

Meta-Programming and MDE with Rascal

Jeroen van den Bos, Mark Hills, Paul Klint, Tijs van der Storm
and Jurgen J. Vinju

CWI, INRIA ATEAMS, NFI

2nd Workshop on Algebraic Methods in Model-Based Software
Engineering (AMMSE 2011)
Zurich, Switzerland, 30 June 2011



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Rascal: From Algebraic Specification to Meta-Programming

Or...



A Plug for the Paper

- Lessons learned: ASF to ASF+SDF to Rascal
- Some background: design principles of Rascal
- Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications



A Plug for the Paper

- Lessons learned: ASF to ASF+SDF to Rascal
- Some background: design principles of Rascal
- Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications



A Plug for the Paper

- Lessons learned: ASF to ASF+SDF to Rascal
- Some background: design principles of Rascal
- Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications



A Plug for the Paper

- Lessons learned: ASF to ASF+SDF to Rascal
- Some background: design principles of Rascal
- Overviews of several Rascal applications, with a focus on MDE and (briefly) linking to existing algebraic specifications



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



What's Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.



What's Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.



What's Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.



What's Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.



What's Rascal?

Rascal is

- a programming language
- for source code analysis and transformation
- with rich data types, higher-order functions,
- specialized control flow, and advanced pattern matching, including matching over concrete syntax.

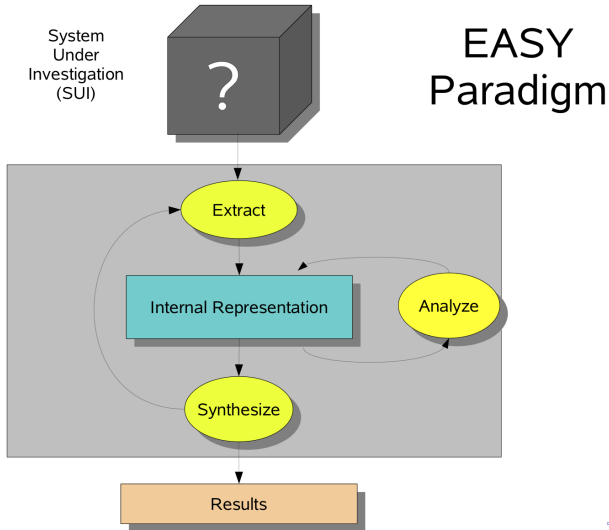


Rascal Features

- Familiar, C or Java-like syntax
- Immutable data
- Rich built-in data types and pattern matching
- Domain-specific constructs (traversals, comprehensions, regular expressions, fixed-point computation)
- Arbitrary context-free grammars with generalized parsing
- String templates
- Java and Eclipse integration



Extract, Analyze, SYNthesize



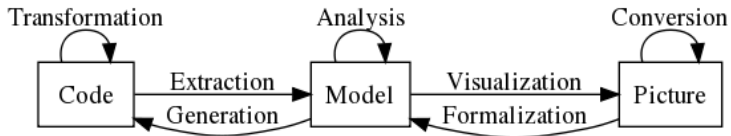
Rascal is EASY

Rascal follows the EASY paradigm:

- Information is *Extracted* from the program, such as the program's abstract syntax
- This information is then used to *Analyze* the program, for instance to check consistency, generate a control flow graph, or bind names to definitions
- Finally, the extracted information and the analysis results are used to *Synthesize* the desired results, such as by transforming the code or generating visualizations



Domain Analysis for Rascal: Meta-Programming



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Which Language is This?

```
PROCEDURE Swap(VAR x, y: INTEGER);  
VAR  
  temp: INTEGER;  
BEGIN  
  temp := x;  
  x := y;  
  y := temp  
END Swap;
```



Why Look at a Standard Programming Language?

- Similar challenges across standard PLs, DSLs, modelling languages, etc
- Similar desired functionality: IDEs, consistency checking, program analysis, code generation, etc



Why Look at Oberon-0?

- Part of work done for tools competition at this year's LDTA
- Focused on features as a showcase for Rascal – shows what one could do for a language defined in Rascal
- Features include checkers, code generation, visualization, IDE menu links
- Not too Oberon specific: features shown are ones you could use for your own language



Goals of Oberon-0 Implementation

- Modular
- Functional
- Visual



Goals of Oberon-0 Implementation

- Modular
- Functional
- Visual



Goals of Oberon-0 Implementation

- Modular
- Functional
- Visual



Parsing in Rascal

- Grammars defined using Rascal grammar definition notation
- A Rascal program then builds a Java-based parser for the grammar
- Parser is GLL with filtering rules used to remove ambiguities



Example: Oberon-0 Grammar

syntax Statement

= assign: Ident var " := " Expression exp

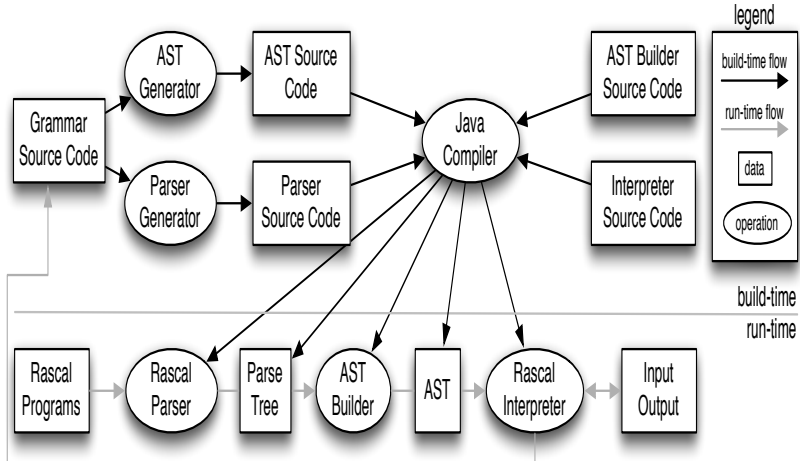
| ifThen: "IF" Expression condition "THEN"
 {Statement ";" }+ body
 ElsIfPart*
 ElsePart?
 "END"

| whileDo: "WHILE" Expression condition "DO"
 {Statement ";" }+ body
 "END"

;



Rascal Meta-Programming Architecture



Code Outlining Example: Java in Eclipse

```
}  
  
public void print(IValue arg, IEvaluatorContext eval){  
    PrintWriter currentOutputStream = eval.getStdOut();  
  
    synchronized(currentOutputStream){  
        try{  
            if(arg.getType().isStringType()){  
                currentOutputStream.print(((IString) arg).getValue().toString());  
            }else if(arg.getType().isSubtypeOf(Factory.Tree)){  
                currentOutputStream.print(TreeAdapter.viewOf((IConstructor) arg));  
            }  
        }  
    }  
}
```

- IO(ValueFactory)
- print(IValue, IEvaluatorContext) : void
- iprint(IValue, IEvaluatorContext) : void
- iprintln(IValue, IEvaluatorContext) : void
- println(IValue, IEvaluatorContext) : void
- rprintln(IValue, IEvaluatorContext) : void
- rprint(IValue, IEvaluatorContext) : void
- readFile(IString) : IValue
- exists(SourceLocation, IEvaluatorContext) : IValue
- lastModified(SourceLocation, IEvaluatorContext) : IValue



Outlining Support in Rascal: Building the Outline

- Outlines are built over the concrete syntax of a language
- Labels indicate the display name in the outline view
- Locations allow the user to jump to the outlined item
- Once the outliner is registered, the runtime keeps the view up to date as the source is edited



Code Outlining Example: Oberon-0 in Rascal

The screenshot displays the Rascal IDE interface. The main editor window shows the source code for a Rascal program. The code defines a procedure named 'Multiply' and a procedure named 'Divide'. The 'Multiply' procedure is currently selected in the Outline view on the right. The Outline view shows a tree structure of the code, with the 'Multiply' procedure expanded to show its internal structure, including constants, types, variables, and nested procedures like 'Divide' and 'BinSearch'.

```
PROCEDURE Multiply;
  VAR x, y, z: INTEGER;
BEGIN
  Read(x);
  Read(y);
  z := 0;
  WHILE x > 0 DO
    IF x MOD 2 = 1 THEN
      z := z + y
    END ;
    y := 2*y;
    (* Dag *)
    x := x DIV 2 END;
  Write(x);
  Write(y);
  Write(z);
  WriteLn
END Multiply;

(* def *)

PROCEDURE Divide;
  VAR x, (* Q *) y, r, q, w: INTEGER;
BEGIN
```

Outline view structure:

- Constants
- Types
- Variables
- Procedures
 - Nesting
 - Multiply
 - Constants
 - Types
 - Variables
 - x, y, z
 - Divide
 - BinSearch



Annotators

- Annotators allow annotations to be added to language constructs and displayed in the editor
- Typical examples: name resolution, type checking – want errors to be displayed graphically to users, marking error locations

```
public Module checkModule(Module x) {  
  m = implode(x);  
  <m, st> = resolve(m);  
  errors = { error(l, s) | <l, s> <- st.scopeErrors };  
  if (errors == {}) {  
    errors = check(m, st.symbolTable);  
  }  
  return x[@messages = errors];  
}  
  
registerAnnotator("l4", checkModule);
```



Annotator Example: Type Checking Oberon-0

The screenshot shows an IDE window titled '*collatz.l3' containing the following Oberon-0 code:

```
1 MODULE Collatz;  
2  
3 VAR even, odd : INTEGER;  
4  
5 PROCEDURE doCollatz();  
6   VAR current : INTEGER;  
7     currentEven : BOOLEAN;  
8  
9   PROCEDURE computeEven();  
10  BEGIN  
11    IF current MOD 2 = 0 THEN  
12      currentEven := even  
13    ELSE  
14      currentE  
15    END  
16  END computeEven;  
17
```

Line 12 is highlighted in blue. A yellow tooltip points to the assignment 'currentEven := even', displaying the error: 'Cannot assign value of type INTEGER, expected type BOOLEAN'. The 'Outline' window on the right shows a tree structure with 'Variables' expanded to show 'even, odd'.



Contributors

- Contributors provide a way to add more advanced functionality
- Each contribution is a menu item – execution is triggered by the user
- Examples: interaction with external tools, compilation, visualization



An Example Contributors Menu

```
END printSequence;  
BEGIN  
  Read(current);  
  printSequence()  
END doCollatz;  
BEGIN  
  even := 1;  
  odd := 0;  
  doCollatz()  
END Collatz.
```

Run As ▶
Debug As ▶
Validate
Team ▶
Compare With ▶
Replace With ▶
Pretty Print
WikiText ▶
Preferences...
Oberon ▶
Remove from Context ⚙️ ↕️

Compile to C
Compile to Java
Format
Obfuscate (protect your precious oberon0 code!)
Show control flow graphs
Compile to Java bytecode and run

Writable



Visualization Contribution: Control Flow Graph

The image shows a screenshot of an IDE with two windows. The left window displays Rascal code for a swap function, and the right window displays a Control Flow Graph (CFG) visualization of that code.

```
collatz.i3  collatz.i4  test.i1  swap.i3  Figure
```

```
20= BEGIN
21   swap2(w,x);
22   swap2(x,y);
23   swap2(y,z);
24   swap2(z,w)
25   END swap4;
26
27= PROCEDURE swap3Twice(VAR x,y,z : INTEG
28= BEGIN
29   swap3(x,y,z);
30   swap3(x,y,z)
31   END swap3Twice;
32
33= BEGIN
34   a := 1;
35   b := 2;
36   c := 3;
37   d := 4;
38   Write(a); Write(b); Write(c); Write(
39   swap4(a,b,c,a); swap3Twice(a,b,c); s
40   Write(a); Write(b); Write(c); Write(
41   END Swap.
```

The CFG visualization shows a linear flow of operations:

- start Swap
- a := 1
- b := 2
- c := 3
- d := 4
- Write(a)
- Write(b)
- Write(c)
- Write(d)
- WriteLn()
- swap4(a, b, c, a)
- swap3Twice(a, b, c)
- swap3Twice(b, c, d)



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages**
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Goals

- First, define a language, with support tools, for entities
- Then, extend this to support packages for modularity
- Next, extend this language to support entity instances
- Finally, add modular extensions to the language

Note: Work by Tijs van der Storm, presented at LWC'11 by Jurgen J. Vinju



Goals

- First, define a language, with support tools, for entities
- Then, extend this to support packages for modularity
- Next, extend this language to support entity instances
- Finally, add modular extensions to the language

Note: Work by Tijs van der Storm, presented at LWC'11 by Jurgen J. Vinju

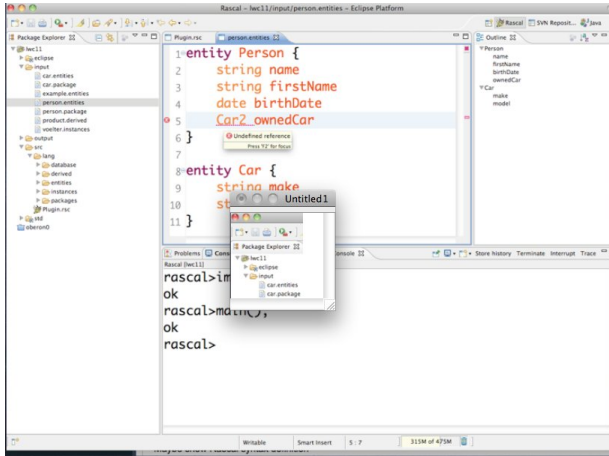


Entities and Instances

- Immediate IDE: highlighting, folding, error marking, etc
- Java and SQL generation
- Online checking and error marking



An IDE for Entities



Generating Java Code

```
1 public class Person {
2
3     private java.lang.String name;
4     public java.lang.String getName() {
5         return this.name;
6     }
7     public void setName(java.lang.String name) {
8         this.name = name;
9     }
10 }

1 public class Car {
2
3     private java.lang.String make;
4     public java.lang.String getMake() {
5         return this.make;
6     }
7     public void setMake(java.lang.String make) {
8         this.make = make;
9     }
10 }
```

```
rascal>import Plugin;
ok
rascal>main();
ok
```



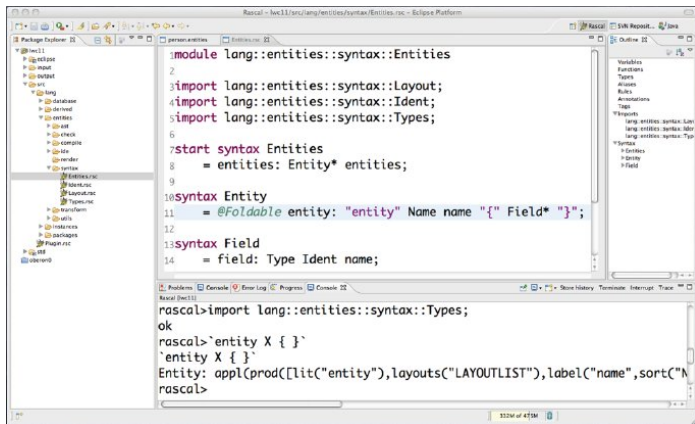
Generating SQL Code

```
1
2create table Person (
3  _id int primary key,
4  name varchar ,
5  firstName varchar ,
6  birthDate date ,
7  ownedCar int foreign key references Car(_id)
8);
9
10create table Car (
11  _id int primary key,
12  make varchar ,
13  model varchar
14);
15
16
```

```
rascal>import Plugin;
ok
rascal>main();
ok
```



Defining Entity Concrete Syntax



```
module lang::entities::syntax::Entities
2
3 import lang::entities::syntax::Layout;
4 import lang::entities::syntax::Ident;
5 import lang::entities::syntax::Types;
6
7 start syntax Entities
8   = entities: Entity* entities;
9
10 syntax Entity
11   = @Foldable entity: "entity" Name name "{" Field* "}";
12
13 syntax Field
14   = field: Type Ident name;
```

```
Rascal (Jvarkit)
rascal>import lang::entities::syntax::Types;
ok
rascal>`entity X { `
`entity X { `
Entity: appl(prod([lit("entity"),layouts("LAYOUTLIST"),label("name"),sort("N
rascal>
```



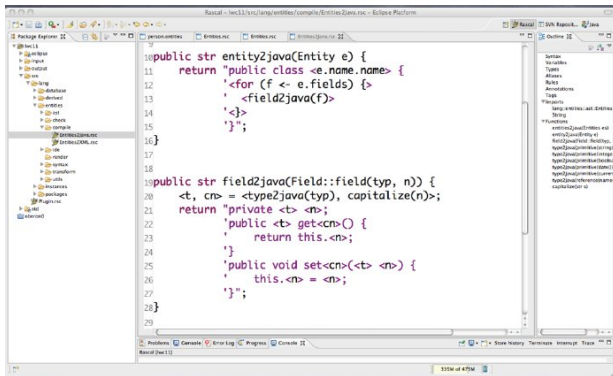
Defining Entity Abstract Syntax

```
1 module lang::entities::ast::Entities
2
3 data Entities
4   = entities(list[Entity] entities);
5
6 data Entity
7   = entity(Name name, list[Field] fields);
8
9 data Field
10  = field(Type \type, str name);
11
12 data Type
13  = primitive(PrimitiveType primitive)
14  | reference(Name name);
```

int: 2
rascal>2 + 2
int: 4
rascal>3 + 3
int: 6
rascal>



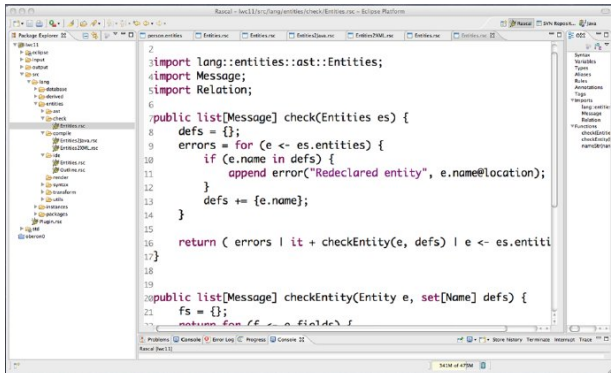
String Template-based Code Generation



```
10 public str entity2java(Entity e) {
11     return "public class <e.name.name> {
12         <for (f <- e.fields) {>
13             <field2java(f)>
14         <>>
15     }";
16 }
17
18
19 public str field2java(Field::field(typ, n)) {
20     <t, cn> = <type2java(typ), capitalize(n)>;
21     return "private <t> <cn>;
22         public <t> get<cn>() {
23             return this.<cn>;
24         }
25         public void set<cn>(<t> <cn>) {
26             this.<cn> = <cn>;
27         }";
28 }
29
```



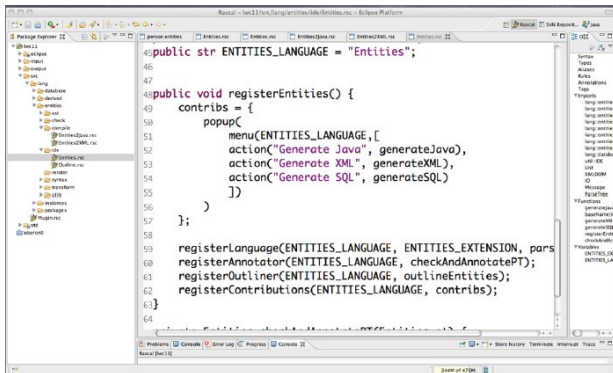
Checking Entities



```
2
3import lang::entities::ast::Entities;
4import Message;
5import Relation;
6
7public list[Message] check(Entities es) {
8    defs = {};
9    errors = for (e <- es.entities) {
10        if (e.name in defs) {
11            append error("Redeclared entity", e.name@location);
12        }
13        defs += {e.name};
14    }
15
16    return ( errors | it + checkEntity(e, defs) | e <- es.entiti
17}
18
19
20public list[Message] checkEntity(Entity e, set[Name] defs) {
21    fs = {};
22    return for (f <- e.fields) {
```



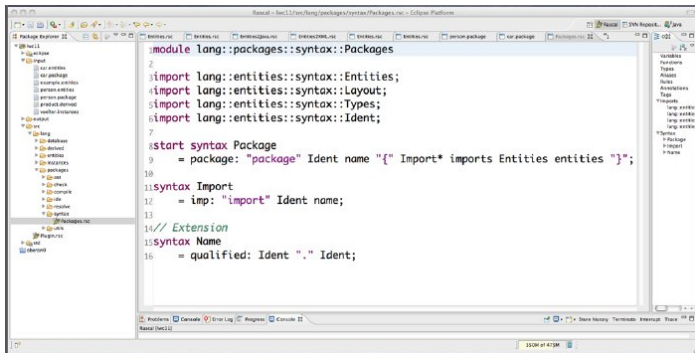
Registering the Contributors



```
45 public str ENTITIES_LANGUAGE = "Entities";
46
47
48 public void registerEntities() {
49     contribs = {
50         popup(
51             menu(ENTITIES_LANGUAGE, [
52                 action("Generate Java", generateJava),
53                 action("Generate XML", generateXML),
54                 action("Generate SQL", generateSQL)
55             ])
56         )
57     };
58
59     registerLanguage(ENTITIES_LANGUAGE, ENTITIES_EXTENSION, pars
60 registerAnnotator(ENTITIES_LANGUAGE, checkAndAnnotatePT);
61 registerOutliner(ENTITIES_LANGUAGE, outlineEntities);
62 registerContributions(ENTITIES_LANGUAGE, contribs);
63 }
64
```



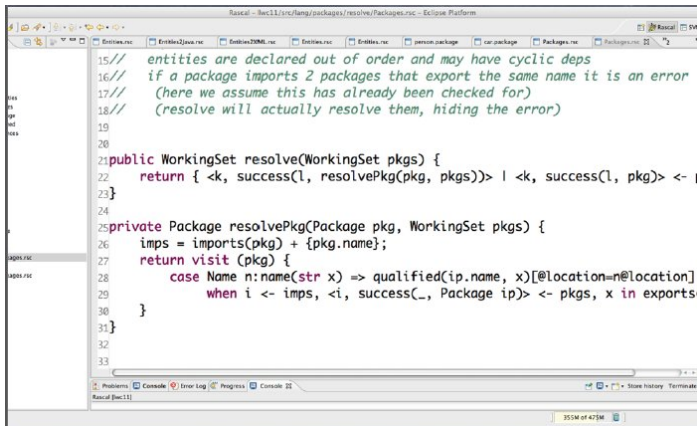
Adding Packages: Package Concrete Syntax



```
1 module lang::packages::syntax::Packages
2
3 import lang::entities::syntax::Entities;
4 import lang::entities::syntax::Layout;
5 import lang::entities::syntax::Types;
6 import lang::entities::syntax::Ident;
7
8 start syntax Package
9   = package: "package" Ident name "{" Import* imports Entities entities "}";
10
11 syntax Import
12   = imp: "import" Ident name;
13
14 // Extension
15 syntax Name
16   = qualified: Ident "." Ident;
```



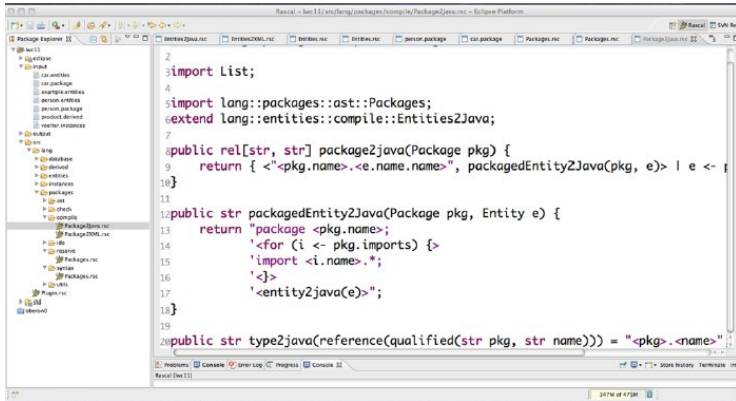
Resolving Names



```
Rascal - /src/lang/packages/resolve/Packages.rsc - Eclipse Platform
Entities.rsc  Entities2Java.rsc  Entities2XML.rsc  Entities.rsc  Entities.rsc  person.package  car.package  Packages.rsc  Packages.rsc
15// entities are declared out of order and may have cyclic deps
16// if a package imports 2 packages that export the same name it is an error
17// (here we assume this has already been checked for)
18// (resolve will actually resolve them, hiding the error)
19
20
21public WorkingSet resolve(WorkingSet pkgs) {
22    return { <k, success(l, resolvePkg(pkg, pkgs))> | <k, success(l, pkg)> <- l
23}
24
25private Package resolvePkg(Package pkg, WorkingSet pkgs) {
26   imps = imports(pkg) + {pkg.name};
27    return visit (pkg) {
28        case Name n:name(str x) => qualified(ip.name, x)[@location=n@location]
29            when i <- imps, <i, success(_, Package ip)> <- pkgs, x in exports
30    }
31}
32
33
Problems Console Error Log Progress Console
Rascal [Rascal]
```



Generating Java Code



```
2
3 import List;
4
5 import lang::packages::ast::Packages;
6 extend lang::entities::compile::Entities2Java;
7
8 public rel[str, str] package2java(Package pkg) {
9     return { <"<pkg.name>.<e.name.name>", packagedEntity2Java(pkg, e) | e <-
10 }
11
12 public str packagedEntity2Java(Package pkg, Entity e) {
13     return "package <pkg.name>;
14         '<for (i <- pkg.imports) {>
15             'import <i.name>.*;
16             '<>
17             'entity2java(e)";
18 }
19
20 public str type2java(reference(qualified(str pkg, str name))) = "<pkg>.<name>"
```



Results

- 4 languages defined in total
- 5 total IDEs (1 for Rascal, plus 4 more)
- 3 checkers defined
- 3 Java code generators created
- 1 SQL code generator created
- 2 XML code generators created
- Total SLOC: 950



DERRIC

- Developed for digital forensics
- Allows specification of file formats using a DSL
- Compiled to optimized Java code
- Total code size: 1871 SLOC
- Highly competitive in both speed and precision
- Developed by a PhD student working at the NFI (Dutch Forensics Institute)



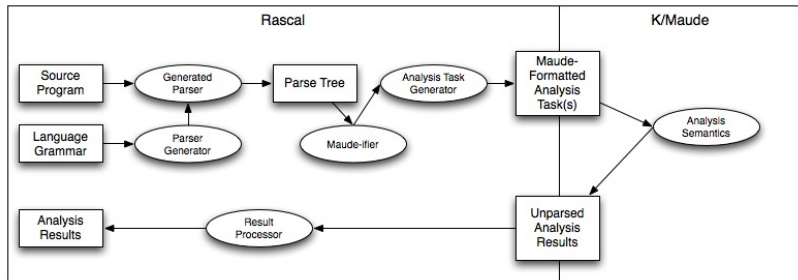
Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications**
- 5 Rascal: Future Development Plans



Integrating Rascal with K Specifications in Maude

- Contributors in Rascal-based IDEs are not limited to those written in Rascal
- Example: linking a Rascal-based front-end with a Maude-based analysis framework



What is Needed to Make the Link (Rascal)?

- Grammar for the language
- *Maudeifier* to generate Maude-readable form of program
- Support for starting, reading from, writing to, and stopping Maude
- Support for preparing individual tasks and reading back results
- Eclipse interaction to display results



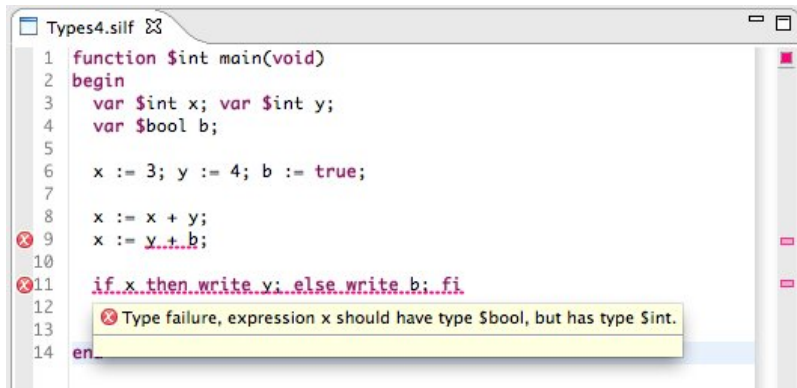
What is Needed to Make the Link (Maude)?

- Specification support for Rascal source locations (if used)
- Result generation in parsable format (not necessarily human readable)



Displaying Analysis Results

Information from the external tool can be used to set up annotations...



The screenshot shows a code editor window titled "Types4.silf". The code is as follows:

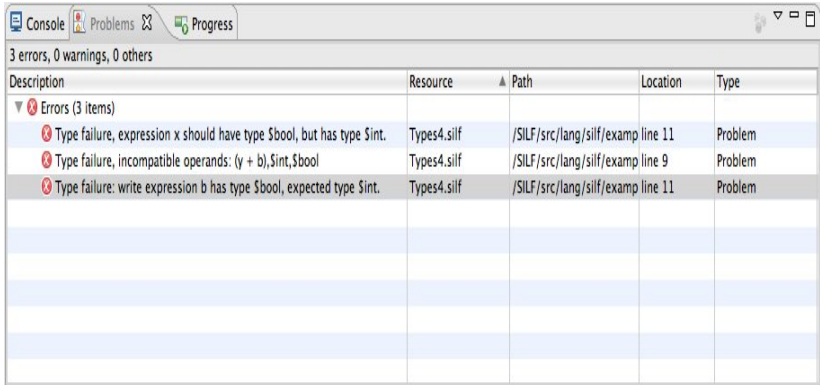
```
1 function $int main(void)
2 begin
3   var $int x; var $int y;
4   var $bool b;
5
6   x := 3; y := 4; b := true;
7
8   x := x + y;
9   x := y.+b;
10
11  if x then write v; else write b; fi
12
13
14 en
```

Red 'X' markers are placed to the left of lines 9 and 11. A yellow tooltip box is positioned over line 11, containing the text: "Type failure, expression x should have type \$bool, but has type \$int."



Displaying Analysis Results (2)

... and to add other information, such as entries in an Eclipse Problems view.



The screenshot shows the Eclipse IDE's Problems view. The title bar includes 'Console', 'Problems', and 'Progress'. Below the title bar, it indicates '3 errors, 0 warnings, 0 others'. The main area is a table with the following columns: Description, Resource, Path, Location, and Type. There are three error entries:

Description	Resource	Path	Location	Type
✘ Errors (3 items)				
✘ Type failure, expression x should have type \$bool, but has type \$int.	Types4.silf	/SILF/src/lang/silf/examp	line 11	Problem
✘ Type failure, incompatible operands: (y + b), \$int, \$bool	Types4.silf	/SILF/src/lang/silf/examp	line 9	Problem
✘ Type failure: write expression b has type \$bool, expected type \$int.	Types4.silf	/SILF/src/lang/silf/examp	line 11	Problem



Outline

- 1 Introduction to Rascal
- 2 Rascal for Language Development
- 3 Developing Modelling Languages
- 4 Tying into Existing Specifications
- 5 Rascal: Future Development Plans



Rascal Development: A Rough Future Timeline

- Syntax: features and documentation finished by the end of September for an early October release
- Performance: ongoing work on improving the performance of program evaluation, with special focus on function call and pattern match performance
- Type Checking: currently uses a runtime type system, switching over to static system – work mostly done, but integrating into Rascal more closely, improving performance so it can run constantly as files are edited



Rascal Development: A Rough Future Timeline

- Syntax: features and documentation finished by the end of September for an early October release
- Performance: ongoing work on improving the performance of program evaluation, with special focus on function call and pattern match performance
- Type Checking: currently uses a runtime type system, switching over to static system – work mostly done, but integrating into Rascal more closely, improving performance so it can run constantly as files are edited



Rascal Development: A Rough Future Timeline

- Syntax: features and documentation finished by the end of September for an early October release
- Performance: ongoing work on improving the performance of program evaluation, with special focus on function call and pattern match performance
- Type Checking: currently uses a runtime type system, switching over to static system – work mostly done, but integrating into Rascal more closely, improving performance so it can run constantly as files are edited



Rascal in Moving to Eclipse.org!



Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications



Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications



Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications



Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications



Wrapping Up

- The Rascal Language
- Rascal for Language Development
- MDE: Entities
- MDE: DERRIC
- Linking to Existing Specifications



For More Information...

- Rascal: <http://www.rascal-mpl.org>
- IMP: <http://www.eclipse.org/imp>
- CWI SEN1: <http://www.cwi.nl/sen1>

