Introduction to Software Language Engineering using Rascal

Tijs van der Storm
(with help of Jouke Stoel)

university of groningen

CWI

Rascal = Functional Metaprogramming language

Rascal = Functional Metaprogramming language

- Immutable variables
- Higher order functions
- Static safety, with local type inference

Rascal = Functional Metaprogramming

???

Metaprogramming

From Wikipedia, the free encyclopedia

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs […]

Rascal = Functional Metaprogramming

- “Code as data”
- Program that generates/analyzes other programs
- Syntax definitions and parsing
- Pattern matching and rewriting mechanisms
- Visiting / traversal of Tree structure

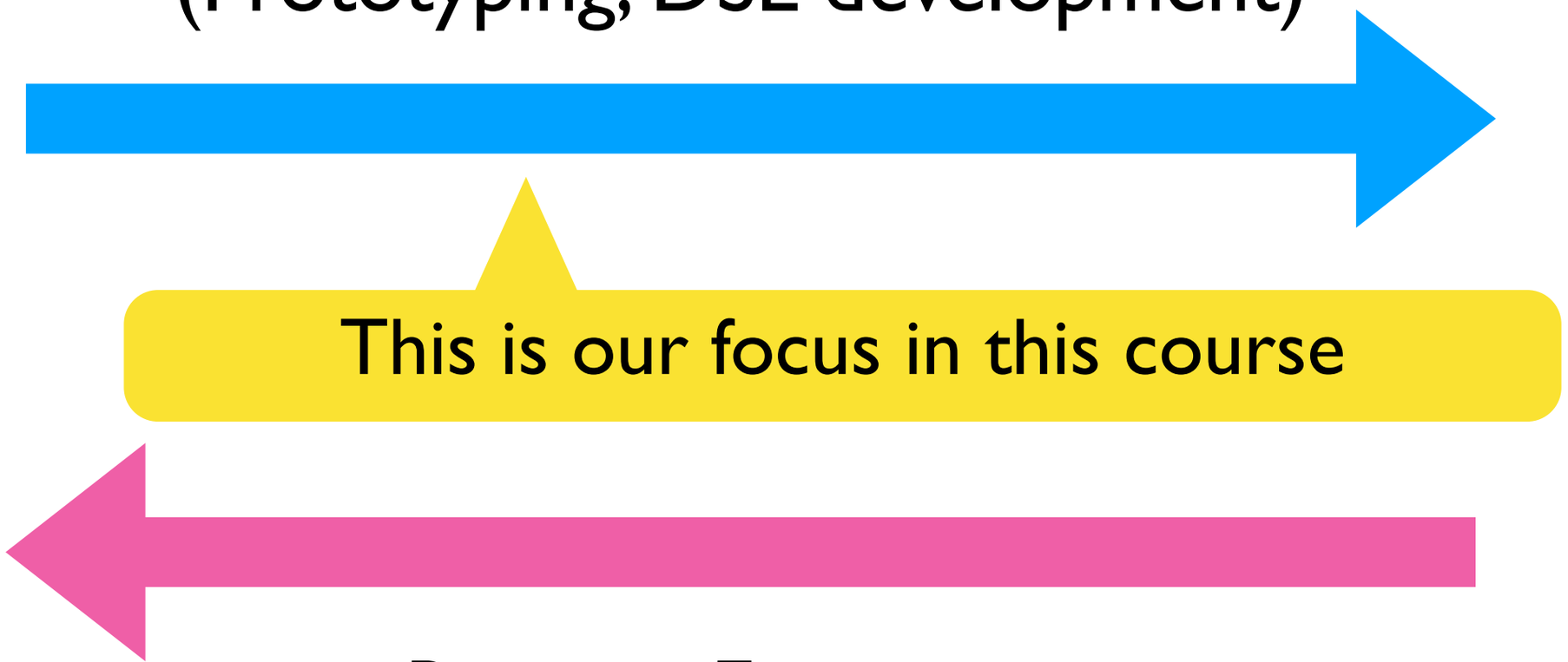
Rascal can be used for…

Forward Engineering
(Prototyping, DSL development)

Reverse Engineering
(Analysis, detectors, renovations)

Rascal can be used for…

Forward Engineering
(Prototyping, DSL development)



Reverse Engineering
(Analysis, detectors, renovations)

Why learn *yet* another new language?


Java
Grep
ANTLR
awk

Basic concepts

Functional immutability with an imperative syntax

- All data is immutable

- Can write code that looks like ‘mutating variables’

Functional immutability with an imperative syntax

- **All data is immutable**

- Can write code that looks like ‘mutating variables’

Common Data Types

- e.g. Sets, Lists, Maps, Tuples and Relations
- Can all be used in comprehensions

Sets

- Unordered collection of values
- Elements are all of the same static type
- All elements are distinct
- Allows all sorts of powerful operations like comprehensions, difference, slicing, etc..
- `import Set;` for convenient functions on sets
- See: http://docs.rascal-mpl.org/unstable/Rascal/#Values-Set

Lists

- Ordered sequence of values
- Elements are all of the same static type
- Allows for duplicate entries
- Allows all sorts of powerful operations like comprehensions, difference, slicing, etc..
- `import List;` for convenient functions on sets
- See: http://docs.rascal-mpl.org/unstable/Rascal/#Values-List

Tuples

- Ordered sequence of elements
- Tuples are fixed sized
- Elements may be of different types
- Each element can have a label
- See: http://docs.rascal-mpl.org/unstable/Rascal/#Values-Tuple

Relations

- All elements have the same static tuple type
- Set of Tuples
- Next to the set operations allows for composition, joining, transitive closure, etc
- `import Relation;` for convenient functions on relations
- See: http://docs.rascal-mpl.org/unstable/Rascal/#Values-Relation

Source Locations

- Provide a uniform way to represent files on local or remote storage

- Can have different schemes
  - `file:///`
  - `project://`
  - `http://`
  - etc …

- Can contain text location markers

- See: http://docs.rascal-mpl.org/unstable/Rascal/#Values-Location

String templates

- Easy way to “generate” strings

- Often used for source-to-source transformation

Pattern Matching

- Determines whether pattern matches a given value
- One of Rascal's most powerful features
- Can bind matches to local variables
- Can be used in many places
- May result in multiple matches so employs local backtracking
- See http://docs.rascal-mpl.org/unstable/RascalConcepts/#RascalConcepts-PatternMatching

Different types of matching

type-based matching

```
int x := 3;
```

structural matching

```
event(x, y) := event("a", "b");
```

anti-matching

```
event("c", "d") !:= event("a", "b");
```

list matching

```
[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];
```

set matching

```
{1, *x} := {4, 5, 6, 1, 2, 3};
```

deep matching

```
/transition(e, "idle") := ast;
/state(x, _, /transition(_, x)) := ast;
```

element matching

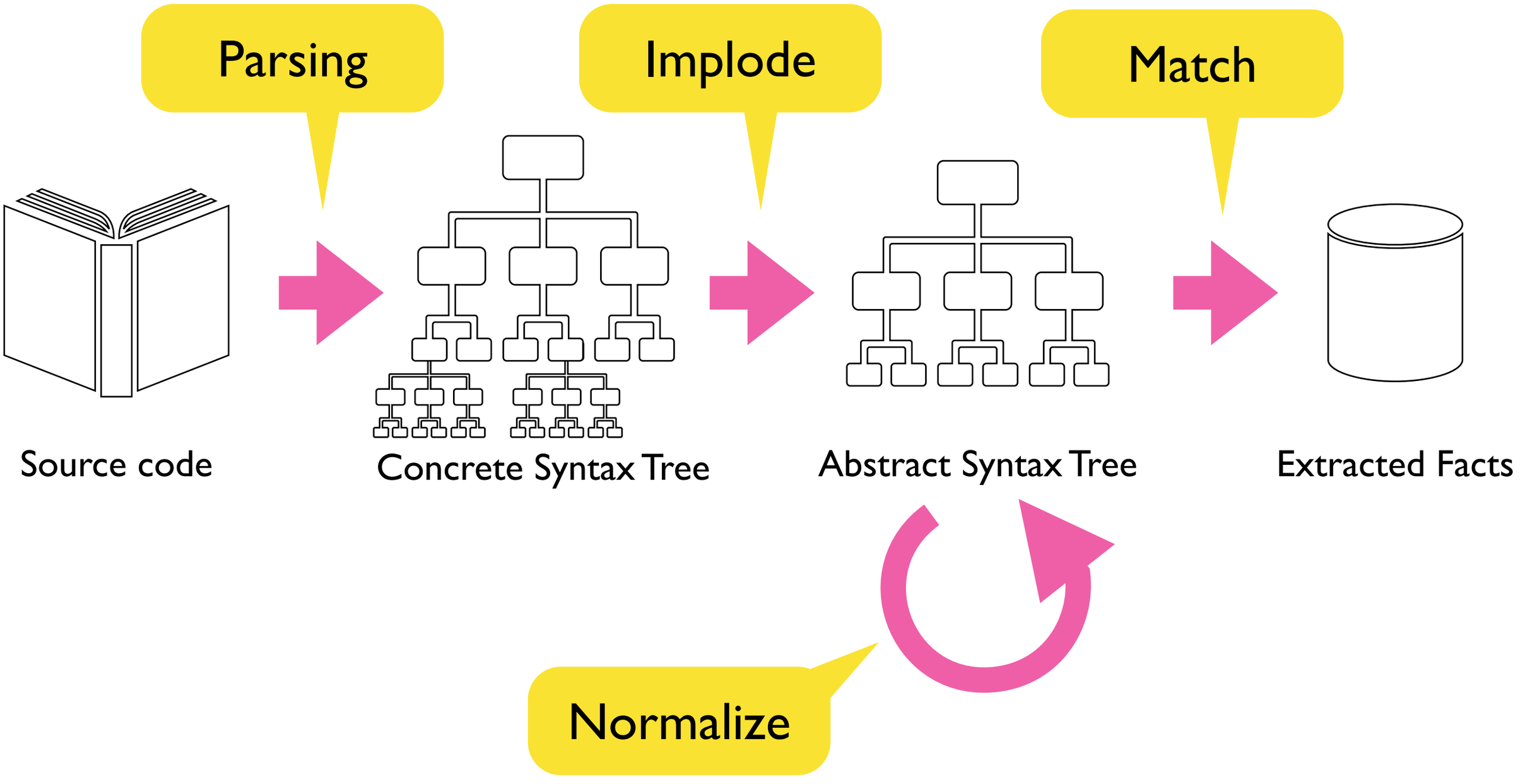
```
3 ← {1,2,3}
int x ← {1,2,3}
```

regular expressions

```
/[A-Za-z]*/ := “09090aap noot mies”
```

Tools for Language Engineering

Extracting facts using parsing



Describes all possible strings which can be produced

```
start syntax Program = "begin" Stat* "end";

syntax Statement
  = "i                  " Stat* "else" Stat* "fi"
  | Id
  | "while" Expression "do" Stat* "od"
  ;

syntax Expression
  = Id
  | "(" Expression ")"
  | left Expression "*" Expression
                    on "+" Expression

lex                                  -9\-]*;

layout Whitespace = [\ \t\n\r]*;
```

Nonterminal

Terminal

Layout characters

Abstract Syntax (– ADT)

Same information as concrete tree but more abstract

```
data Program = program(list[Stat] stats);

data Stat
  = assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
  ;

data Expr
  = id(str name)
  | mult(Expr lhs, Expr rhs)
  | add(Expr lhs, Expr rhs)
  ;
```

Abstract Data Type

Escape for keywords

Extracting information from an Abstract Syntax Tree

- Use Pattern Matching
  - Match on structure
  - Match on values
  - Deep matching
- See http://docs.rascal-mpl.org/unstable/RascalConcepts/#RascalConcepts-PatternMatching

Find all assigned variables

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
```

Iterate over all Stats in subtree

Only match on ‘assign’

```
// find assigned identifiers
for (Stat s ← program.stats, assign(str id, Expr _) := s) {
  println("Found id: <id>");
}
```

Wildcard

Find all assigned variables

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
```

Direct iterate and match on ‘assign’

```
// find assigned identifiers
for (assign(str id, Expr _) ← program.stats) {
  println("Found id: <id>");
}
```

Find all *nested* assigned variables

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
```

What if an assignment is nested?

Find all *nested* assigned variables

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
```

Use a deep match

```
for (/assign(str id, Expr _) ← program) {
  println("Found id: <id>");
}
```

Find the variable named “x”

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
```

Match on value

```
for (/assign("x", Expr expr) ← program) {
  println(“Expr assigned to x: <expr>");
}
```

Transforming an Abstract Syntax Tree

- Use visit-statement
  - reach all nodes
  - not composible
- See http://docs.rascal-mpl.org/unstable/Rascal/#Expressions-Visit

Rename “x” to “y”

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
  ;

data Expr
  = id(str name)
```

Rename “x” to “y”

```
data Program = program(list[Decl] decls, list[Stat] stats);

data Stat
  = \if(Expr cond, list[Stat] \tr, list[Stat] \f)
  | assign(str id, Expr val)
  | \while(Expr cons, list[Stat] body)
  ;

data Expr
  = id(str name)
```

Visit all the nodes in the tree

Rewrite / replace node

```
  p = visit(p) {
    case assign("x", Expr val) ⇒ assign("y",val)
    case id("x") ⇒ id("y")
  }
```

Both assignment and use are replaced

Visit strategies

- **`top-down`**: root to leafs (default)
- **`top-down-break`**: root to leafs but stop on case match
- **`bottom-up`**: leafs to root
- **`bottom-up-break`**: leafs to root but stop on case match
- **`innermost`**: bottom-up fix point (repeat until no more changes)
- **`outermost`**: top-down fix point

Tips and Tricks

Debugging Rascal

- Poor man's debugging
  - Using `println` or `bprintln` (in comprehensions)
    - You need to import IO for this!
- "Rich man's" debugging
  - Using the debugger
    - You have to open the debug perspective manually!

Common errors
Undeclared variable

- Forgetting to import a module, i.e.:

- Function is declared private

```
rascal>l = [1,2,3];
list[int]: [1,2,3]
rascal>size(l);
|prompt:///|(0,4,<1,0>,<1,4>): Undeclared variable: size
Advice: |http://tutor.rascal-mpl.org/Errors/Static/UndeclVariable.html|
```

- Solution for above example: `import List;`

Common errors
CallFailed

- Calling a function with the wrong arguments

```
void someFunc(str a) {
  println(a);
}
```

```
rascal>someFunc("a");
a
ok
rascal>someFunc(2);
|prompt:///|(9,1,<1,9>,<1,10>): CallFailed(
  |prompt:///|(9,1,<1,9>,<1,10>),
  [2])
        at $root$(|prompt:///|(0,12,<1,0>,<1,12>))
```

Common Error: Root cause analysis

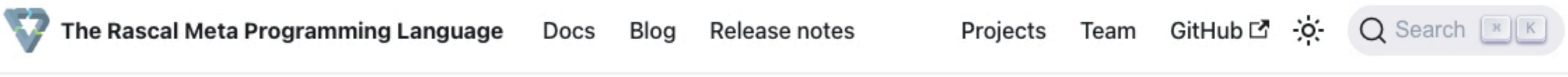
- Slice your problem by
- Importing the problematic module directly
- Use ‘delta’-debugging (comment out 50% of the code and try to reimport, add 25% again and try again, etc)

Looking for how you can do stuff in Rascal?

Looking for how you can do stuff in Rascal?

Browse and search the documentation

- https://www.rascal-mpl.org/docs/GettingStarted/



Looking for how you can do stuff in Rascal?

Take a look at the Rascal Cheatsheet:

Rascal Cheat Sheet

http://www.rascal-mpl.org
http://tutor.rascal-mpl.org
https://github.com/cwi-swat/rascal

Modules

```
module Example

import ParseTree;         // import
extend lang::std::Layout; // "inherit"
```

Declarations

```
// Algebraic data types (ADT)
data Exp
  = var(str x)          // unary constructor
  | add(Exp l, Exp r); // binary constructor

data Person        // keyword parameter
  = person(int id, bool married=false);

alias Age = int; // type alias

anno loc Exp@location; // annotation

private real PI = 3.14; // variables

// Functions:  signatures are lists of patterns
// May have keyword parameters.
void f(int x) { println(x); }      // block style
int inc(int x) = x+1;              // rewrite style
int inc0(int x) = x+1 when x == 0; // side condition
default int inc0(int x) = x;       // otherwise

// Test functions (invoke from console with :test)
test bool add() = 1+2 == 3;

// randomized test function
test bool comm(int x, int y) = x+y == y+x;

// Foreign function interface to Java
@javaClass{name.of.javaClass.with.Method}
java int method();
```

```
// Context-free grammars
start syntax Prog       // start symbol
  = prog:  Exp* exps    // production
  | stats:  {Stat ";"}* // separated list
  | stats:  {Stat ";"}+ // one-or-more sep.  list
  | "private"?  Func;   // optional

syntax Exp
  = var:  Id
  | left mul:  Exp l "*" Exp r     // or right, assoc
  | left div:  Exp!div "/" Exp!div // reject
  > left add:  Exp l "+" Exp r     // ">" = priority
  | bracket "(" Exp ")";

lexical Comment
  = ^"#" ![\n]* $; // begin/end markers

lexical Id
  = ([a-zA-Z] !<<         // look behind restriction
    [a-zA-Z][a-zA-Z0-9_]* // character classes
    !>> [a-zA-Z0-9_])     // lookahead restriction
    \ Reserved;           // subtract keywords

layout Layout // for whitespace/comments
  = [\ \t\n\r]*;

keyword Reserved   // keyword class
  = "if" | "else"; // finite langs
```

Statements

```
// Standard control-flow
if (E) S;
if (E) S; else S;
while (E) S;
do S; while(E);
continue; break;
return; return E;

// Loop over all bindings produce by patterns
for (i <- [0..10]) S; // Loop 10 times

fail;     // control backtracking
append E; // add to loop result list
```

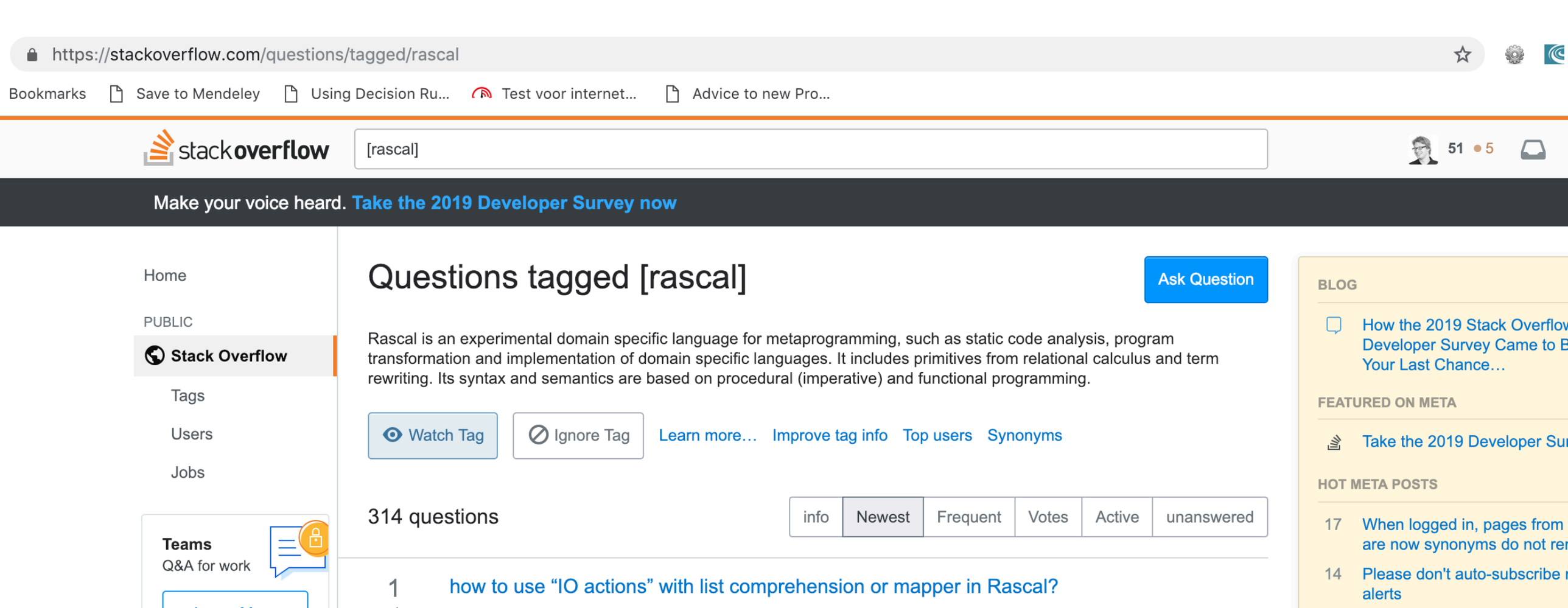
```
// Pattern-based switch-case
switch (E) {
  case P: S;   // do something
  case P => E  // rewrite it
  default:  S; // otherwise
}

// Traversal with visit; like switch, but matches
// at arbitrary depth of value
visit (E) {
  case P: S;  // do something
  case P => E // rewrite something
  case P => E when E
}
insert E; // rewrite subject as statement

// Strategies:  bottom-up, innermost, outermost,
// top-down-break, bottom-up-break
top-down visit (E) {}

try S;      // pattern-based try-catch
catch P: S; // match to catch
finally S;

throw E; // throw values

// Fix-point equation solving;
// iterates until all params are stable
solve (out,ins) {
  out[b] = ( {} | it + ins[s] | s <- succ[b] );
  ins[b] = (out[b] - kill[b]) + gen[b];
};

x = 1;               // assignment
nums[0] = 1;         // subscript assignment
nums[1,3..10] = 2;   // sliced (see below)
p.age = 31;          // field assignment
ast@location = l;    // annotation update
<p, a> = <"ed", 30>; // destructuring

// A op=E  == A = A op E
A += E; A -= E; A *= E;
A /= E; A &= E;
```

https://github.com/cwi-swat/rascal-cheat-sheet/raw/master/sheet.pdf

Looking for how you can do stuff in Rascal?

Search on StackOverflow

https://stackoverflow.com/questions/tagged/rascal



Good luck!

- Next up: get up to speed in Rascal
- https://github.com/cwi-swat/rascal-wax-on-wax-off