EASY Meta-Programming with Rascal

Leveraging the Extract-Analyze-SYnthesize Paradigm

Tijs van der Storm
storm@cwi.nl / @tvdstorm

CWI
Centrum Wiskunde & Informatica

university of groningen

Joint work with (amongst others):
*Bas Basten, Mark Hills, Anastasia Izmaylova, **Paul Klint**, Davy Landman, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, **Tijs van der Storm**, **Jurgen Vinju**, Vadim Zaytsev*

About me...

- Group leader Software Analysis & Transformation (SWAT) at Centrum Wiskunde & Informatica (CWI)
- Professor of Software Engineering University of Groningen (RUG)
- Research topics: Domain-specific languages, programming languages, language engineering, model-driven engineering
- Co-designer of **Rascal**, a metaprogramming language and language workbench

http://www.cwi.nl/~storm

Some recent papers

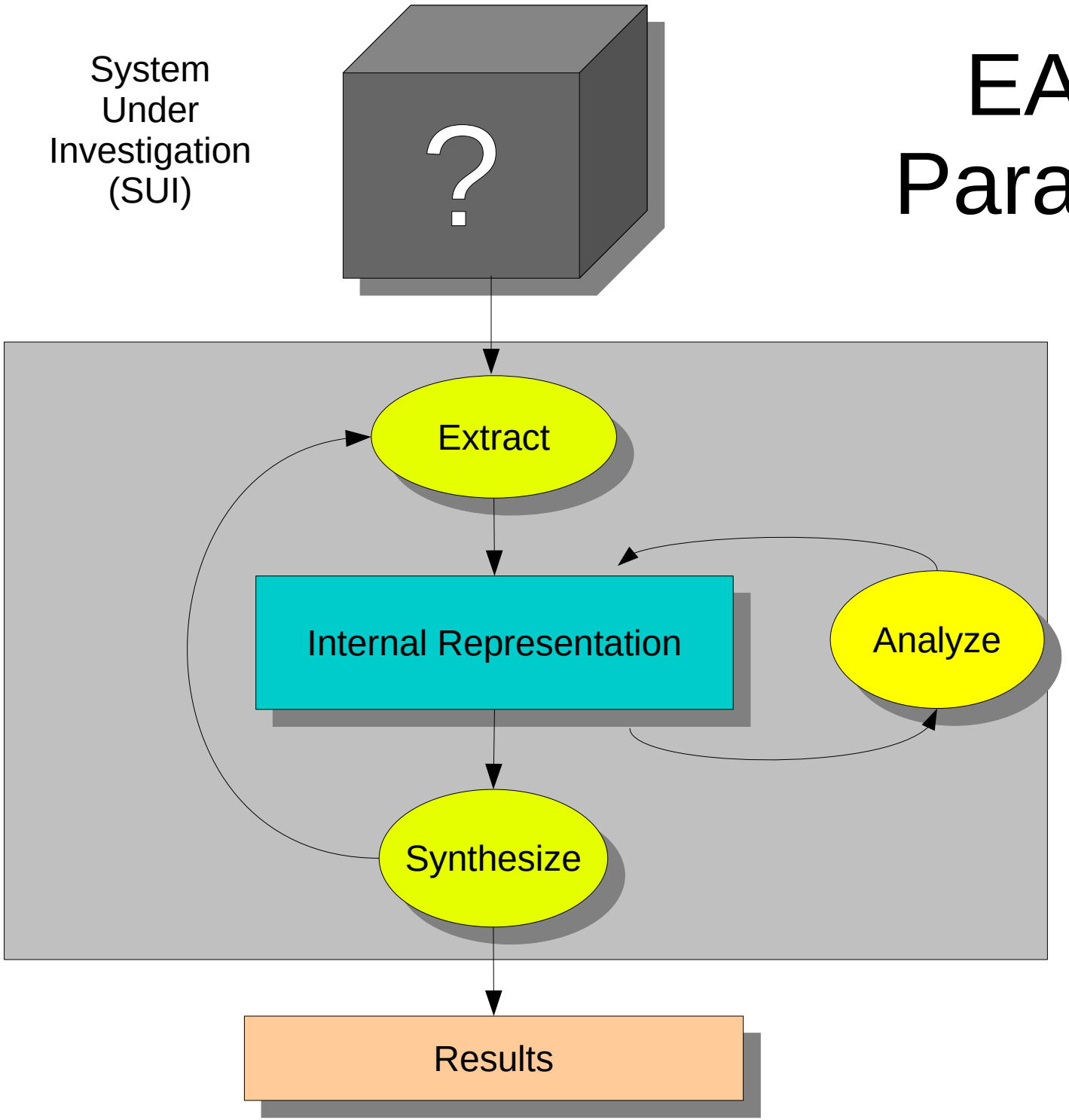
- A principled approach to REPL interpreters
- Block-Based Syntax from Context-Free Grammars
- MATLAB doesn't love me: an essay
- Bacatá: Notebooks for DSLs, Almost for Free
- High-fidelity metaprogramming with separator syntax trees
- Concrete Syntax with Black Box Parsers
- AlleAlle: bounded relational model finding with unbounded data

What are the *Technical* Software Evolution Challenges?

- How to parse source code/data files/models?
- How to extract facts from them?
- How to perform computations on these facts?
- How to generate new source code (trafo, refactor, compile)?
- How to synthesize other information?

EASY: Extract-Analyze-SYnthesize Paradigm

EASY Paradigm



What tools are available?

- Lexical tools: Grep, Awk; also Perl, Python, Ruby
  - Regular expressions have limited expressivity
  - Hard to maintain
- Compiler tools: yacc, bison, CUP, ANTLR
  - Only automate front-end part
  - Everything else programmed in C, Java, ..
- Attribute Grammar tools: FNC2, Eli, JastAdd, …
  - Only analysis, no transformation

What Tools are Available?

- Relational Analysis tools: Grok, Rscript
  - Strong in analysis
- Transformation tools: ASF+SDF, Stratego, TOM, TXL
  - Strong in transformation
- Logic languages: Prolog
- Many others …

	Extract	Analyze	Synthesize
Lexical tools	++	+/-	--
Compiler tools	++	+/-	+/-
Attribute grammar tools	++	+/-	--
Relational tools	--	++	--
Transformation tools	--	+/-	++
Logic languages	+/-	+/-	+/-
Our goal for Rascal	**++**	**++**	**++**

Why a new Language?

- No current technology spans the full range of EASY steps
- There are many fine technologies but they are
  - highly specialized with steep learning curves
  - hard to learn unintegrated technologies
  - not integrated with a standard IDE
  - hard to extend

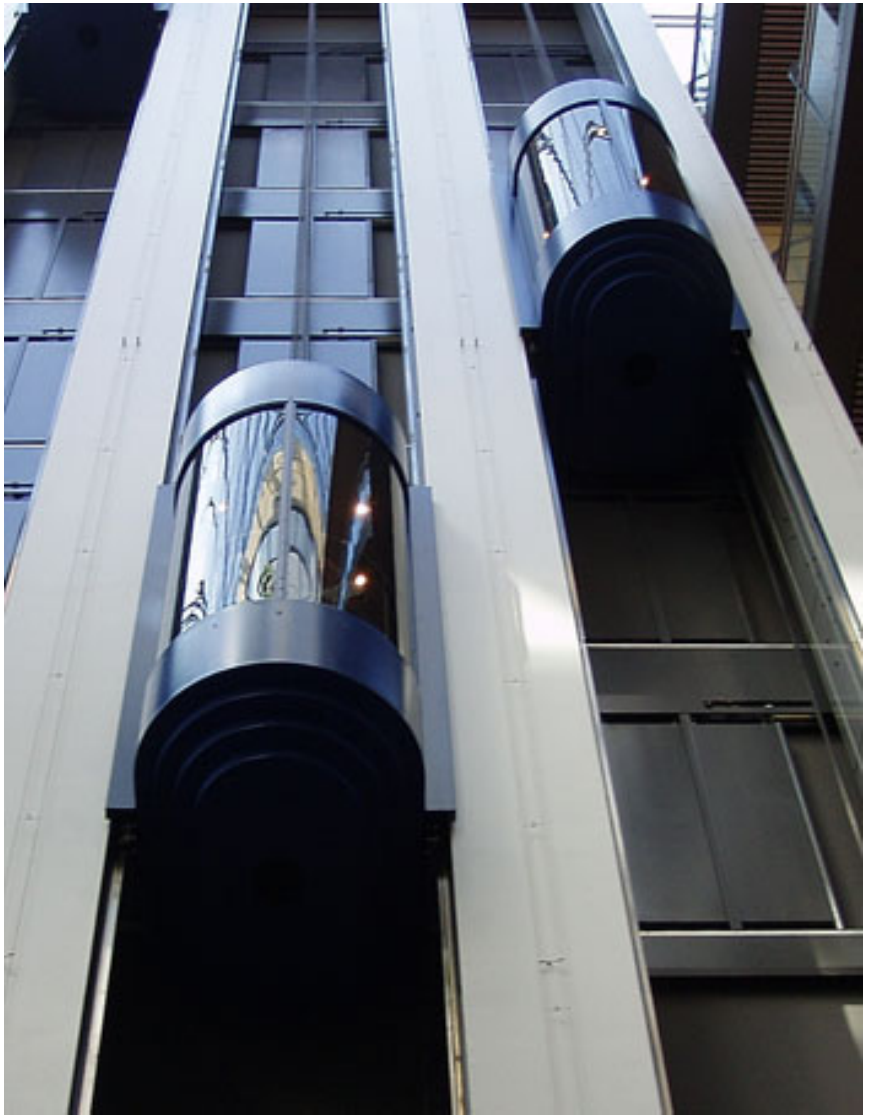
Goal

Keep all benefits of advanced (academic) tools and unify them in a **new**, **extensible**, **teachable** framework

The Swiss Army Knife of Metaprogramming

rASCAL

Rascal Elevator Pitch



Rascal Elevator Pitch

- Sophisticated built-in data types
- Immutable data
- Static safety
- Generic types
- Local type inference
- Pattern Matching
- Syntax definitions and parsing
- Concrete syntax
- Visiting/traversal
- Comprehensions
- Higher-order
- Familiar syntax
- Java and Eclipse integration
- Read-Eval-Print (REPL)

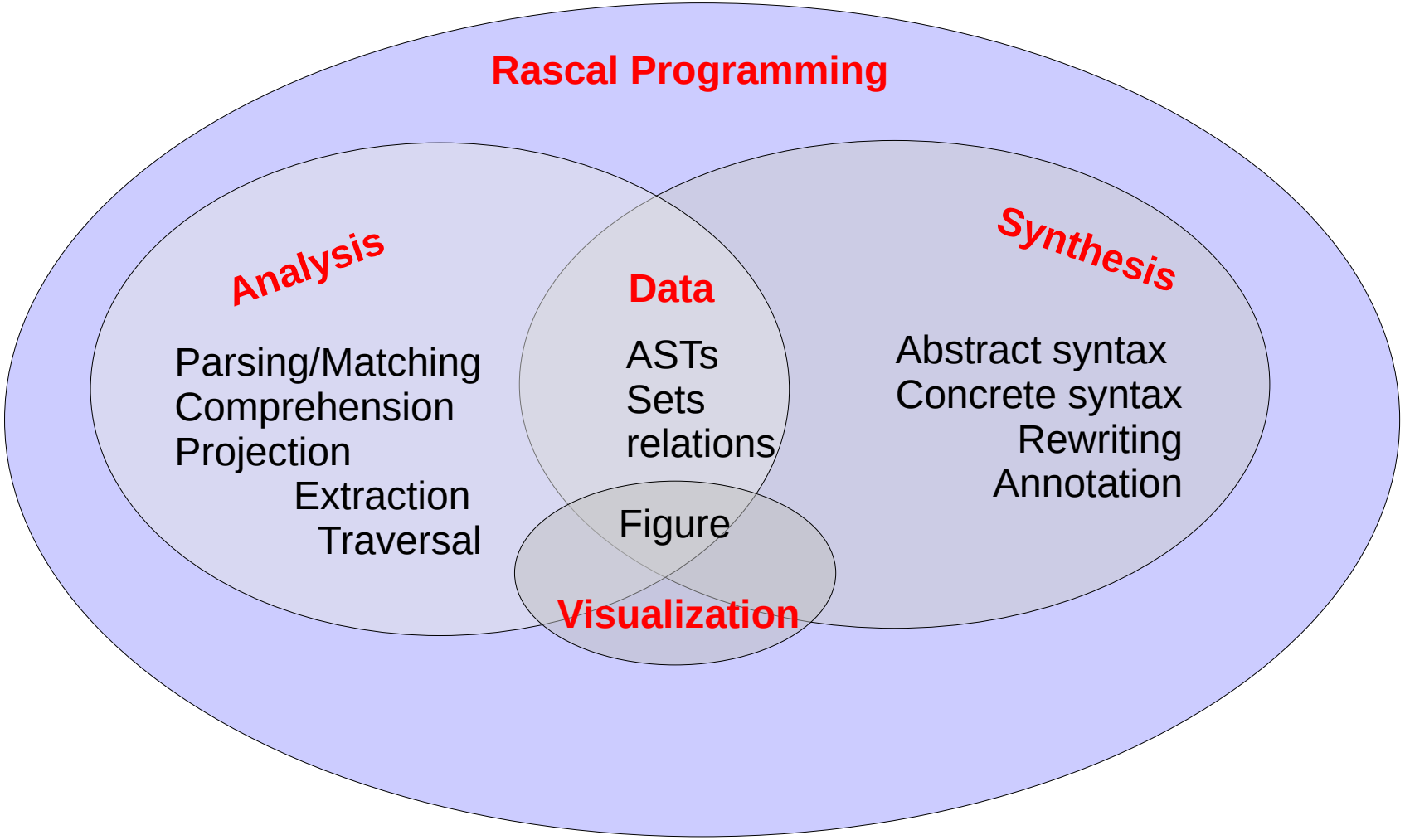
Rascal ...

- is a new language for meta-programming
- is based on *Syntax Analysis*, *Term Rewriting*, *Relational Calculus*
- extended super set (regarding features not syntax!) of ASF+SDF and Rscript
- relations used for sharing and merging of facts for different languages/modules
- embedded in the Eclipse IDE
- easily extensible with Java code

Rascal design based on ...

- **Principle of least surprise**
  - Familiar (Java-like) syntax
  - Imperative core
- **What you see is what you get**
  - No heuristics (or at least as few as possible)
  - *Explicit* preferred over *implicit*
- **Learnability**
  - Layered design
  - Low barrier to entry

Bridging Gaps



One-stop-shop

Some Classical Examples

- Read-Eval-Print
- Hello
- Factorial
- ColoredTrees

Read-Eval-Print

```
rascal>1 + 1
int: 2

rascal>[1,2,3]
list[int]: [1,2,3]

rascal>[1,2,3] + [9,5,1]
list[int]: [1,2,3,9,5,1]
```

List concatenation

Read-Eval-Print

```
rascal>{1,2,3}
set[int]: {1,2,3}

rascal>{1,1,1}
set[int]: {1}

rascal>{1,2,3} + {9,5,1}
set[int]:{1,2,3,9,5}
```

Sets do not contain duplicates

Set union

Read-Eval-Print

Set comprehension

Number range (excludes upper bound!)

```
rascal>{i*i | i←[1..10]}
set[int]: {1,4,9,16,25,36,...}

rascal>{i*i | i←[1..10], i%2==0}
set[int]: {4,16,36,…}
```

List comprehension

```
rascal>[i*i | i←[1..10]]
list[int]: [1,4,9,16,25,36,...]
```

Read-Eval-Print

```
rascal>import IO;
ok
rascal>for (i <- [1..10]) {
>>>>>>>>  println("<i> * <i> = <i * i>");
>>>>>>>>}
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81
list[void]: []
```

String interpolation

Recall: upto (not including) upperbouind 10

Hello (on the command line)

```
rascal > import IO;
ok

rascal> println("Hello, my first Rascal program");
Hello, my first Rascal program
ok
```

Hello (as function in module)

```
module demo::Hello
import IO;
public void hello() {
  println("Hello, my first Rascal program");
}
```

```
rascal > import demo::Hello;
ok
rascal> hello();
Hello, my first Rascal program
ok
```

Factorial

```
module demo::Factorial
public int fac(int N){
  return N <= 0 ? 1 : N * fac(N - 1);
}
```

```
rascal> import demo::Factorial;
ok

rascal> fac(47);
int: 258623241511168180642964355153611979969197632389120000000000
```

Numbers can be arbitrarily large

Types and Values

- Atomic: bool, num, int, real, str, loc (source code location), datetime
- Structured: list, set, map, tuple, rel (n-ary relation), abstract data type, parse tree
- Type system:
  - Types can be parameterized (polymorphism)
  - All function signatures are explicitly typed
  - Inside function bodies types can be inferred (local type inference)

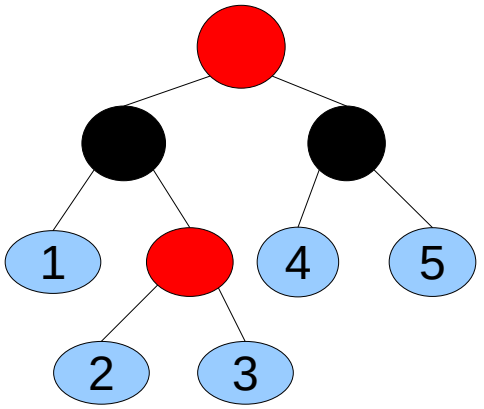
Type	Example
bool	true, false
int, real	1, 0, -1, 123, 1.023e20, -25.5
str	“abc”, “values is <x>”
loc	|file:///etc/passwd|
datetime	$2010-07-15T09:15:23.123+03:00$
tuple[t_1, ..., t_n]	<1,2>, <“john”, 43, true>
list[t]	[], [1], [1,2,3], [true, 2, “abc”]
set[t]	{}, {1,3,5,7}, {“john”, 4.0}
rel[t_1, ..., t_n]	{<1,10,100>,<2,20,200>}
map[t, u]	(), (“a”:1, “b”:2,”c”:3)
node	f, add(x,y), g(“abc”,[2,3,4])

User-defined data types

- Algebraic Data Types: named alternatives
  - name acts as constructor
  - can be used in patterns
- Named fields (access/update via . notation)
- All data types are a subtype of the standard type `node`
  - Permits very generic operations on data
- Parse trees resulting from parsing source code are represented by the datatype `ParseTree`

ColoredTrees: *CTree*

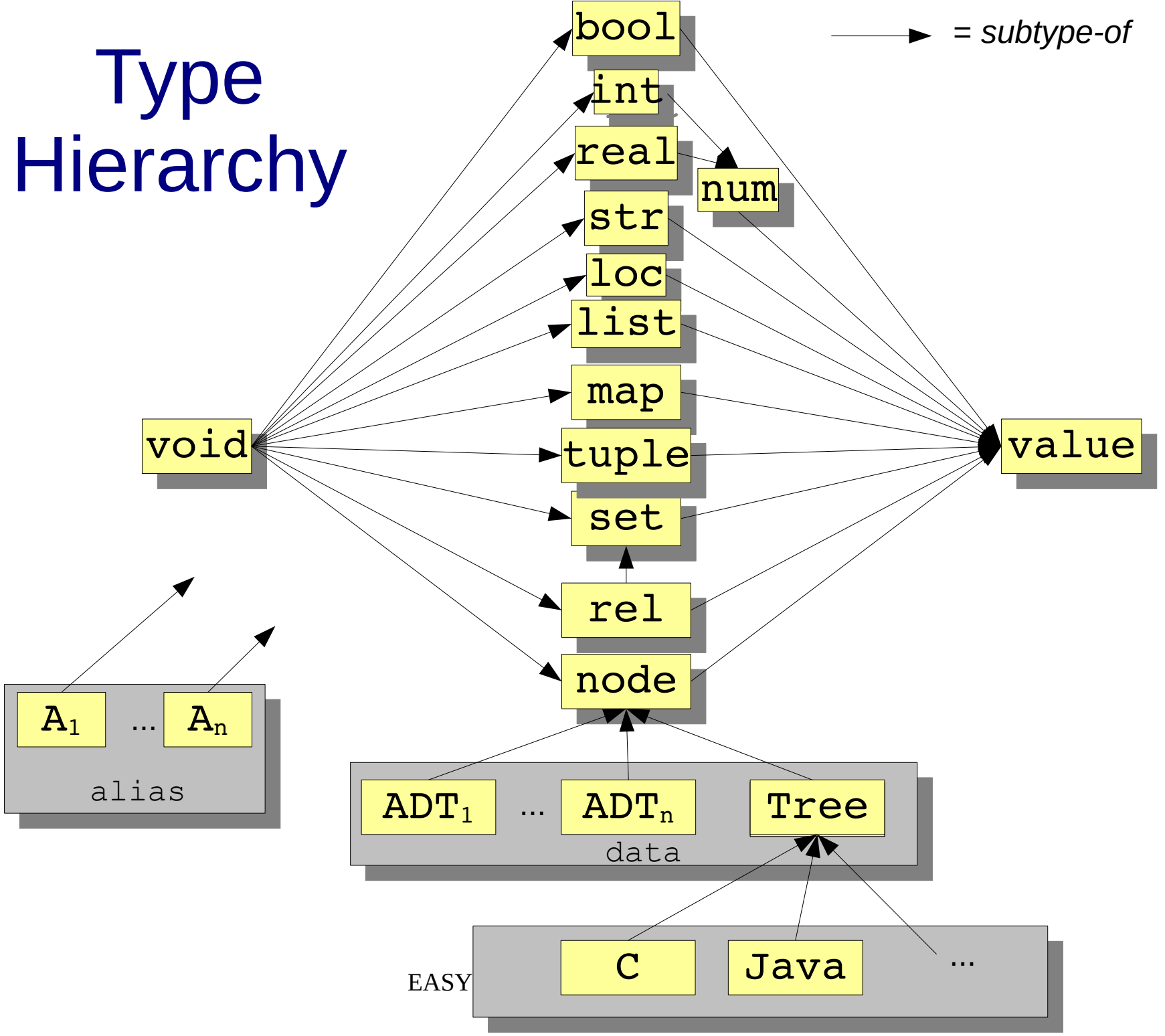
```
data CTree =  leaf(int N)
            | red(CTree left, CTree right)
            | black(Ctree left, Ctree right) ;

rb = red(black(leaf(1), red(leaf(2), leaf(3))),
         black(leaf(4), leaf(5)));
```



Abstract Syntax

```
data STAT = asgStat(Id name, EXP exp)
          | ifStat(EXP exp,list[STAT] thenpart,
                                list[STAT] elsepart)
          | whileStat(EXP exp, list[STAT] body)
          ;
```

Type Hierarchy



Pattern matching

Given a pattern and a value:

- Determine whether the pattern matches the value
- If so, bind any variables occurring in the pattern to corresponding subparts of the value

Pattern matching

Pattern matching is used in:

- Explicit match operator `Pattern := Value`
- Switch: matching controls case selection
- Visit: matching controls visit of tree nodes

Function calls: matching controls dynamic dispatch

Patterns

Regular: Grep/Perl like regular expressions

```
/^<before:\W*><word:\w+><after:.*$>/
```

Abstract: match data types

```
whileStat(Exp, Stats*)
```

Concrete: match parse trees

```
(STAT)`while <Exp e> do <Stat* s> od `
```

Regular Patterns

```
rascal>/[a-z]+/ := "abc"
bool: true
rascal>/rac/ := "abracadabra";
bool: true
rascal>/^rac/ := "abracadabra";
bool: false
rascal>/rac$/ := "abracadabra";
bool: false
```

Regular Patterns

```
rascal>if(/\W<x:[a-z]+>/ := "@abc34") println("x = <x>");
x = abc
ok
```

- Matches non-word character (`\W`) followed by one or more letters
- Binds text matched by `[a-z]+` to variable `x` (is only available in the body of the if statement)
- Prints: `x = abc`
- Regular patterns are tricky (in any language)!

Patterns

Abstract patterns support:

- List matching: [P1, ..., Pn]
- Set matching: {P1, ..., Pn}
- Named subpatterns: N:P
- Anti-patterns: !P
- Descendant: /N

Can be combined/nested in arbitrary ways

List Matching

```
rascal> L = [1, 2, 3, 1, 2];
list[int]: [1,2,3,1,2]

rascal> [X*, 3, X] := L;
bool: true

rascal> X;
Error: X is undefined

rascal> if([X*, 3, X] := L) println("X = <X>");
X = [1, 2]
ok
```

List pattern

X* is a (sub)list variable and of type list[int]

X is bound but has limited scope

List matching provides **associative** (A) matching

Set Matching

```
rascal> S = {1, 2, 3, 4, 5};
set[int]: {1,2,3,4,5}

rascal> {3, Y*} := S;
bool: true

rascal> if({3, Y*} := S) println("Y = <Y>");
Y = {5,4,2,1}
ok
```

Set pattern

Y* is a (sub)set variable of type set[int] Y

Set matching provides **associative**, **commutative**, **identity** (ACI) matching

Note

- List and Set matching are non-unitary
- E.g., [L*, M*] := [1, 2] has three solutions:
  - L == [ ], M == [1,2]
  - L == [1], M == [2]
  - L == [1,2], M == [ ]
- In boolean expressions, matching, etc. solutions are generated when failure occurs later on (local backtracking)

Descendant Matching

```
whileStat(_, /ifStat(_,_,_))
```

Match a while statement
that contains an if statement
at arbitrary depth

Enumerators and Tests

- Enumerate the elements in a value
- Tests determine properties of a value
- Enumerators and tests are used in comprehensions

And in if, for, while, etc.

Enumerators

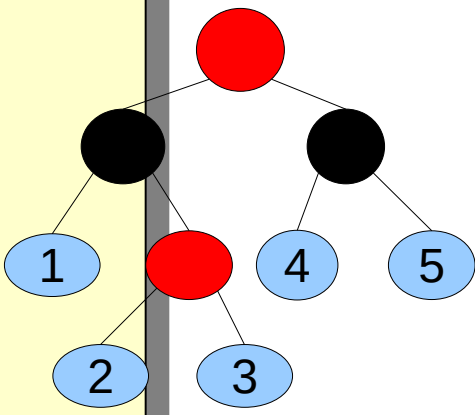
- Elements of a list or set
- The tuples in a relation
- The key/value pairs in a map
- The elements in a datastructure (in various orders!)

```
int x <- { 1, 3, 5, 7, 11 }
int x <- [ 1 .. 10 ]
asgStat(Id name, _) <- P
```

Comprehensions

- Comprehensions for lists, sets and maps
- Enumerators generate values; tests filter them

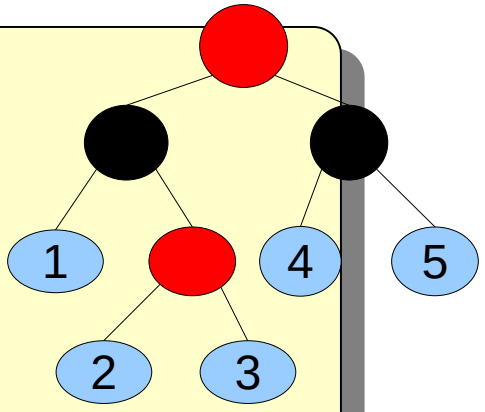
```
rascal> {n * n | int n ← [1 .. 10], n % 3 == 0};
set[int]: {9, 36, 81}

rascal> [ n | /leaf(int n) ← rb ];
list[int]: [1,2,3,4,5]

rascal> {name | /asgStat(id name, _) ← P};
{ ... }
```



Control structures

- Combinations of enumerators and tests drive the control structures
- for, while, all, one

```
rascal> for(/int n ← rb, n > 3){ println(n);}
4
5
ok
rascal> for(/asgStat(Id name, _) ← P, size(name)>10){
    println(name);
}
...
```



Counting words in a string

```
int countWords(str S){
  int count = 0;
  for(/[a-zA-Z0-9]+/ := S){
      count += 1;
  }
  return count;
}
```

Iterates over all matches

countWords("'Twas brillig, and the slithy toves") => 6

Switching

- A switch does a top-level case distinction

```
switch (P){
case whileStat(EXP Exp, Stats*):
    println("A while statement");
case ifStat(Exp, Stats1*, Stat2*):
    println("An if statement");
}
```

A switch is a potential code smell:
you can also use functions dispatched by patterns

Pattern-based Dispatch

- A conventional function has formal parameters:
  - int factorial(int n) { … }
- In Rascal, also patterns can be used as formal parameters.
- At the call site, pattern matching determines which funcion to call => pattern-based dispatch.

Pattern-directed Invocation: 99 bottles of beer

99 Bottles of Beer

99 bottles of beer on the wall, 99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down, pass it around, 97 bottles of beer on the wall.

…
1 bottle of beer on the wall, 1 bottle of beer.
Take one down, pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.

99 Bottles of Beer

```
module demo::basic::Bottles
import IO;

str bottles(0)     = "no more bottles";
str bottles(1)     = "1 bottle";
default str bottles(int n) = "<n> bottles";

void sing(){
  for(n <- [99 .. 0]){
      println("<bottles(n)> of beer on the wall, <bottles(n)> of beer.");
      println("Take one down, pass it around, <bottles(n-1)> of beer on the wall.\n");
  }
  println("No more bottles of beer on the wall, no more bottles of beer.");
  println("Go to the store and buy some more, 99 bottles of beer on the wall.");
}
```

Example

Code generation: Generating getters and setters

Generating Getters and Setters (1)

- Given:
  - A class name
  - A mapping from names to types

Required:

- Generate the named class with getters and setters

Input

```
public map[str, str] fields = (
  "name" : "String",
  "age" : "Integer",
  "address" : "String"
);
```

Field name of type String

Field age of type Integerr

Field address of type String

```
genClass("Person", fields)
```

Generate class person with these fields

Expect Output

```
public class Person {

    private Integer age;
    public void setAge(Integer age) {  this.age = age;   }
    public Integer getAge() { return age;   }


    private String name;
    public void setName(String name) { this.name = name;   }
    public String getName() {  return name;   }


    private String address;
    public void setAddress(String address) {  this.address = address;   }
    public String getAddress() {  return address;   }
}
```

Generating Getters and Setters

```
public str genClass(str name, map[str,str] fields) {
 return "
   public class <name > {
     <for (x <- fields) {
       str t = fields[x];
       str n = capitalize(x);>
       private <t> <x>;
       public void set<n>(<t> <x>) { this.<x> = <x>;  }
       public <t> get<n>() { return <x>;  }
     <}>
   }
";
}
```

String with computed interpolations

Red is interpolated

Blue is literal

Generating Getters and Setters

```
public str genClass(str name, map[str,str] fields) {
 return "
   'public class <name > {
   '  <for (x <- fields) {
   '    str t = fields[x];
   '   str n = capitalize(x);>
   '  private <t> <x>;
   '  public void set<n>(<t> <x>) { this.<x> = <x>;  }
   '  public <t> get<n>() { return <x>;  }
   ' <}>
   '}
";
}
```

String with computed interpolations

Red is interpolated

Blue is literal

Text before ' is ignored

Creating Language Processors and IDEs: Scaling to Java, PHP, ...

Scaling to Java

- We re-use the Java infrastructure of Eclipse
- A general model for Eclipse resources:
  `util::Resources`
- A model for Java entities:
  `lang::java::jdt::m3::Core`
- A model for Java ASTs:
  `lang::java::jdt::m3::AST`

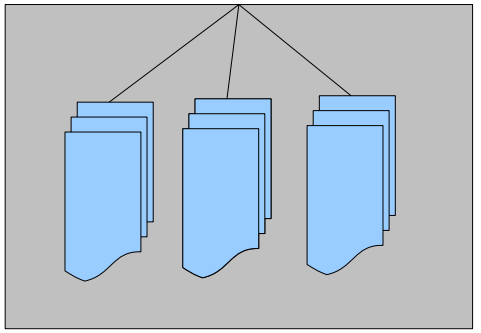
Scaling to Java

packages
classes
implements
extends
methods
methodDecls
constructors
fields
fieldDecls

Access Java facts via access functions

types
variables
modifiers
declaredTopTypes
declaredSubTypes
methodBodies → AST
declaredFields
calls

Resource



Java Project

Eclipse Resource

```
data Resource = root(set[Resource] projects)
              | project(loc id, set[Resource] contents)
              | folder(loc id, set[Resource] contents)
              | file(loc id);
```

```
rascal>loc my_project = |project://org.eclipse.imp.pdb.values|;
loc: |project://org.eclipse.imp.pdb.values|
rascal>import util::Resources;
ok
rascal>getProject(my_project)
Resource: project(
  |project://org.eclipse.imp.pdb.values|,
  {
    ...

    {folder(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts|,
            {file(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts/IBool.java|),
             file(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts/INumber.java|),
             file(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts/ITuple.java|),
             file(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts/IRelationWriter.j
             file(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts/IMapWriter.java|)
             folder(|project://org.eclipse.imp.pdb.values/src/org/eclipse/imp/pdb/facts/util|,
                    { … }
             ...
            }
```

Get the resources of the project

Scaling to Java

- Rascal also provides in `lang::java::jdt::m3::Core` additional data types to represent high-level Java entities (packages, classes, interfaces, methods, …)
- … and in `lang::java::jdt::m3::AST` Java ASTs: `Declaration, Expression, ...` to represent all constructs in Java code.
- … and functions to retrieve this information from a Java project

AST for Java Declaration

```
data Declaration
    = \compilationUnit(list[Declaration] imports, list[Declaration] types)
    | \compilationUnit(Declaration package, list[Declaration] imports, list[Declaration] types)
    | \enum(str name, list[Type] implements, list[Declaration] constants, list[Declaration] body)
    | \enumConstant(str name, list[Expression] arguments, Declaration class)
    | \enumConstant(str name, list[Expression] arguments)
    | \class(str name, list[Type] extends, list[Type] implements, list[Declaration] body)
    | \class(list[Declaration] body)
    | \interface(str name, list[Type] extends, list[Type] implements, list[Declaration] body)
    | \field(Type \type, list[Expression] fragments)
    | \initializer(Statement initializerBody)
    | \method(Type \return, str name, list[Declaration] parameters, list[Expression] exceptions,
Statement impl)
    | \method(Type \return, str name, list[Declaration] parameters, list[Expression] exceptions)
    | \constructor(str name, list[Declaration] parameters, list[Expression] exceptions, Statement impl)
    | \import(str name)
    | \package(str name)
    | \package(Declaration parentPackage, str name)
    | \variables(Type \type, list[Expression] \fragments)
    | \typeParameter(str name, list[Type] extendsList)
    | \annotationType(str name, list[Declaration] body)
    | \annotationTypeMember(Type \type, str name)
    | \annotationTypeMember(Type \type, str name, Expression defaultBlock)
    // initializers missing in parameter, is it needed in vararg?
    | \parameter(Type \type, str name, int extraDimensions)
    | \vararg(Type \type, str name)
    ;
```

AST for Java Expression

```
data Expression
    = \arrayAccess(Expression array, Expression index)
    | \newArray(Type \type, list[Expression] dimensions, Expression init)
    | \newArray(Type \type, list[Expression] dimensions)
    | \arrayInitializer(list[Expression] elements)
    | \assignment(Expression lhs, str operator, Expression rhs)
    | \cast(Type \type, Expression expression)
    | \characterLiteral(str charValue)
    | \newObject(Expression expr, Type \type, list[Expression] args, Declaration class)
    | \newObject(Expression expr, Type \type, list[Expression] args)
    | \newObject(Type \type, list[Expression] args, Declaration class)
    | \newObject(Type \type, list[Expression] args)
    | \qualifiedName(Expression qualifier, Expression expression)
    | \conditional(Expression expression, Expression thenBranch, Expression elseBranch)
    | \fieldAccess(bool isSuper, Expression expression, str name)
    | \fieldAccess(bool isSuper, str name)
    | \instanceof(Expression leftSide, Type rightSide)
    | \methodCall(bool isSuper, str name, list[Expression] arguments)
    | \methodCall(bool isSuper, Expression receiver, str name, list[Expression] arguments)
    | \null()
    | \number(str numberValue)
    | \booleanLiteral(bool boolValue)
    | \stringLiteral(str stringValue)
    | \type(Type \type)
    | \variable(str name, int extraDimensions)
    | \variable(str name, int extraDimensions, Expression \initializer)
    | \bracket(Expression expression)
    | \this()
    | \this(Expression thisExpression)
    | \super()
    | \declarationExpression(Declaration declaration)
    | \infix(Expression lhs, str operator, Expression rhs)
    | \postfix(Expression operand, str operator)
    | \prefix(str operator, Expression operand)
    | \simpleName(str name)
    | \markerAnnotation(str typeName)
    | \normalAnnotation(str typeName, list[Expression] memberValuePairs)
    | \memberValuePair(str name, Expression \value)
    | \singleMemberAnnotation(str typeName, Expression \value)
```

AST for Java Statement

```
data Statement
    = \assert(Expression expression)
    | \assert(Expression expression, Expression message)
    | \block(list[Statement] statements)
    | \break()
    | \break(str label)
    | \continue()
    | \continue(str label)
    | \do(Statement body, Expression condition)
    | \empty()
    | \foreach(Declaration parameter, Expression collection, Statement body)
    | \for(list[Expression] initializers, Expression condition, list[Expression] updaters, Statement body)
    | \for(list[Expression] initializers, list[Expression] updaters, Statement body)
    | \if(Expression condition, Statement thenBranch)
    | \if(Expression condition, Statement thenBranch, Statement elseBranch)
    | \label(str name, Statement body)
    | \return(Expression expression)
    | \return()
    | \switch(Expression expression, list[Statement] statements)
    | \case(Expression expression)
    | \defaultCase()
    | \synchronizedStatement(Expression lock, Statement body)
    | \throw(Expression expression)
    | \try(Statement body, list[Statement] catchClauses)
    | \try(Statement body, list[Statement] catchClauses, Statement \finally)
    | \catch(Declaration exception, Statement body)
    | \declarationStatement(Declaration declaration)
    | \while(Expression condition, Statement body)
    | \expressionStatement(Expression stmt)
    | \constructorCall(bool isSuper, Expression expr, list[Expression] arguments)
    | \constructorCall(bool isSuper, list[Expression] arguments)
    ;
```

Extract All Classes

```
rascal>import lang::java::jdt::m3::Core;
ok
rascal>mmm = createM3FromEclipseProject(my_project);
M3: m3(
  |project://org.eclipse.imp.pdb.values|,
  annotations={
    <|java+method:///org/eclipse/imp/pdb/facts/io/binary/BinaryWriter/IdentityValue/
accept(org.eclipse.imp.pdb.facts.visitors.IValueVisitor)|,|java+interface:///java/lang/
Override|>,
    <|java+method:///org/eclipse/imp/pdb/facts/impl/primitive/BigIntegerValue/
divide(org.eclipse.imp.pdb.facts.IRational)|,|java+interface:///java/lang/Override|>,

<|java+method:///org/eclipse/imp/pdb/facts/type/ExternalType/lubWithExternal(org.eclipse.
imp.pdb.facts.type.Type)|,|java+interface:///java/lang/Override|>,
    <|java+method:///org/eclipse/imp/pdb/test/random/RandomRationalGenerator/next()|,|
java+interface:///java/lang/Override|>,
    <|java+method:///org/eclipse/imp/pdb/facts/impl/primitive/RationalValue/
less(org.eclipse.imp.pdb.facts.IReal)|,|java+interface:///java/lang/Override|>,

<|java+method:///org/eclipse/imp/pdb/facts/type/AliasType/lubWithConstructor(org.eclipse.
imp.pdb.facts.type.Type)|,|java+interface:///java/lang/Override|>,
...
```

mmm gets M3 model as value

Extract All Classes

Retrieve the classes

```
package org.eclipse.imp.pdb.facts.exceptions;

public class NullTypeException extends FactTypeUseException {
     private static final long serialVersionUID = -4201840676263159311L;

     public NullTypeException() {
          super("A null reference as a type");
     }
}
```

```
rascal>classes(mmm);

|java+class:///org/eclipse/imp/pdb/facts/exceptions/NullTypeException|,
|java+class:///org/eclipse/imp/pdb/test/persistent/TestList|,
|java+class:///org/eclipse/imp/pdb/facts/type/IntegerType|,
|java+class:///org/eclipse/imp/pdb/facts/type/SourceLocationType|,
|java+class:///org/eclipse/imp/pdb/facts/util/Set4|,
|java+class:///org/eclipse/imp/pdb/test/random/DataIterable|,
|java+class:///org/eclipse/imp/pdb/test/persistent/TestAnnotations|,
|java+class:///org/eclipse/imp/pdb/facts/type/StringType|,
|java+class:///org/eclipse/imp/pdb/test/reference/TestRandomValues|,
|java+class:///org/eclipse/imp/pdb/test/random/RandomNumberGenerator|,
...
```

Printing Methods per Class

```
module CountMethods
import Prelude;
import lang::java::jdt::m3::Core;

loc project = |project://org.eclipse.imp.pdb.values|;

void countMethods(){
  M3 mmm = createM3FromEclipseProject(project);

  my_classes = {e | <c, e> <- declaredTopTypes(mmm), isClass(e)};
  my_methods = (e : size(declaredMethods(mmm)[e]) | e <- my_classes);
  for(m <- my_methods)
      println("<m> : <my_methods[m]>");
}
```

Printing Methods per Class

```
rascal>import CountMethods;
ok

rascal>countMethods();
|java+class:///org/eclipse/imp/pdb/facts/exceptions/NullTypeException| : 1
|java+class:///org/eclipse/imp/pdb/test/persistent/TestList| : 1
|java+class:///org/eclipse/imp/pdb/facts/type/IntegerType| : 14
|java+class:///org/eclipse/imp/pdb/facts/type/SourceLocationType| : 12
|java+class:///org/eclipse/imp/pdb/facts/util/Set4| : 14
|java+class:///org/eclipse/imp/pdb/test/random/DataIterable| : 2
|java+class:///org/eclipse/imp/pdb/test/persistent/TestAnnotations| : 1
|java+class:///org/eclipse/imp/pdb/facts/type/StringType| : 12
|java+class:///org/eclipse/imp/pdb/test/reference/TestRandomValues| : 1
|java+class:///org/eclipse/imp/pdb/test/random/RandomNumberGenerator| : 2
|java+class:///org/eclipse/imp/pdb/facts/exceptions/UnexpectedMapValueTypeException| : 1
|java+class:///org/eclipse/imp/pdb/facts/exceptions/IllegalAnnotationDeclaration| : 2
|java+class:///org/eclipse/imp/pdb/facts/util/Set5| : 14
|java+class:///org/eclipse/imp/pdb/facts/impl/AbstractValueFactoryAdapter| : 69
...
```

Enough

- Ok, that was quite a lot of information
- Rascal is for Meta-programming
  - Code analysis
  - Code transformation
  - Code generation
  - Code visualization
- It is a normal programming language
- Learn it using the Tutor view and the console

Debugging your code

- Classical and effective:
  - IO::println, Util::ValueUI:text, tree
- Interactive debugger in Eclipse
- Documentation:
  - http://tutor.rascal-mpl.org
  - Also in Eclipse tutor view (more recent)
- Online coding questions:
  - ===> stackoverflow.com [rascal] tag

Internships at CWI and SWAT.engineering

- Where: at Centrum Wiskunde & Informatica
- Challenges:
  - Live programming, DSLs, Pattern recognition, IDEs for grammars, type systems, compiler construction
- Profile: analytical, programmer, creative, ambitious, but otherwise no restrictions
- Support: we have supervised hundreds of MSE thesis projects