



**sets**

**Lists,** **and** **grammars**

**in Rascal**

Vadim Zaytsev, Universiteit van Amsterdam  
10 July 2014, SATT<sub>o</sub>SE 2014



# Rich (immutable) data

- Built-in sophisticated types:
  - lists
  - sets
  - maps
  - tuples
  - relations
- with comprehensions and many operators

```
rascal> [1..10]
```

```
list[int]: [1,2,3,4,5,6,7,8,9,10]
```

```
rascal> [x/2 | x <- [1..10]]
```

```
list[int]: [0,1,1,2,2,3,3,4,4,5]
```

```
rascal> {x/2 | x <- [1..10]} + {4,5,6}
```

```
set[int]: {6,5,4,3,2,1,0}
```



# Syntax definitions

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/...
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs



# Syntax definitions

**lexical** Id = [A-Za-züäöß]+ !>> [A-Za-züäöß];

**lexical** Num = [0-9]+ !>> [0-9];

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/...
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs



# Syntax definitions

```
start syntax System = Line+;  
syntax Line = Num ":" {Id ","}+ "." ;
```

- Define lexical syntax
- Define context-free syntax
- Define whitespace/layout/...
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs



# Syntax definitions

- Define lexical syntax **layout**  $WS = [\backslash \backslash t \backslash n \backslash r]^* !>> [\backslash \backslash t \backslash n \backslash r];$
- Define context-free syntax
- Define whitespace/layout/...
- Get GLL parser for free
- Define an algebraic data type
- Automatically implode parse trees to ASTs



# Patterns

- Pattern matching
  - on concrete syntax
  - on lists
  - on sets
  - on trees
  - ...
- Pattern-driven dispatch

```
rascal> {int x, str y} := {2}
```

```
bool: false
```

```
rascal> {int x, str y} := {2, "3"}
```

```
bool: true
```

```
rascal> {int x, *y, str z} := {2,2,2,"3",4,"2"}
```

```
bool: true
```



# Other pattern kinds

- **Regular:** grep/Perl like regular expressions
  - `/^<before:\W*><word:\w+><after:.*$/`
- **Abstract:** match data types
  - `whileStat(Exp, Stats*)`
- **Concrete:** match parse trees
  - `` while <Exp> do <Stats*> od ``





# Pattern-directed invocation

Prolog?

```
bool eqfp(fpnt(), fpnt()) = true;  
bool eqfp(fpopt(), fpopt()) = true;  
bool eqfp(fpplus(), fpplus()) = true;  
bool eqfp(fpstar(), fpstar()) = true;  
bool eqfp(fpempty(), fpempty()) = true;  
bool eqfp(fpmany(L1), fpmany(L2)) = multiseteq(L1,L2);  
default bool eqfp(Footprint pi, Footprint xi) = false;
```



# Switch/case

**switch(p)**

{

**case** (DCGFun) `[]` => ["ε"];

**case** (DCGFun) `<Word n>` =>

["<n>" | "<n>"==toLowerCase("<n>")];

**case** (DCGFun) `(<{DCGFun ", "}\* args>)` =>

[\*getTags(a) | a <- args];

**case** (DCGFun) `<Word f> (<{DCGFun ", "}\* as>)` =>

["<f>"] + [\*getTags(a) | a <- as];

**default ...**

}



# Visitor

```
@contributor{Bas Basten - Bas.Basten@cwi.nl (CWI)}
```

```
@contributor{Mark Hills - Mark.Hills@cwi.nl (CWI)}
```

```
module Operations
```

```
import AST;
```

```
import IO;
```

```
public Company cut(Company c) {
```

```
  return visit (c) {
```

```
    case employee(name, [*ep,ip:intProp("salary",salary),*ep2])
```

```
      => employee(name, [*ep,ip[intVal=salary/2],*ep2])
```

```
  }
```

```
public int total(Company c) {
```

```
return (0 | it+salary | /employee(name, [*ep,ip:intProp("salary",salary),*ep2]) <- c);
```

```
}
```



# ADTs and visitors

```

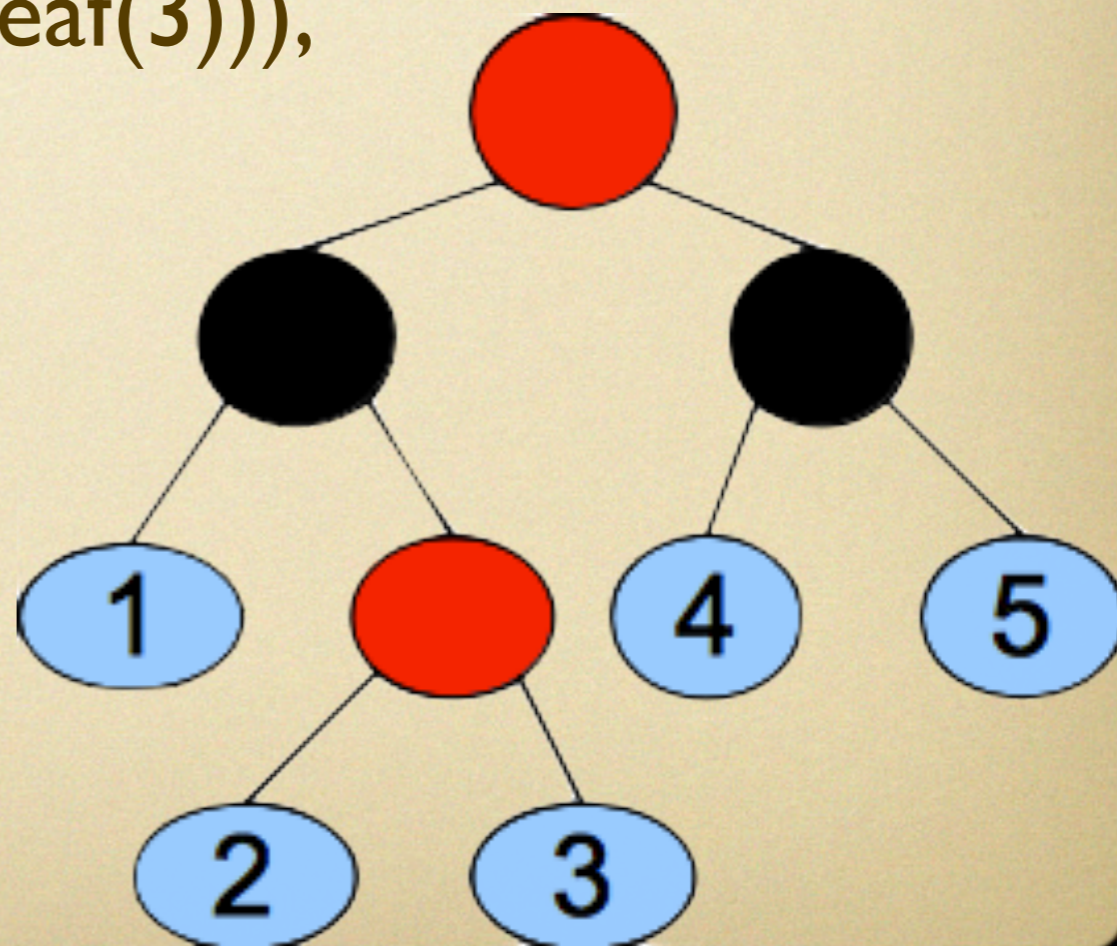
data CTree = leaf(int N)
            | red(CTree left, CTree right)
            | black(CTree left, CTree right) ;
  
```

```

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

```

public int cntRed(CTree t) {
  int c = 0;
  visit(t){case red(_, _): c += 1;};
  return c;
}
  
```



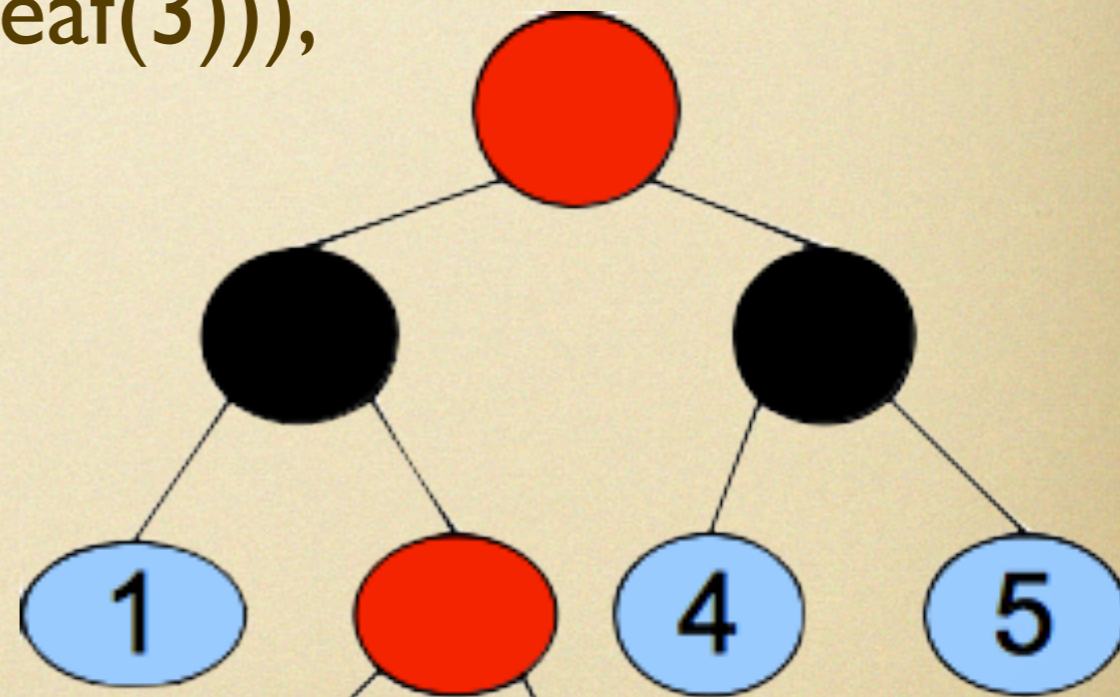


# ADTs and visitors

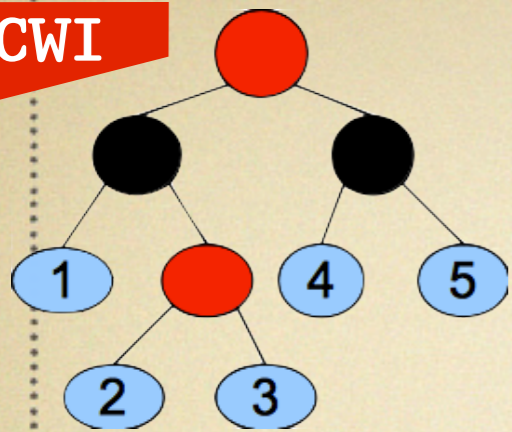
```
data CTree = leaf(int N)
           | red(CTree left, CTree right)
           | black(CTree left, CTree right) ;
```

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

```
public int cntRed(CTree t) {
  int c = 0;
  visit(t){case red(_, _): c += 1;};
  return c;
}
```



```
public int cnt2(CTree t) = size([b | /b:red(_,_) := t]);
```

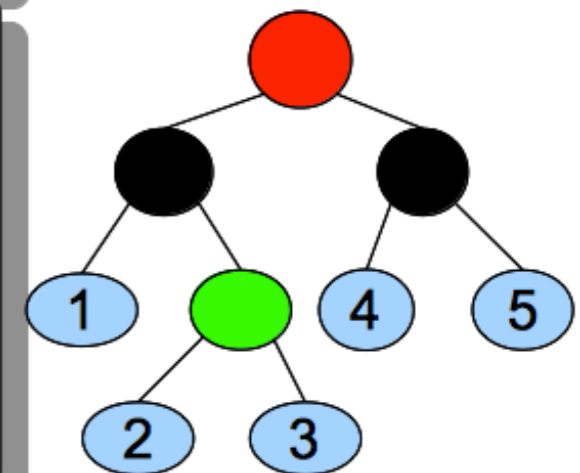
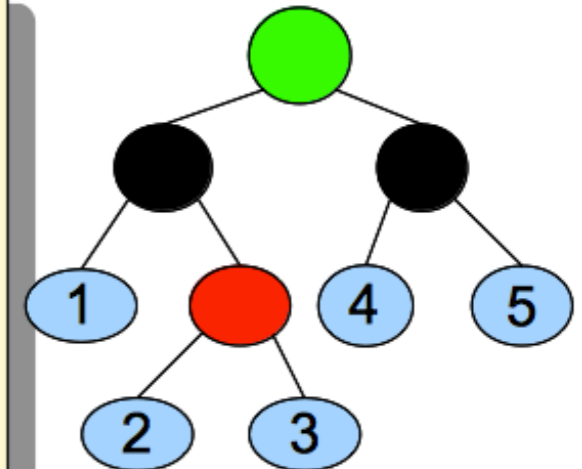
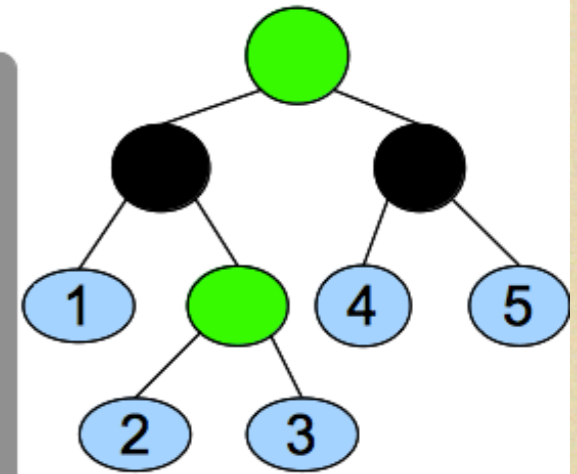


# Full/shallow/deep

```
public CTree frepl(CTree T) {
  return visit (T) {
    case red(CTree T1, Ctree T2) => green(T1, T2)
  };
}
```

```
public Ctree srepl(CTree T) {
  return top-down-break visit (T) {
    case red(Ctree T1, CTree T2) => green(T1, T2)
  };
}
```

```
public Ctree drepl(Ctree T) {
  return bottom-up-break visit (T) {
    case red(CTree T1, CTree T2) => green(T1, T2)
  };
}
```



[vadim@grammarware.net](mailto:vadim@grammarware.net)



- <http://rascal-mpl.org>
- <http://ask.rascal-mpl.org>
- <http://tutor.rascal-mpl.org>



questions

