

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*



We believe in tools

Tools
versus
Diversity

=>

Make your own tools

=>

Work

=>






Rascal

=>

Less Work



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

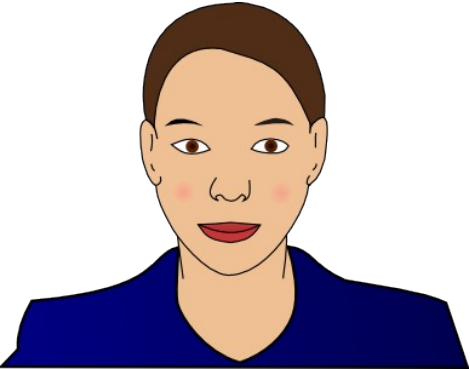


Meet Bernd



- 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



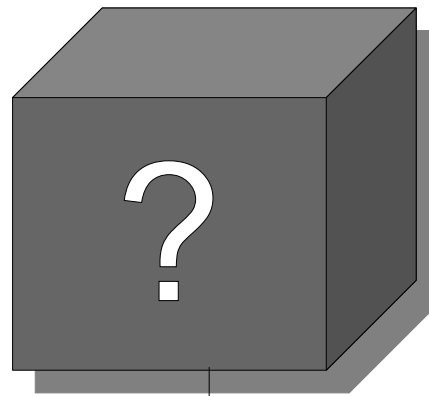
What are their *Technical Challenges?*

- 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?

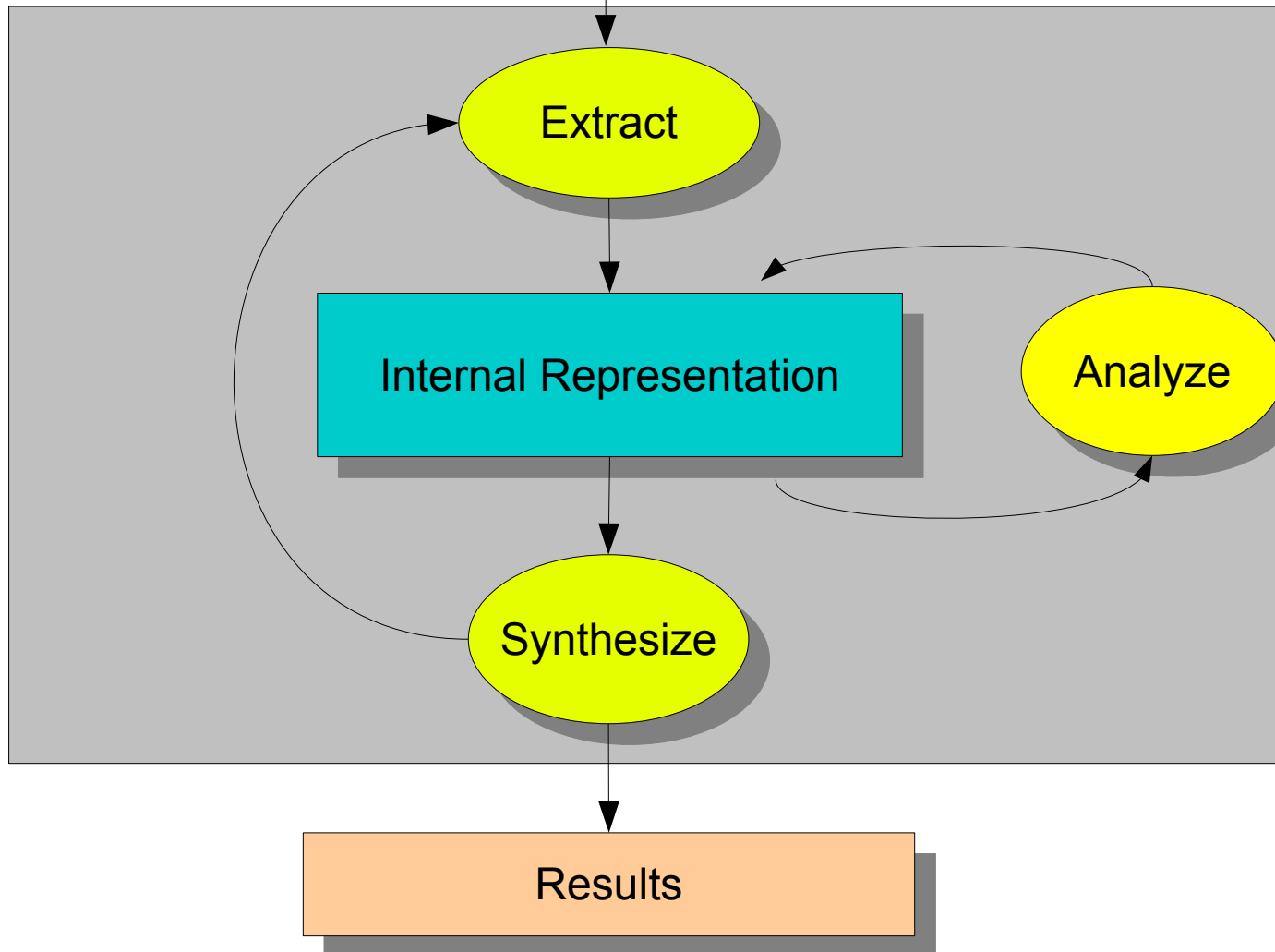


EASY: Extract-Analyze-SYnthesize Paradigm

System Under Investigation (SUI)

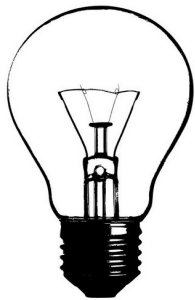


EASY Paradigm



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



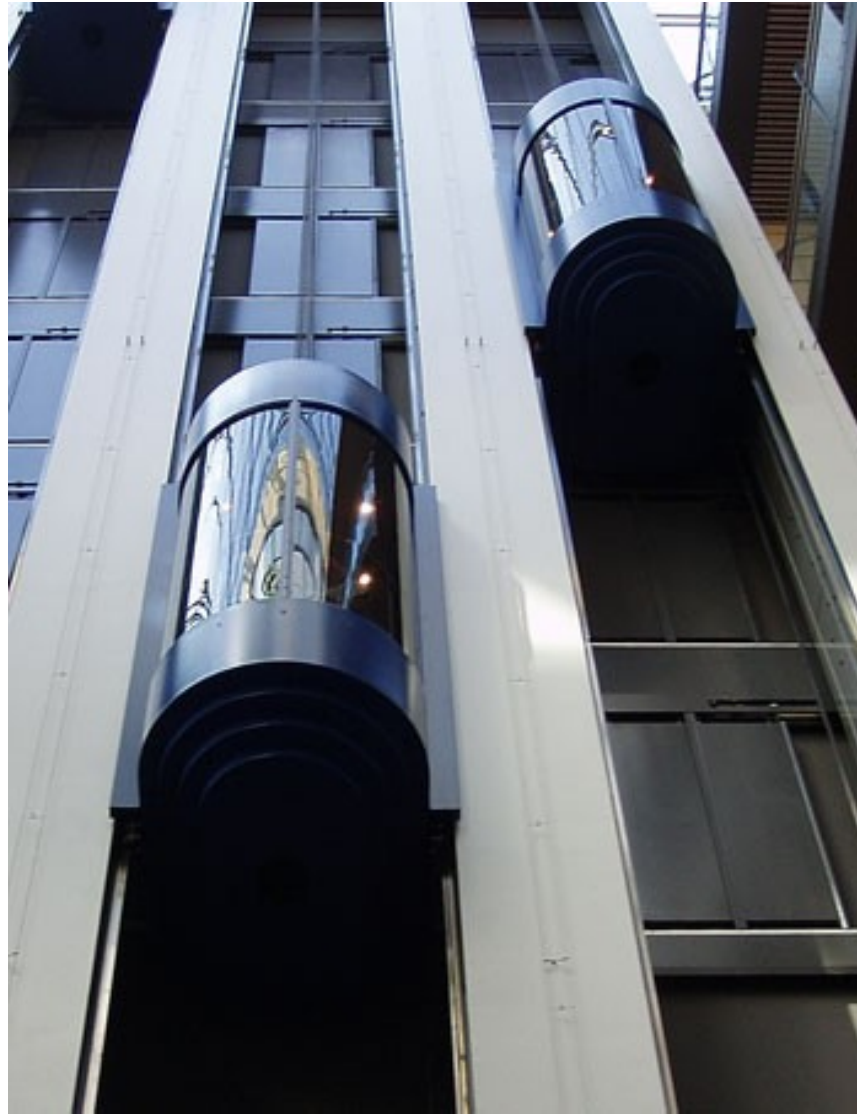
Goal

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



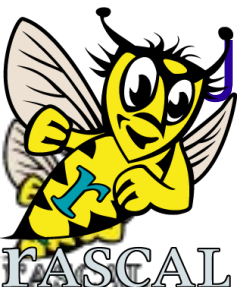
Rascal design based on ...

- **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

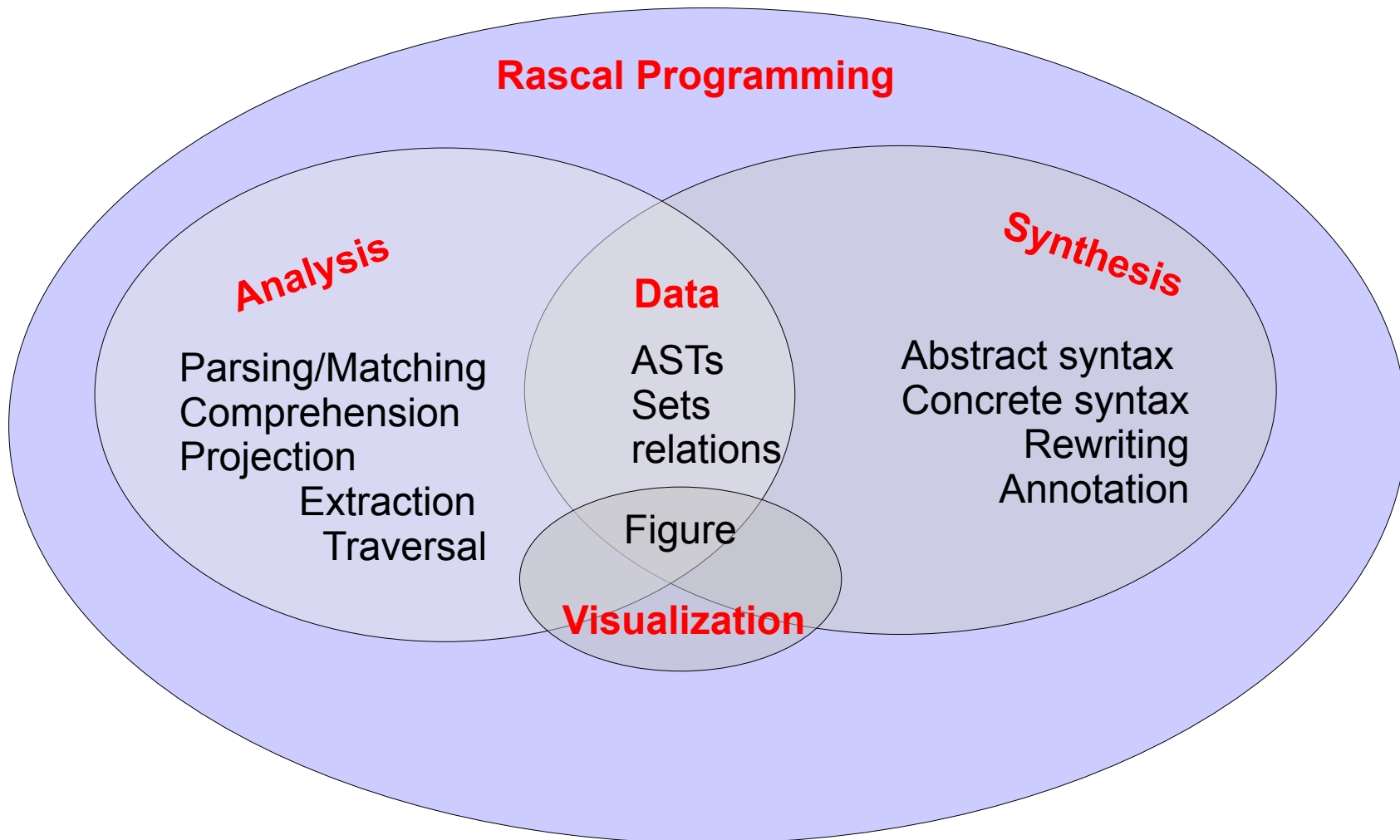


Rascal provides

- 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

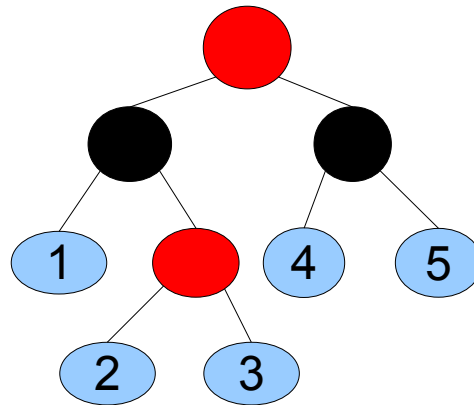
- Named alternatives
 - name acts as constructor
 - can be used in patterns
- Named fields (access/update via . notation)
- All datastructures are a subtype of the standard type node
 - Permits very generic operations on data
- Parse trees resulting from parsing source code are represented by the datatype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



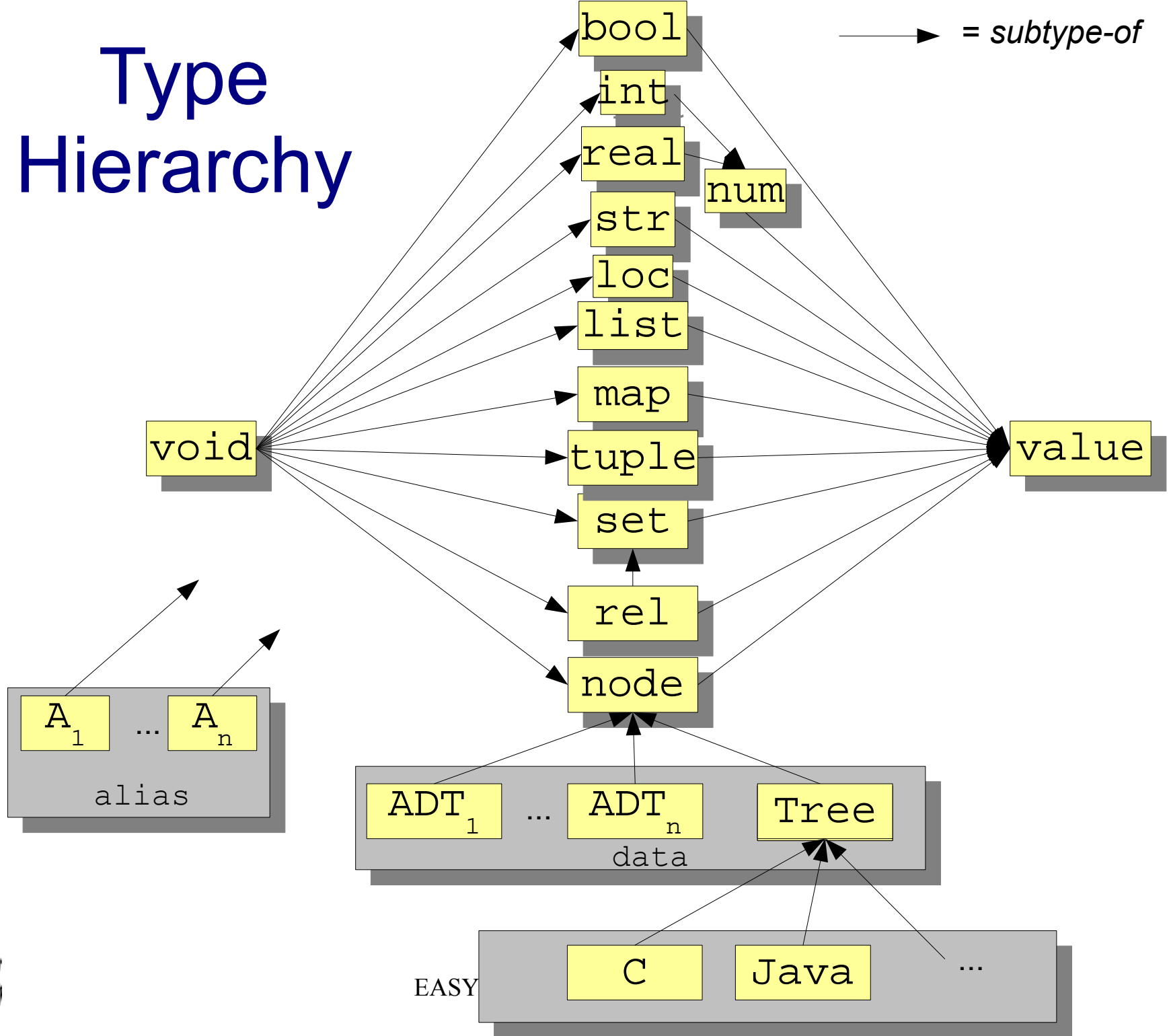
Abstract Syntax

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

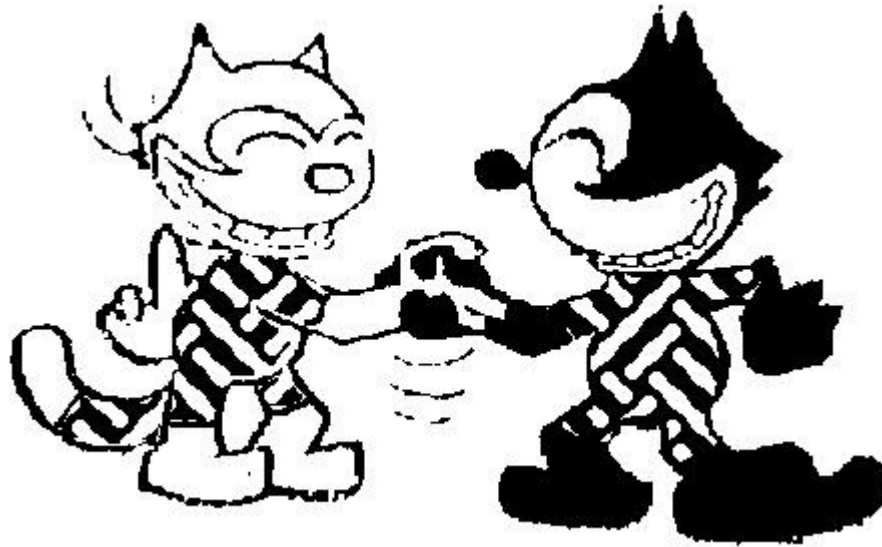


Type Hierarchy

→ = *subtype-of*



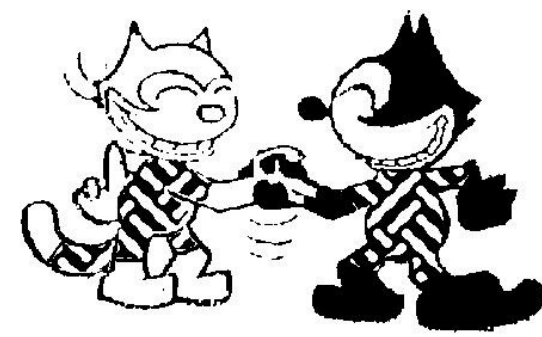
Pattern matching



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

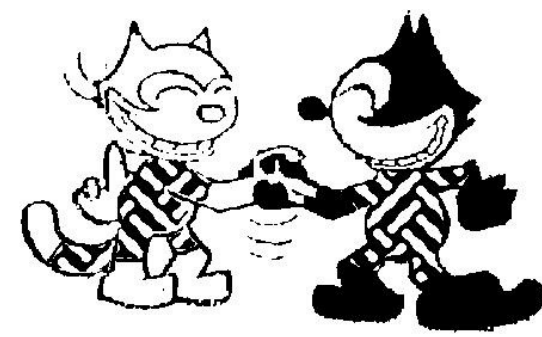


Pattern matching is used in:

- 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
```



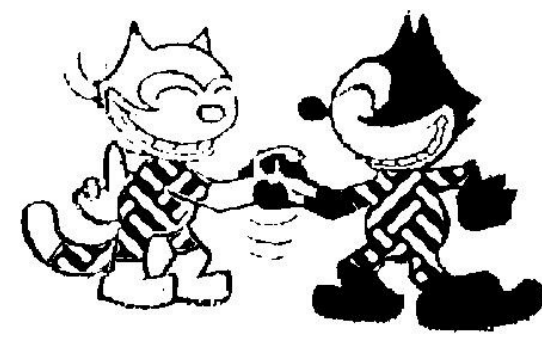
Regular Patterns

```
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>");
```

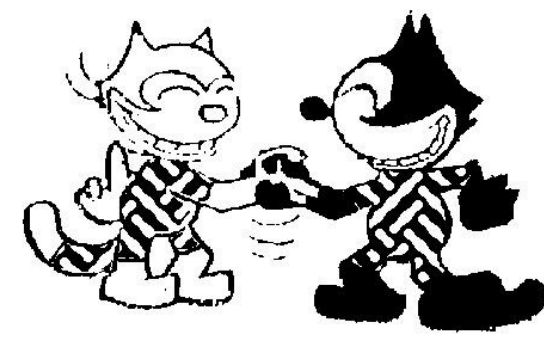
```
X = [1, 2]
```

```
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
```

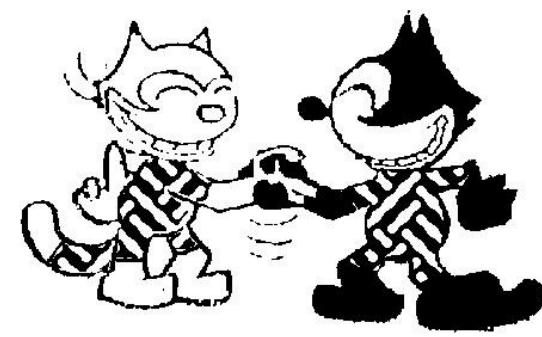
Y* is a set variable
and abbreviates
set[int] Y

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

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



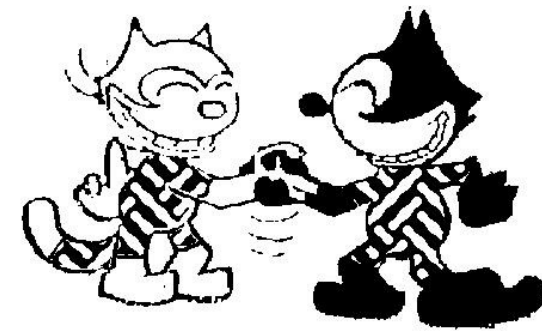
Note



- 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
```



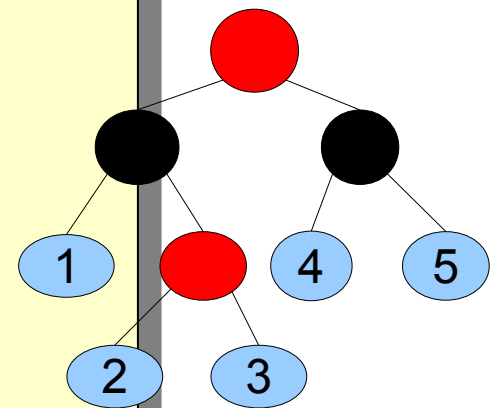
Comprehensions

- 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);}
```

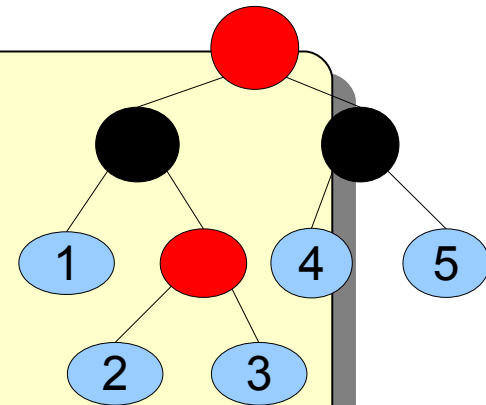
```
4
```

```
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!

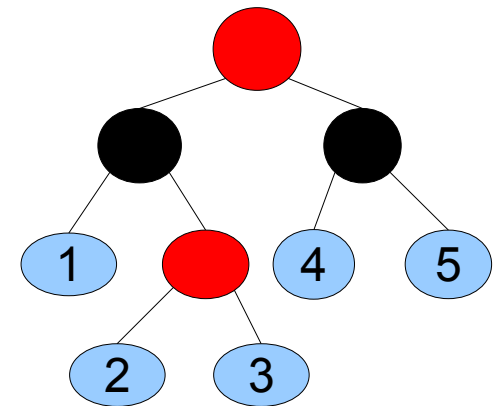
- Ok, that was quite a lot of information
- Rascal is for Meta-Programming
 - Code analysis
 - Code transformation
 - Code generation
 - Code visualization
- It is a normal programming language
- Learn it using the Tutor view and the Console



Visiting

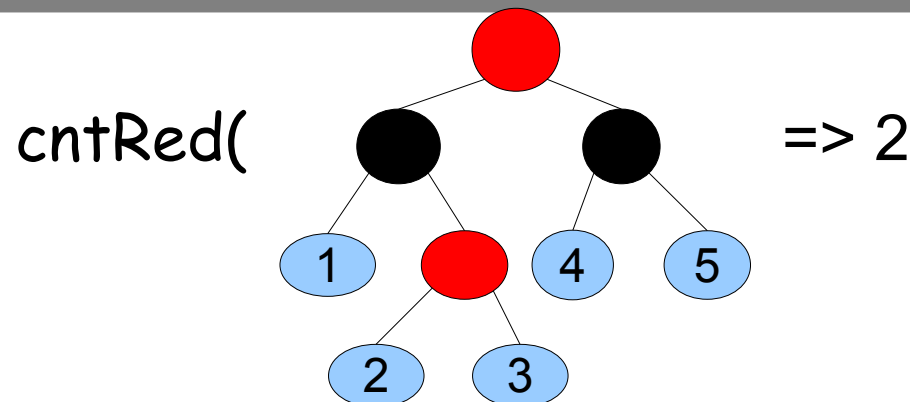
- Recall the **visitor design pattern**:
 - Decouples traversal, and
 - Action per visited node
- A **visit** does a complete traversal

Recall the coloured trees (*CTree*):



Count all Red Nodes (switch + recursion)

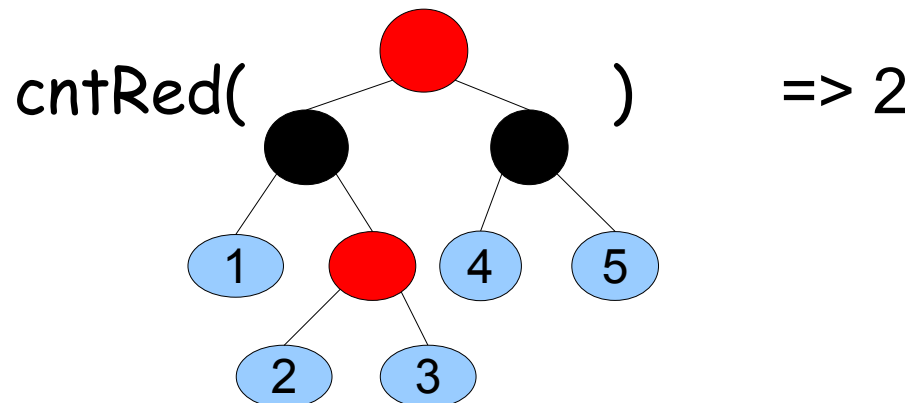
```
public int cntRed(CTree t) {  
    switch(t){  
        case leaf(_): return 0;  
        case red(l,r): return 1 + cntRed(l) + cntRed(r);  
        case black(l,r): return cntRed(l) + cntRed(r);  
    };  
}
```



Count all Red Nodes (using visit)

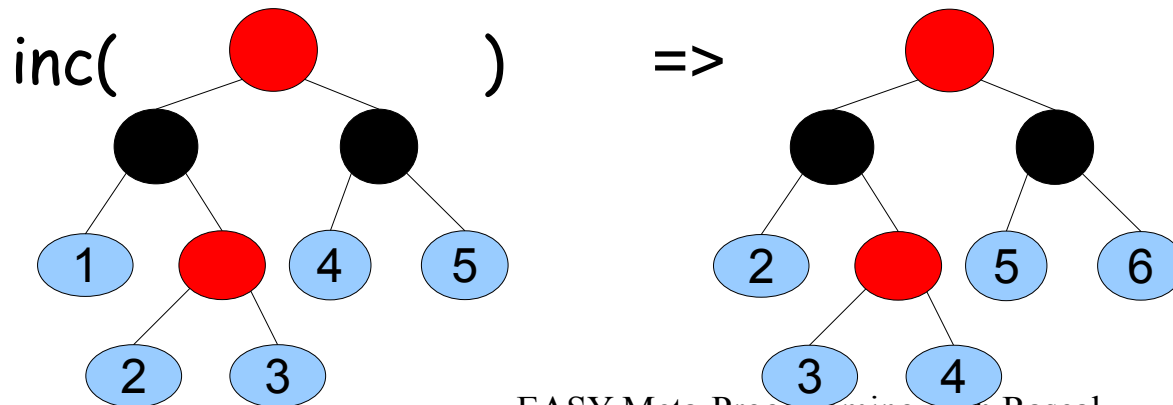
```
public int cntRed(CTree t) {  
    int c = 0;  
    visit(t){  
        case red(__,__): c += 1;  
    };  
    return c;  
}
```

Visit traverses the complete tree and modifies c



Increment all leaves in a CTree

```
public CTree inc(CTree T) {  
    switch(t){  
        case leaf(i): return leaf(i + 1);  
        case red(l,r): return red(inc(l),inc(r));  
        case black(l,r): return black(inc(l),inc(r));  
    };  
}
```

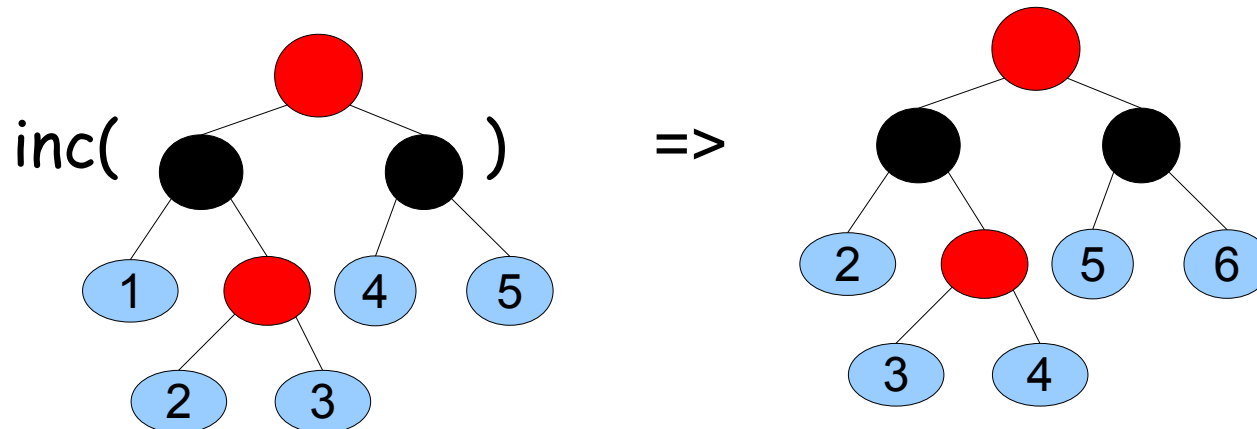


Increment all leaves in a CTree

```
public CTree inc(CTree T) {  
  return visit(T) {  
    case int N => N + 1;  
  };  
}
```

Visit traverses the complete tree and returns modified tree

Matching by cases and local subtree replacement



Note

- This code is insensitive to the number of constructors
 - Here 3: leaf, black and red
 - In Java or Cobol: hundreds
- Lexical/abstract/concrete matching
- List/set matching
- Visits can be parameterized with a strategy

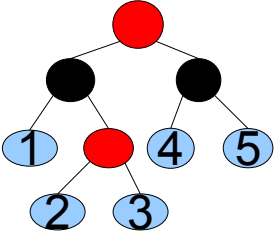


Let's add **green** nodes

```
data CTree green(CTree left, CTree right);
```

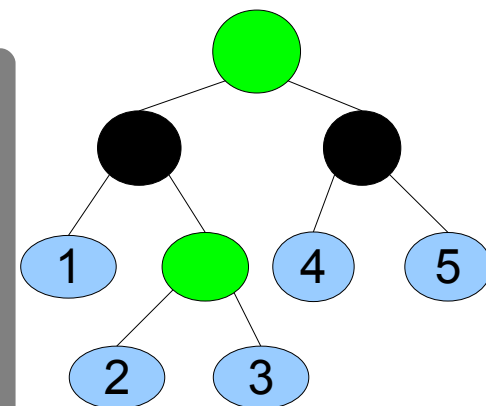
Problem: convert **red** nodes into **green** nodes



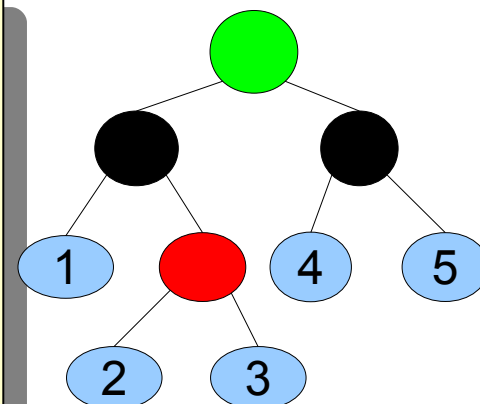


Full/shallow/deep replacement

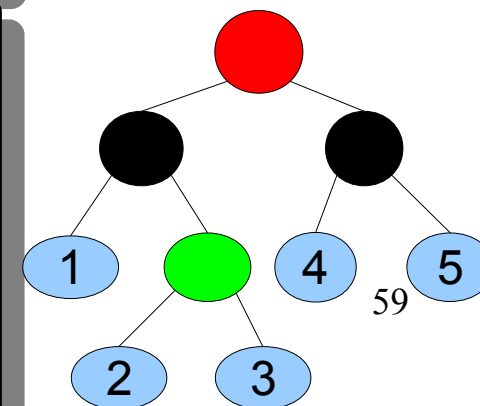
```
public CTree frepl(CTree T) {
    return visit (T) {
        case red(CTree T1, Ctree T2) => green(T1, T2)
    };
}
```



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



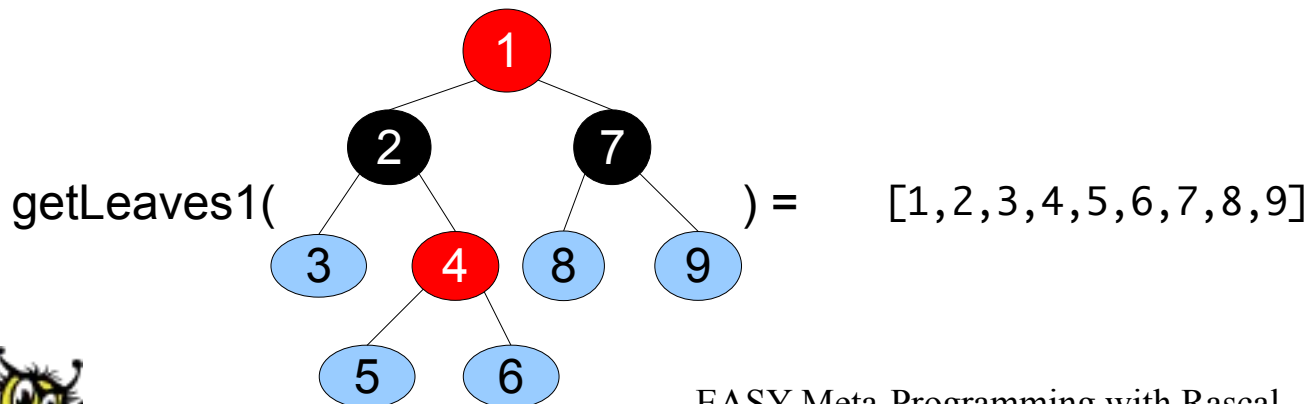
```
public Ctree drepl(Ctree T) {
    return bottom-up-break visit (T) {
        case red(Ctree T1, Ctree T2) => green(T1, T2)
    };
}
```



Different ways to Traverse a Tree, 1

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

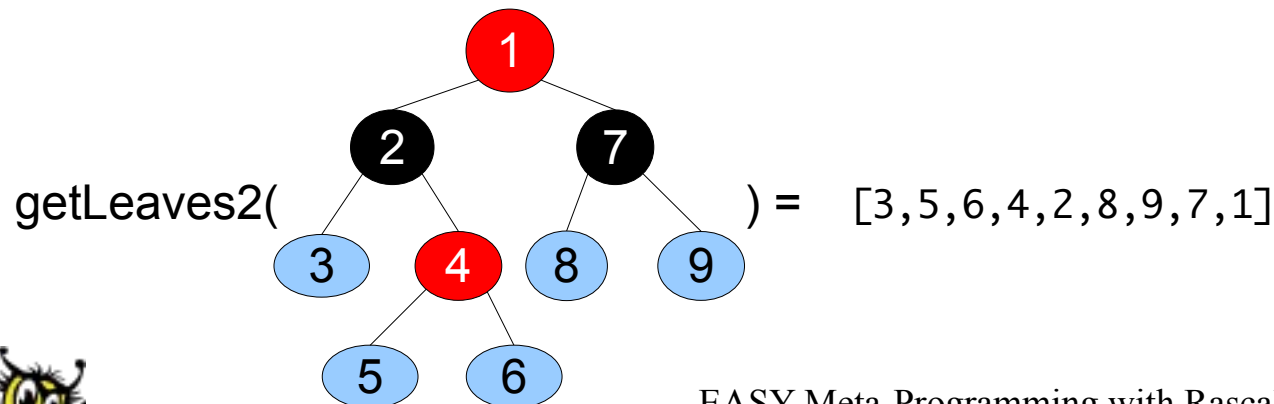
public list[int] getLeaves1(CTree t){
  switch(t) {
    case leaf(n):      return [n];
    case black(n,l,r): return [n] + getLeaves1(l) + getLeaves1(r);
    case red(n,l,r):   return [n] + getLeaves1(l) + getLeaves1(r);
  };
}
```



Different ways to Traverse a Tree,2

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

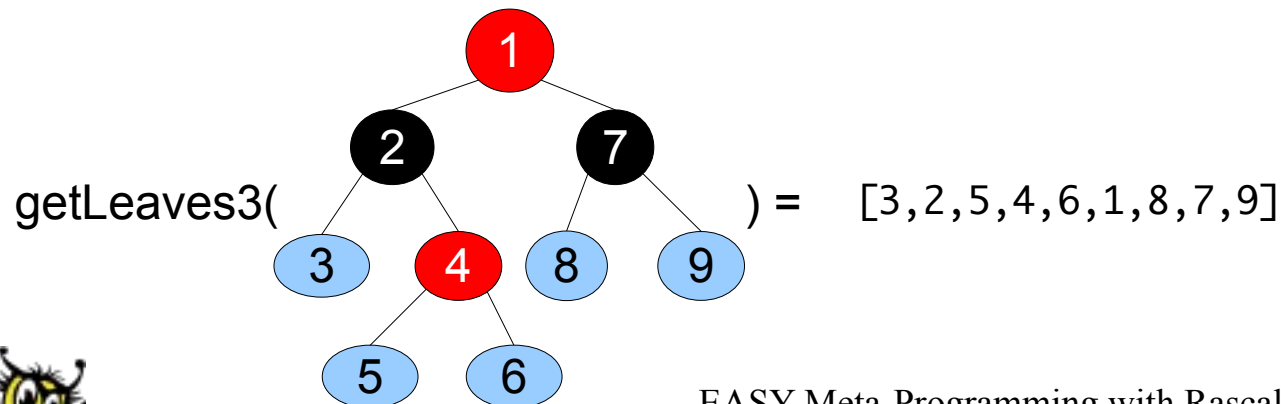
```
public list[int] getLeaves2(CTree t){
  switch(t) {
    case leaf(n):      return [n];
    case black(n,l,r): return getLeaves2(l) + getLeaves2(r) + [n];
    case red(n,l,r):   return getLeaves2(l) + getLeaves2(r) + [n];
  };
}
```



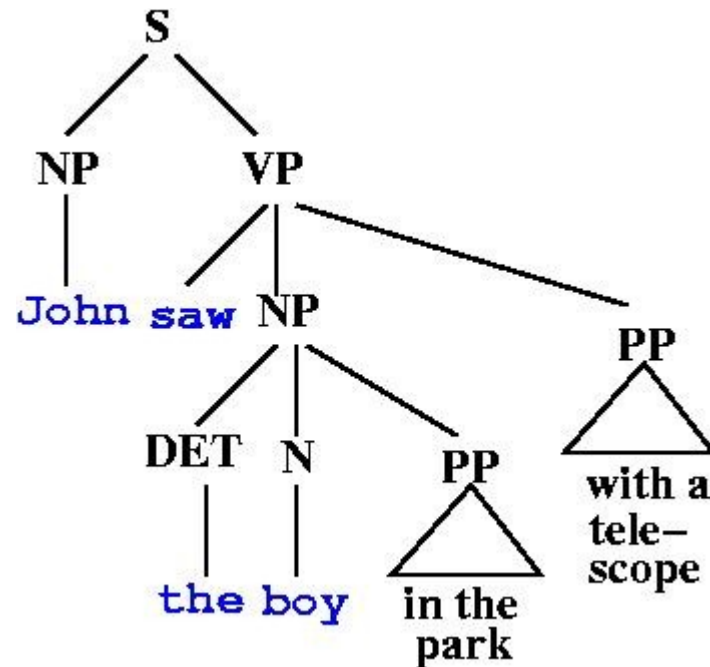
Different ways to Traverse a Tree,3

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

```
public list[int] getLeaves3(CTree t){
  switch(t) {
    case leaf(n):      return [n];
    case black(n,l,r): return getLeaves3(l) + [n] + getLeaves3(r);
    case red(n,l,r):   return getLeaves3(l) + [n] + getLeaves3(r);
  };
}
```



Syntax and Parsing



Given a grammar and a sentence find the structure of the sentence and discover its **parse tree**



Syntax and Parsing

- Uses a new formalism that is based on (and improves upon) the Syntax Definition Formalism (**SDF**)
- Modular grammar definitions
- Integrated lexical and context-free parsing
- A complete grammar can be imported and can be used for:
 - Parsing source code (parse functions)
 - Matching concrete code patterns
 - Synthesizing source code



Syntax for Exp

```
module demo::lang::Exp::Concrete::NoLayout::Syntax  
  
lexical IntegerLiteral = [0-9]+;  
  
start syntax Exp =  
    IntegerLiteral  
    | bracket "(" Exp "  
    > left Exp "*" Exp  
    > left Exp "+" Exp  
    ;
```



Example

Even numbers:
In many flavours

See Tutor: [Recipes/Basic/Even](#)

Even0: initial version

```
public list[int] even0(int max) {  
  list[int] result = [];  
  for (int i <- [0..max])  
    if (i % 2 == 0)  
      result += i;  
  return result;  
}
```

```
rascal>even0(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Even1: remove type declarations

```
public list[int] even0(int max) {  
  list[int] result = [];  
  for (int i <- [0..max])  
    if (i % 2 == 0)  
      result += i;  
  return result;  
}
```



Even1: remove type declarations

```
public list[int] even1(int max) {  
  result = [];  
  for (i <- [0..max])  
    if (i % 2 == 0)  
      result += i;  
  return result;  
}
```

```
rascal>even1(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Even2: merge for and if

```
public list[int] even1(int max) {  
  result = [];  
  for (i <- [0..max])  
    if (i % 2 == 0)  
      result += i;  
  return result;  
}
```



Even2: merge for and if

```
public list[int] even2(int max) {  
  result = [];  
  for (i <- [0..max], i % 2 == 0)  
    result += i;  
  return result;  
}
```

```
rascal>even2(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Even3: for returns the list (using append)

```
public list[int] even2(int max) {  
    result = [];  
    for (i <- [0..max], i % 2 == 0)  
        result += i;  
    return result;  
}
```



Even3: for returns the list (using append)

```
public list[int] even3(int max) {  
    result = for (i <- [0..max], i % 2 == 0)  
              append i;  
    return result;  
}
```

```
rascal>even3(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Even4: eliminate result variable

```
public list[int] even3(int max) {  
    result = for (i <- [0..max], i % 2 == 0)  
              append i;  
    return result;  
}
```



Even4: eliminate result variable

```
public list[int] even4(int max) {  
    return for (i <- [0..max], i % 2 == 0)  
        append i;  
}
```

```
rascal>even4(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Even5: use comprehension

```
public list[int] even4(int max) {  
  return for (i <- [0..max], i % 2 == 0)  
    append i;  
}
```



Even5: use comprehension

```
public list[int] even5(int max) {  
    return [i | i <- [0..max], i % 2 == 0];  
}
```

```
rascal>even5(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Even6: use abbreviated function declaration

```
public list[int] even5(int max) {  
  return [i <- [0..max], i % 2 == 0];  
}
```



Even6: use abbreviated function declaration

```
public list[int] even6(int max) = [i | i <- [0..max], i % 2 == 0];
```

```
rascal>even5(25);  
list[int]: [0,2,4,6,8,10,12,14,16,18,20,22,24]
```



Pattern-directed Invocation

- A conventional function has formal parameters:
 - `int factorial(int n) { ... }`
- In Rascal, also patterns can be used as formal parameters.
- At the call site, pattern matching determines which function to call => pattern-directed invocation.



Examples

Pattern-directed

Invocation:

99 bottles of beer

See Tutor: [Recipes/Basic/BottlesOfBeer](#)

99 Bottles of Beer

99 bottles of beer on the wall, 99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall.

98 bottles of beer on the wall, 98 bottles of beer.
Take one down, pass it around, 97 bottles of beer on the wall.

...

1 bottle of beer on the wall, 1 bottle of beer.
Take one down, pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.



99 Bottles of Beer

```
module demo::basic::Bottles
import IO;

str bottles(0)    = "no more bottles";
str bottles(1)    = "1 bottle";
default str bottles(int n) = "<n> bottles";

public void sing(){
  for(n <- [99 .. 1]){
    println("<bottles(n)> of beer on the wall, <bottles(n)> of beer.");
    println("Take one down, pass it around, <bottles(n-1)> of beer on the wall.\n");
  }
  println("No more bottles of beer on the wall, no more bottles of beer.");
  println("Go to the store and buy some more, 99 bottles of beer on the wall.");
}
```



Examples

Pattern-directed
Invocation:
derivatives

See Tutor: [Recipes/Common/Derivative](#)

Recall from Calculus:

The Derivative of a Function

- $dN / dX = 0$, for constant N
- $dX / dX = 1$
- $dX / dY = 0$, when $X \neq Y$
- $d(E1 + E2) / dx = d E1 / dX + d E2 / dX$
- $d(E1 * E2) / dX = (d E1 / dX * E2) + (E1 * d E2 / dX)$



Representing Expressions

```
data Exp = con(int n)
         | var(str name)
         | mul(Exp e1, Exp e2)
         | add(Exp e1, Exp e2)
         ;
```

```
public Exp E = add(mul(con(3), var("y")), mul(con(5), var("x")));
```

$3 * y + 5 * x$



Derivative in Rascal

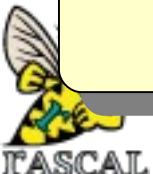
- $dN / dX = 0$, for constant N
- $dX / dX = 1$
- $dX / dY = 0$, when $X \neq Y$
- $d(E1 + E2) / dx = d E1 / dX + d E2 / dX$
- $d(E1 * E2) / dX = (d E1 / dX * E2) + (E1 * d E2 / dX)$

Exp $dd(\text{con}(n), \text{var}(V)) = \text{con}(0);$

Exp $dd(\text{var}(V1), \text{var}(V2)) = \text{con}((V1 == V2) ? 1 : 0);$

Exp $dd(\text{add}(\text{Exp } e1, \text{Exp } e2), \text{var}(V)) = \text{add}(dd(e1, \text{var}(V)), dd(e2, \text{var}(V)));$

Exp $dd(\text{mul}(\text{Exp } e1, \text{Exp } e2), \text{var}(V)) =$
 $\text{add}(\text{mul}(dd(e1, \text{var}(V)), e2), \text{mul}(e1, dd(e2, \text{var}(V))));$



But ...

```
rascal> dd(E, var("x"));
```

```
Exp: add(  
  add(  
    mul(  
      con(0),  
      var("y")),  
    mul(  
      con(3),  
      con(0))),  
  add(  
    mul(  
      con(0),  
      var("x")),  
    mul(  
      con(5),  
      con(1))))
```

We expect

$$d(3 * y + 5 * x) / dx$$

=

5

We need simplification!



Simplifying Expressions

Exp simp(add(con(n), con(m))) = con(n + m);

Exp simp(mul(con(n), con(m))) = con(n * m);

Exp simp(mul(con(1), Exp e)) = e;

Exp simp(mul(Exp e, con(1))) = e;

Exp simp(mul(con(0), Exp e)) = con(0);

Exp simp(mul(Exp e, con(0))) = con(0);

Exp simp(add(con(0), Exp e)) = e;

Exp simp(add(Exp e, con(0))) = e;

default Exp simp(Exp e) = e;

```
Exp simplify(Exp e){
  return bottom-up visit(e){
    case Exp e1 => simp(e1)
  }
}
```



Victory!

```
rascal>simplify(dd(E, var("x")));  
Exp: con(5)
```

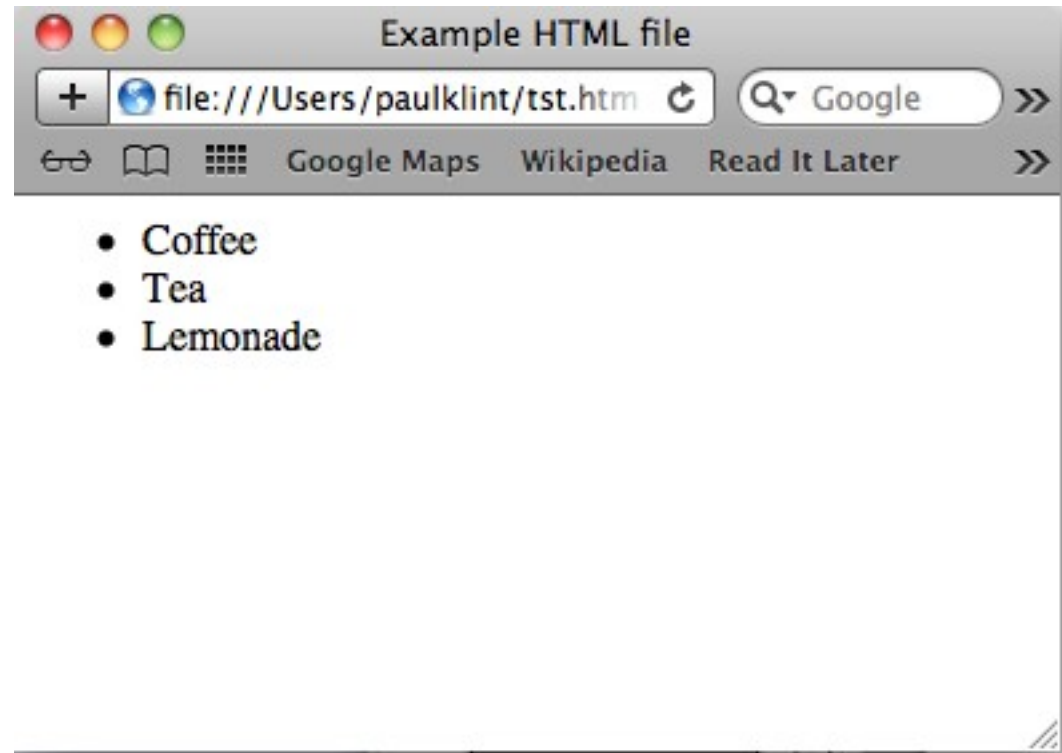


Example

Generating
HTML

Generating HTML

```
<html>
<title>Example HTML file</title>
<body>
<ul>
<li>Coffee</li>
<li>Thee</li>
<li>Lemonade</li>
</ul>
</body>
</html>
```



Generating Drinks Example

```
module HTML
import IO;

public str item(str op, str content) = "\<<op>\><content>\</<op>\>\n";

public str html(str title, str content) =
    item("html", item("title", title) + item("body", content));
public str ul(str content) = item("ul", content);
public str li(str content) = item("li", content);
```

```
rascal>item("li", "Coffee")
str: "\<li>Coffee\</li>\n"

rascal>println(item("li", "Coffee"))
<li>Coffee</li>
ok
```

```
rascal>println(html("ex1", ul(li("coffee") + li("tea"))))
<html><title>ex1</title>
<body><ul><li>coffee</li>
<li>tea</li>
</ul>
</body>
</html>
ok
```



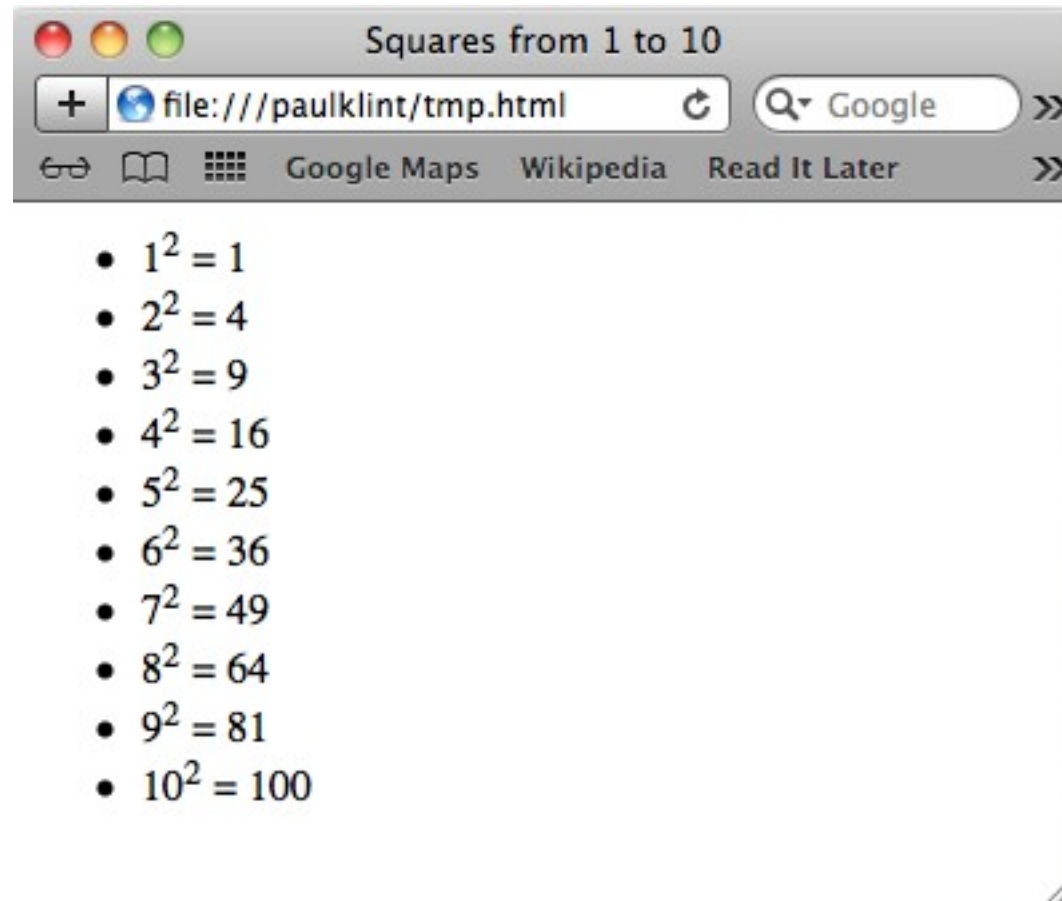
Saving to a file

```
rascal>writeFile(|file:///paulklint/tst.html|,  
                html("ex1", ul(li("coffee") + li("tea"))))  
ok
```

And load in your browser to see the effect ...



Exercise, generate squares



A Solution

```
module HTML

import IO;

public str item(str op, str content) = "\<<op>\><content>\</<op>\>\n";
public str html(str title, str content) =
    item("html", item("title", title) + item("body", content));
public str ul(str content) = item("ul", content);
public str li(str content) = item("li", content);

public str squared(int n) = li("<n><item(\"sup\", \"2\")> = <n*n>");

public str squares(int max) =
    html("Squares from 1 to <max>",
        ul("<for(int i <- [1 .. max])><squared(i)><}>"));
};

public void save(str name, str text){
    writeFile(|file:///paulklint/| + name, text);
}

public str ex1() = html("ex1", ul(li("coffee") + li("tea")));
```



Example

Job interview:
FizzBuzz!

A test from job interviews

Write a program that prints the numbers from 1 to 100.

But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz".

For numbers which are multiples of both three and five print "FizzBuzz".

Surprisingly: a substantial amount of applicants fails!



Exercise: write your fizzbuzz

```
rascal>fizzbuzz();  
1  
2  
Fizz  
4  
Buzz  
Fizz  
7  
8  
Fizz  
Buzz  
11  
Fizz  
13  
14  
FizzBuzz  
...
```



FizzBuzz Solutions

```
public void fizzbuzz() {
  for(int n <- [1 .. 100]){
    fb = ((n % 3 == 0) ? "Fizz" : "") + ((n % 5 == 0) ? "Buzz" : "");
    println((fb == "") ? "<n>" : fb);
  }
}
```

```
public void fizzbuzz2() {
  for (n <- [1..100])
    switch(<n % 3 == 0, n % 5 == 0> {
      case <true,true> : println("FizzBuzz");
      case <true,false> : println("Fizz");
      case <false,true> : println("Buzz");
      default: println(n);
    }
}
```

```
public void fizzbuzz3() {
  for (n <- [1..100]) {
    if (n % 3 == 0) print("Fizz");
    if (n % 5 == 0) print("Buzz");
    else if (n % 3 != 0) print(n);
    println("");
  }
}
```



Example

Generating
getters
and
setters

Generating Getters and Setters (1)

- Given:
 - A class name
 - A mapping from names to types

Required:

- Generate the named class with getters and setters



Input

```
public map[str, str] fields = (  
  "name" : "String",  
  "age" : "Integer",  
  "address" : "String"  
);
```

Field name of type String

Field age of type Integer

Field address of type String

```
genClass("Person", fields)
```

Generate class person
with these fields



Expect Output

```
public class Person {  
  
    private Integer age;  
    public void setAge(Integer age) { this.age = age; }  
    public Integer getAge() { return age; }  
  
    private String name;  
    public void setName(String name) { this.name = name; }  
    public String getName() { return name; }  
  
    private String address;  
    public void setAddress(String address) { this.address = address; }  
    public String getAddress() { return address; }  
}
```



Generating Getters and Setters

```
public str genClass(str name, map[str,str] fields) {  
  return "
```

String with computed interpolations

```
  public class <name> {
```

```
    <for (x <- fields) {
```

Red is interpolated

```
      str t = fields[x];
```

```
      str n = capitalize(x);>
```

Blue is literal

```
      private <t> <x>;
```

```
      public void set<n>(<t> <x>) { this.<x> = <x>; }
```

```
      public <t> get<n>() { return <x>; }
```

```
    <>>
```

```
  }
```

```
};
```

```
}
```



Generating Getters and Setters

```
public str genClass(str name, map[str,str] fields) {
```

```
  return "
```

String with computed interpolations

```
  'public class <name> {
```

```
  ' <for (x <- fields) {
```

Red is interpolated

```
  '   str t = fields[x];
```

```
  '   str n = capitalize(x);>
```

Blue is literal

```
  ' private <t> <x>;
```

```
  ' public void set<n>(<t> <x>) { this.<x> = <x>; }
```

```
  ' public <t> get<n>() { return <x>; }
```

```
  ' <>>
```

```
  ' }
```

```
  " ,
```

Text before ' is ignored

```
  }
```



Other features

- Solve equations using fixed point iteration
- Parser generation from syntax definitions
- Get/set fields of ADTs
- Exception handling
- Annotations
- Parameterized types
- Higher order functions
- Many libraries ...



Information

See:

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



Join Us in Creating Something Beautiful

- Feedback α -version
- Criticism on design
- Suggest additions
- Case studies
- Tool support
- Tutorials

