  - A first stage generates something that a second stage interprets
  - Or an interpreter works on some kind of low-level language, where higher-level languages are transformed into those lower level languages.
- Interpreters should be used more in practice. This talk shows how tool support could look like.

CONTENTS

Oitok – the oAW Interpreter ToolKit

- **Oitok** is the framework on which this talk is based.
- It is in **an early stage of development** and not yet released officially.
- Oitok is intended for **relatively simple and small languages** (typical DSLs). It is not suitable for building „real“ VMs.
- In the context of this talk, Oitok is more important to **illustrate how to build interpreters** as opposed to being a production-ready framework for mission critical interpreter construction.
  - This will hopefully change in the future.

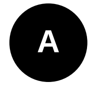
Overall Oitok Process



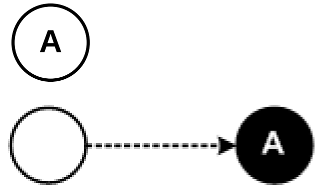
CONTENTS

Key concepts in Interpreters

- **Symbol**
  - A named entity such as a variable



- **SymbolRef**
  - A reference to a symbol

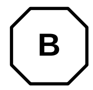


- **RValue** (Right-Value)
  - Something that can be evaluated and yields a value, such as an expression or a SymbolRef

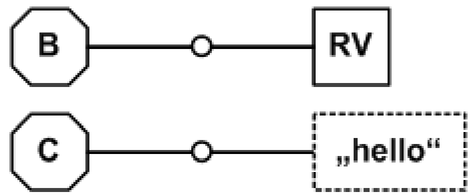
- **Literal**
  - An RValue with a fixed value, such as 1 or „hello“

Key concepts in Interpreters II

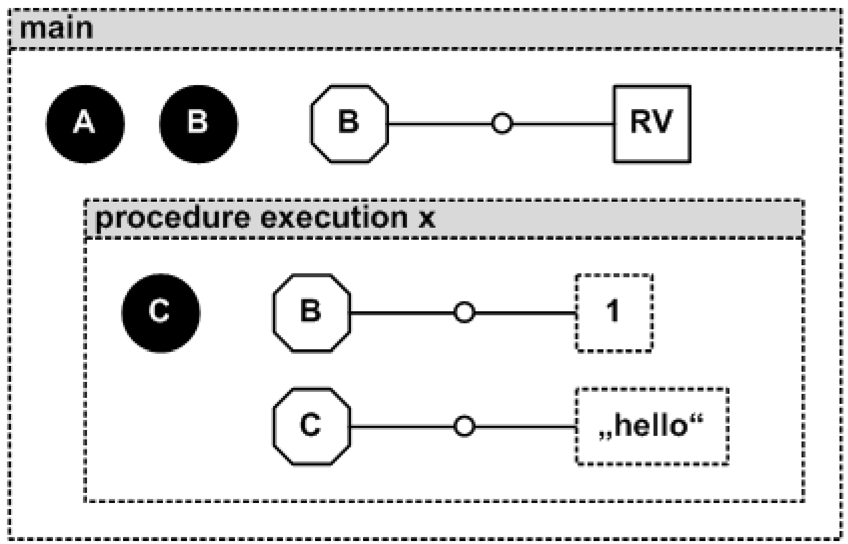
- **LValue** (Left-Value)
  - Something to which something else can be assigned, such as a Symbol or a SymbolRef.



- **Binding**
  - Assigns the value of an RValue to an Lvalue

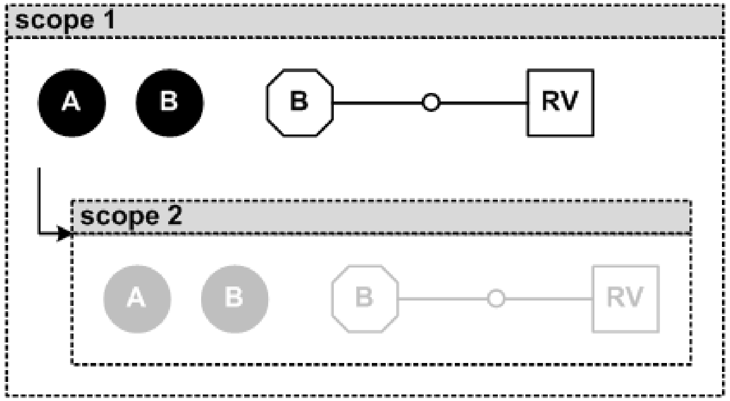


- **Scope**
  - An area of the program (execution) where symbols are defined and bindings are valid.
  - Can be nested.
  - Are kept on a stack.

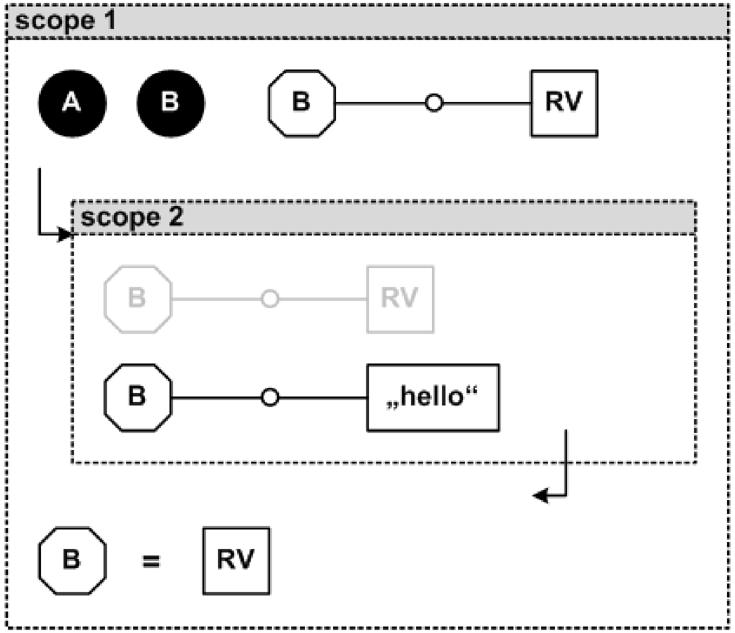


Key concepts in Interpreters III

- **Scope Creation**
  - A new scope is created as a clone of an existing scope
  - Changes to symbols and bindings in the new scope do not affect original scope.

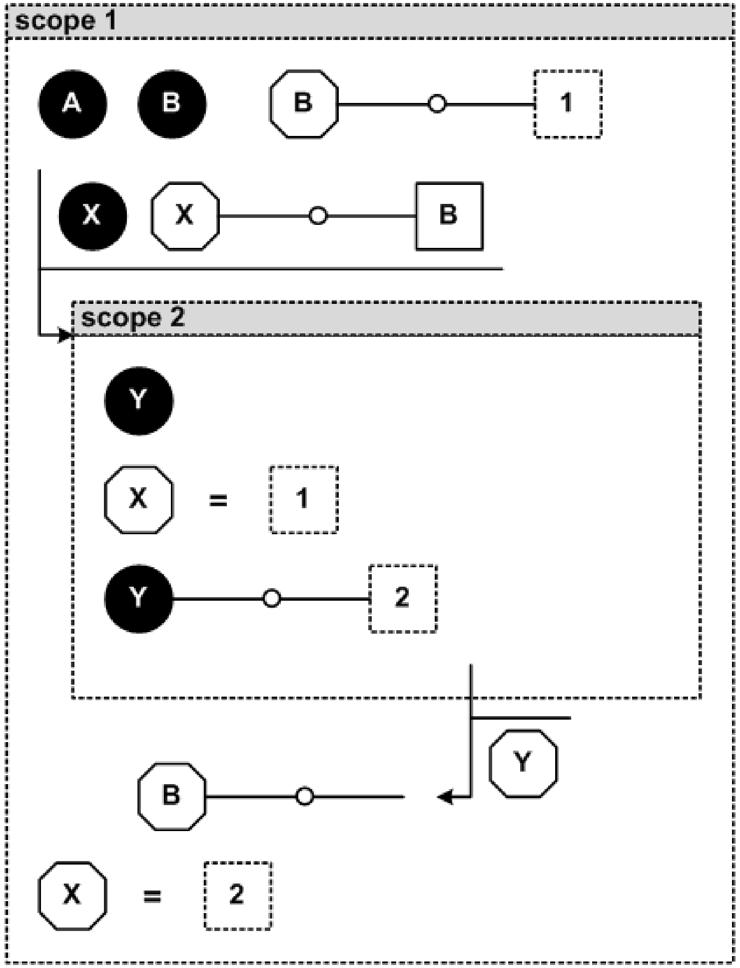


- **Scope Termination**
  - The current scope is terminate, all its symbols and bindings are lost.
  - We return to original scope, where symbols and bindings are unchanged, like before the scope creation.



Key concepts in Interpreters IV

- **Parameter Passing**
  - When creating a scope, new initial bindings can be created. This is called parameter passing.
  - When a scope is terminated, one or more values are returned to the original scope.
  - They can be bound to symbols in the original scope.
  - Parameters can also be passed by reference; changes to them in the new scope also affect the original scope.



Key concepts in Interpreters II

- **Call**
  - A call jumps to a different **location** in the code
  - Typically, **scope creation** is inherent in a call
  - If you call something procedure-like, the call is a **statement**
  - If you call something function-like (i.e. it returns a value) the call is an **RValue**
- In Oitok, a *call* is basically a wrapper around scope creation, termination and parameter passing.
- We also create call stack for error reporting.

CONTENTS

Example Program in Example Language

- Calculating the Factorial using a simple language.

```
local A Fac Res in
  Fac := function(X)
    if X = 0 then
      return 1
    else local Xm Fm T in
      Xm := X - 1
      Fm := Fac(Xm)
      T := X * Fm
      return T
    end
  end
  A := 5
  Res := Fac(A)
end
```

- The language has variables, simple expressions, functions, assignments, blocks and calls.

- **Challenge:**
  How can we build an interpreter for this language?

Building an interpreter

- We need to be able to **parse the textual syntax**
  - We use oAW Xtext to define a grammar, generate an editor as well as a parser and an AST.
- We then need to **implement the interpreter** working on the AST created by the parser.
  - We simplify this task by **mapping** the AST concepts onto the general interpreter concepts above.
  - We then build the interpreter in Scala, supported by a framework that knows about the generic interpreter concepts.
- Because the generic concepts are a kind of AST for an AST, it is called **Higher Order Abstract Syntax**.

CONTENTS

Step 1: Defining the Grammar

- The grammar is defined using oAW **Xtext‘s grammar editor**

- It is basically an EBNF notation

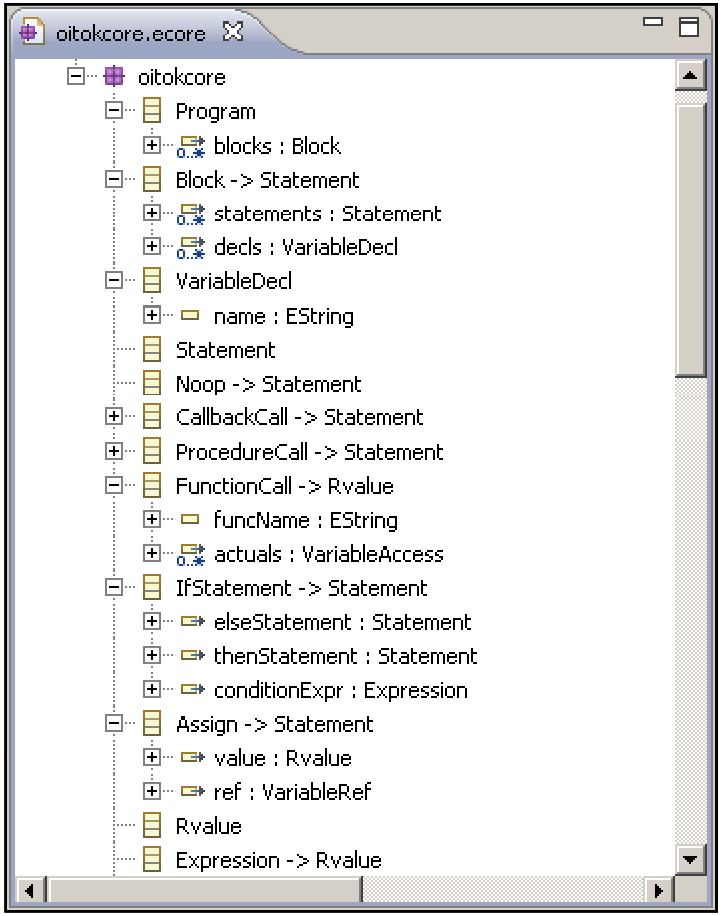
- Note how some rules are collapsed.

oitokcore.xtxt

```
Program:
    (blocks+=Block)*;

Block:
    "local" (decls+=VariableDecl)* "in"
        (statements+=Statement)*
    "end";

VariableDecl:
    name=ID;

Statement:
    Noop | Assign | Block | DebugStatement |
    ProcedureCall | CallbackCall | IfStatement |
    ReturnStatement;

Noop: "noop";

CallbackCall:

ProcedureCall:

FunctionCall:
    funcName=ID "(" (actuals+=VariableAccess)* ")";

IfStatement:

Assign:

Rvalue:

Expression:

ListOp:

ListMod:
```

Step 1a: The generated AS

- Xtext derives an **abstract syntax** for the language from the grammar.
- It is represented as an **ecore metamodel** so it can be processed using Eclipse/oAW easily.
- Note how
  - the rules from the grammar become **meta classes**
  - Attributes become **properties**
  - and all the **concrete syntax stuff is lost**.



Step 2: Static Semantics (Constraints)

- Static Semantics is defined against the **abstract syntax** using regular oAW constraints against the generated meta model.

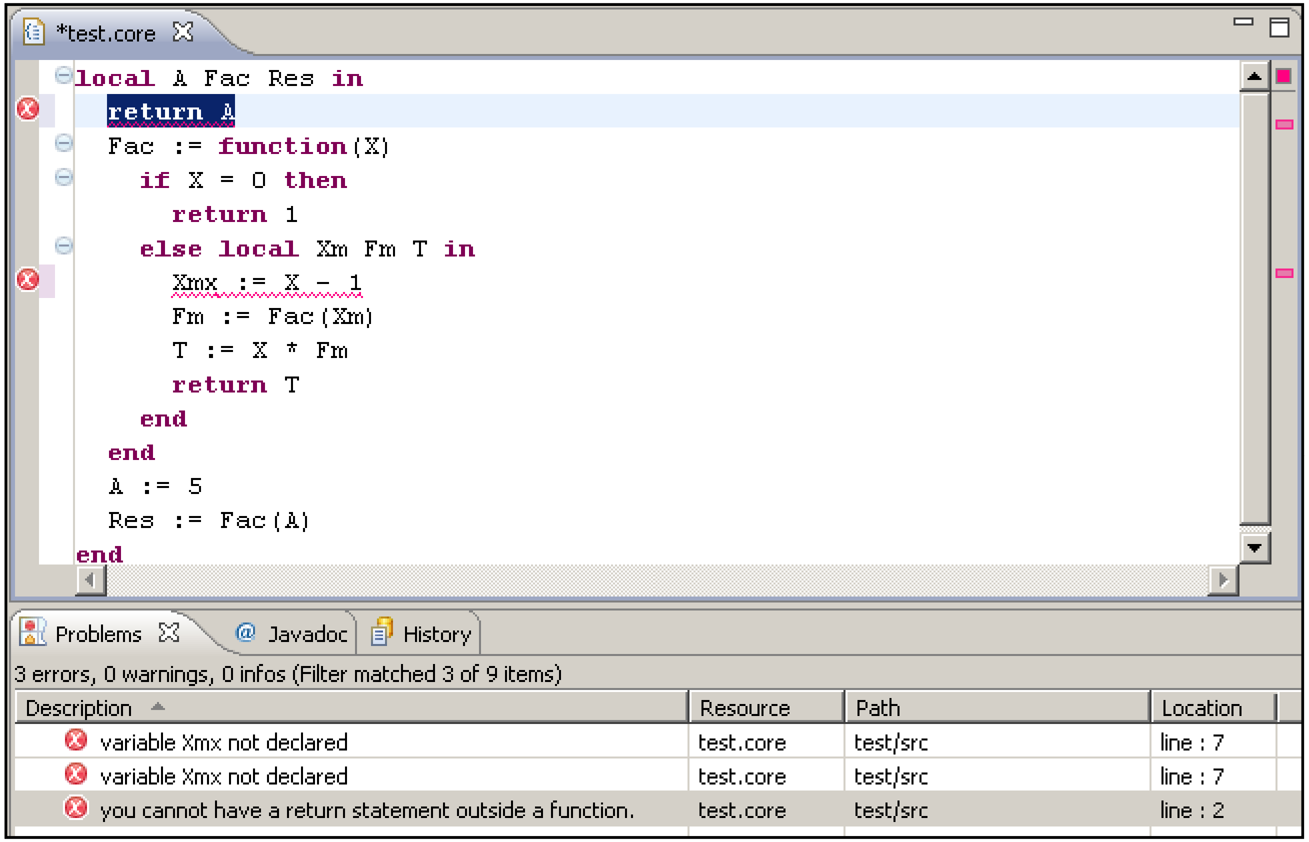
Checks.chk

```
import oitokcore;

extension org::openarchitectureware::oitok::core::Extensions;

context ReturnStatement ERROR "you cannot have a return statement outside a function." :
   allParents().exists( p | FunctionDeclaration.isInstance(p) );

context VariableRef ERROR "variable "+referencedVariable+" not declared":
    isDeclared( this, referencedVariable );

context VariableAccess ERROR "variable "+referencedVariable+" not declared":
    isDeclared( this, referencedVariable );

context Assign ERROR "variable "+ref.referencedVariable+" not declared":
    isDeclared( this, ref.referencedVariable);

context FunctionCall ERROR "function "+funcName+" not declared":
    isDeclared( this, funcName);

context ProcedureCall ERROR "procedure "+procName+" not declared":
    isDeclared( this, procName);

context VariableDecl ERROR "variable named "+name+" is duplicate":
    ((Block)eContainer).decls.select(d|d.name == name).size == 1;

context Formal if ProcedureDeclaration.isInstance(eContainer) ERROR "formal named "+name+
    ((ProcedureDeclaration)eContainer).formals.select(d|d.name == name).size == 1;

context Formal if FunctionDeclaration.isInstance(eContainer) ERROR "formal named "+name+"
    ((FunctionDeclaration)eContainer).formals.select(d|d.name == name).size == 1;
```

CONTENTS

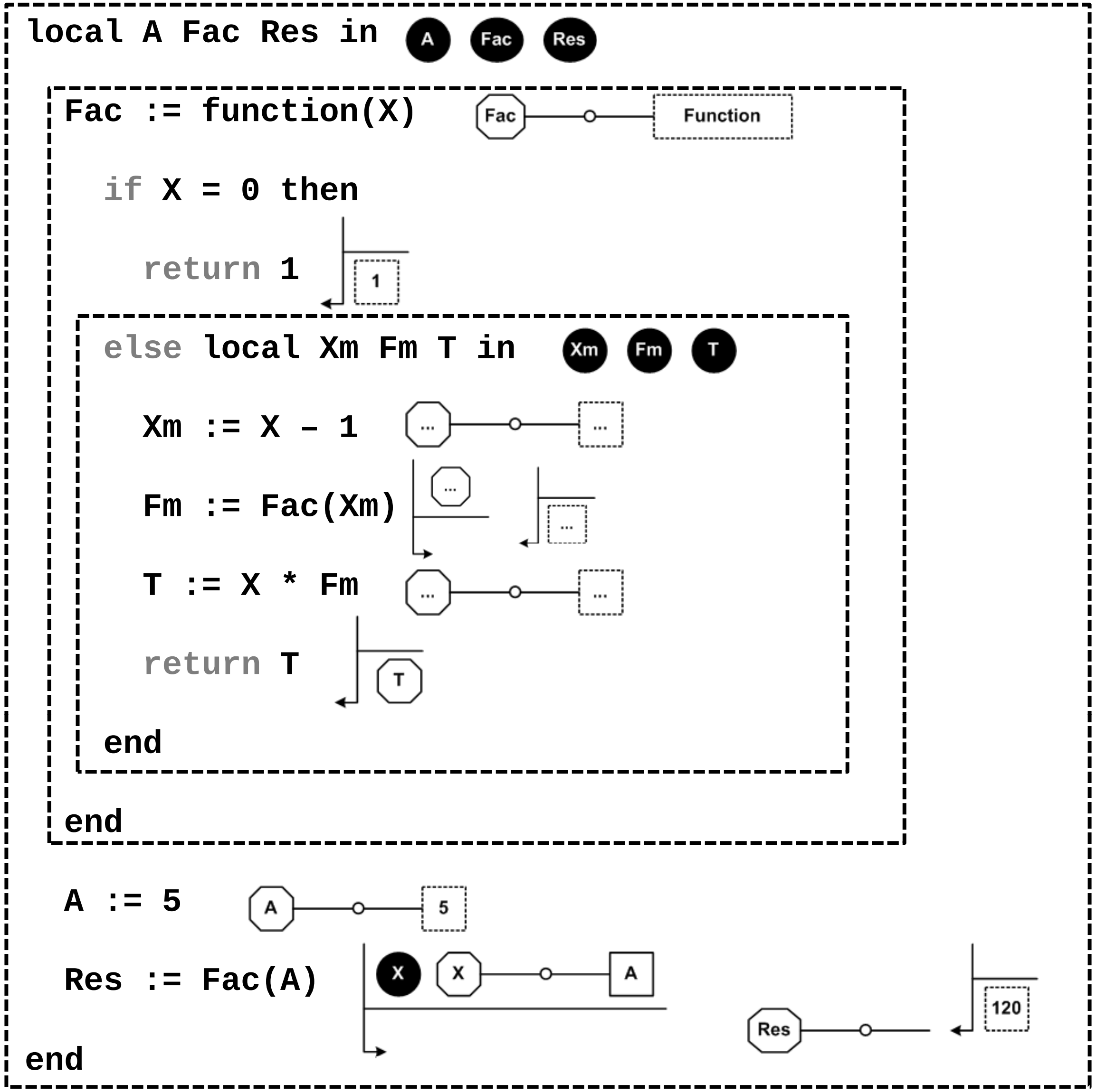
Step 3: Editing the Program

- Usingthe **editor generated with Xtext**, editing programs in this language is very convenient.
  - Syntax Highlighting, Code Completion, Constraint Checking



CONTENTS

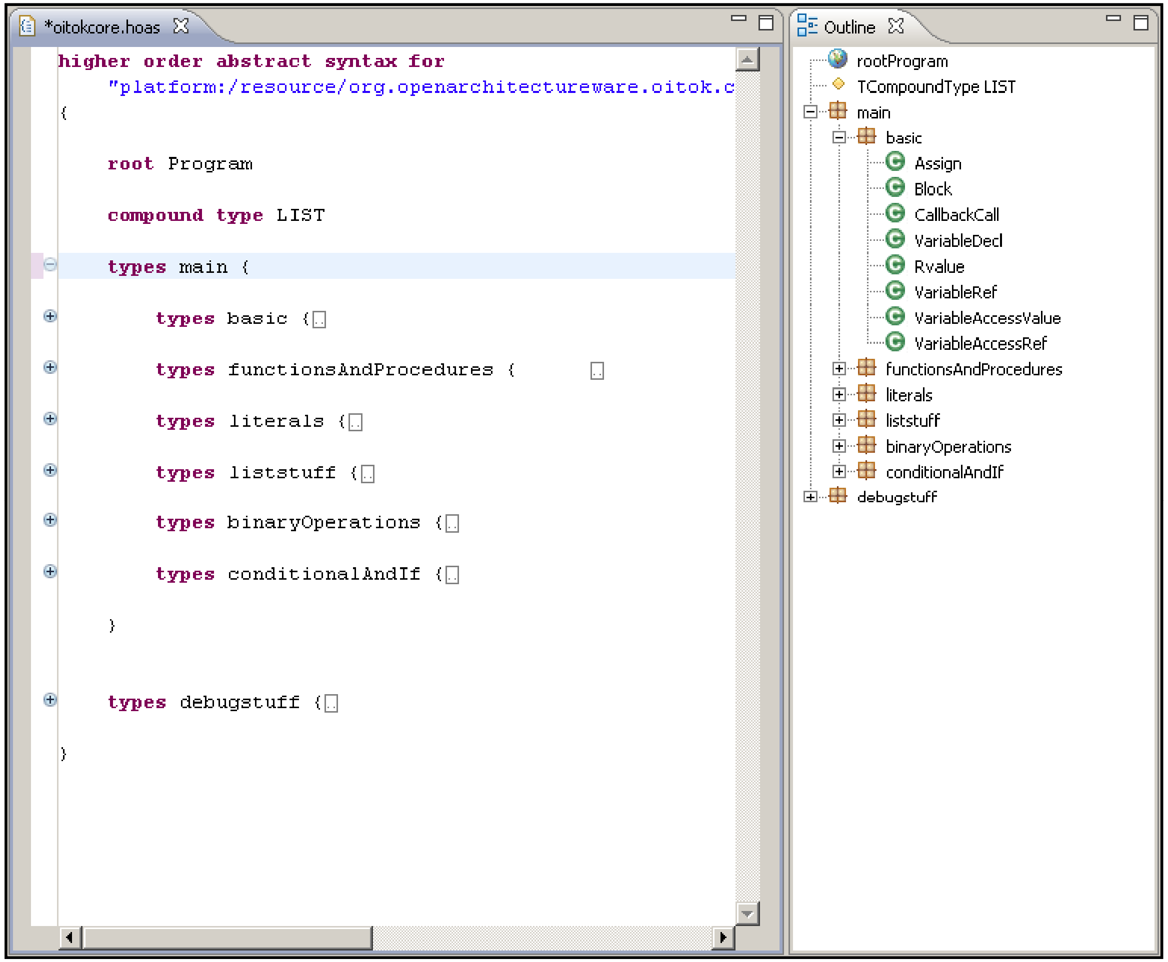
Step 4: Mapping towards HOAS / Example Program



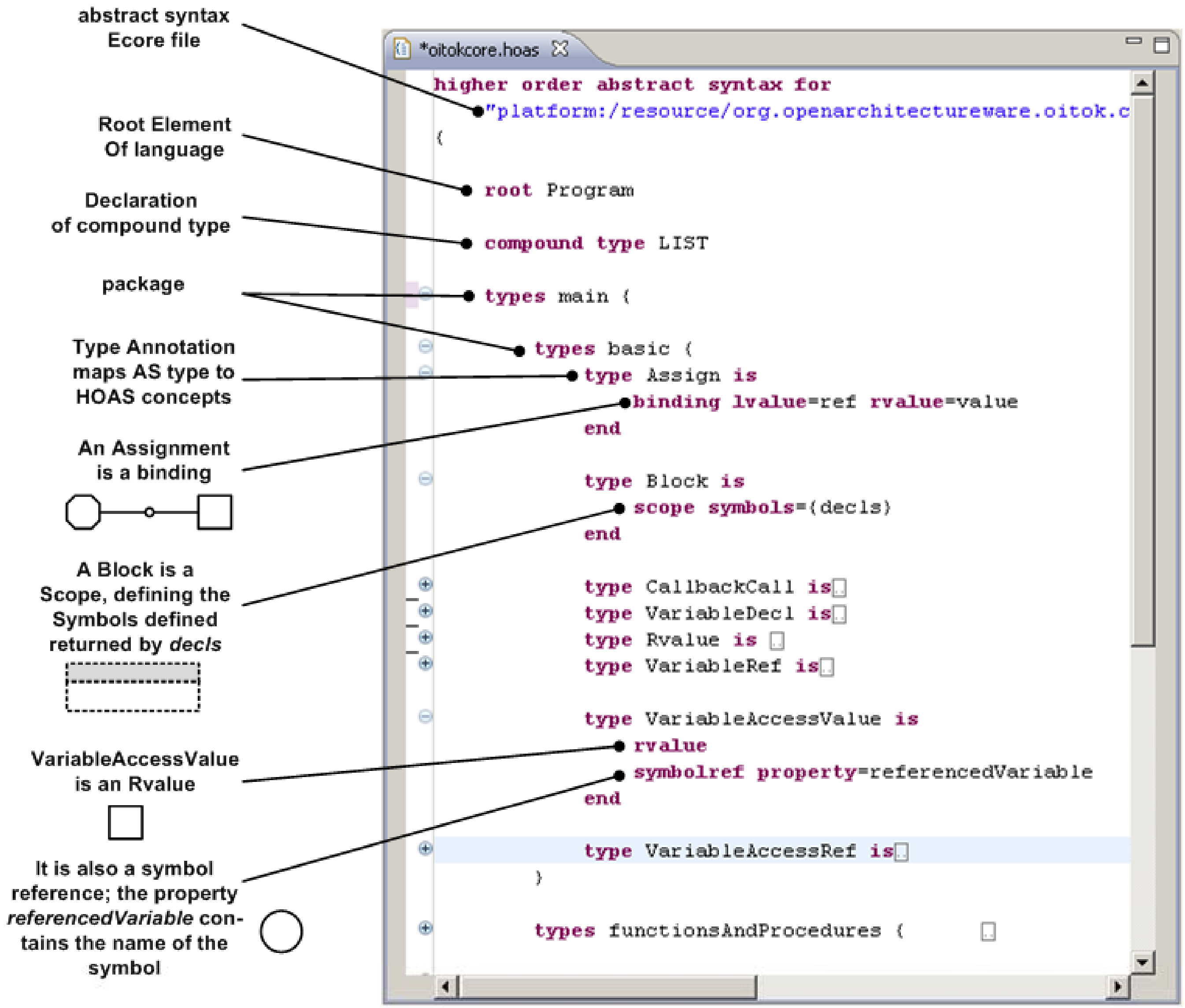
- Most of the concepts in the example language can be **mapped** to the generic interpreter HOAS.
- The diagram on the left mixes example program text with the HOAS notation introduced before.

Step 4: Mapping File

- We first run the **Oitok Wizard** that creates an interpeter project for a given Xtext syntax.
- To keep things modular **the HOAS mapping** is kept in **sepearate file**.
- The HOAS file specified the AS ecore file for which it contains the HOAS mapping
- The editor is of course also built with Xtext ☺

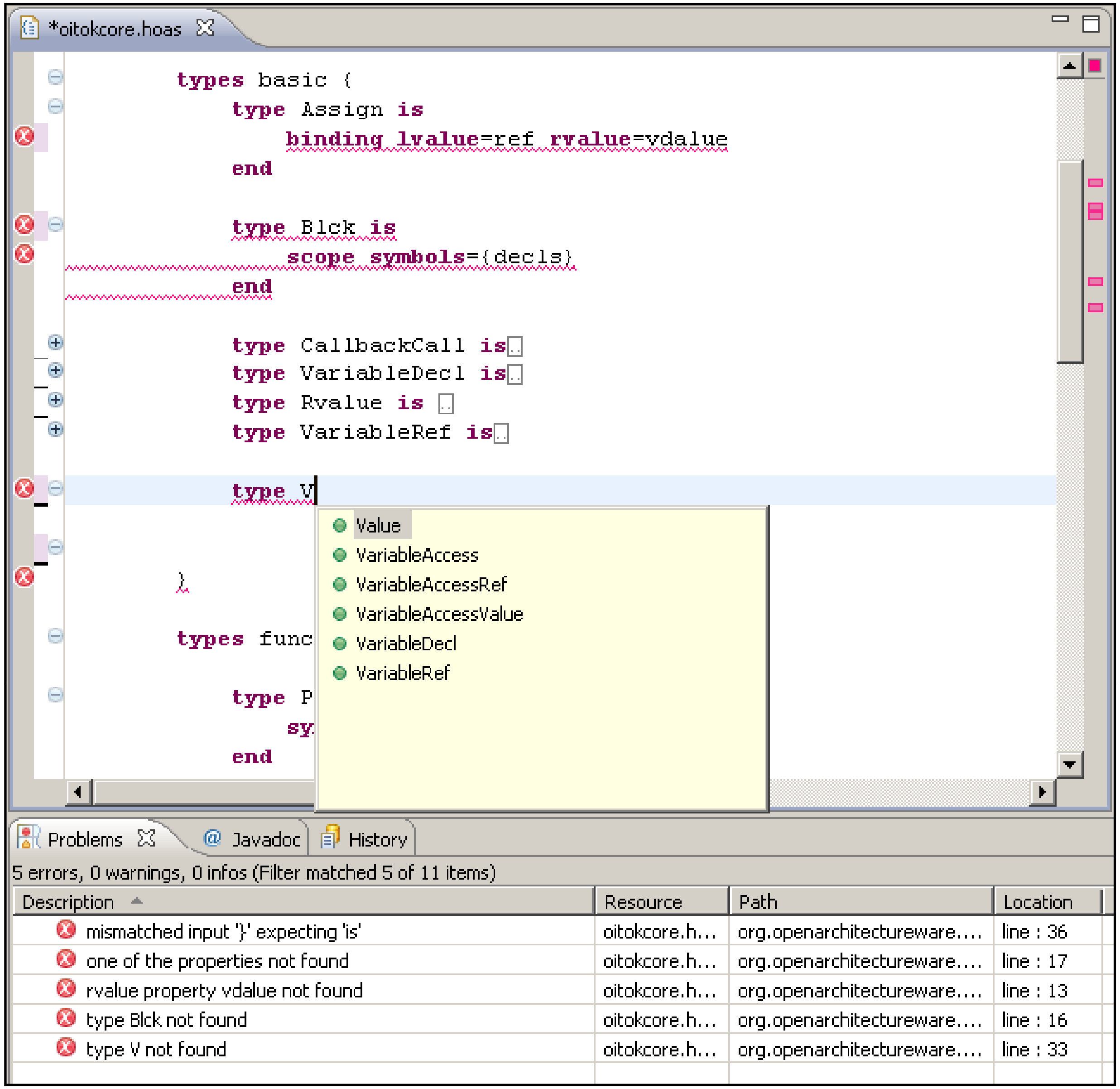


Step 4: Mapping File II



Step 4: Mapping File III

- Since the HOAS editor knows about the ecore file for which the HOAS is defined, it provides **code completion** and **constraint checks** into the ecore file.



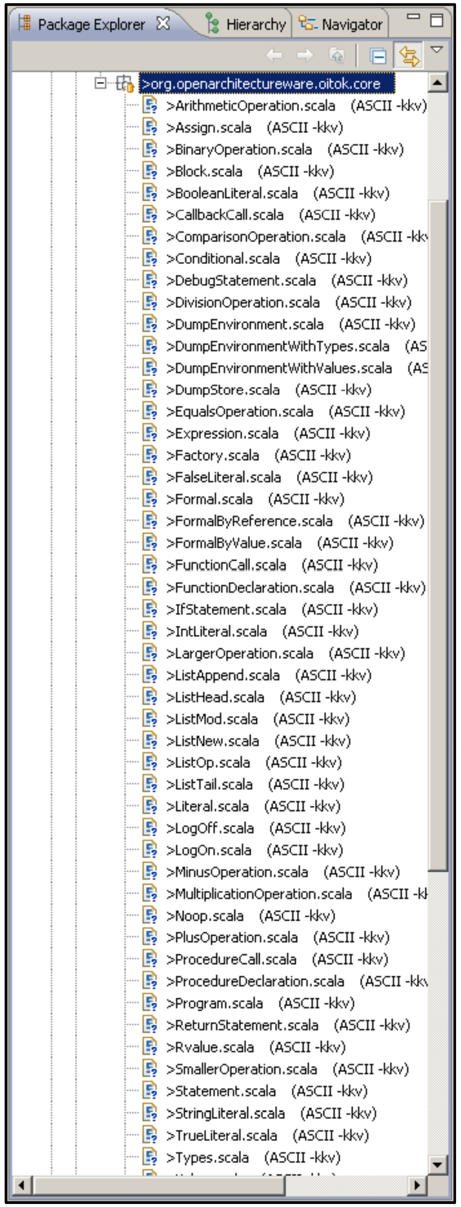
CONTENTS

Step 5: Generating the Scala Code

- The next step is to **generate the Scala classes** representing the language‘s abstract syntax.

- This is not unlike the EMF Java generator,
  - However, it generates Scala ☺
  - The generated code provides read access to the model only
  - And takes into account the HOAS mapping.

- For each HOAS concept, there is a **trait** in the Oitok framework which is mixed into the class
  - The generator **generates the implementation** of the abstract methods of the trait.



Step 5: Example Traits

- **HSymbolRef:**
whenever something is a reference to a symbol

```
package org.openarchitectureware.oitok.fw.hoas;

trait HSymbolRef extends HLValue {

  def hreferencedName(): String

  def hsymbolName() = hreferencedName()

}
```

- **HRValue**
the important characteristic is that it can be evaluated to return a 2-tuple consisting of value and type.

```
package org.openarchitectureware.oitok.fw.hoas;

import org.openarchitectureware.oitok.fw.types._

trait HRValue extends HHoasRoot  {

  def hevaluate(): Tuple2[Any,TType] = {
    throw new ProgramTerminated("for some strange reason, hevaluate() is not
  }

  def hevaluate(expectedType: TType): Tuple2[Any,TType] = {
    throw new ProgramTerminated("for some strange reason, hevaluate() is not
  }
}
```

- **HBinding:**

```
package org.openarchitectureware.oitok.fw.hoas;

trait HBinding extends HHoasRoot {

  def lvalue(): HLValue

  def rvalue(): HRValue

}
```

- **HScope:**

```
package org.openarchitectureware.oitok.fw.hoas;

trait HScope extends HHoasRoot  {

  var isolated = false

  def setIsolated( b: Boolean ) = isolated = b

  def hisIsolated() = isolated

  def hintroducedSymbols(): List[HSymbol]

}
```

Step 6: Writing the Interpreter itself

- Subclass the framework class **Engine** and overwrite *handleInternal* and *evaluateInternal*.

```
class CoreEngine(factory: ElementFactory) extends Engine(factory) {

  def handleInternal( el: Base ): Unit = el match {
    case o:_ => handleDefault(o)
  }

  override def evaluateInternal( element: HRValue ): Any = element match {
    case _ => super.evaluate(element)
  }

}
```

- *handleInternal* deals with statements (things that do not yield a value)
- *evaluateInternal* evaluates things that return values.
- Use Scala's **pattern matcher** to treat your AS classes specifically

Step 6: Writing the Interpreter itself II

- Implement a **case block for each of your AS classes** that have been generated by the Oitok generator.

```
class CoreEngine(factory: ElementFactory) extends Engine(factory) {

  def handleInternal( el: Base ): Unit = el match {
    case p:Program => ...
    case b:Block => ...
    case a:Assign => ...
    ...
    case o:_ => handleDefault(o)
  }

  override def evaluateInternal( element: HRValue ): Any = element match {
    case b:BinaryOperation => ...
    case fd:FunctionDeclaration => ...
    ...
    case _ => super.evaluate(element)
  }

}
```

Step 6: Writing the Interpreter itself III

```
def handleInternal( el: Base ): Unit = el match {
  case p:Program => handle( p.blocks() )
```

- Handling a program means to **handle all the blocks** in the program sequentially
  - *handle(...)* is a framework method that automatically takes care of **scoping**, if the handled element is a *HScope*.

```
def handleInternal( el: Base ): Unit = el match {
  case p:Program => handle( p.blocks() )
  case b:Block => handle( b.statements() )
```

- Handling a block means to **handle all the statements** in the block sequentially
  - *Statement* is the superclass of *Block*, *Program*, *IfStatement*, etc.

Step 6: Writing the Interpreter itself IV

```
def handleInternal( el: Base ): Unit = el match {
  case p:Program => handle( p.blocks() )
  case b:Block => handle( b.statements() )
  case a:Assign => rt.env().bind( a )
```

- Handling an assignment simply means **calling the *bind(..)* method** on the current environment (re: scopes!).
  - Since binding is a HOAS concept, the framework knows what to do in case of a binding – no coding required.

```
def handleInternal( el: Base ): Unit = el match {
  case p:Program => handle( p.blocks() )
  case b:Block => handle( b.statements() )
  case a:Assign => rt.env().bind( a )
  case f:FunctionDeclaration => handle( f.statements() )
```

- Handling a *FunctionDeclaration* (when it is executed!) simply means **to execute all the statements** in the function body.
  - The framework handles scopes

Step 6: Writing the Interpreter itself V

```
def handleInternal( el: Base ): Unit = el match {
  case p:Program => handle( p.blocks() )
  case b:Block => handle( b.statements() )
  case a:Assign => rt.env().bind( a )
  case f:FunctionDeclaration => handle( f.statements() )
  case ret:ReturnStatement => {
    val res = ret.evalRvalue()
    throw new ScopeTerminated(res)
  }
```

- In case of a **return statement**, we evaluate the *RValue* that comes with the return statement, and then we throw a *ScopeTerminated* exception passing the result.

- Finally, the *IfStatement* should be self-explaining by now.

```
def handleInternal( el: Base ): Unit = el match {
  ...
  case i:IfStatement => handleIfStatement(i)
}

def handleIfStatement( ifs: IfStatement ) = {
  val (v:Boolean,t) = ifs.evalConditionExpr()
  if ( v ) handle( ifs.thenStatement )
  else handle( ifs.elseStatement )
}
```

Step 6: Writing the Interpreter itself VI

```
override def evaluateInternal( element: HRValue ): Any = element match {
  case va:VariableAccessValue => {
    rt.env().resolveStorage( va.referencedVariable() ).asTuple()
  }
```

- Evaluating a *VariableAccessValue* is as simple as **returning a 2-tuple (value, type)** of the variable that‘s referenced by the *VariableAccessValue*

```
override def evaluateInternal( element: HRValue ): Any = element match {
  case va:VariableAccessValue => {
    rt.env().resolveStorage( va.referencedVariable() ).asTuple()
  }
  case fd:FunctionDeclaration => fd
```

- In case a *FunctionDeclaration* shows up in an **RValue position** (e.g. on the right side of an assignment) the value of the function declaration is itself.
  - This assigns it to its symbol in the symbol table

Step 6: Writing the Interpreter itself VII

```
override def evaluateInternal( element: HRValue ): Any = element match {
  ...
  case b:BinaryOperation => {
    val (l:Int,lt) = b.evalLeft(INT)
    val (r:Int,rt) = b.evalRight(INT)
    b match {
      case _:MultiplicationOperation => (l * r, INT)
      case _:MinusOperation => (l - r, INT)
      case _:EqualsOperation => (l == r, BOOLEAN)
      case _ => super.evaluateInternal(element)
    }
  }
```

- For *BinaryOperations* we first **evaluate the two arguments** (which returns value and type)
- Then we **perform** the arithmetic or comparison **operations** itself
- We then **return the 2-tuple** (new value, new type) as the value of this RValue

Step 6: Writing the Interpreter itself VIII

```
override def evaluateInternal( element: HRValue ): Any = element match {
  ...
  case c:FunctionCall => evaluateCall(c)
  ...
}
```

- Because a HCall is a HOAS feature, you can simply **let the framework handle** the call.
  - A *FunctionCall* is an *call,* it specifies the list of actuals and the type of what it can call.
  - A *FunctionDeclaration* is a *callable*, and it specifies the formals.

```
types functionsAndProcedures {

    type FunctionCall is
        symbolref property = funcName
        rvalue
        call target=FunctionDeclaration
             actuals=actualArgs
    end

    type FunctionDeclaration is
        scope symbols={formalArgs}
        rvalue type=FUNC
        callable formals=formalArgs
    end

}
```

Step 6: Writing the Interpreter itself IX

- **That‘s it.**

- The code on the right is the **complete code** that needs to be written for the interpreter

- (Of course you also need the grammar and the HOAS file).

```
class CoreEngine(factory: ElementFactory) extends Engine(factory) {

  def handleInternal( el: Base ): Unit = el match {
    case p:Program => handle( p.blocks() )
    case b:Block => handle( b.statements() )
    case a:Assign => rt.env().bind( a )
    case f:FunctionDeclaration => handle( f.statements() )
    case ret:ReturnStatement => returnFromScope( ret.evalRvalue() )
    case i:IfStatement => {
      val (v:Boolean,t) = i.evalConditionExpr()
      if ( v ) {
        handle( i.thenStatement )
      } else {
        handle( i.elseStatement )
      }
    }
    case o:_ => handleDefault(o)
  }

  override def evaluateInternal( element: HRValue ): Any = element match {
    case va:VariableAccessValue => {
      rt.env().resolveStorage( va.referencedVariable() ).asTuple()
    }
    case b:BinaryOperation => {
      val (l:Int,lt) = b.evalLeft(INT)
      val (r:Int,rt) = b.evalRight(INT)
      b match {
        case _:MultiplicationOperation => (l * r, INT)
        case _:MinusOperation => (l - r, INT)
        case _:EqualsOperation => (l == r, BOOLEAN)
        case _ => super.evaluateInternal(element)
      }
    }
    case fd:FunctionDeclaration => fd
    case c:FunctionCall => evaluateCall(c)
    case _ => super.evaluateInternal(element)
  }

}
```

CONTENTS

Execution Trace

- The execution trace of a program can be written to *stderr*

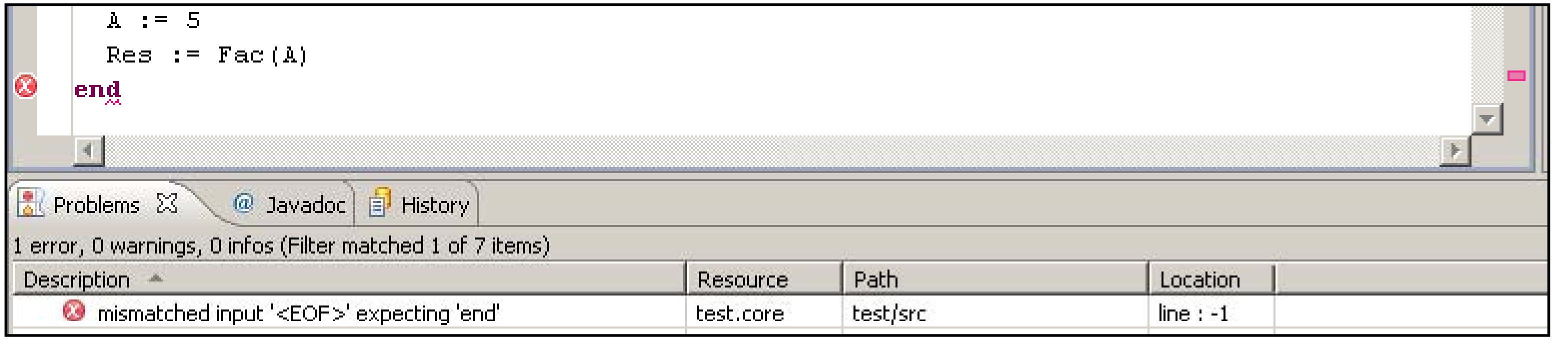
```
local A B F in
  F := function(X)
    return X
  end
  B := 10
  A := F(B)
end
```

```
:: Program[blocks={Block[] } ]
:: Block[statements={Assign[] Assign[] Assign[] } decls={A B F } ]
->(1)E:
  :: Assign[value=FunctionDeclaration[] ref=ref(F) ]
  bind F <- FunctionDeclaration[formals={X } statements={ReturnStatement[] } ]
  :: Assign[value=10 ref=ref(B) ]
  bind B <- 10
  :: Assign[value=ref(F) ref=ref(A) ]
  :: FunctionDeclaration[formals={X } statements={ReturnStatement[] } ]
  ref F <- {FunctionDeclaration[formals={X } statements={ReturnStatement[] } ],FUNC}
  ref B <- {10,INT}
  ref A <- {}
  E:A,B=10,F=FunctionDeclaration[]
  ->(2)E:A,B=10,F=FunctionDeclaration[]
    bind X <- 10
    :: ReturnStatement[rvalue=ref(X) ]
    !! terminate scope with (10,INT)
  <-E:A,B=10,F=FunctionDeclaration[],X=10
  (1)E:A,B=10,F=FunctionDeclaration[]
  bind A <- 10
<-E:A=10,B=10,F=FunctionDeclaration[]
```

Error Reporting

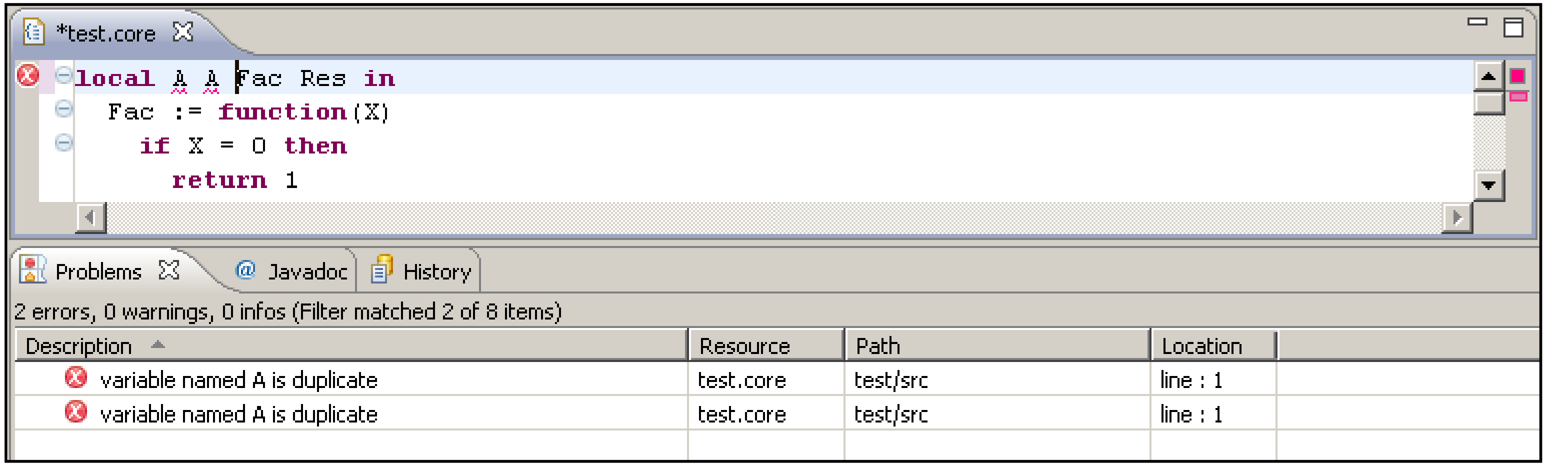
- There are **three kinds** of errors:
  - Parsing Errors
  - Constraint Violations
  - Runtime Errors

- **Parsing Errors** are reported by antlr either inside the editor or when parsing the program in batch mode.



Error Reporting II

- **Constraint Violations** are also reported in the editor...



- ... as well as during parsing in **batch mode:**

```
program validation failed:
- line 14: you cannot have a return statement outside a function.
```

Error Reporting III

- **Runtime Errors** are reported as *Program-Terminated* exceptions (at runtime of course!):
  - You'll get a call stack, incl. line numbers

```
program terminated:
VariableAccessValue[Fm]: type INT expected, but evaluated to UNDEF
  at line 17, FunctionCall[Fac]
  at line 11, FunctionCall[Fac]
  at line 11, FunctionCall[Fac]
  at line 11, FunctionCall[Fac]
  at line 11, FunctionCall[Fac]
```

```
local A Fac Res in
  Fac := function(X)
    if X = 0 then
      local A in
        A := "hallo"
        A := 1
      end
    else local Xm Fm T in
      Xm := X - 1
      Fm := Fac(Xm)
      T := X * Fm
      return T
    end
  end
  A := 5
  Res := Fac(A)
end
```

Testing

- There‘s **special testing support:**
  Testing environments, symbol bindings and types at the end and in arbitrary other places in the program

Test Case extends the EngineTestCase class

You have to implement the runTest() method

then call run to execute an actual program

This uses Scala' practical """ very long string with line breaks """ feature ☺

Here we express how the environment should be at the end of the program

This is where we instantiate the engine we want to test

FunctionTests.scala | CoreTestsRunner.scala | factory.xpt | Factory.scala

```
class FunctionTests extends EngineTestCase("first experiments with functions") {

  override def runTest() = {
    run("""
      local A Fac Res in
        Fac := function(X)
          if X = 0 then
            return 1
          else local Xm Fm T in
            Xm := X - 1
            Fm := Fac(Xm)
            T := X * Fm
            return T
          end
        end
        A := 5
        Res := Fac(A)
      end
      """, "E:A=5,Fac=FunctionDeclaration[],Res=120" );
  }

  override def createRuntime( prg: String ): Rt =
    StandaloneEngine.run(prg, "counter" -> new CounterCallback())

}
```

CONTENTS

Simple Expression Language

- Simple **Expression Language** with arithmetic and comparion features

```
1: 6 = 2 * (2+1);
2: 20 = (4+1)*2 + (4+1)*2;
3: false = true && false;
4: true = (true || false) && true;
5: true = 1 > 2 == (2 > 3);
6: true = (2 * 16 + (40 - 8)) ==
             (12 + (4+1)*2 + (4+1)*2 + 12 + (4+1)*2 + (4+1)*2 +
              12 + (4+1)*2 + (4+1)*2 + 12 + (4+1)*2 + (4+1)*2) / (1+1);
```

- Interpreter ca. 120 lines of code

- To be **included in other DSLs** that need to use a simple expression language

CONTENTS

Why Scala?

- Scala is generally a **very nice language** (my opinion: this is what Java 8 should be!)
- The resulting program is **much more concise** than the equivalent in Java – it has this Ruby/Groovy feel to it.
- However, it is **statically typed** (with a very advanced type system and compiler) and hence has much better performance than Ruby + Co.
- Finally, it is specifically useful for **working with trees** (such as ASTs) because of its match/case facilities.
- **And also:** I really wanted to learn and evangelize it ☺

Is Scala ready for real-world use?

- IDE
- Complex: Conceptual complexity vs. „chaotic“ complexity eg in C++

CONTENTS

More?

- **openArchitectureWare:** http://openarchitectureware.org

- **Scala Language**
  Web Site: *http://scala-lang.org*
  Scala Book: *http://artima.com/shop/forsale*
  Podcast: *http://se-radio.net/podcast/2007-07/episode-62-martin-odersky-scala*

- **Languages Stuff** in General
  Recommended Book:

  Concepts, Techniques, and Models of Computer Programming

  by Peter Van Roy, Seif Haridi