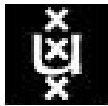


EASY Meta-Programming with Rascal

Leveraging the Extract-Analyze-SYnthesize Paradigm

Paul Klint & Jurgen Vinju



UNIVERSITEIT VAN AMSTERDAM



Centrum Wiskunde & Informatica

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE








centre de recherche LILLE - NORD EUROPE

Joint work with (amongst others):

*Bas Basten, Mark Hills, Anastasia Izmaylova, Davy Landman,
Arnold Lankamp, Bert Lisser, Atze van der Ploeg,
Tijs van der Storm, Vadim Zaytsev*



Cast of Our Heroes

- Alice, system administrator 
- Bernd, forensic investigator 
- Charlotte, financial engineer 
- Daniel, multi-core specialist 
- Elisabeth, model-driven engineering specialist 





Meet Alice

- Alice is security administrator at a large online marketplace
- **Objective:** look for security breaches
- **Solution:**
 - Extract relevant information from system log files, e.g. failed login attempts in Secure Shell
 - Extract IP address, login name, frequency, ...
 - Synthesize a security report

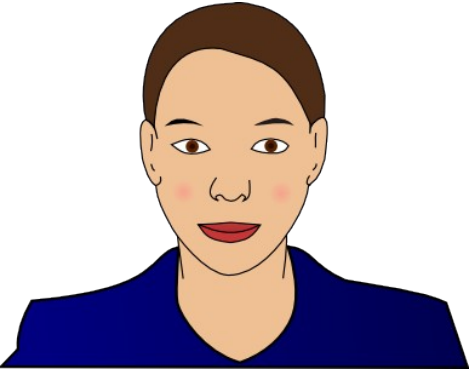


Meet Bernd



- Bernd: investigator at German forensic lab
- **Objective:** finding common patterns in confiscated digital information in many different formats. This is very labor intensive.
- **Solution:**
 - Design DERRICK a domain-specific language for this type of investigation
 - Extract data, analyze the used data formats and synthesize Java code to do the actual investigation





Meet Charlotte

- Charlotte works at a large financial institution in Paris
- **Objective:** connect legacy software to the web
- **Solution:**
 - extract call information from the legacy code, analyze it, and synthesize an overview of the call structure
 - Use entry points in the legacy code as entry points for the web interface
 - Automate these transformations



Meet Daniel



- Daniel is concurrency researcher at one of the largest hardware manufacturers worldwide
- **Objective:** leverage the potential of multi-core processors and find concurrency errors
- **Solution:**
 - extract concurrency-related facts from the code (e.g., thread creation, locking), analyze these facts and synthesize an abstract automaton
 - Analyze this automaton with third-party verification tools





Meet Elisabeth

- Elisabeth is software architect at an airplane manufacturer
- **Objective:** Model reliability of controller software
- **Solution:**
 - describe software architecture with UML and add reliability annotations
 - Extract reliability information and synthesize input for statistics tool
 - Generate executable code that takes reliability into account



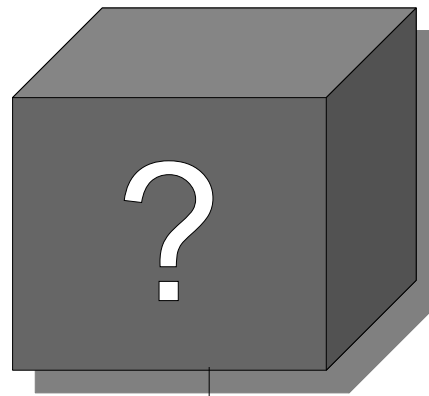
What are their *Technical Challenges?*

- 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 (trafo, refactor, compile)?
- How to synthesize visualizations, charts?

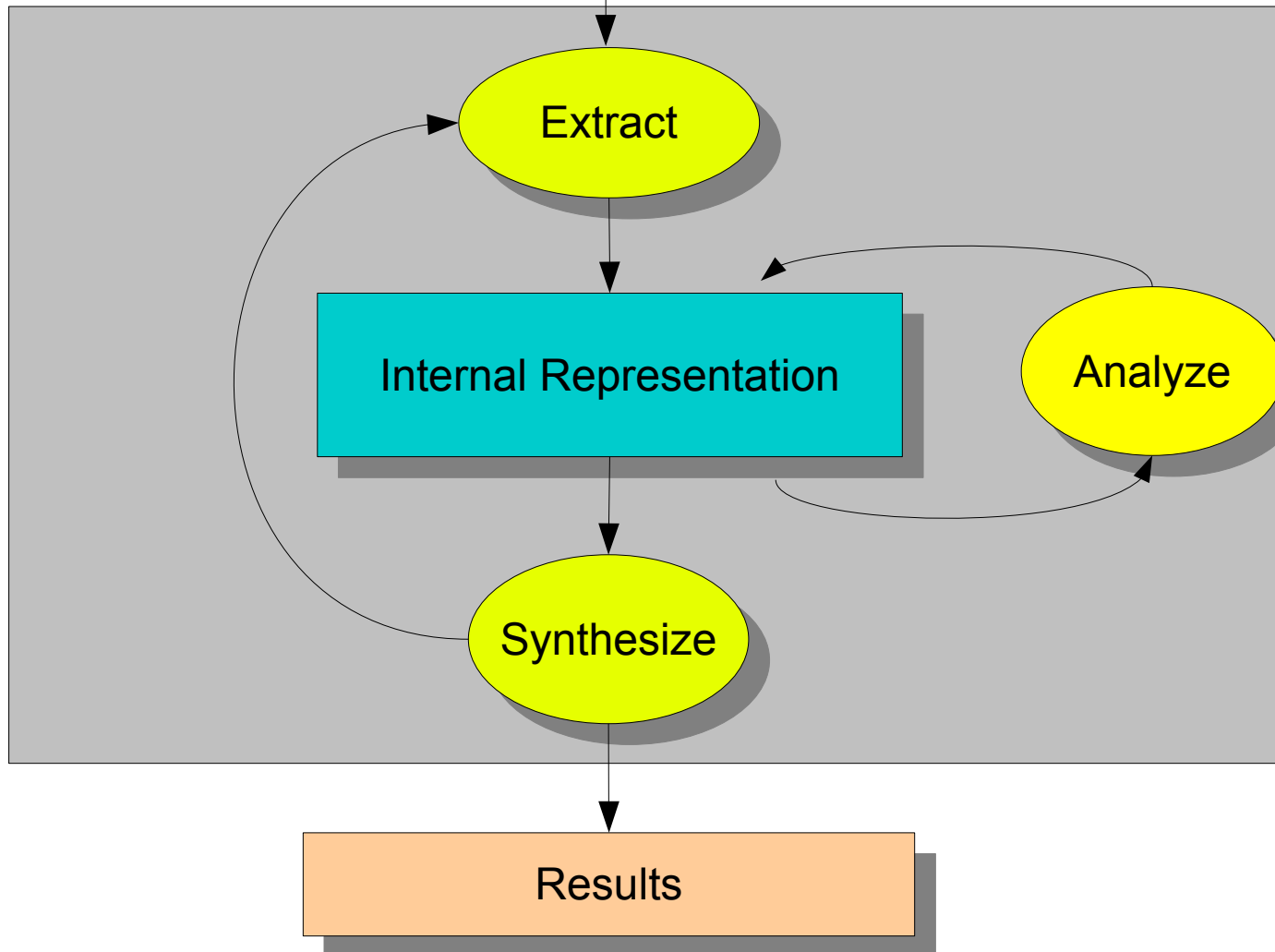


EASY: Extract-Analyze-SYnthesize Paradigm

System Under Investigation (SUI)



EASY Paradigm



Why a new Language?

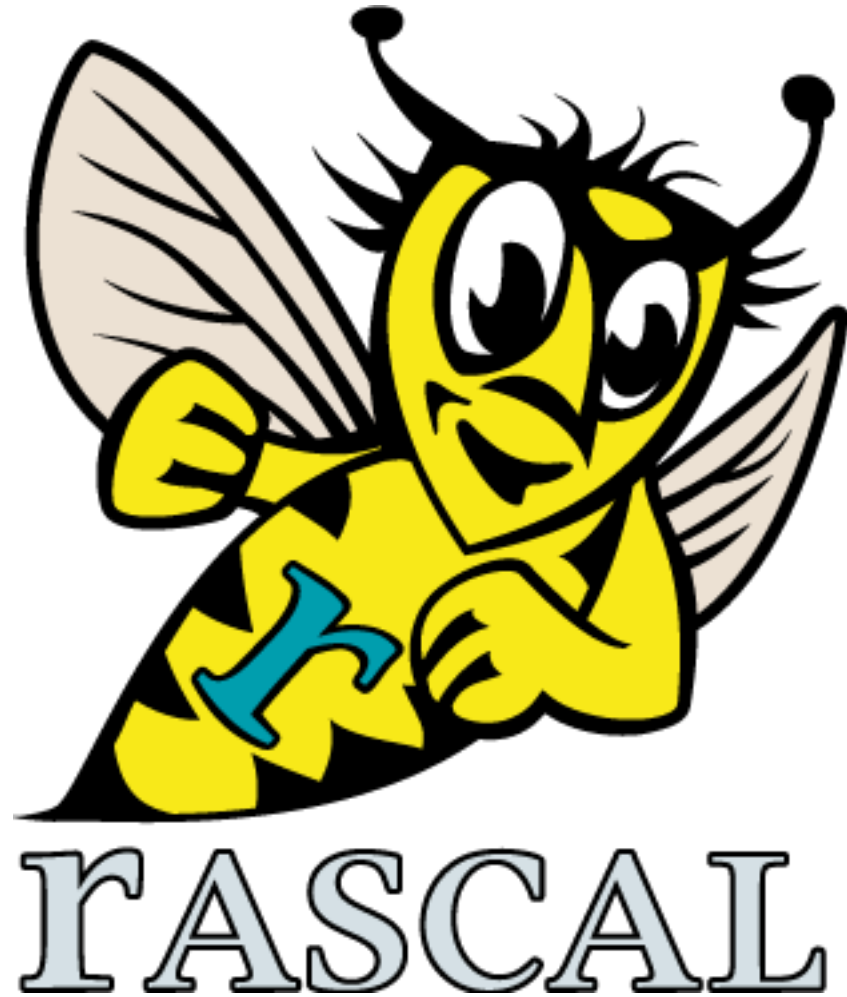
- No current technology spans the full range of EASY steps
- There are many fine technologies but they are
 - highly specialized with steep learning curves
 - hard to learn unintegrated technologies
 - not integrated with a standard IDE
 - hard to extend



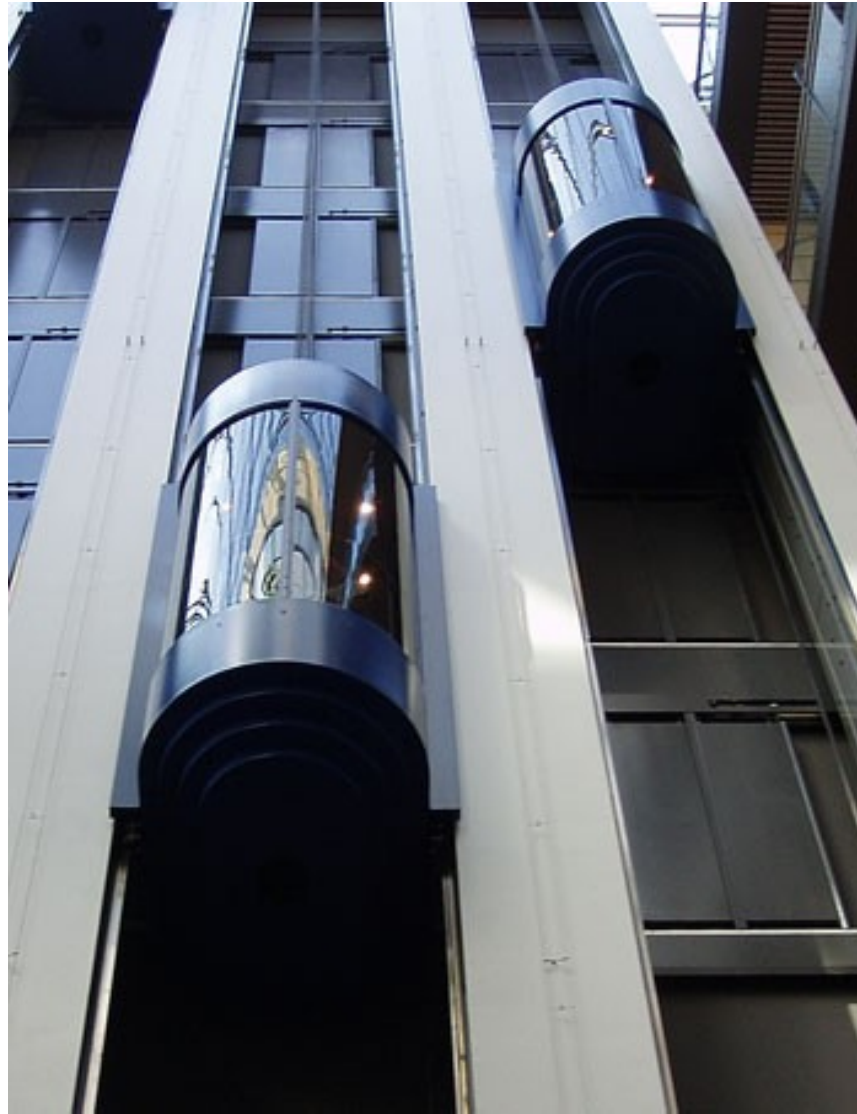
Goal

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

Here comes Rascal to the Rescue



Rascal Elevator Pitch



EASY Meta-Programming with Rascal



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



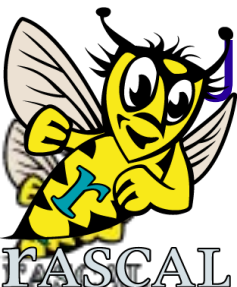
Rascal design based on ...

- **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

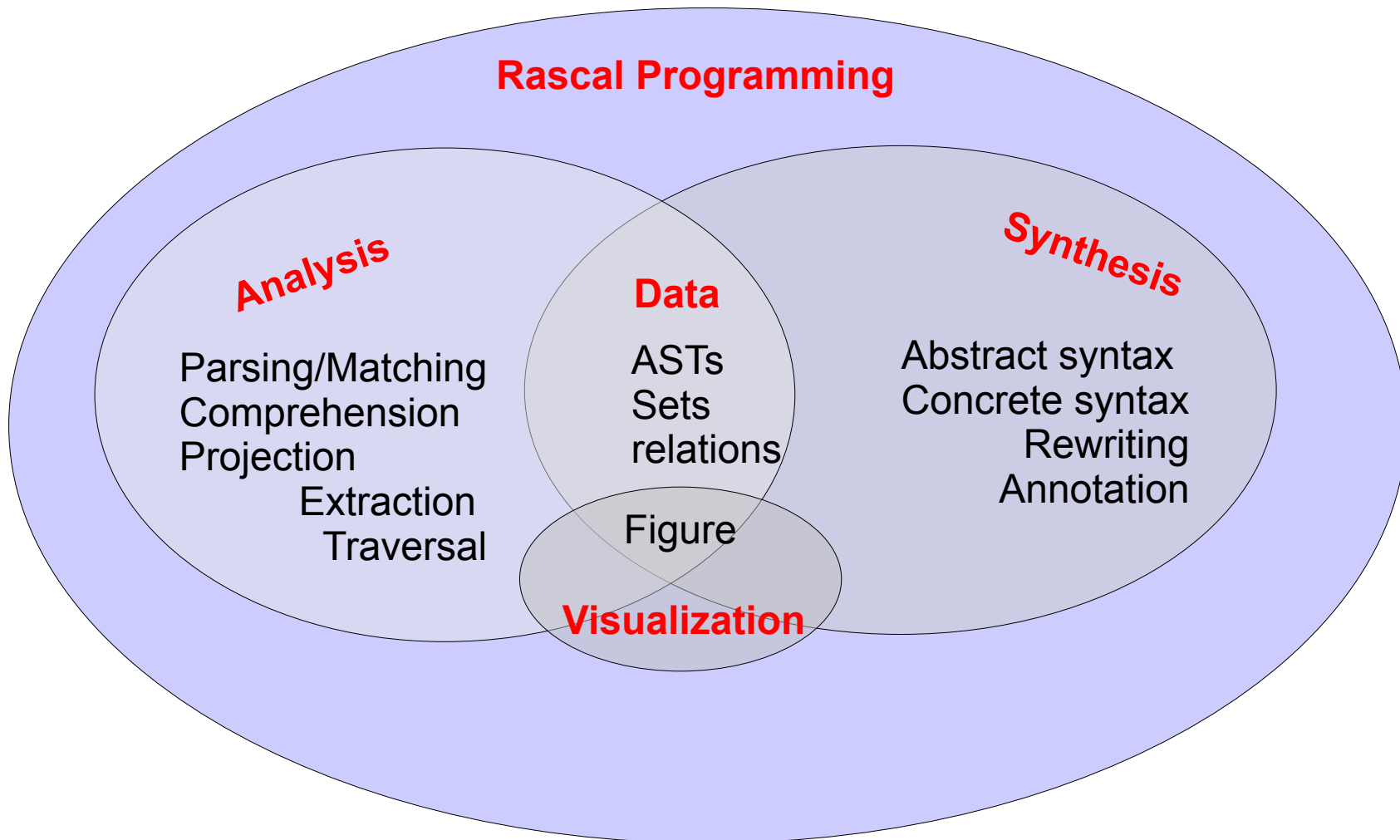


Rascal provides

- Rich (immutable) data: lists, sets, maps, tuples, relations, ... with comprehensions and many operators
- Syntax definitions & parser generation
- Syntax trees, tree traversal
- Pattern matching (text, trees, lists, sets, ...) and pattern-directed invocation
- Code generation (string templates & trees)
- Java and Eclipse (IMP) integration



Bridging Gaps



One-stop-shop

Cool parsers

Deal of the day:
Cheap type checkers

Fancy visualization

Just in: new modeling gadgets



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}
```

Sets do not
contain duplicates

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

Set union

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



Read-Eval-Print

Set comprehension

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

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



Read-Eval-Print

String
interpolation

```

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
10 * 10 = 100
list[void]: []

```



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

```
rascal > import demo::basic::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: 2586232415116818064296435515361197996
9197632389120000000000
```



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, \dots, 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, \dots, 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 datastructures

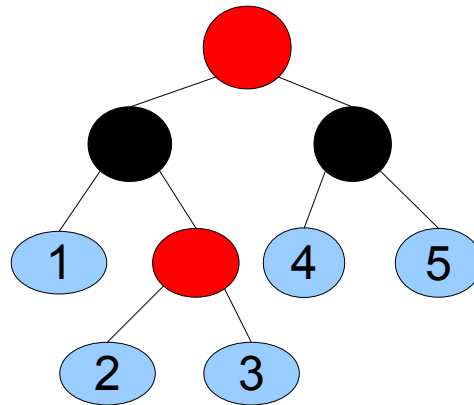
- Named alternatives
 - name acts as constructor
 - can be used in patterns
- Named fields (access/update via . notation)
- All datastructures are a subtype of the standard type node
 - Permits very generic operations on data
- Parse trees resulting from parsing source code are represented by the datatype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



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)));
```



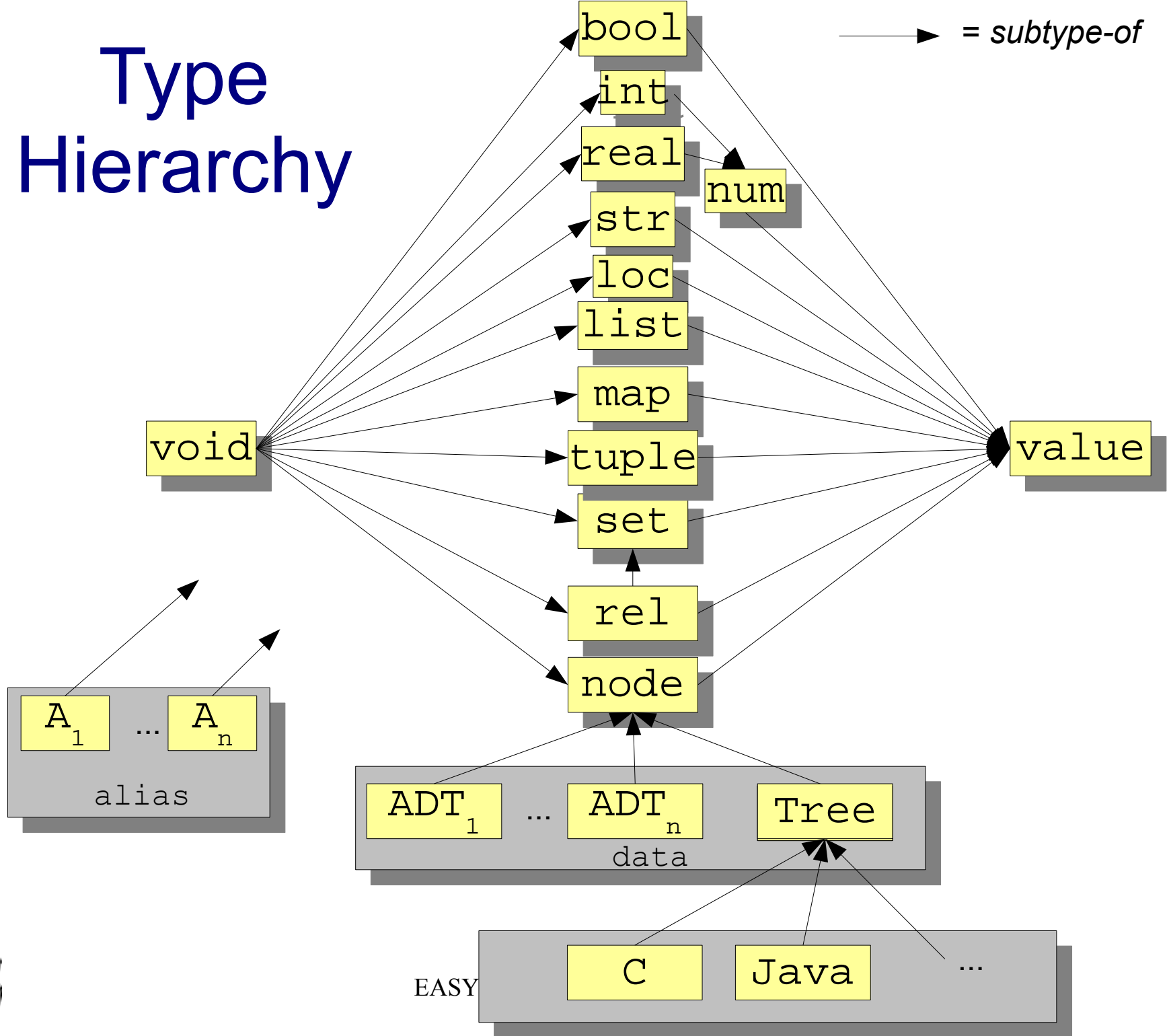
Abstract Syntax

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

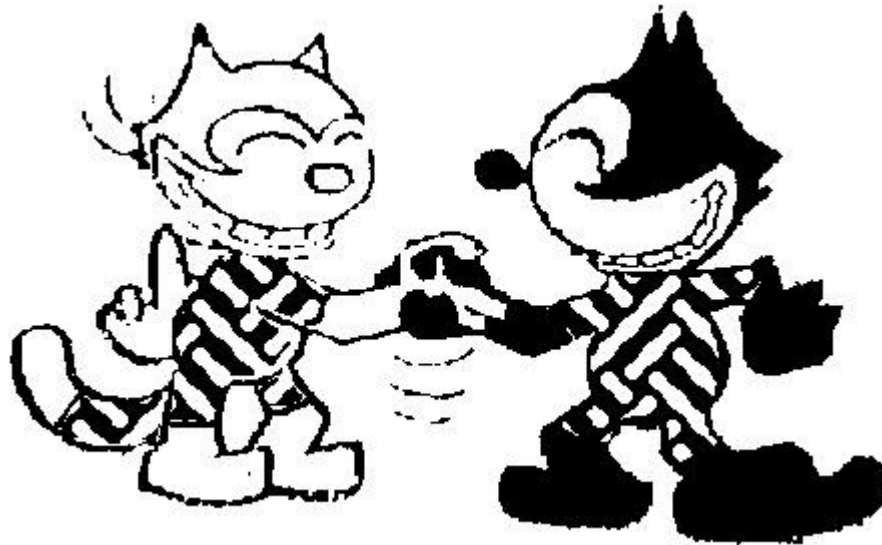


Type Hierarchy

→ = *subtype-of*



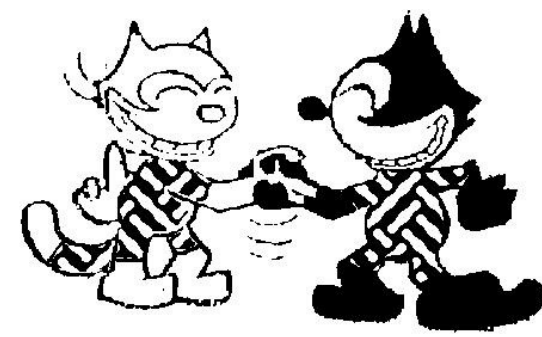
Pattern matching



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

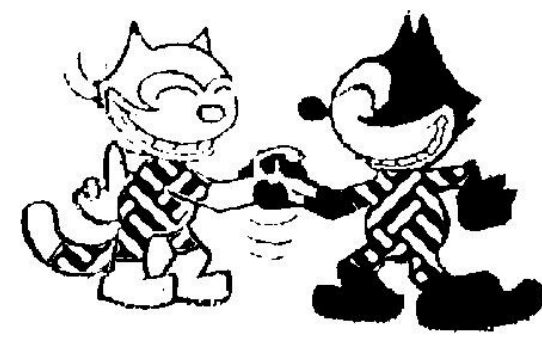


Pattern matching is used in:

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



Patterns



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 `
```



Regular Patterns

```
rascal>/[a-z]+/ := "abc"  
bool: true
```

```
rascal>/rac/ := "abracadabra";  
bool: true
```

```
rascal>/^rac/ := "abracadabra";  
bool: false
```

```
rascal>/rac$/ := "abracadabra";  
bool: false
```



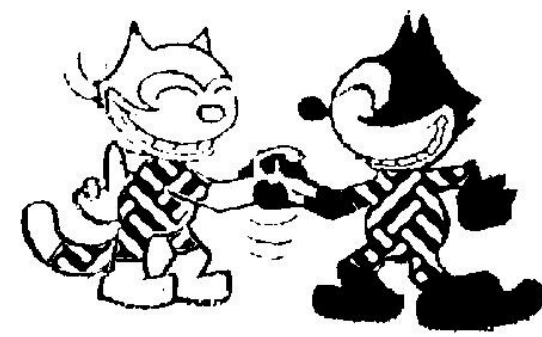
Regular Patterns

```
rascal>if( /\W<x:[a-z]+>/ := "12abc34" )
      println( "x = <x>" );
ok
```

- Matches non-word characters (`\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: `abc`.
- **Regular patterns are tricky (in any language)!**



Patterns



Abstract/Concrete patterns support:

- **List matching:** $[P1, \dots, Pn]$
- **Set matching:** $\{ P1, \dots, Pn \}$
- **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]
```

List pattern

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

```
bool: true
```

X* is a list variable
and abbreviates
list[int] X

```
rascal> X;
```

```
Error: X is undefined
```

X is bound but has
limited scope

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

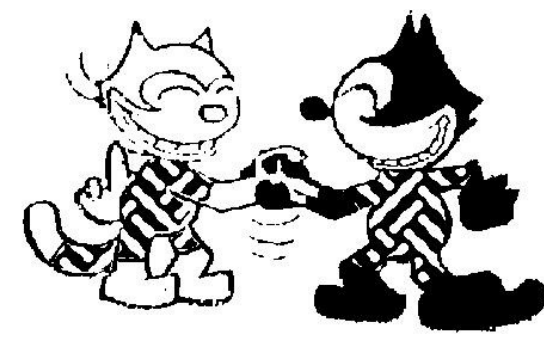
```
X = [1, 2]
```

```
ok
```

List matching provides
associative (A) matching



Set Matching



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

Set pattern

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

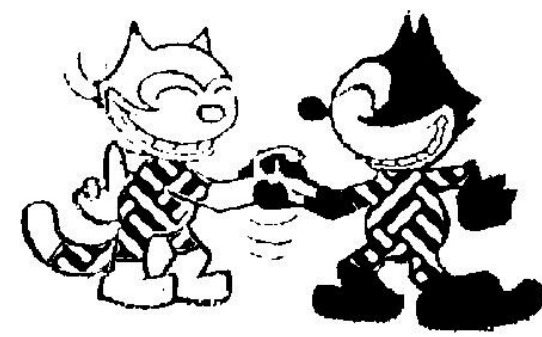
Y* is a set variable
and abbreviates
set[int] Y

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

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



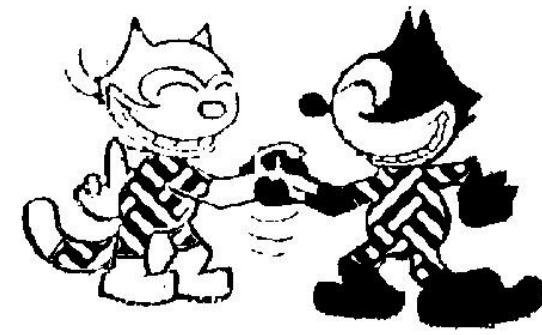
Note



- 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**



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
```



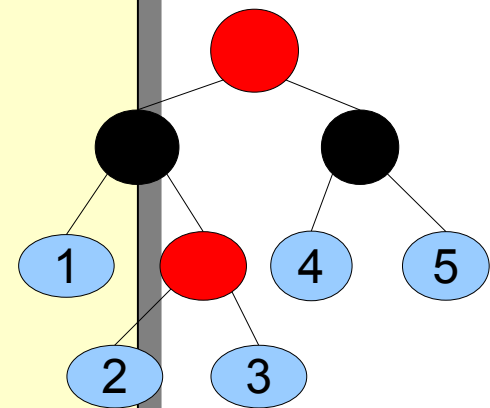
Comprehensions

- 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);}
```

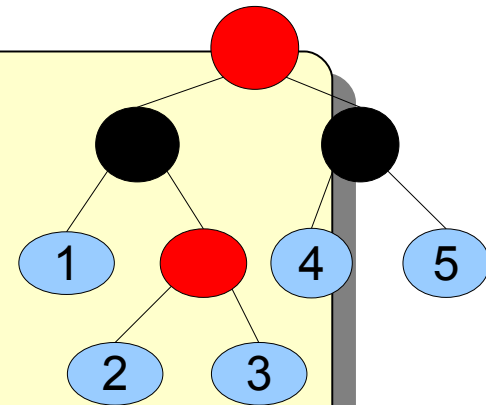
```
4
```

```
5
```

```
ok
```

```
rascal> for(/asgStat(Id name, _) ← P, size(name)>10){  
    println(name);  
}
```

```
...
```



Counting words in a string

```
public 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");  
}
```



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

