SWAT

Rascal
The Metaprogramming Language

Summer School on Software Technologies and Software Languages

Vadim Zaytsev, SWAT, CWI

rASCAL

Joint work with (amongst others):

Bas Basten, Mark Hills, Anastasia Izmaylova, **Paul Klint**, Davy Landman, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Michael Steindorfer, **Tijs van der Storm**, **Jurgen Vinju**.

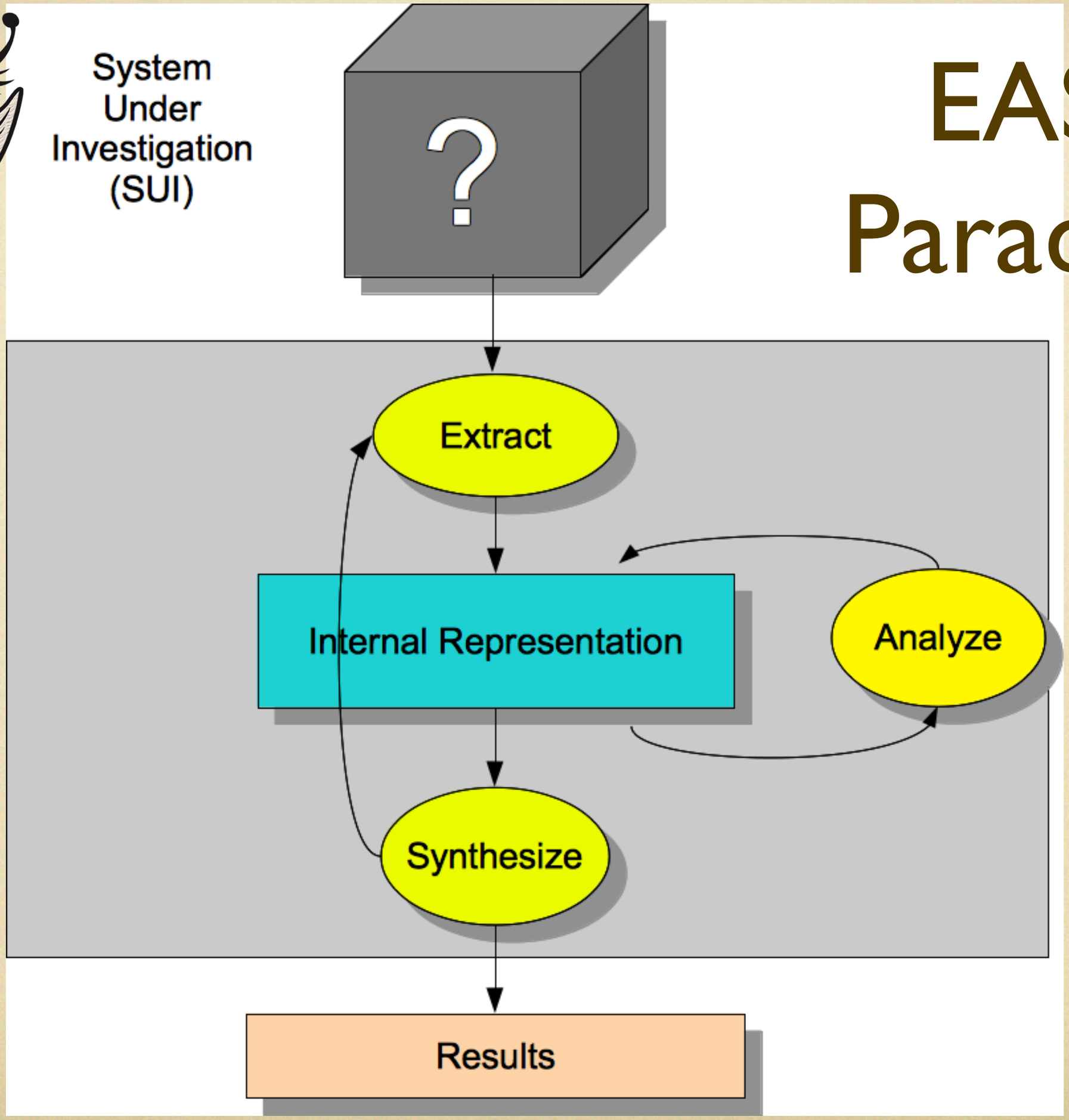
CWI

Technical 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 (transform, refactor, compile)?
- How to synthesize other information?

EASY: Extract-Analyze-SYnthesize Paradigm

EASY Paradigm



CWI

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 **ASF+SDF** and **Rscript**
  - in a new, *unified*, *extensible*, *teachable* framework

CWI

Rascal keywords

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

CWI

Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

presumably familiar

CWI

Rascal design

- Java-like syntax
- Embedded in Eclipse

installs as a plugin

- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

CWI

Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

low barrier to entry,
learn features as you go

CWI

Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

concrete syntax matching

CWI

Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

traversals, matching, …

CWI

Rascal design

- Java-like syntax
- Embedded in Eclipse
- Layered design
- Syntax analysis
- Term rewriting
- Relational calculus

relations for sharing/merging of facts for different languages

Rascal features

CWI

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}
```

CWI

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

CWI

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

CWI

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

CWI

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

layout WS = [\ \t\n\r]* !>> [\ \t\n\r];

CWI

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
```

CWI

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 `

CWI

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;
```

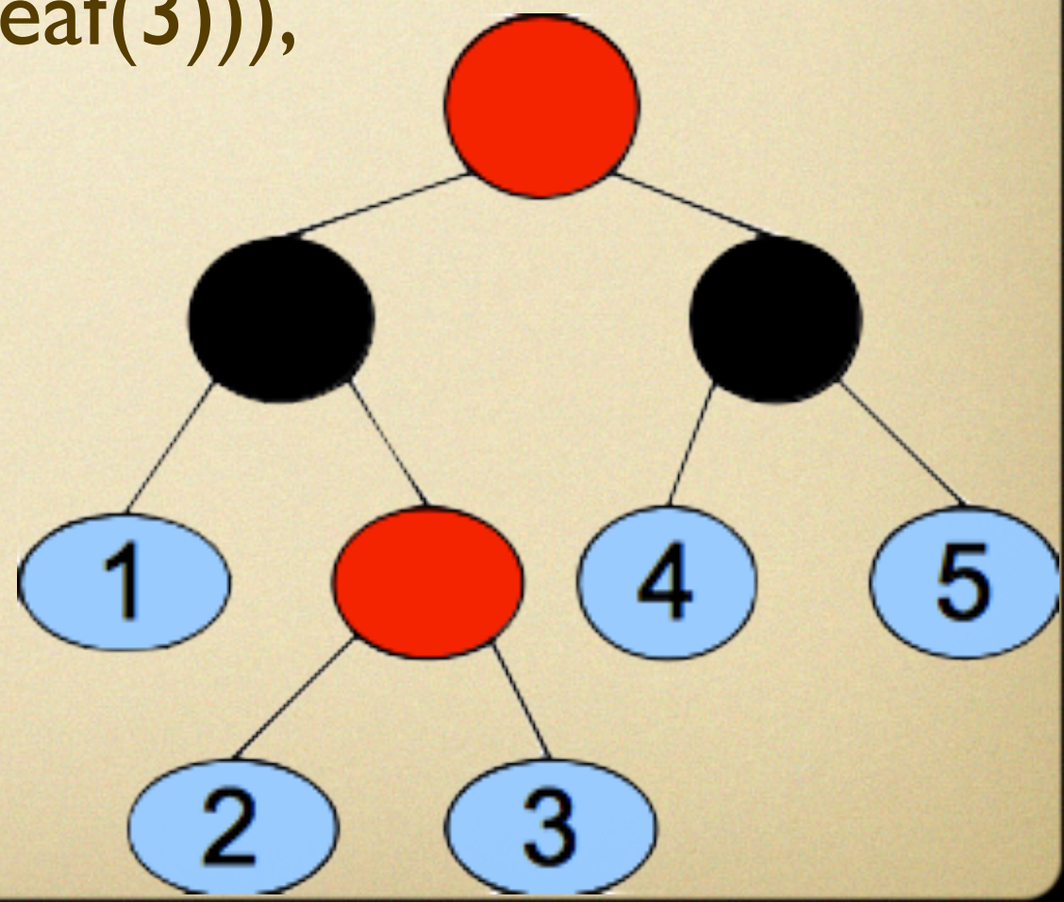
CWI

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;
}
```



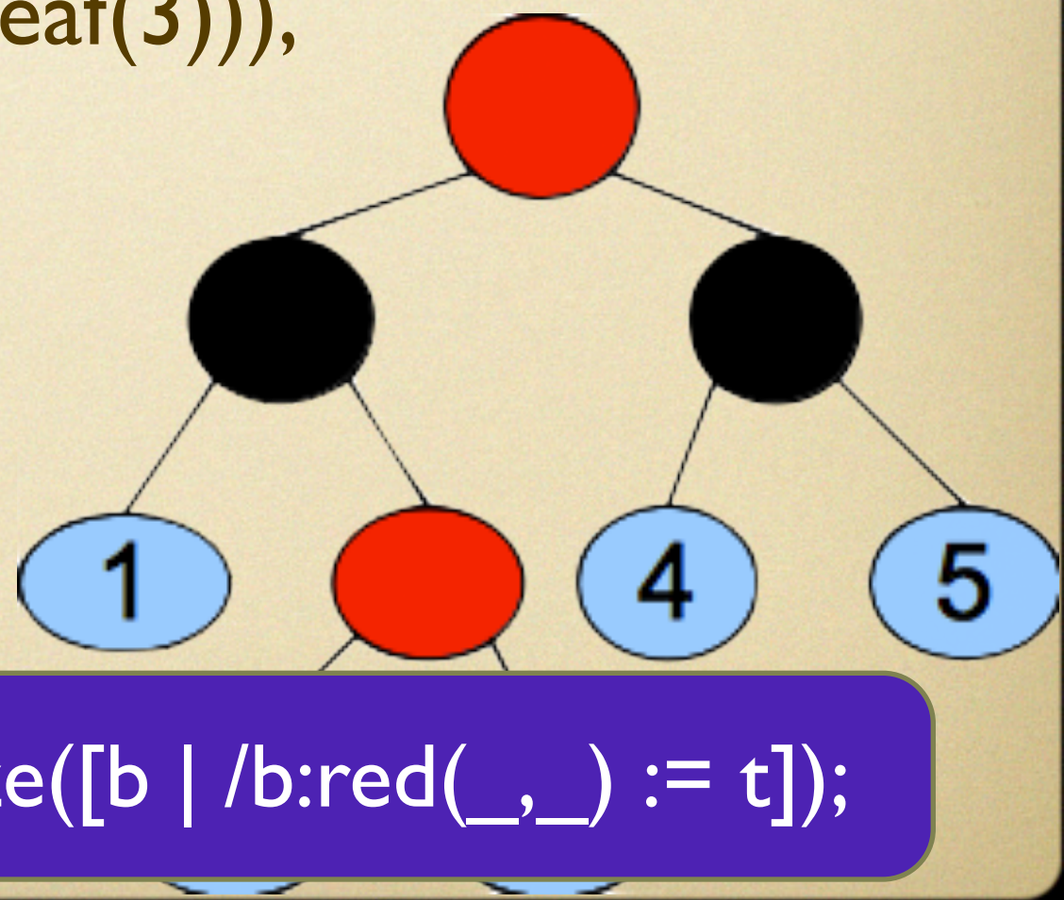
CWI

ADTs and visitors

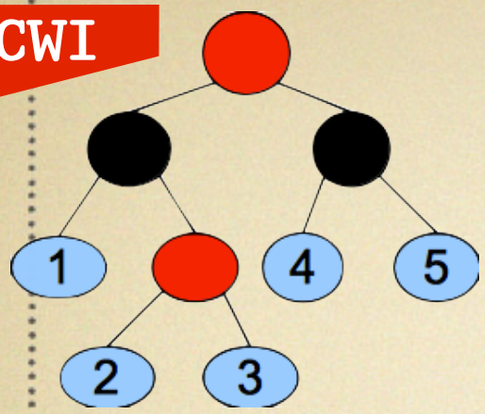
```
data CTree =  leaf(int N)
            | red(CTree left, CTree right)
            | black(CTree left, CTree right) ;
```

```
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;
}
```

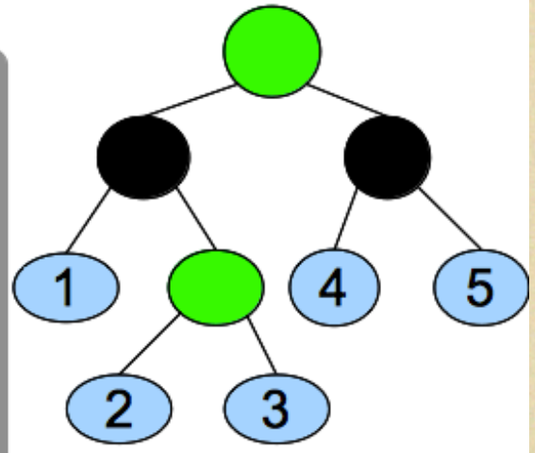



```
public int cnt2(CTree t) = size([b | /b:red(_,_) := t]);
```

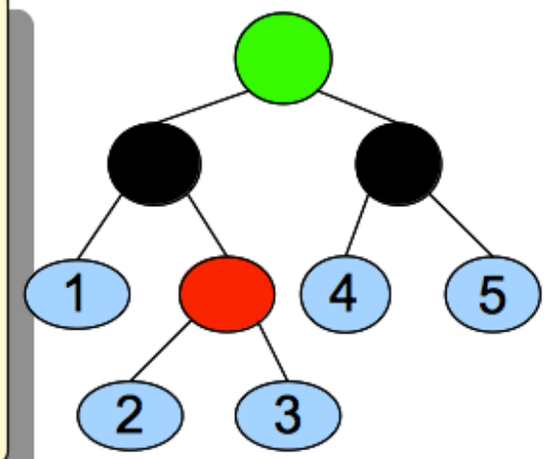


Full/shallow/deep

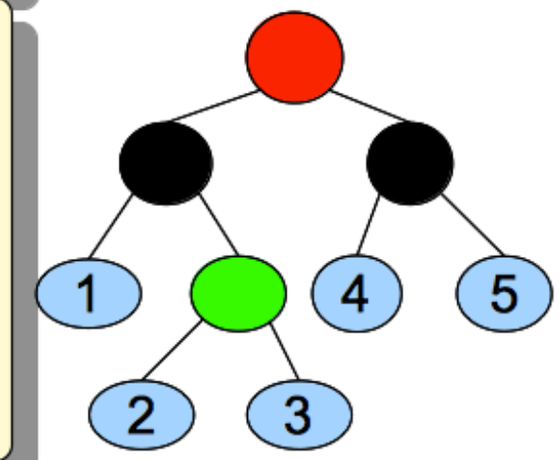
```
public CTree frepl(CTree T) {
    return visit (T) {
       case red(CTree T1, Ctree T2) => green(T1, T2)
    };
}
```




```
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)
    };
}
```



CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Example

```
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 ...
}
```

CWI

Hackathon: NCLOC

@contributor{Vadim Zaytsev - vadim@grammarware.net - SWAT, CWI}

```
module NCLOC

import IO;
import ParseTree;
import List;

lexical OneLineComment = "//" ![\n]* >> [\n];
lexical CodeLine = ![/\n]* meat OneLineComment? >> [\n];
start syntax SCModel = {(OneLineComment | CodeLine) "\n"}+ "\n"?;
layout WS = [\ \t]* !>> [\ \t];

public void main(list[str] args)
  = println(size([l | /CodeLine l := parse(#start[SCModel],|cwd:///|+args
[0]), "<l.meat>" != ""]));
```

CWI

101companies

@contributor{Bas Basten - Bas.Basten@cwi.nl (CWI)}
@contributor{Mark Hills - Mark.Hills@cwi.nl (CWI)}
module Operations

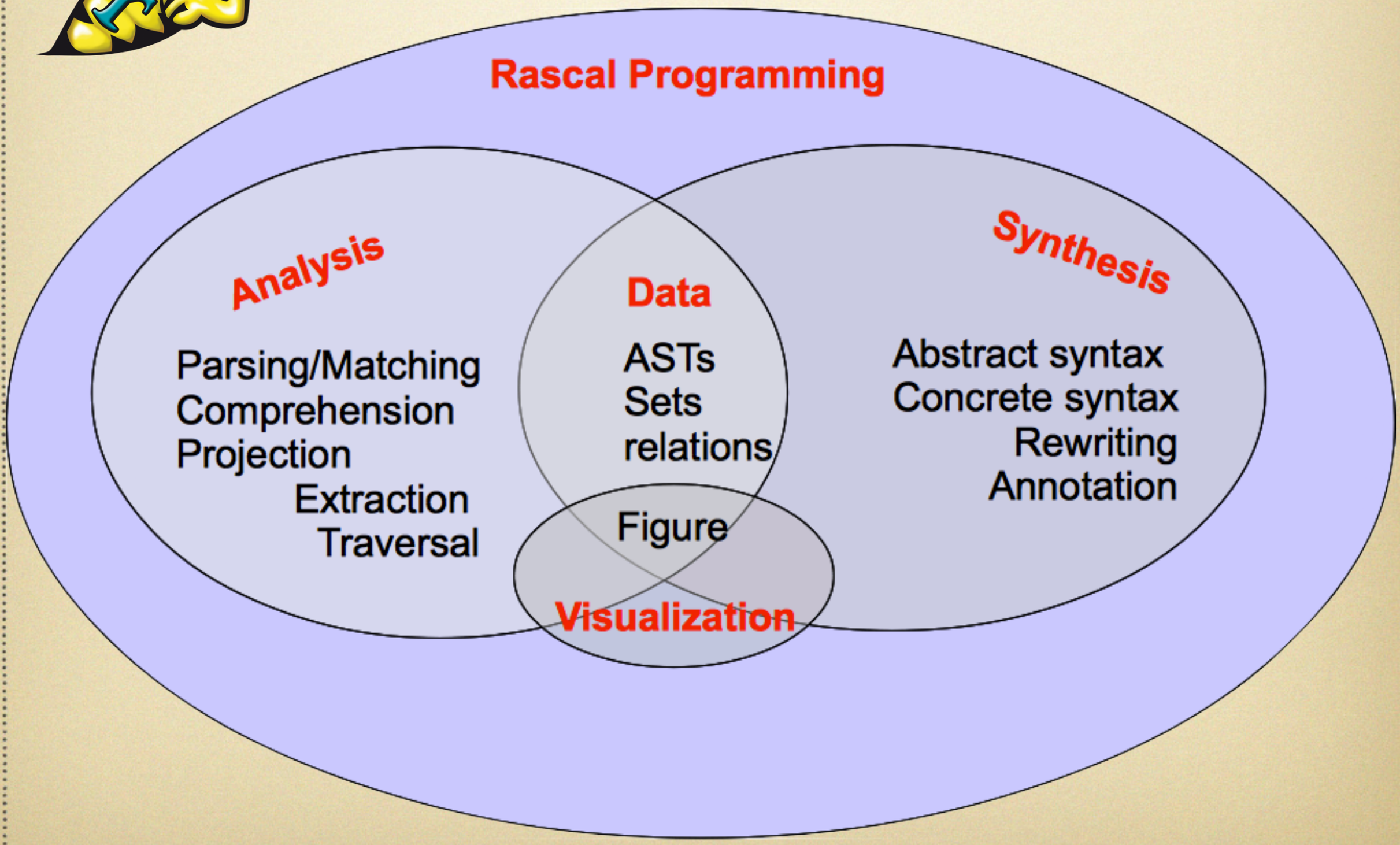
```
import AST;
import IO;

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);
}
```

CWI

Bridging the gaps



vadim@grammarware.net

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

Questions?