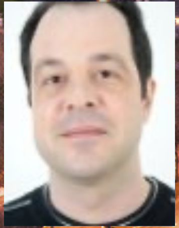
Domain Specific Language Design (2IMP20)

Loek Cleophas



Ivan Kurtev

Department of Mathematics and Computer Science — Software Engineering and Technology cluster

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Introduction to Rascal

Rascal =

Functional
Metaprogramming
Language

Introduction to Rascal

Rascal =

Functional
Metaprogramming
Language

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

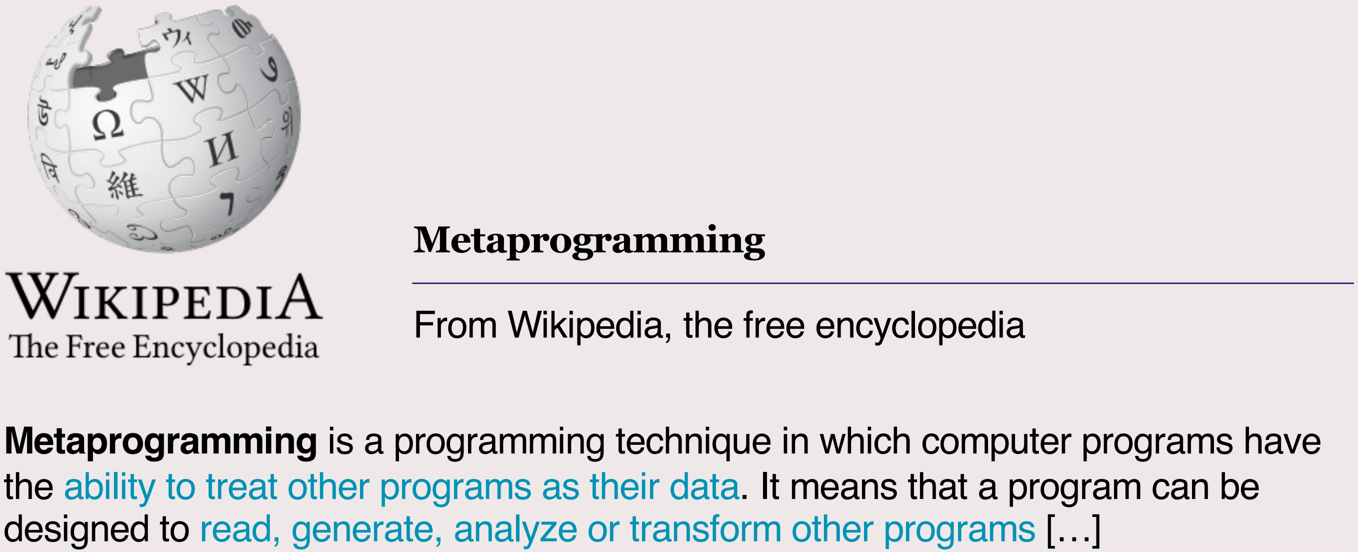
Introduction to Rascal

Rascal =

Functional
Metaprogramming
Language

???

Introduction to Rascal



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 […]

Introduction to Rascal

Rascal =

Functional

Metaprogramming

Language

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

Introduction to Rascal

Rascal can be used for…

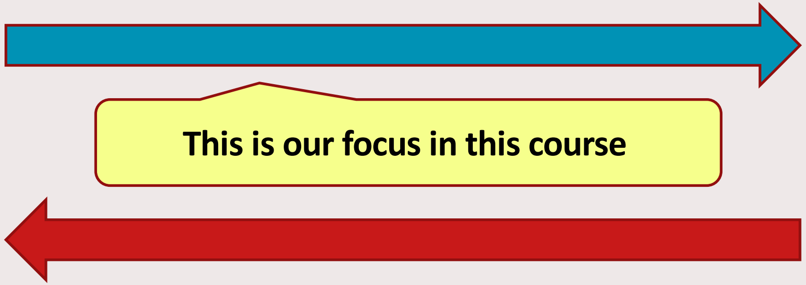
Forward Engineering
(Prototyping, DSL development)

Backward Engineering
(Analysis, detectors, renovations)

Introduction to Rascal

Rascal can be used for…

Forward Engineering
(Prototyping, DSL development)



This is our focus in this course

Backward Engineering
(Analysis, detectors, renovations)

Introduction to Rascal

1 Basic concepts

Introduction to Rascal

Functional immutability with an imperative syntax

- All data is immutable

Introduction to Rascal

Common Data Types

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

Introduction to Rascal

Sets

- Unordered sequence 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: https://www.rascal-mpl.org/docs/Rascal/Expressions/Values/Set/

Introduction to Rascal

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 lists
- See: https://www.rascal-mpl.org/docs/Rascal/Expressions/Values/List/

Introduction to Rascal

Tuples

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

Introduction to Rascal

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: https://www.rascal-mpl.org/docs/Rascal/Expressions/Values/Relation/

Introduction to Rascal

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 (line + column, possibly for both begin and end)
- See: https://www.rascal-mpl.org/docs/Rascal/Expressions/Values/Location/

Introduction to Rascal

String templates

- Easy way to “generate” strings
- Often used for source-to-source transformation
- See: https://www.rascal-mpl.org/docs/Recipes/Common/StringTemplate/

Introduction to Rascal

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 https://www.rascal-mpl.org/docs/Recipes/BasicProgramming/PatternMatching/

Introduction to Rascal

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”`

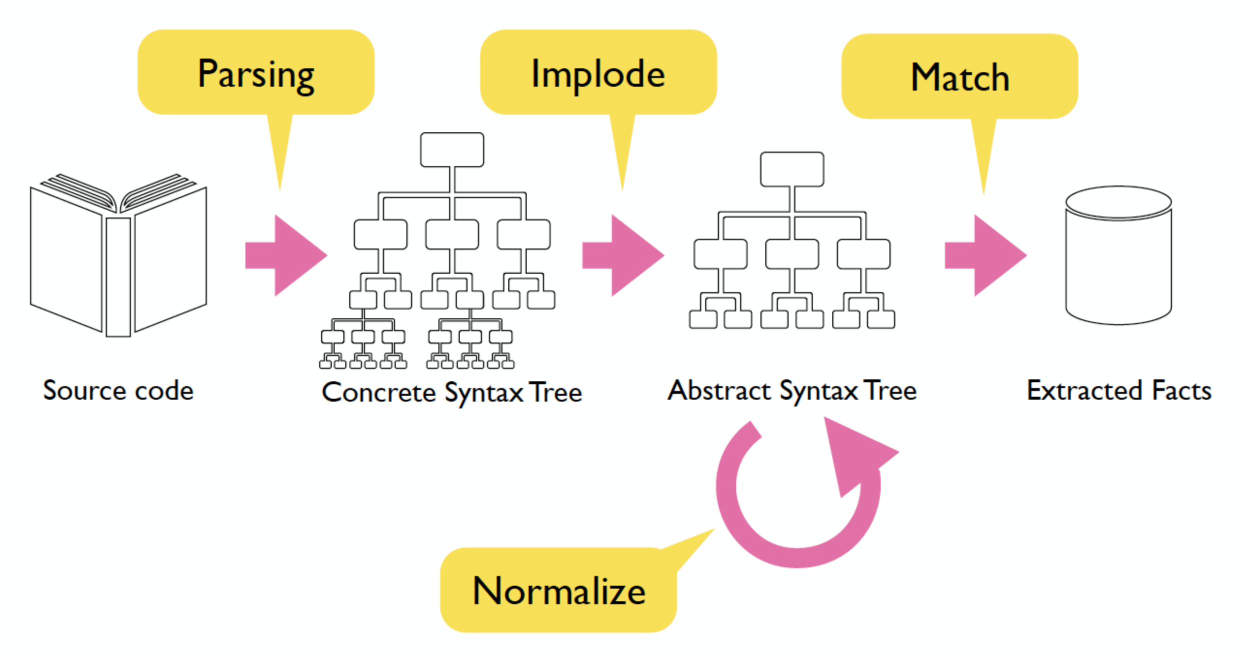
Questions?

Introduction to Rascal

2 Tools for language engineering

Introduction to Rascal

Extracting facts using parsing



Introduction to Rascal

Concrete Syntax

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

syntax Statement
  = "if" Expression "then" Stat* "else" Stat* "fi"
  | Id ":=" Expression
  | "while" Expression "do" Stat* "od" ;

syntax Expression
  = Id
  | "(" Expression ")"
  | left Expression "*" Expression
  > left Expression "+" Expression ;

lexical Id = [A-Za-z][A-Za-z0-9\-]*;

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

Introduction to Rascal

Describes all possible strings which can be produced

Concrete Syntax

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

syntax Statement
  = "if" Expression "then" Stat* "else" Stat* "fi"
  | Id ":=" Expression
  | "while" Expression "do" Stat* "od" ;

syntax Expression
  = Id
  | "(" Expression ")"
  | left Expression "*" Expression
  > left Expression "+" Expression ;

lexical Id = [A-Za-z][A-Za-z0-9\-]*;

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

Introduction to Rascal

Describes all possible strings which can be produced

Concrete Syntax

Nonterminal

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

syntax Statement
  = "if" Expression "then" Stat* "else" Stat* "fi"
  | Id ":=" Expression
  | "while" Expression "do" Stat* "od" ;

syntax Expression
  = Id
  | "(" Expression ")"
  | left Expression "*" Expression
  > left Expression "+" Expression ;

lexical Id = [A-Za-z][A-Za-z0-9\-]*;

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

Introduction to Rascal

Describes all possible strings which can be produced

Concrete Syntax

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

syntax Statem
  = "if" Expression "then" Stat* "else" Stat* "fi"
  | Id ":=" Expression
  | "while" Expression "do" Stat* "od" ;

syntax Expression
  = Id
  | "(" Expression ")"
  | left Expression "*" Expression
  > le                "+" Expression ;

lexical Id = [A-Za-z][A-Za-z0-9\-]*;

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

Terminal

Nonterminal

Terminal

Layout characters

Introduction to Rascal

Abstract Syntax (= ADT)

```
data Program = program(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)
  | mult(Expr lhs, Expr rhs)
  | add(Expr lhs, Expr rhs) ;
```

Introduction to Rascal

ADT)

Same information as concrete tree but more abstract

```
data Program = program(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)
  | mult(Expr lhs, Expr rhs)
  | add(Expr lhs, Expr rhs) ;
```

Introduction to Rascal

Same information as concrete tree but more abstract

ADT)

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

data
   \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)
  | mult(Expr lhs, Expr rhs)
  | add(Expr lhs, Expr rhs) ;
```

Abstract Data Type

Introduction to Rascal

ADT)

Same information as concrete tree but more abstract

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

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

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

Abstract Data Type

Escape for keywords

Questions?

Introduction to Rascal

Extracting information from an Abstract Syntax Tree

- Use Pattern Matching
  - Match on structure
  - Match on values
  - Deep matching
- See: https://www.rascal-mpl.org/docs/Recipes/BasicProgramming/PatternMatching/ and https://www.rascal-mpl.org/docs/Rascal/Patterns/

Introduction to Rascal

Find all assigned variables

```
data Program = program(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) ;




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

Introduction to Rascal

Find all assigned variables

```
data Program = program(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

Introduction to Rascal

Find all assigned variables

```
data Program = program(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> ");
}
```

Introduction to Rascal

Find all assigned variables

```
data Program = program(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 the assignment is nested?

Introduction to Rascal

Find all assigned variables (including *nested* ones)

```
data Program = program(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>");
}
```

Introduction to Rascal

Find the variable named "x"

```
data Program = program(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 a specific value

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

Introduction to Rascal

Transforming an Abstract Syntax Tree

- Use visit statement
  - Reach all nodes
  - Not composable
- See https://www.rascal-mpl.org/docs/Rascal/Expressions/Visit/

Introduction to Rascal

Rename "x" to "y"

```
data Program = program(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)

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

Introduction to Rascal

Rename "x" to "y"

```
data Program = program(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
   = id(s       e)

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

Visit all the nodes in the tree

Rewrite/replace node

Both assignment and use are replaced

Introduction to Rascal

Visiting strategies

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

Questions?

Introduction to Rascal

Debugging Rascal:

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

Introduction to Rascal

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/UndeclaredVariable.html|
```

- Solution for above example: `import List;`

Introduction to Rascal

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>,<1,102>))
```

Introduction to Rascal

Common errors: 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)

Introduction to Rascal

Looking for how you can do stuff in Rascal?

Browse and search the documentation

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

www.rascal-mpl.org/docs/Rascal/
Rascal Language Reference | The Rascal Meta Programming Language
The Rascal Meta Programming Language
Docs
Blog
Release notes
Projects
Team
GitHub
Search
Getting Started with Rascal
Getting Help
Recipes
Rascalopedia
Rascal Language Reference
Rascal Language Reference

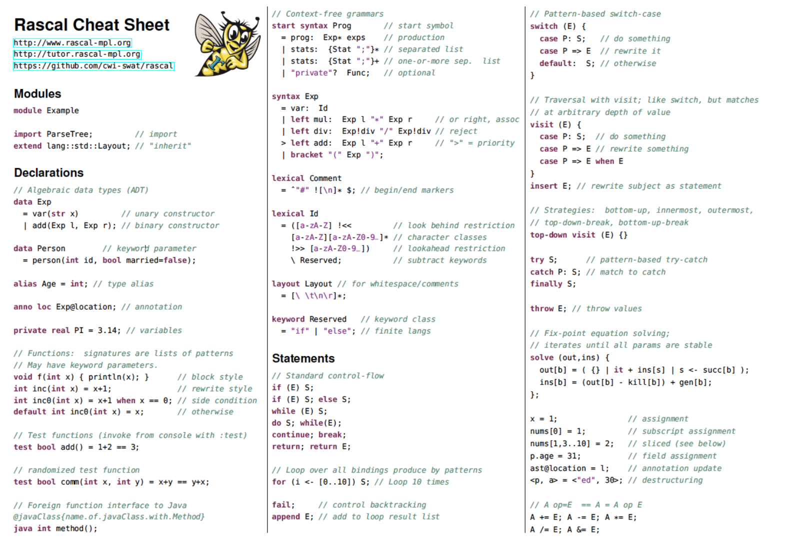
Rascal Language Reference

rascal-0.28.2

Introduction to Rascal

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;
```

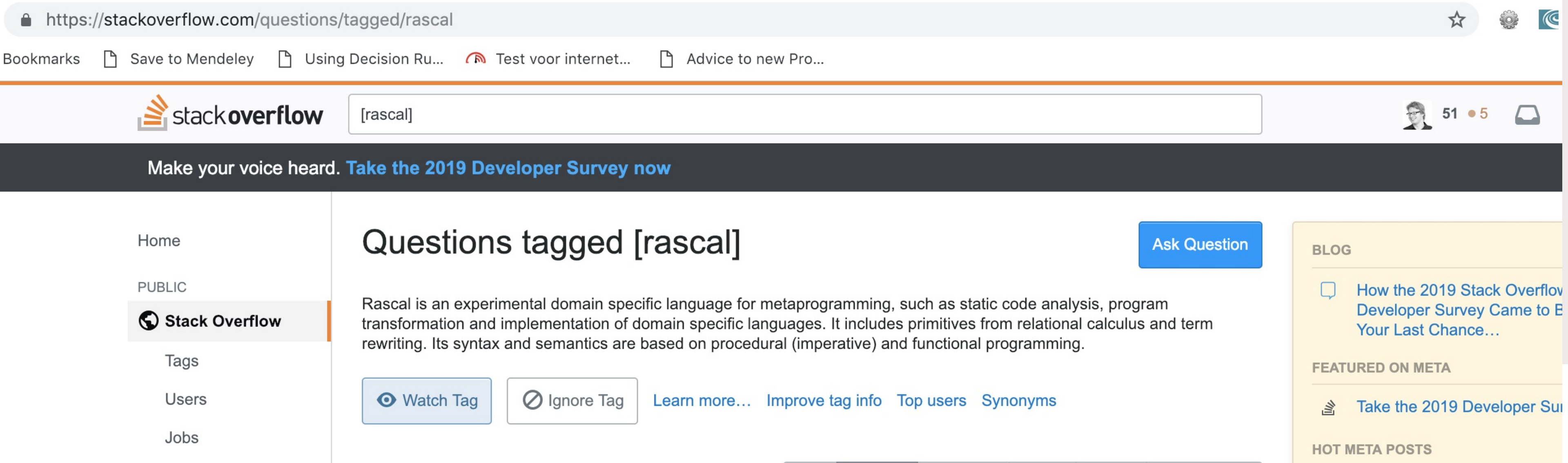
https://github.com/cwi-swat/rascal-cheat-sheet/raw/master/sheet.pdf

Introduction to Rascal

Looking for how you can do stuff in Rascal?

Search on StackOverflow

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



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

[rascal]

Make your voice heard. Take the 2019 Developer Survey now

Questions tagged [rascal]

Rascal is an experimental domain specific language for metaprogramming, such as static code analysis, program transformation and implementation of domain specific languages. It includes primitives from relational calculus and term rewriting. Its syntax and semantics are based on procedural (imperative) and functional programming.

Introduction to Rascal

Some “hello world” exercises on Rascal:

https://www.rascal-mpl.org/docs/Recipes/BasicProgramming/