

# EASY Meta-Programming with Rascal

*Leveraging the Extract-Analyze-SYNthesize Paradigm*

Paul Klint



UNIVERSITEIT VAN AMSTERDAM



INSTITUT NATIONAL  
DE RECHERCHE  
EN INFORMATIQUE  
ET EN AUTOMATIQUE








centre de recherche LILLE - NORD EUROPE

*Joint work with: Emilie Balland, Bas Basten, Arnold  
Lankamp, Tijs van der Storm, Jurgen Vinju*



# 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

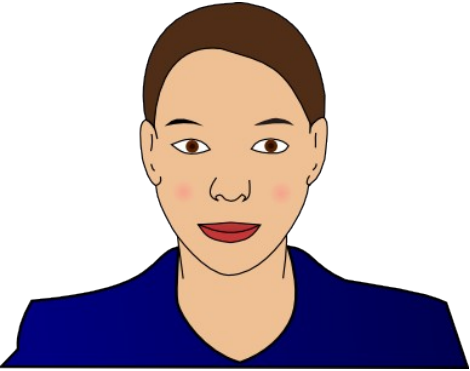


# Meet Bernd



- **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 Common Problems?

- How to parse source code?
- How to extract facts from it?
- How to perform computations on these facts?
- How to generate new source code?
- How to synthesize other information?
- How to do meta-programming?

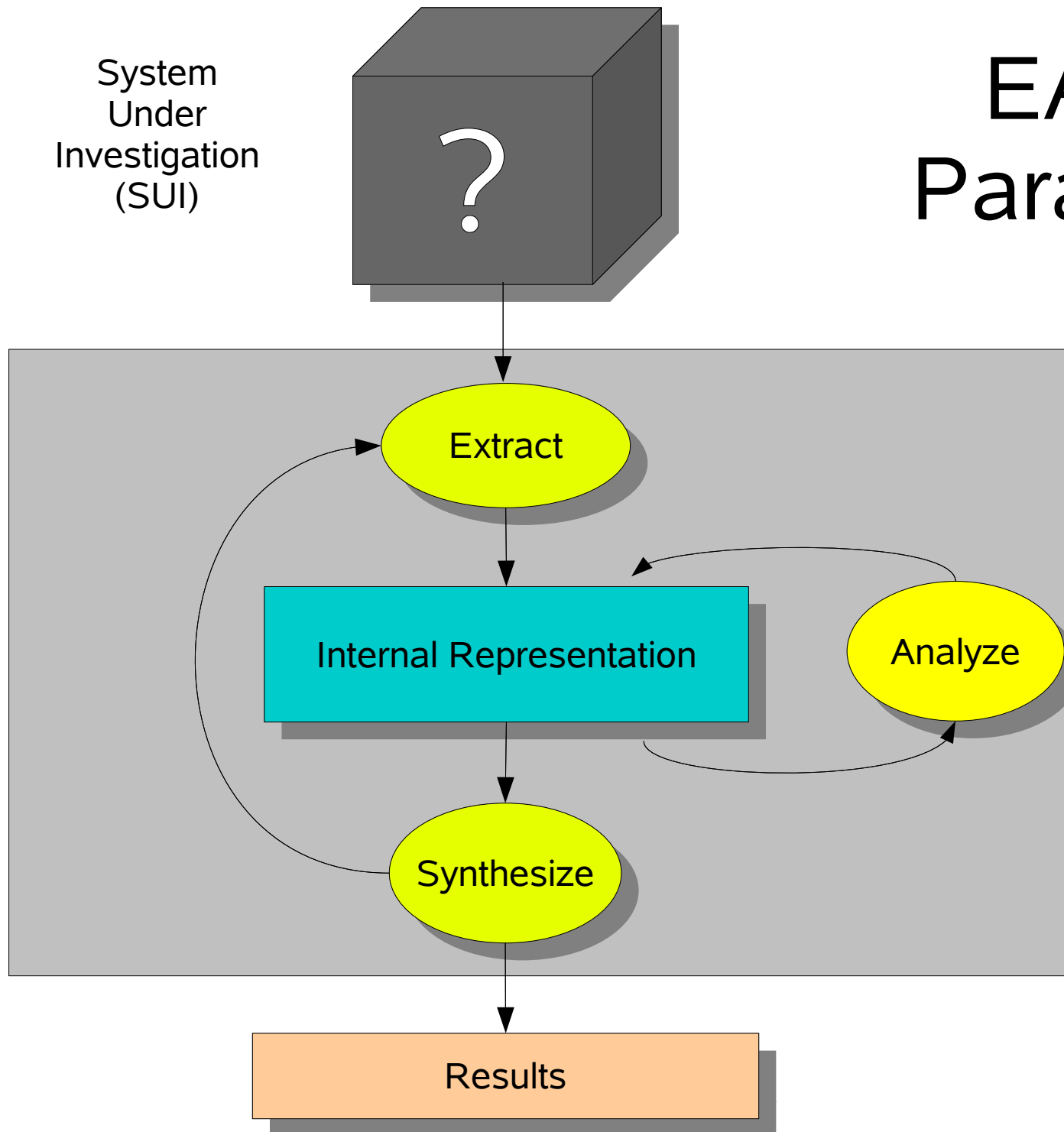


**EASY: Extract-Analyze-SYnthesize Paradigm**



System  
Under  
Investigation  
(SUI)

# EASY Paradigm



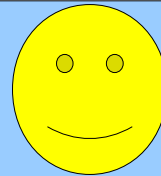
# What tools are available to our heroes?

- **Lexical tools**: Grep, Awk; also Perl, Python, Ruby
  - Regular expressions have limited expressivity
  - Hard to maintain
- **Compiler tools**: yacc, bison, CUP, ANTLR
  - Only automate front-end part
  - Everything else programmed in C, Java, ..
- **Attribute Grammar tools**: FNC2, Eli, JastAdd, ...
  - Only analysis, no transformation



# What Tools are Available to our heroes?

- **Relational Analysis tools**: Grok, Rscript
  - Strong in analysis
- **Transformation tools**: ASF+SDF, Stratego, TOM, TXL
  - Strong in transformation
- **Logic languages**: Prolog
- Many others ...



Apologies if  
your favorite tool  
does not  
appear in this list



	Extract	Analyze	Synthesize
Lexical tools	++	+/-	--
Compiler tools	++	+/-	+/-
Attribute grammar tools	++	+/-	--
Relational tools	--	++	--
Transformation tools	--	+/-	++
Logic languages	+/-	+/-	+/-
<b>Rascal</b>	++	++	++

Some *<code snippets>*  
to illustrate our  
ASF+SDF  
and  
Rscript  
background

ASF+SDF <*snippets*>

*just to give you an idea*

# Boolean Constants (in ASF+SDF)

```
module basic/BoolCon
```

```
exports
```

```
  sorts BoolCon
```

```
  context-free syntax
```

```
    "true"  -> BoolCon
```

```
    "false" -> BoolCon
```

The sort of Boolean constants

The constants **true** and **false**



# Booleans (2)

Boolean "|" Boolean -> Boolean {left}  
Boolean "&" Boolean -> Boolean {left}  
"not" (Boolean) -> Boolean  
"(" Boolean ")" -> Boolean {bracket}

context-free priorities

Boolean "&" Boolean -> Boolean >  
Boolean "|" Boolean -> Boolean

variables

"Bool"[0-9\']\* -> Boolean

The infix operators and & and or |.  
Both are left-associative (**left**)

The prefix function **not**

( and ) may be used as brackets in  
Boolean expressions;  
they are ignored after parsing

**Shorthand** for defining the infix  
operators and & and or |.

Both are left-associative (**left**).  
These rules are promoted to  
context-free syntax rules





# Booleans (3)

equations

[B1] true | Bool = true  
[B2] false | Bool = Bool  
[B3] true & Bool = Bool  
[B4] false & Bool = false  
[B5] not ( false ) = true  
[B6] not ( true ) = false

The meaning of **&**, **|** and **not** operators.

**Observation:** the syntax of equations is not fixed but depends on the syntax definition of the functions.



# Fixed versus user-defined syntax

equations

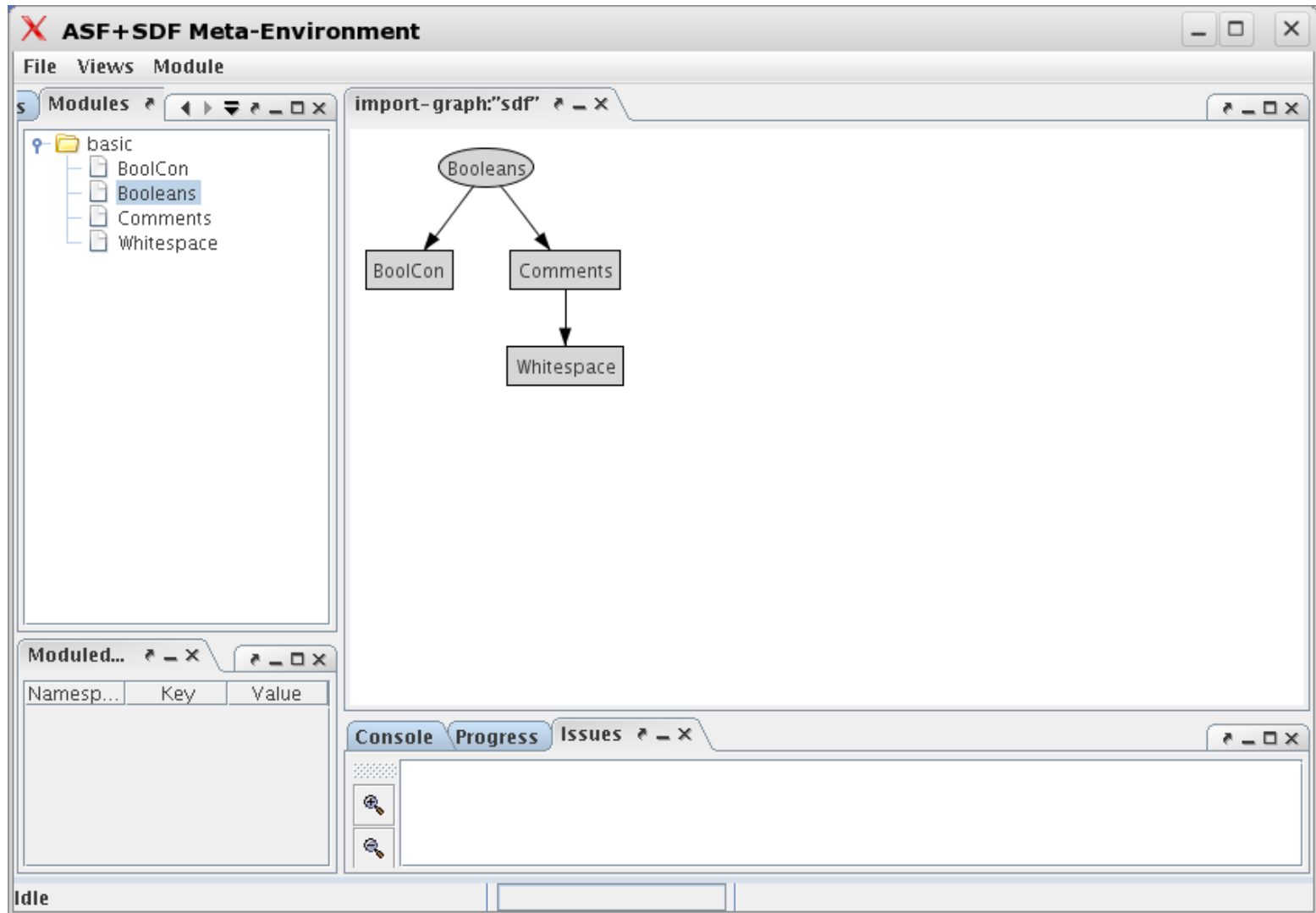
```
[B1] true | Bool = true  
[B2] false | Bool = Bool  
[B3] true & Bool = Bool  
[B4] false & Bool = false  
[B5] not ( false ) = true  
[B6] not ( true ) = false
```

Skeleton syntax for equations

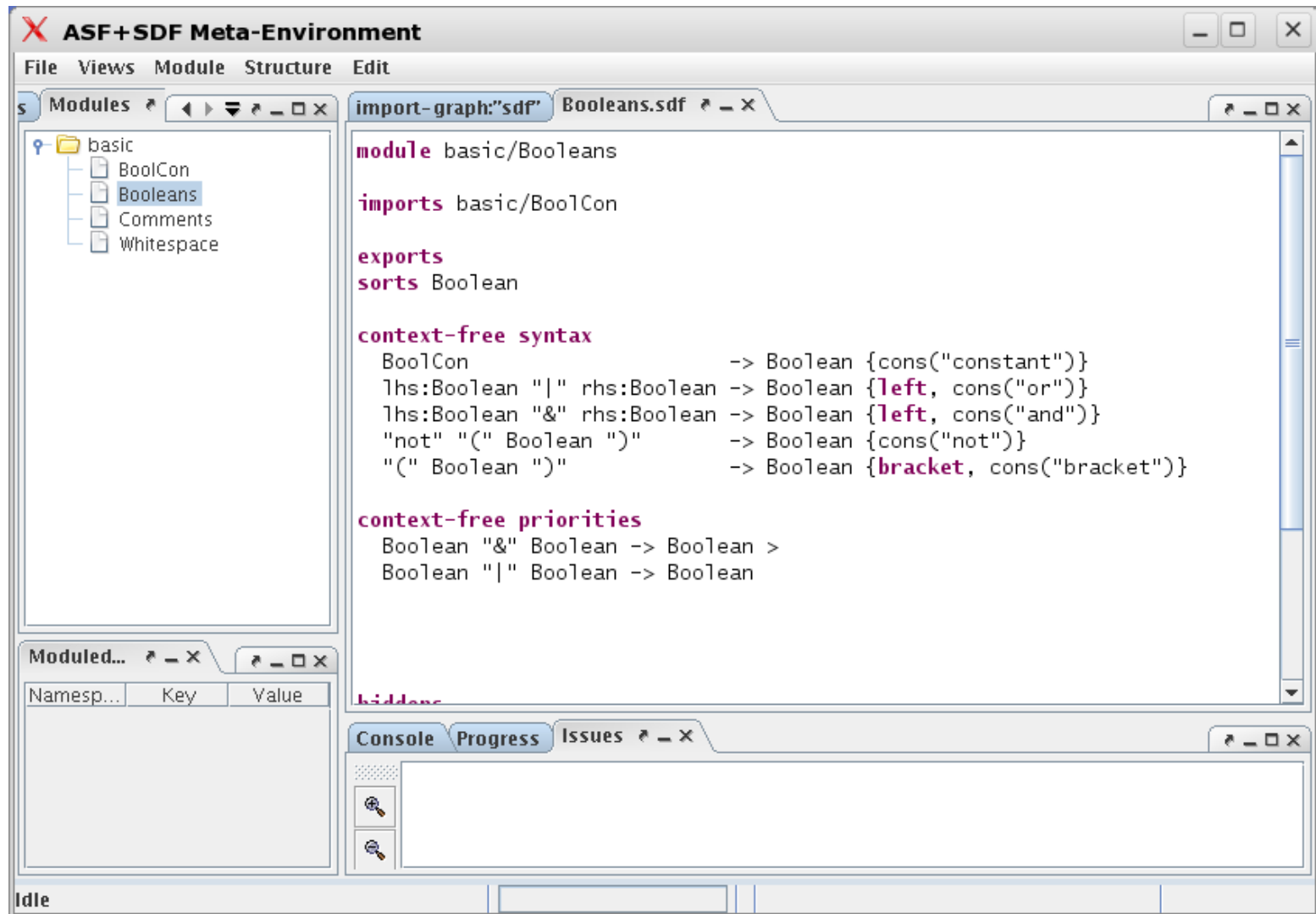
Terms that use user-defined syntax



# Opening Booleans in the Meta-Environment



# Editing Booleans



# Points to Ponder

- Don't be misled by this simple example!
- *“Every module defines a language”* scales from Booleans to Java
- SDF is being used to define the grammar of real languages (COBOL, Java, C, ...)
- ASF+SDF has been used for real mass transformations, defining DSLs, ...



# Example: Cobol transformation

```
module End-If-Trafo
```

```
imports Cobol
```

```
exports
```

```
context-free syntax
```

```
  addEndIf(Program)-> Program {traversal(trafo,continue,top-down)}
```

```
variables
```

```
"Stats"[0-9]*      -> StatsOptIfNotClosed
```

```
"Expr"[0-9]*       -> L-exp
```

```
"OptThen"[0-9]*    -> OptThen
```

```
equations
```

```
[1] addEndIf(IF Expr OptThen Stats) =  
      IF Expr OptThen Stats END-IF
```

```
[2] addEndIf(IF Expr OptThen Stats1 ELSE Stats2) =  
      IF Expr OptThen Stats1 ELSE Stats2 END-IF
```

- Add missing END-IF keywords

Impossible to do with regular expression tools like **grep** since conditionals can be nested

- Equations for the two cases

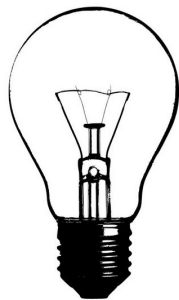


# Lessons



## **Lesson**

Using concrete syntax in transformations avoids the need to define/remember/use hundreds of abstract syntax constructors



## **Lesson**

Every module defines a language with concrete syntax and semantics.  
This scales from Booleans to Cobol/Java!



# Lessons



## **Lesson**

Tight integration between concrete syntax and term rewriting is great but it is an all-or-nothing experience for students



# Rscript

A simple relational calculus language with

- Sets/relations
- Functions, Comprehensions
- Built-in operators: sets, lists
- Built-in functions: domain, range, ...

Used for:

- Analysis of extracted facts from Java, C.



# Rscript: examples

- `rel[str,int] U = {<"y",3>, <"x",3>, <"z", 5>}`

- `int Usize = #U`

- 3

- `rel[int,str] Uinv = inv(U)`

- `{<3, "y">, <3, "x">, <5, "z">}`

- `set[str] Udom = domain(U)`

- `{"y", "x", "z"}`

**domain:**

all elements in lhs of pairs

**range:**

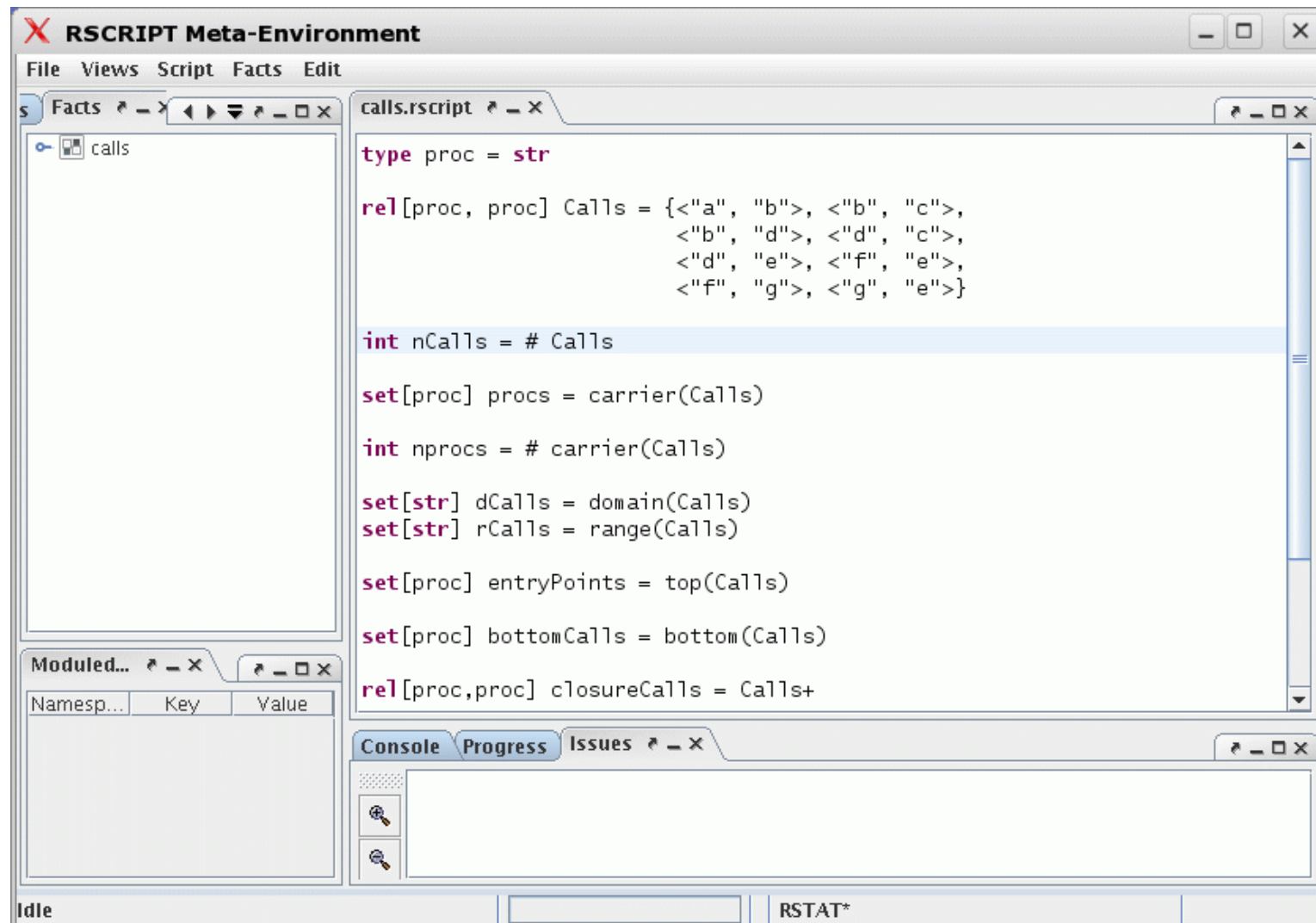
all elements in rhs of pairs

**carrier:**

all elements in lhs or rhs  
of pairs



# Rscript IDE (based on Meta-Environment)



# Lessons



**Lesson**  
Relational calculus is great for  
fact *analysis*

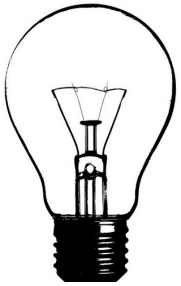


**Lesson**  
Fact *extraction* remains difficult and  
becomes a bottleneck



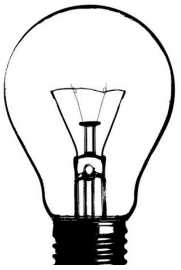
End of  
ASF+SDF  
and  
Rscript  
<*snippets*>

# Lessons



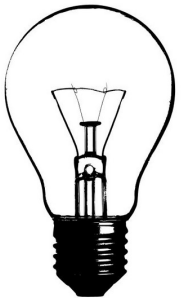
## **Lesson**

SDF-like definitions are essential for defining syntax of programming languages/DSLs



## **Lesson**

Rewrite rules are excellent for transformation



## **Lesson**

Concrete syntax helps simplifying rules



# Where is our ASF+SDF and Rscript background applicable?

	Extract	Analyze	Synthesize
ASF: rewrite rules	--	+/-	++
SDF: grammar rules	++	+/-	--
Rscript: relational calculus	--	++	--

# 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

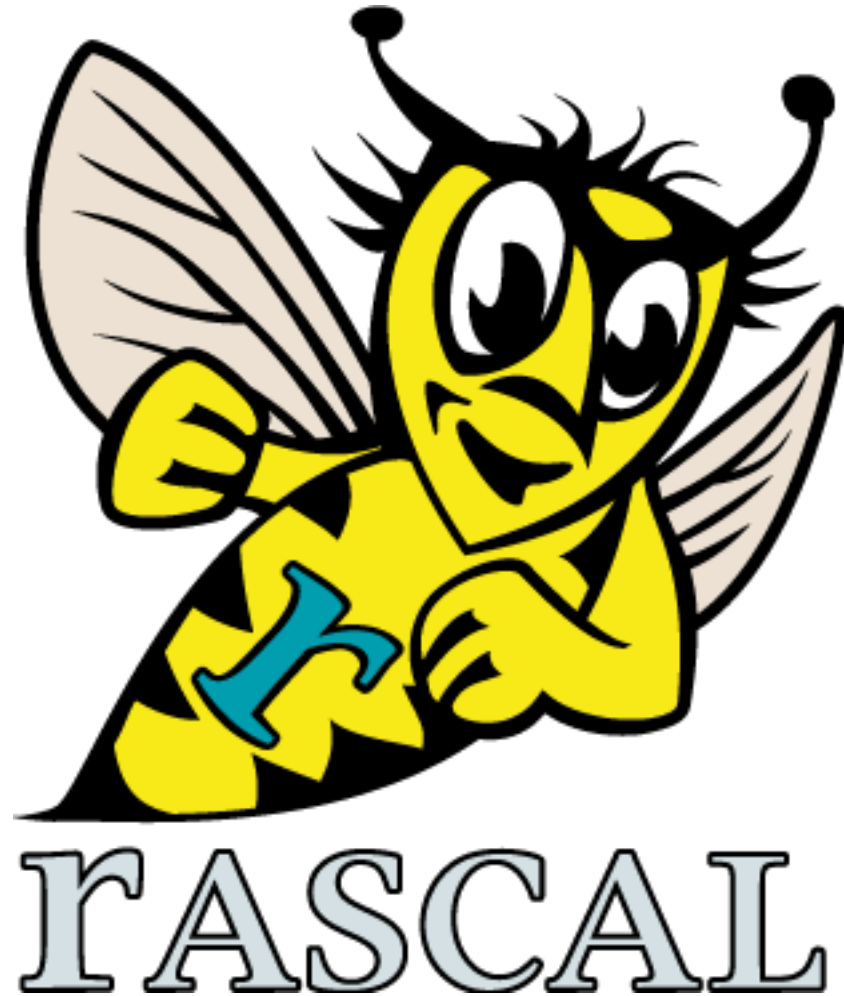


## Goal

Keep all benefits of ASF+SDF and Rscript in a **new, unified, extensible, teachable** framework



# 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



# Design Guidelines

- Principle of least surprise
  - Familiar syntax
  - Imperative core
- What you see is what you get
  - No heuristics (or as few as possible)
  - Explicitness over implicitness
- Learnability
  - Layered design
  - Low barrier to adoption

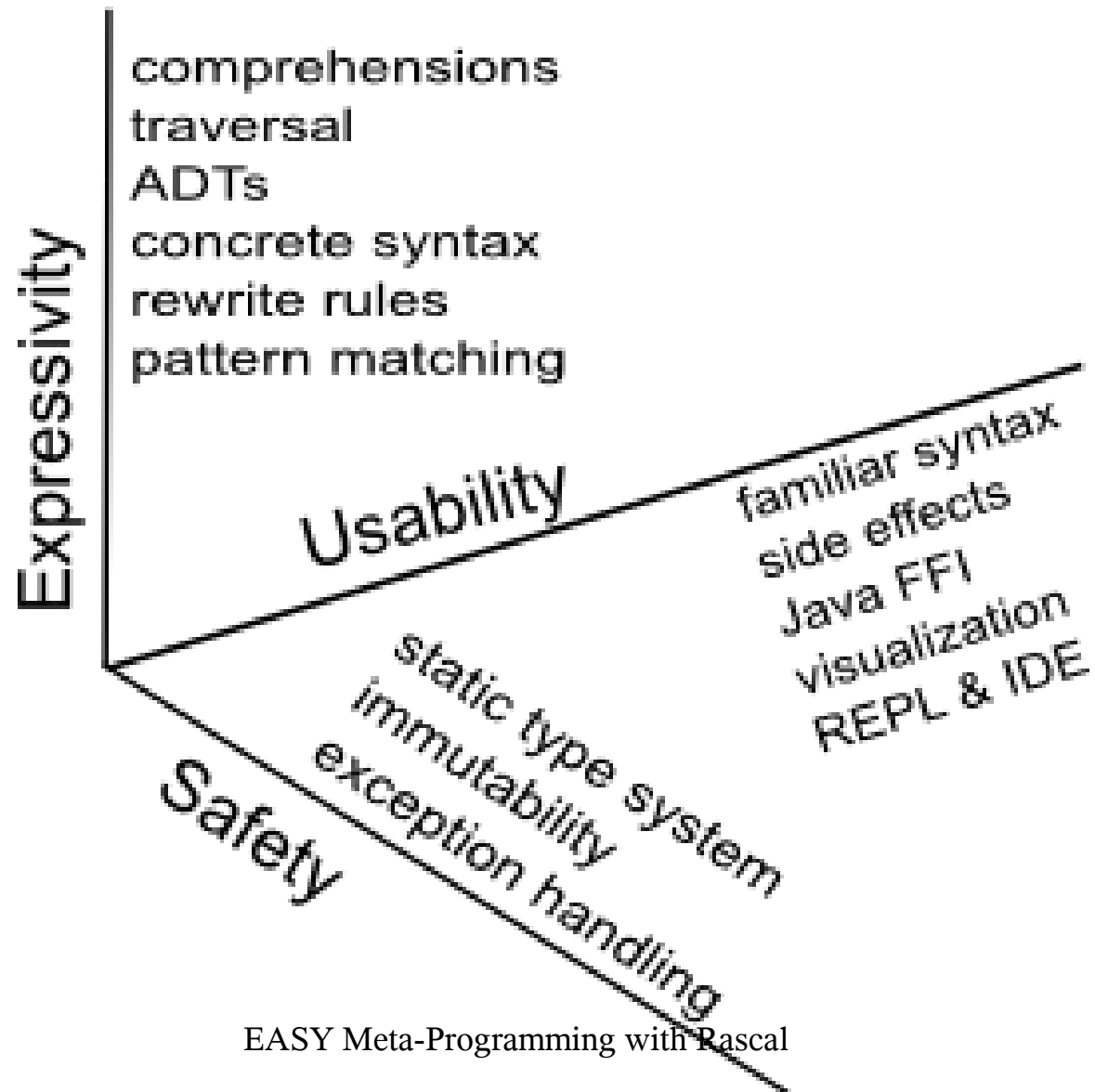


# Rascal Concepts

- Values and Types
- Data structures
- Syntax and Parsing
- Pattern Matching
- Enumerators
- Comprehensions
- Control structures
- Switching
- Visiting
- Functions
- Rewrite rules
- Constraint solving
- Typechecking
- Execution

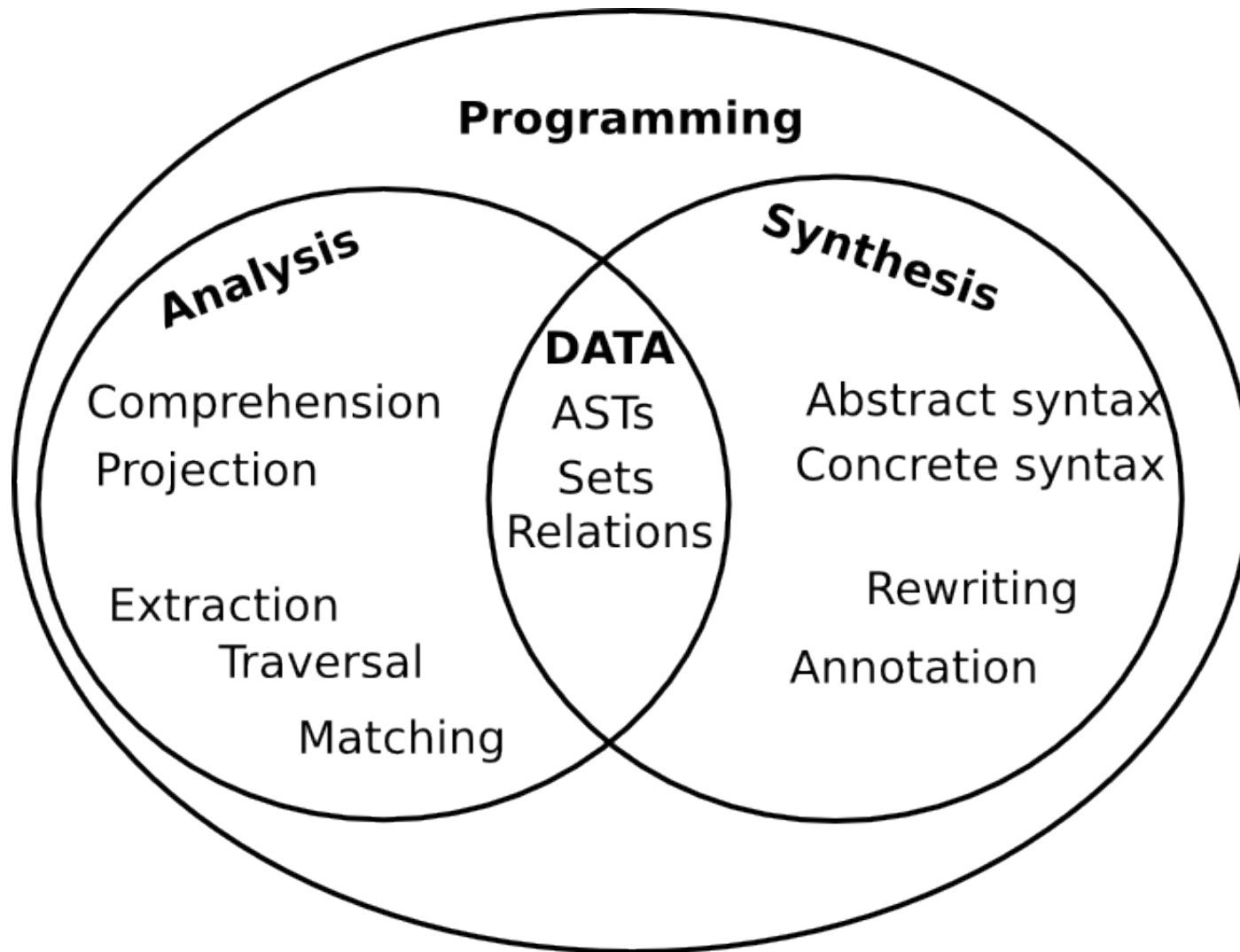


# Dimensions of requirements

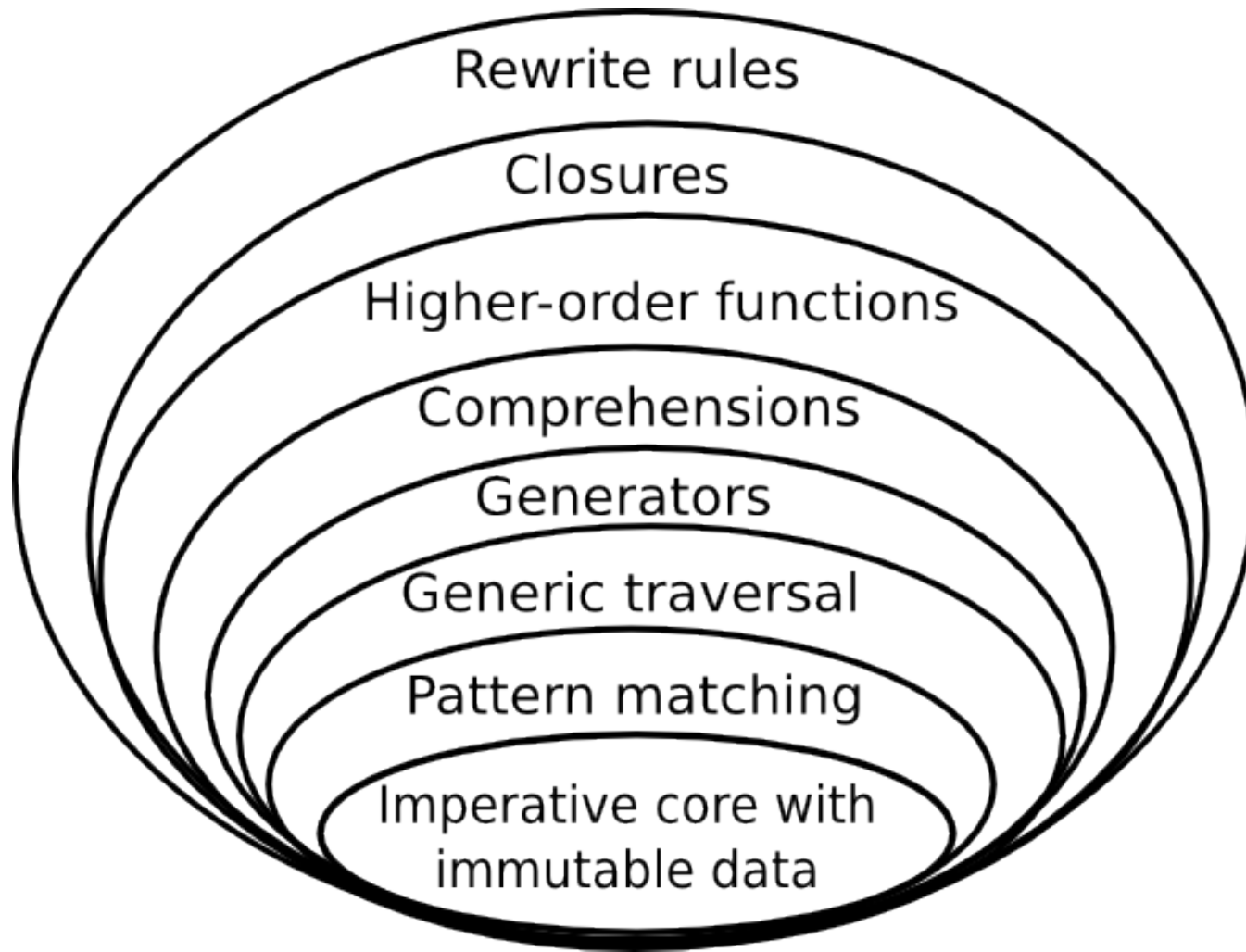




# Bridging analysis and synthesis



# Rascal's layered design



	Extract	Analyze	Synthesize
Values, Types, Datatypes	++	++	++
Syntax analysis and parsing	++	+/-	--
Pattern matching	++	++	+/-
Visitors and Switching	++	++	++
Relations, Enumerators Comprehensions	+/-	++	+/-
Rewrite rules	--	++	++

# Some Classical Examples

- Hello
- Factorial
- ColoredTrees



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

```
rascal > import demo::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, int, real, str, loc (source code location)
- **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	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
str	"abc", "values is <x>"
loc	!file:///etc/passwd
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

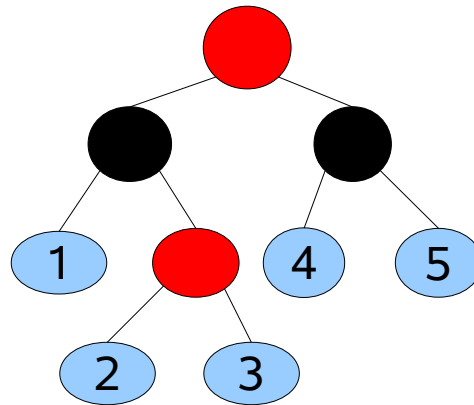
- Named alternatives
  - name acts as constructor
  - can be used in patterns
- Named fields (access/update via . notation)
- All datastructures are a subtype of the standard type *node*
  - Permits very generic operations on data
- Parse trees resulting from parsing source code are represented by the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



# 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)));
```

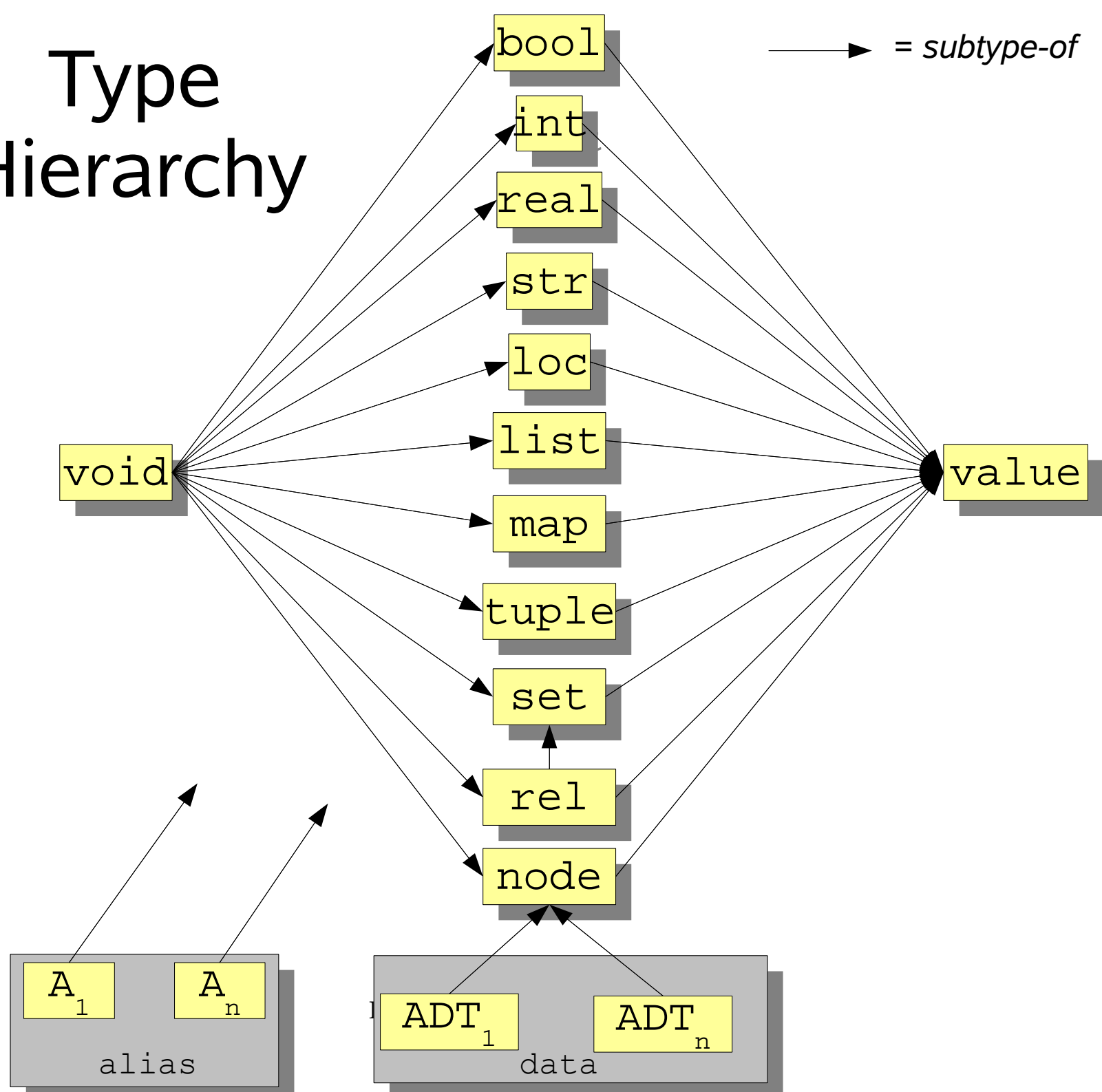


# Abstract Syntax

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



# Type Hierarchy



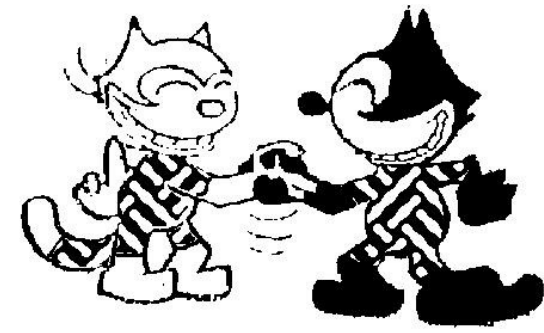
# Pattern matching



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

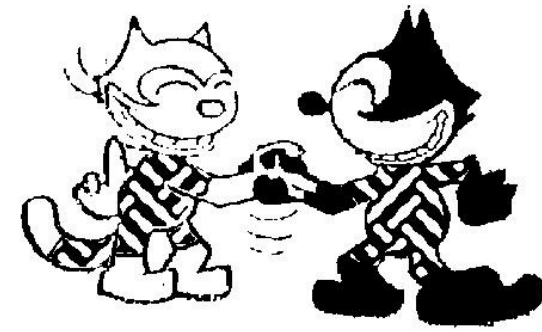


Pattern matching is used in:

- Explicit **match operator** `Pattern := Value`
- **Switch**: matching controls case selection
- **Visit**: matching controls visit of tree nodes
- **Rewrite rules**: determine whether a rule should be applied



# Patterns



**Regular:** Grep/Perl like regular expressions

```
/^<before:\W*><word:\w+><after:.*$>/
```

**Abstract:** match data types

```
whileStat(Exp, Stats*)
```

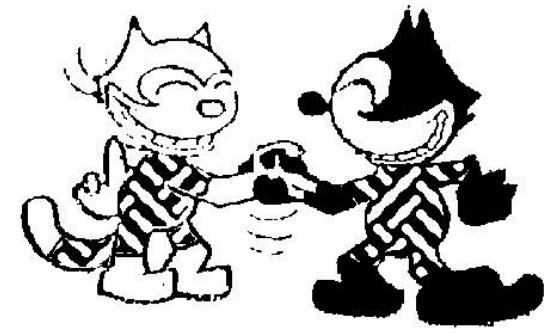
**Concrete:** match parse trees

```
[| while <Exp> do <Stats*> od |]
```





# Patterns



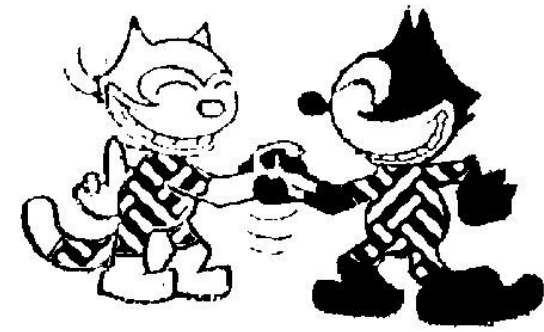
Abstract/Concrete patterns support:

- **List matching:**  $[P_1, \dots, P_n]$
- **Set matching:**  $\{P_1, \dots, P_n\}$
- **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
```

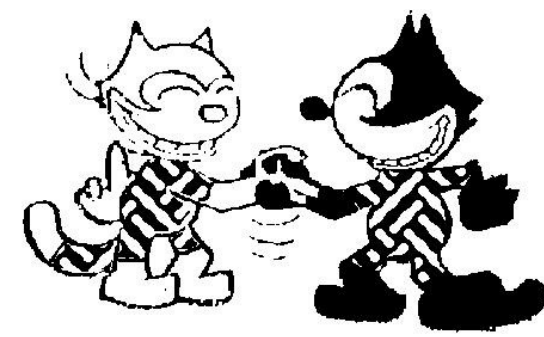
```
rascal> X;  
list[int]: [1,2]
```

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

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
```

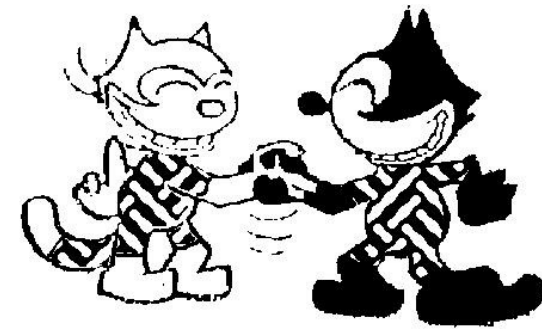
```
rascal> Y;  
set[int]: {1,2,4,5}
```

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

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



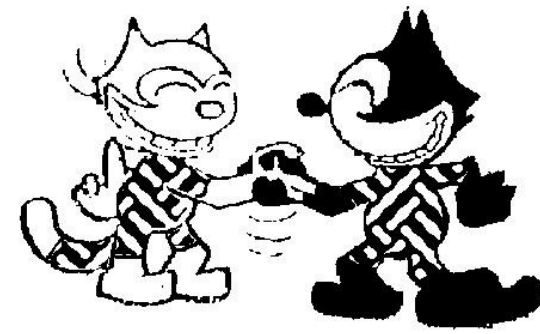
# Note



- 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)
- Side effects are undone (using recovery cache)



# 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
```



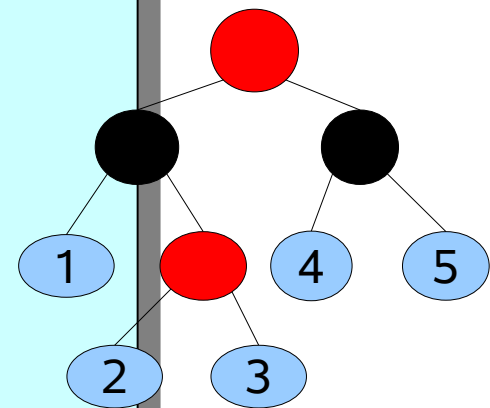
# Comprehensions

- 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;  
}
```

`countWords( "Twas brillig, and the slithy toves" ) => 6`



# Switching

- A **switch** does a top-level case distinction

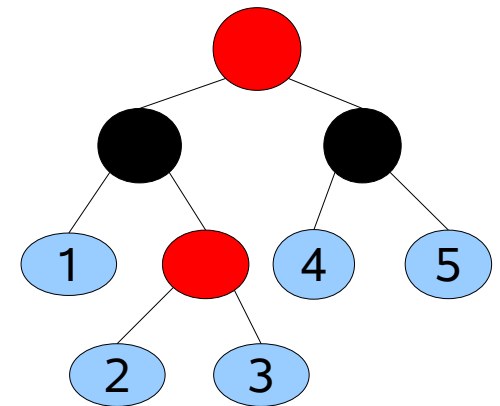
```
switch (P){  
  case whileStat(EXP Exp, Stats*):  
    println("A while statement");  
  case ifStat(Exp, Stats1*, Stats2*):  
    println("An if statement");  
}
```



# Visiting

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

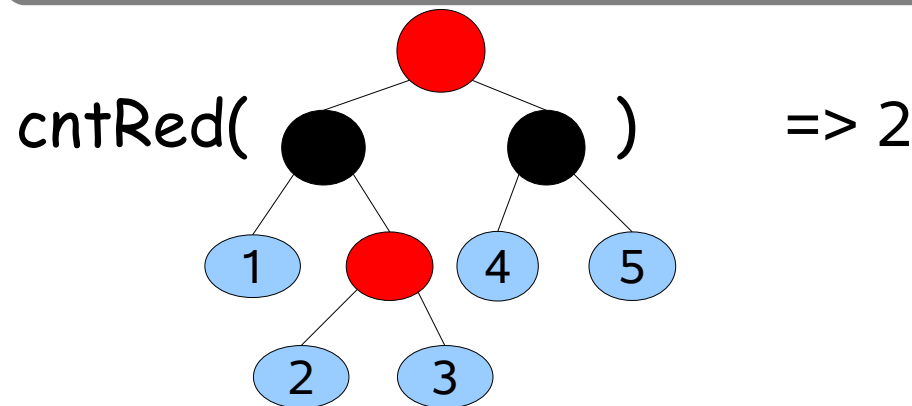
Recall the coloured trees (*CTree*):



# Count all Red Nodes

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

Visit traverses the  
complete tree and  
modifies c

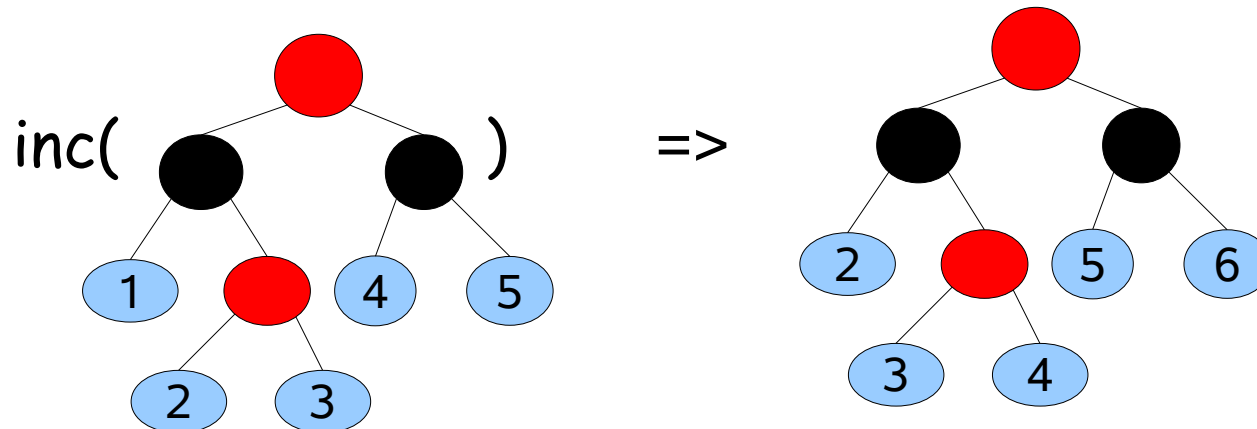


# Increment all leaves in a CTree

```
public CTree inc(CTree T) {  
  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



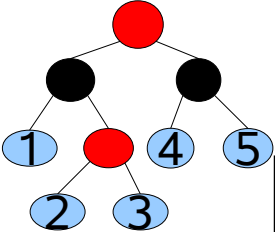
# Let's add **green** nodes

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

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

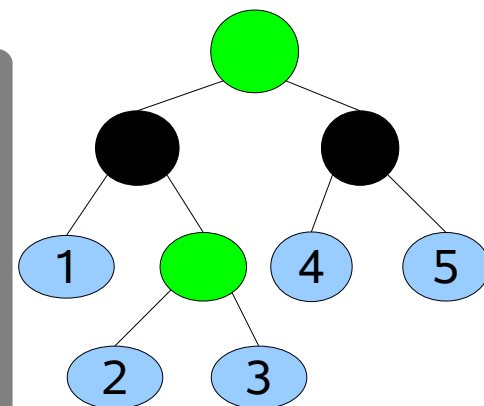




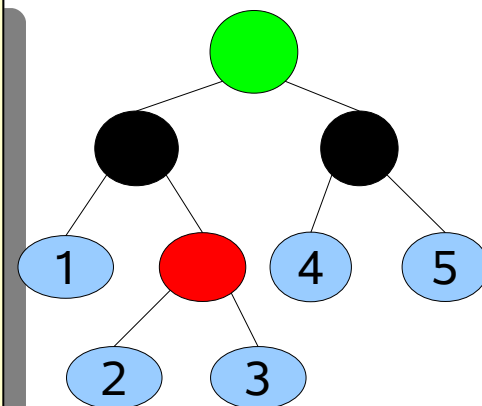


# Full/shallow/deep replacement

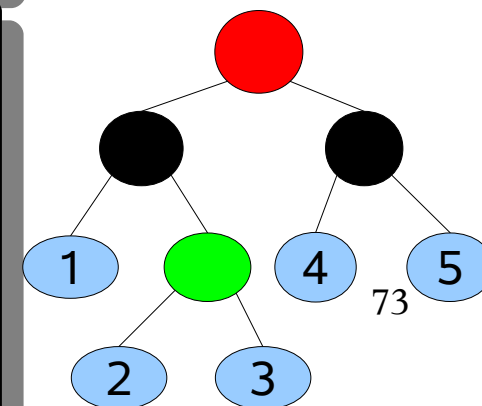
```
public CTree frepl(CTree T) {
    return visit (T) {
        case red(CTree T1, Ctree T2) => green(T1, T2)
    };
}
```



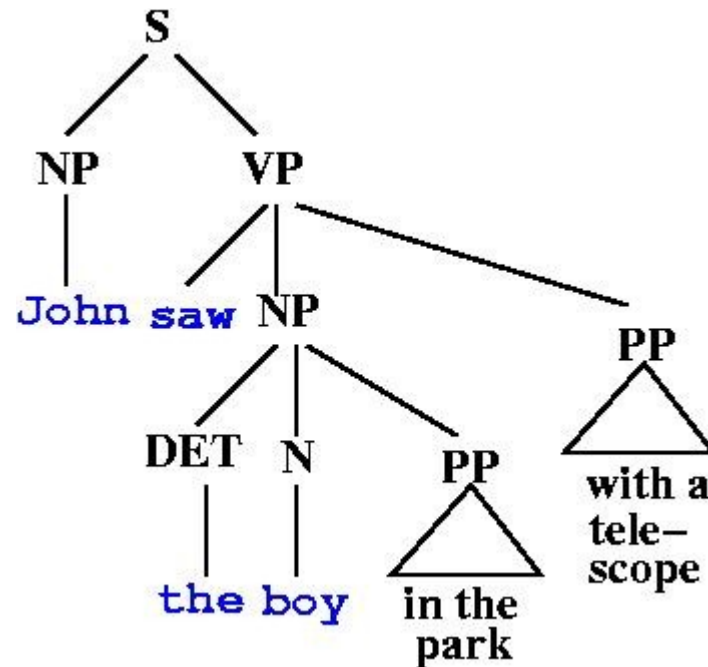
```
public Ctree srepl(CTree T) {
    return top-down-break visit (T) {
        case red(NODE T1, NODE T2) => green(T1, T2)
    };
}
```



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



# Syntax and Parsing



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

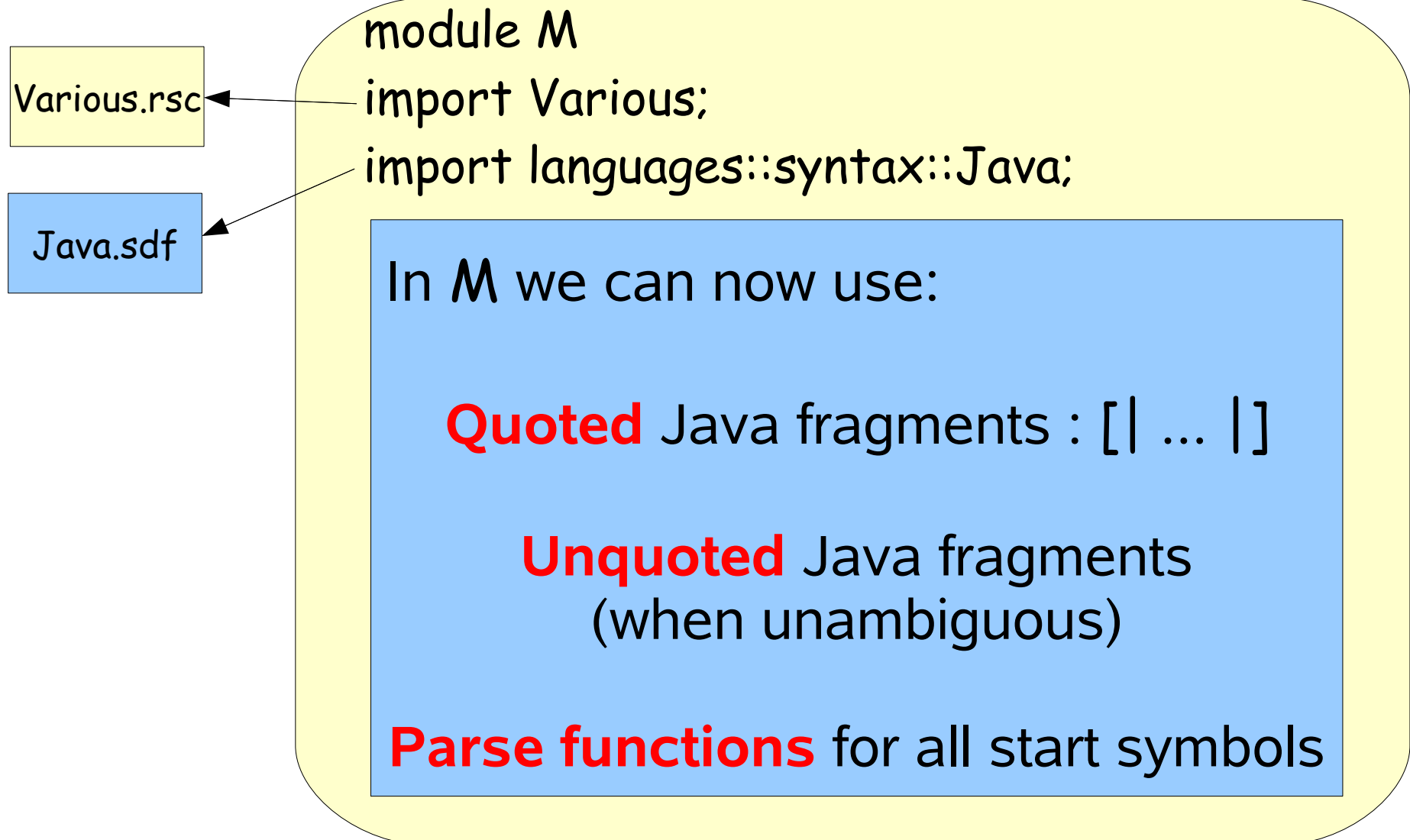


# Syntax and Parsing

- Reuses the Syntax Definition Formalism (**SDF**)
- Modular grammar definitions
- Integrated lexical and context-free parsing
- A complete SDF grammar can be imported and can be used for:
  - Parsing source code (parse functions)
  - Matching concrete code patterns
  - Synthesizing source code



# Importing an SDF module



# Result of importing an SDF module

- A typed parse function becomes available for all start symbols in the grammar, e.g.
  - `CompilationUnit parseCompilationUnit(str file)`

```
module Count
import languages::syntax::Java;
public int countMethods(str file){
    int n = 0;
    for(MethodDeclaration md <- parseCompilationUnit(file))
        n += 1;
    return n;
}
```



# Finding date-related variables

Import the COBOL grammar

```
module DateVars  
import Cobol;
```

```
set[Var] getDateVars(CobolProgram P){
```

Traverse P and  
return all occurrences  
of variables

```
  return {V | Var V <- P,
```

Put variables that  
match in result

```
    /^.*(date|dt|year|yr).*$$/i := toString(V)
```

```
  };
```

```
}
```

Variable name  
matches a date-related  
heuristic



# Generating Getters and Setters (1)

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

Required:

- Generate the named class with getters and setters



# Generating Getters and Setters

```
Class generate(Id name, map[Id,Type] fields) {  
  Decl* decls = [| |];  
  for (id <- domain(fields)) {
```

Syntactic type

```
}
```

Unquoted concrete  
syntax expression

```
return public class <name> { <decls> };  
}
```





# Generating Getters and Setters

```
Class generate(Id name, map[Id,Type] fields) {  
  Decl* decls = [| |];  
  for (id <- domain(fields)) {  
    type = fields[id]; <get, set> = getSetIds(id);  
    decls = [| <decls>  
      private <id> <type>;  
      public <type> <get>() { return <id>;}  
      public void <set>(<type> x) {  
        this.<id> = x;  
      } |];  
  }  
  return public class <name> { <decls> };  
}
```

Quoted concrete  
syntax expression

Variable  
interpolation

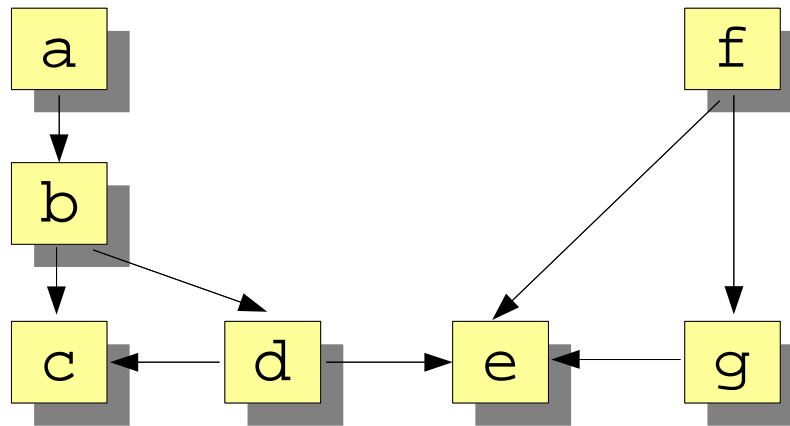


# Other features

- Rewrite rules
- Solve equations using fixed point iteration
- Get/set fields of ADTs
- Exception handling
- Annotations
- Local backtracking, undoing side-effects
- Parameterized types
- Higher order functions



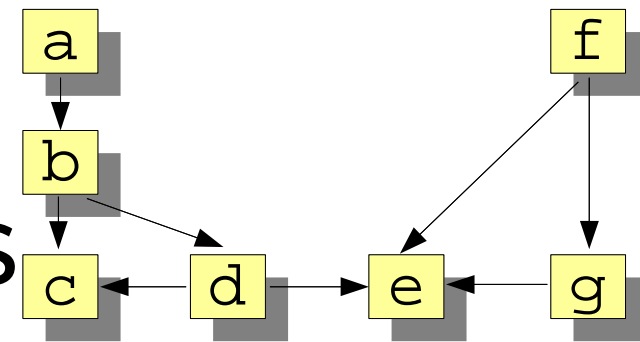
# Analyzing the call structure of an application



```
alias graph[&T] = rel[&T, &T];  
graph[str] calls = {<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">,  
                    <"d", "e">, <"f", "e">, <"f", "g">, <"g", "e">};
```



# Some questions



- How many calls are there?

- `int ncalls = size(calls);`

- 8

Number of elements

- How many procedures are there?

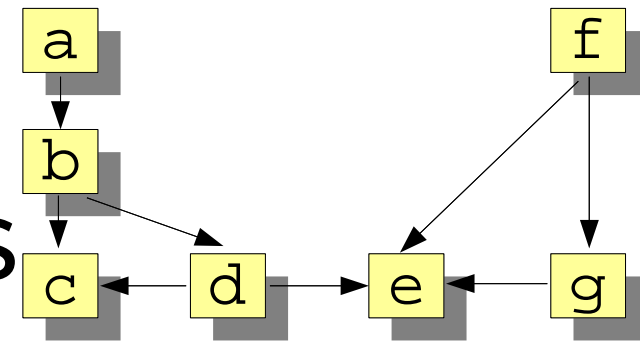
- `int nprocs = size(carrier(calls));`

- 7

All elements in domain or range of a relations



# Some questions



- What are the entry points?
  - `set[str] entryPoints = top(calls)`
  - {"a", "f"}
- What are the leaves?
  - `set[str] bottomCalls = bottom(calls)`
  - {"c", "e"}

The *roots* of a relation  
(viewed as a graph)

The *leaves* of a relation  
(viewed as a graph)



# Intermezzo: Top

- The **roots** of a relation viewed as a graph
- $\text{top}(\{\langle 1,2 \rangle, \langle 1,3 \rangle, \langle 2,4 \rangle, \langle 3,4 \rangle\})$  yields  $\{1\}$
- Consists of all elements that occur on the **lhs** **but not on the rhs** of a tuple
- ```
set[&T] top(graph[&T] R) {  
    return domain(R) - range(R);  
}
```

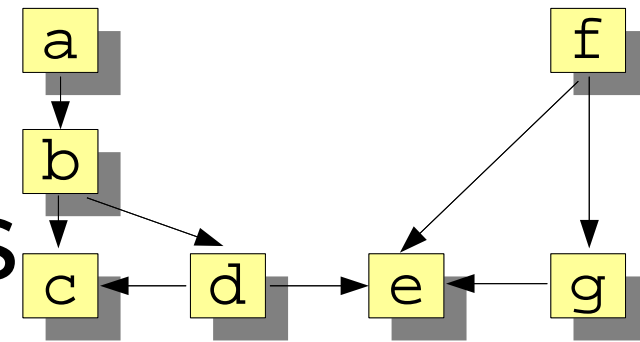


# Intermezzo: Bottom

- The **leaves** of a relation viewed as a graph
- `bottom({<1,2>,<1,3>,<2,4>,<3,4>})` yields `{4}`
- Consists of all elements that occur on the **rhs** **but not on the lhs** of a tuple
- ```
set[&T] bottom(graph[&T] R) {  
    return range(R) - domain(R);  
}
```



# Some questions



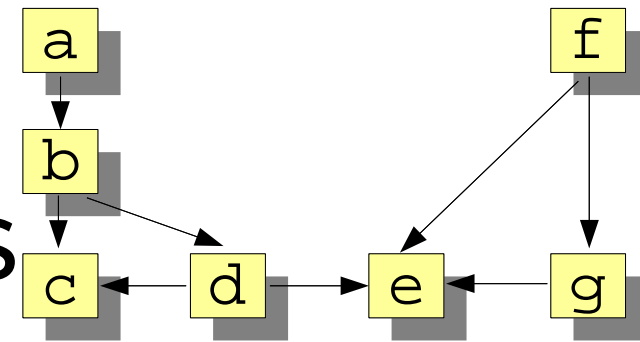
- What are the indirect calls between procedures?
  - `graph[str] closureCalls = calls+`
  - `{<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">, <"d", "e">, <"f", "e">, <"f", "g">, <"g", "e">, <"a", "c">, <"a", "d">, <"b", "e">, <"a", "e">}`
- What are the calls from entry point a?
  - `set[str] calledFromA = closureCalls["a"]`
  - `{"b", "c", "d", "e"}`

The image of domain value "a"





# Some questions



- What are the calls from entry point f?
  - `set[str] calledFromF = closureCalls["f"];`
  - `{"e", "g"}`
- What are the common procedures?
  - `set[str] commonProcs =`  
`calledFromA & calledFromF`
  - `{"e"}`

Intersection

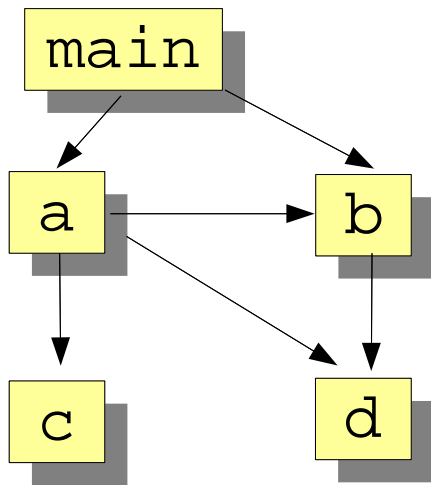


# Component Structure of Application

- Suppose, we know:
  - the call relation between procedures (*Calls*)
  - the component of each procedure (*PartOf*)
- Question:
  - Can we lift the relation between procedures to a relation between components (*ComponentCalls*)?
- This is usefull for checking that real code conforms to architectural constraints



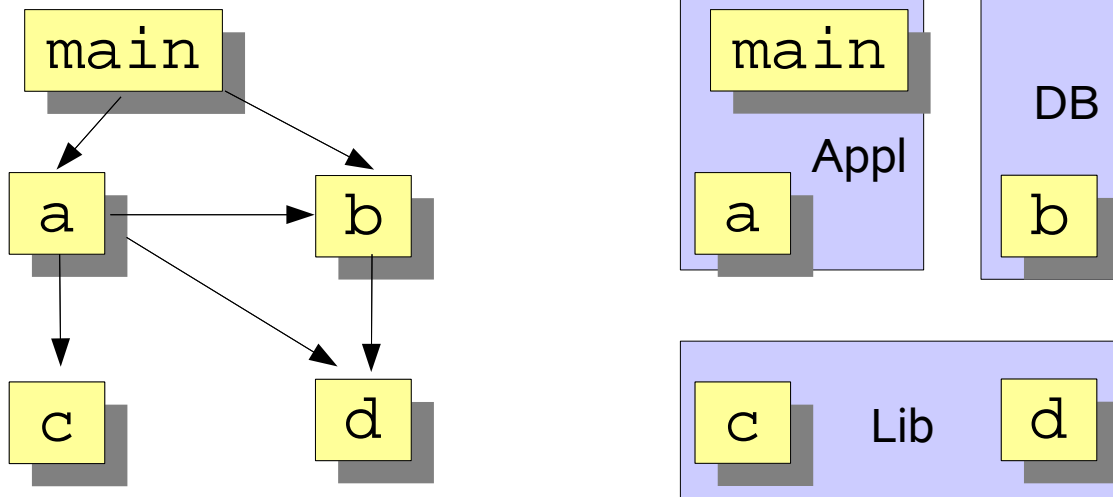
# Calls



```
alias proc = str;  
alias comp = str;  
rel[proc,proc] Calls = {<"main", "a">, <"main", "b">, <"a", "b">,  
                        <"a", "c">, <"a", "d">, <"b", "d">};
```



# PartOf

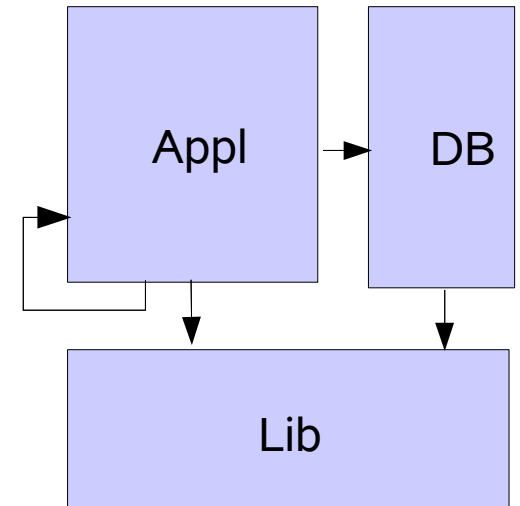
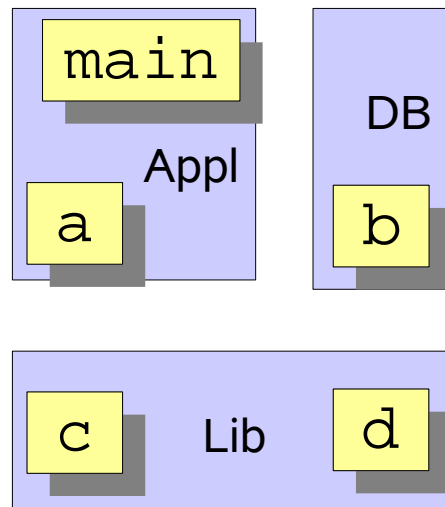
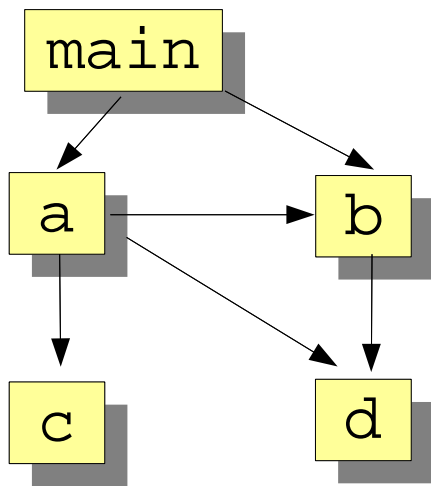


```
set[comp] Components = {"Appl", "DB", "Lib"};
```

```
rel[proc, comp] PartOf =  
  <"main", "Appl">, <"a", "Appl">, <"b", "DB">,  
  <"c", "Lib">, <"d", "Lib">;
```



# lift



```
rel[comp,comp] lift(rel[proc,proc] aCalls, rel[proc,comp] aPartOf) {
  return { <C1, C2> | <proc P1, proc P2> <- aCalls,
    <comp C1, comp C2> <- aPartOf[P1] x aPartOf[P2] };
}
```

```
rel[comp,comp] ComponentCalls = lift(Calls2, PartOf);
```

**Result:** {<"DB", "Lib">, <"Appl", "Lib">, <"Appl", "DB">, <"Appl", "Appl">}

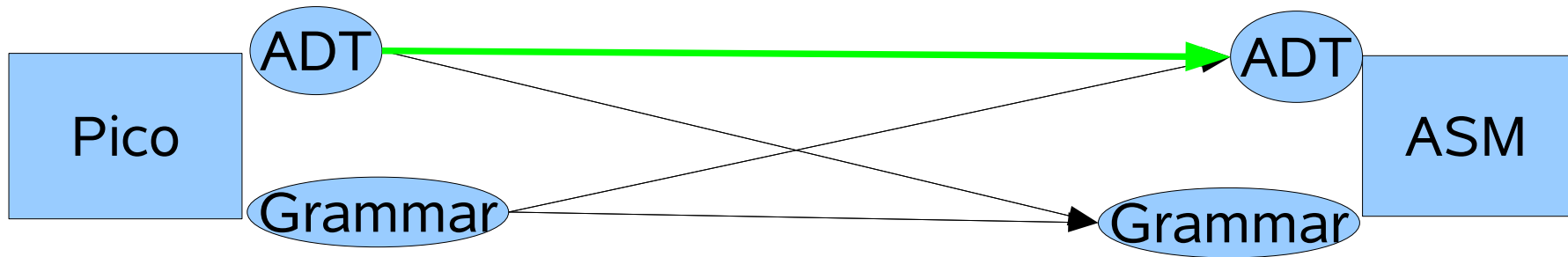


# A Code Generation Example

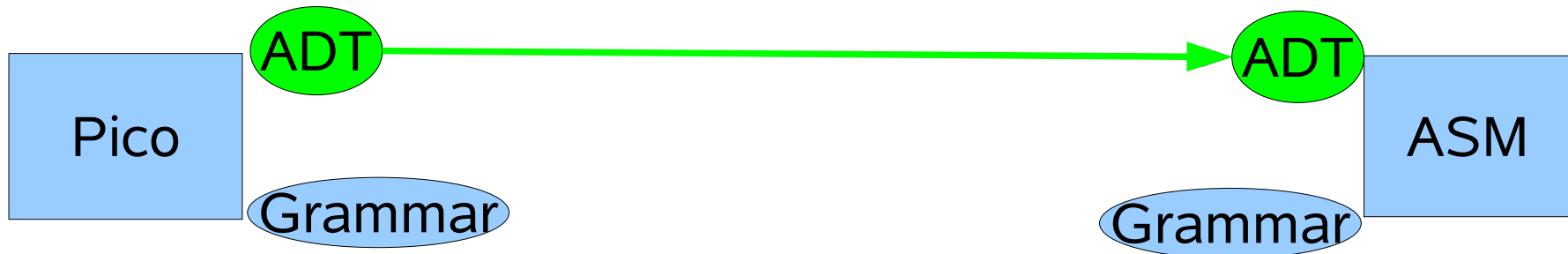
- Given the toy language Pico
- Given a simple, stack-based, assembly language
- Problem: compile Pico to Assembly Language



# Design Choices



# Design Choices



ADT	Grammar
Rewrite rules	Visit/switch
Functional	Global State





# Pico Abstract Syntax (1)

```
module demo::PicoAbstract::PicoAbstractSyntax
```

```
public data TYPE =
```

```
    natural | string;
```

```
public alias PicoId = str;
```

```
public data EXP =
```

```
    id(PicoId name)
```

```
    | natCon(int iVal)
```

```
    | strCon(str sVal)
```

```
    | add(EXP left, EXP right)
```

```
    | sub(EXP left, EXP right)
```

```
    | conc(EXP left, EXP right)
```

```
;
```



# Pico Abstract Syntax (2)

```
public data STATEMENT =  
  asgStat(PicoId name, EXP exp)  
  | ifStat(EXP exp, list[STATEMENT] thenpart,  
           list[STATEMENT] elsepart)  
  | whileStat(EXP exp, list[STATEMENT] body)  
  ;  
public data DECL =  
  decl(PicoId name, TYPE tp);  
  
public data PROGRAM =  
  program(list[DECL] decls, list[STATEMENT] stats);
```



# Assembly Instructions

```
module demo::PicoAbstract::Assembly
import demo::PicoAbstract::PicoAbstractSyntax;

public data Instr =
  dclNat(PicoId Id) | dclStr(PicoId Id)
| pushNat(int intCon) | pushStr(str strCon)
| rvalue(PicoId Id) | lvalue(PicoId Id)
| pop | copy | assign | add | sub | mul
| label(str label) | go(str label)
| gotrue(str label) | gofalse(str label);
```



# Compilation Problem

Pico  
program

ASM  
list[Instr]

```
private list[Instr] compileExp(EXP exp) { .. }
```

```
private list[Instr] compileStatement(STATEMENT Stat){ ... }
```

```
private list[Instr] compileStatements(list[STATEMENT] Stats){ ... }
```

```
private list[Instr] compileDecls(list[DECL] Decls){ ... }
```

```
public list[Instr] compileProgram(PROGRAM P){ ... }
```



# Compile Expressions

```
private list[Instr] compileExp(EXP exp) {  
  switch (exp) {  
    case natCon(int N): return [pushNat(N)];  
    case strCon(str S): return [pushStr(S)];  
    case id(PicoId Id): return [rvalue(Id)];  
    case add(EXP E1, EXP E2):  
      return [compileExp(E1), compileExp(E2), add];  
    case sub(EXP E1, EXP E2): ...  
    case conc(EXP E1, EXP E2): ...  
  }  
}
```



# Label Generation (using a module variable)

```
private int nLabel = 0;  
  
private str nextLabel(){  
    nLabel += 1;  
    return "L" + toString(nLabel);  
}
```



# Compile Statements (1)

```
private list[Instr] compileStatement(STATEMENT Stat){  
  
    switch (Stat) {  
  
        case asgStat(PicoId Id, EXP Exp):  
            return [lvalue(Id), compileExp(Exp), assign];  
  
        ... (on next slides)  
    }  
}
```



# Compile Statement (2)

```
private list[Instr] compileStatement(STATEMENT Stat){  
    switch (Stat) {  
        ... (on previous slide)  
        case ifStat(EXP Exp, list[STATEMENT] Stats1,  
                    list[STATEMENT] Stats2):{  
            nextLab = nextLabel(); falseLab = nextLabel();  
            return [compileExp(Exp),  
                    gofalse(falseLab),  
                    compileStatements(Stats1),  
                    go(nextLab),  
                    label(falseLab), compileStatements(Stats2),  
                    label(nextLab)];  
        }  
        ... (on next slide)
```





# Compile Statement (3)

```
private list[Instr] compileStatement(STATEMENT Stat){  
  switch (Stat) {  
    ... (on previous slide)  
    case whileStat(EXP Exp, list[STATEMENT] Stats1): {  
      entryLab = nextLabel();  
      nextLab = nextLabel();  
      return [label(entryLab), compileExp(Exp),  
              gofalse(nextLab),  
              compileStatements(Stats1),  
              go(entryLab),  
              label(nextLab)];  
    }  
  }  
}
```



# Compile Statements

```
private list[Instr] compileStatements(list[STATEMENT] Stats){  
  return [ compileStatement(S) | S <- Stats ];  
}
```



# Compile Declarations

```
private list[Instr] compileDecls(list[DECL] Decls){  
  return [ (type == natural) ? dclNat(Id) : dclStr(Id) |  
    decl(PicoId Id, TYPE type) <- Decls];  
}
```



# Compile Pico Program

```
public list[Instr] compileProgram(PROGRAM P){  
    nLabel = 0;  
    if(program(list[DECL] Decls, list[STATEMENT] Series) := P){  
        return [compileDecls(Decls), compileStatements(Series)];  
    } else  
        throw Exception("Cannot happen");  
}
```



# Example of Compilation

```
P = program([decl("x", natural),  
            [ifStat(natCon(5), [asgStat("x", natCon(3))],  
                          [asgStat("x", natCon(4))])]);
```

compileProgram(P) =>

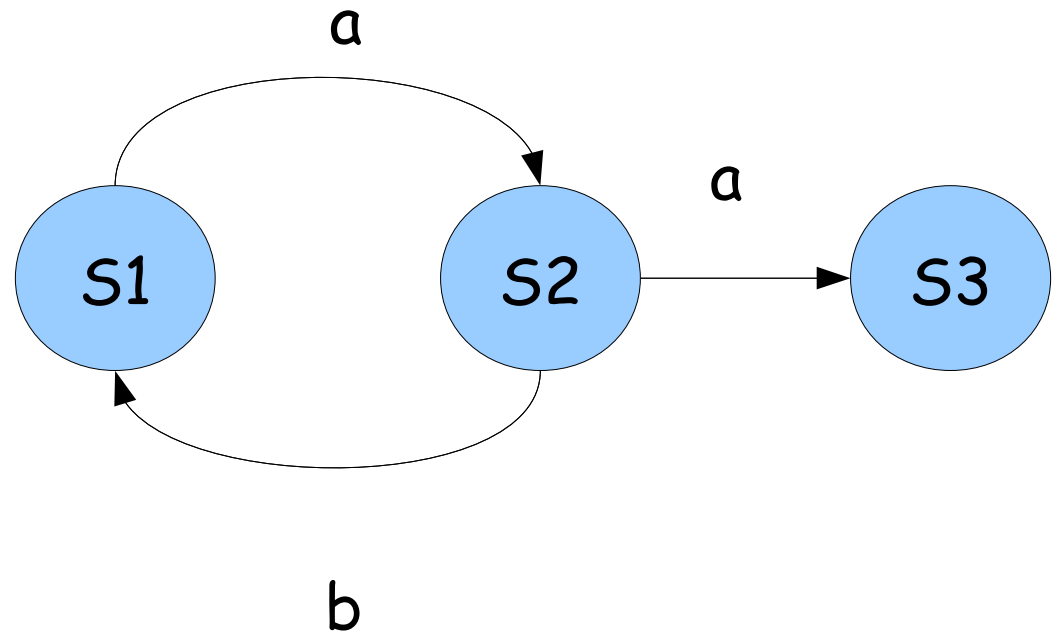
```
[declNat("x"),  
 pushNat(5), gofalse("L4"),  
 lvalue("x"),  
 pushNat(3),  
 assign(),  
 go("L3"),  
 label("L4"),  
 lvalue("x"),  
 pushNat(4),  
 assign(),  
 label("L3")];
```



# State Machine

*(thanks to Görel Hedin)*

```
state S1;  
state S2;  
state S3;  
trans a: S1 -> S2;  
trans b: S2 -> S1;  
trans a: S2 -> S3
```



- An abstract version
- A concrete version



# Abstract Version (1)

```
module FSM
import Graph;
alias StateMachine = list[Declaration];

data Declaration =
    state(str label) |
    transition(str label, str sourceLabel, str targetLabel);

public StateMachine example =
    [state("s1"), state("s2"), state("s3"),
     transition("a", "s1", "s2"), transition("b", "s2", "s1"),
     transition("a", "s2", "s3")];
... (next sheet)
```



# Abstract Version (2)

```
public graph[str] getTransitions(StateMachine sm) {  
    return { <from, to> | transition(_, str from, str to) <- sm };  
}
```

```
public map[str, set[str]] canReach(StateMachine sm) {  
    transitions = getTransitions(sm);  
    closure = transitions+;  
    return ( s : closure[s] | str s <- carrier(transitions) );  
}
```





# State Machine Concrete Syntax

```
module demo/StateMachine/Syntax
imports basic/Whitespace
imports basic/IdentifierCon
exports
  context-free start-symbols
  FSM
  sorts FSM Decl Trans State IdCon
  context-free syntax
    "state" IdCon          -> State
    "trans" IdCon ":" IdCon "->" IdCon -> Trans
    State                  -> Decl
    Trans                  -> Decl
    {Decl ";" }+           -> FSM
```



# Reachability (1)

```
module demo::StateMachine::CanReach
```

```
import demo::StateMachine::Syntax;
```

```
import Graph;
```

```
{Decl ":"}+ example =
```

```
state S1;
```

```
state S2;
```

```
state S3;
```

```
trans a: S1 -> S2;
```

```
trans b: S2 -> S1;
```

```
trans a: S2 -> S3;
```

Yet another,  
**concrete,**  
**unquoted,**  
text fragment.  
Here for {Decl ":"}+

... (next sheet)



# Reachability (2)

module demo::StateMachine::CanReach  
... (previous sheet)

```
public graph[IdCon] getTransitions({Decl ";" }+ fsm){  
  return { <from, to> |  
    trans <IdCon a>: <IdCon from> -> <IdCon to> <- fsm }  
}
```

```
public map[IdCon, set[IdCon]] canReach({Decl ";" }+ fsm){  
  transitions = getTransitions(fsm);  
  closure = transitions+;  
  return ( s : closure[s] | IdCon s <- carrier(transitions) );  
}
```



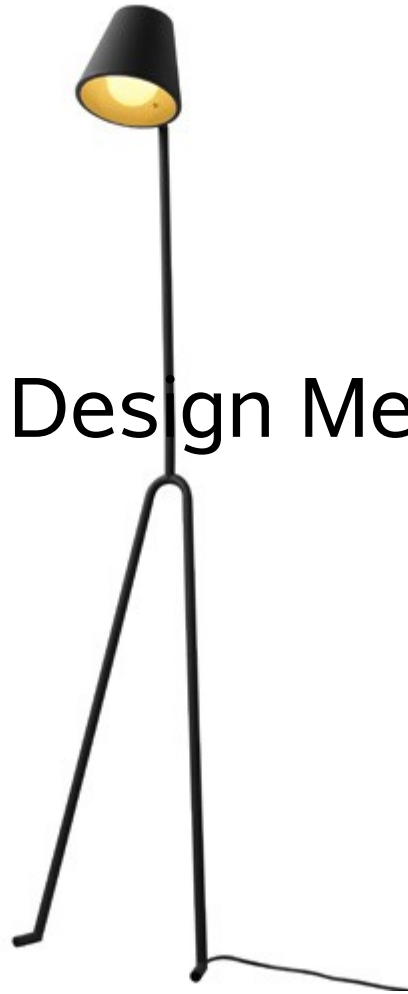
# Computing Dominators

- A node M **dominates** other nodes S in the flow graph iff all path from the root to a node in S contain M

```
public rel[&T, set[&T]] dominators(  
    rel[&T,&T] PRED,    // control flow graph  
    &T ROOT              // entry point)  
{  
    set[&T] VERTICES = carrier(PRED);  
    return { <V, (VERTICES - {V, ROOT})  
            - reachX({ROOT}, {V}, PRED)> | &T V : VERTICES};  
}
```

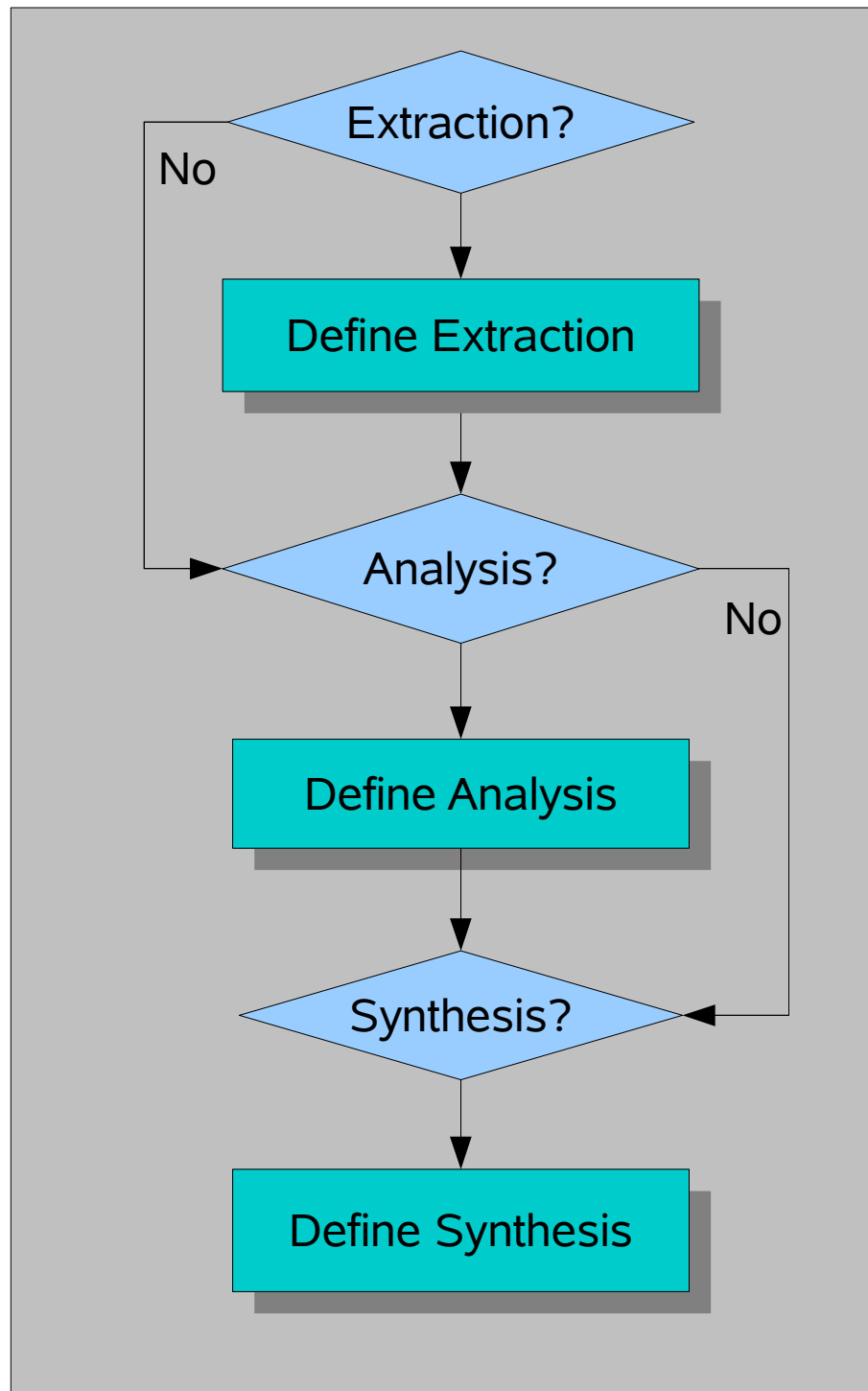


Is there a Rascal Design Methodology?

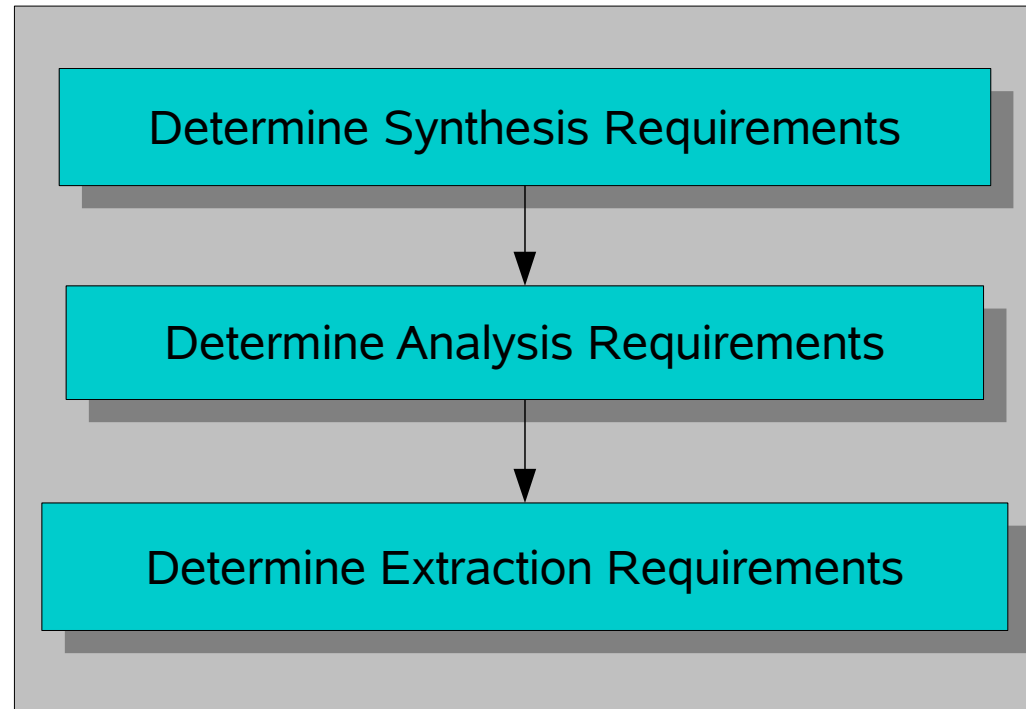


# Rascal

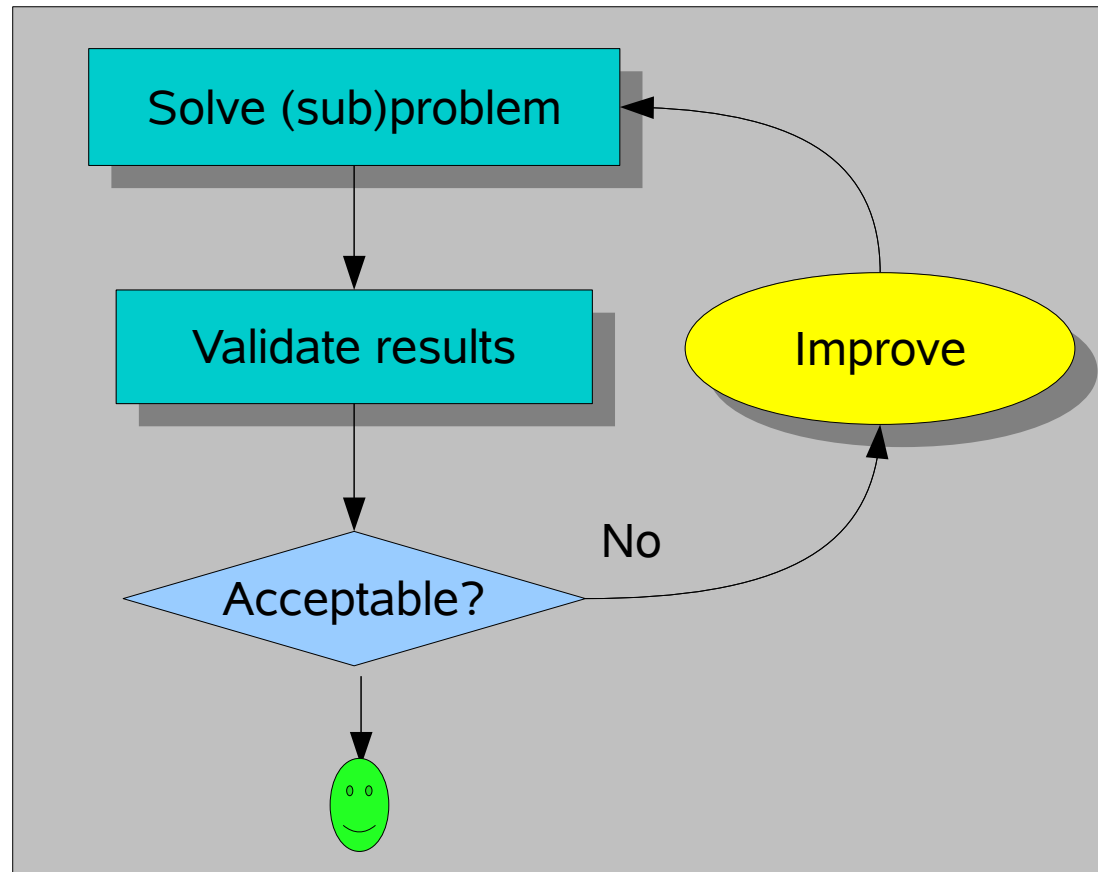
# Workflow



# Requirements Analysis

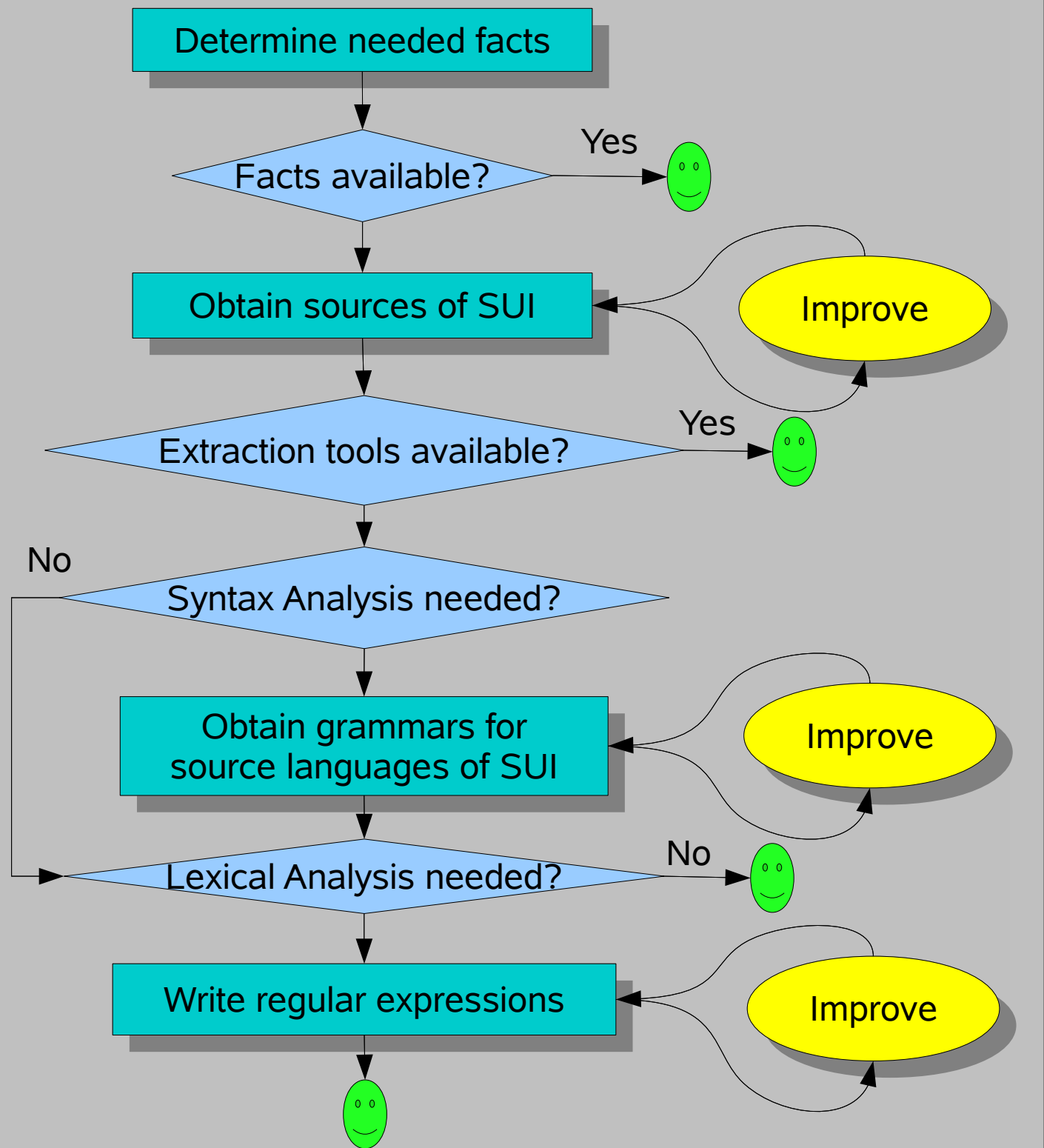


# Validation

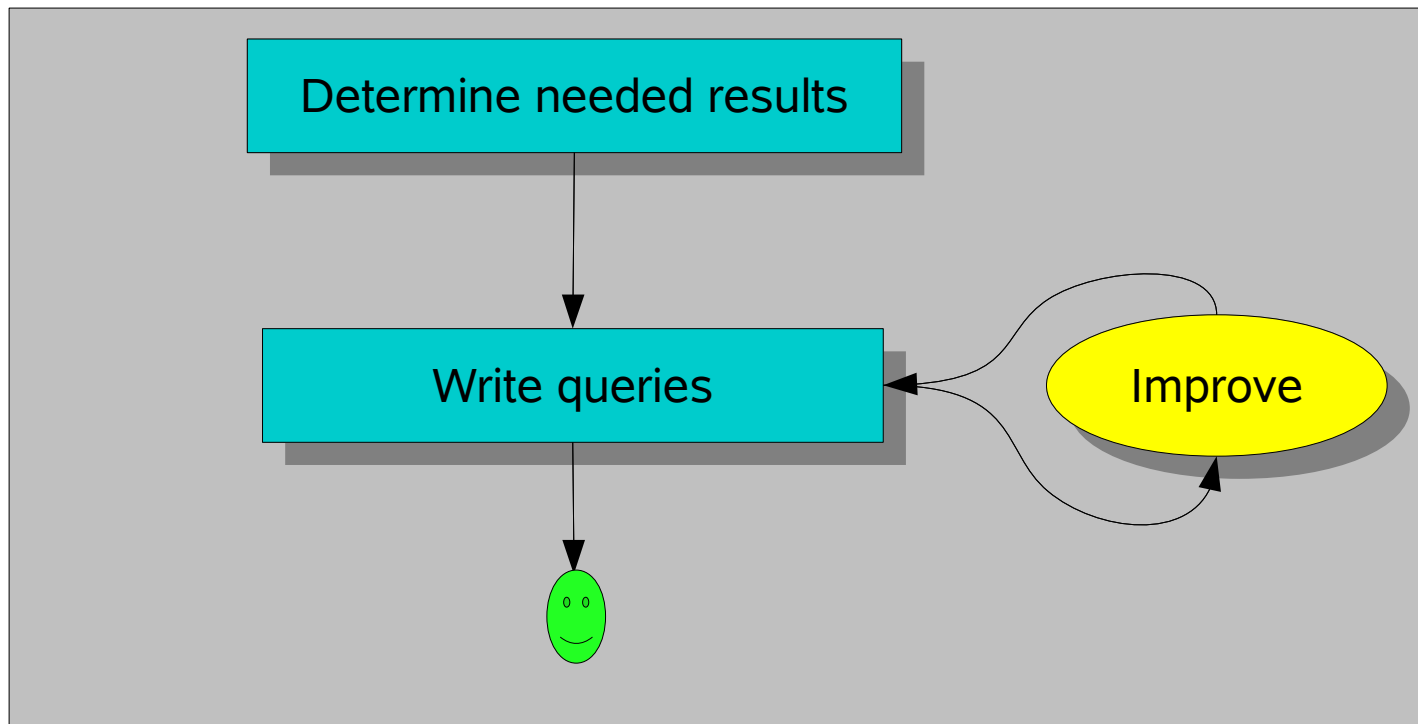




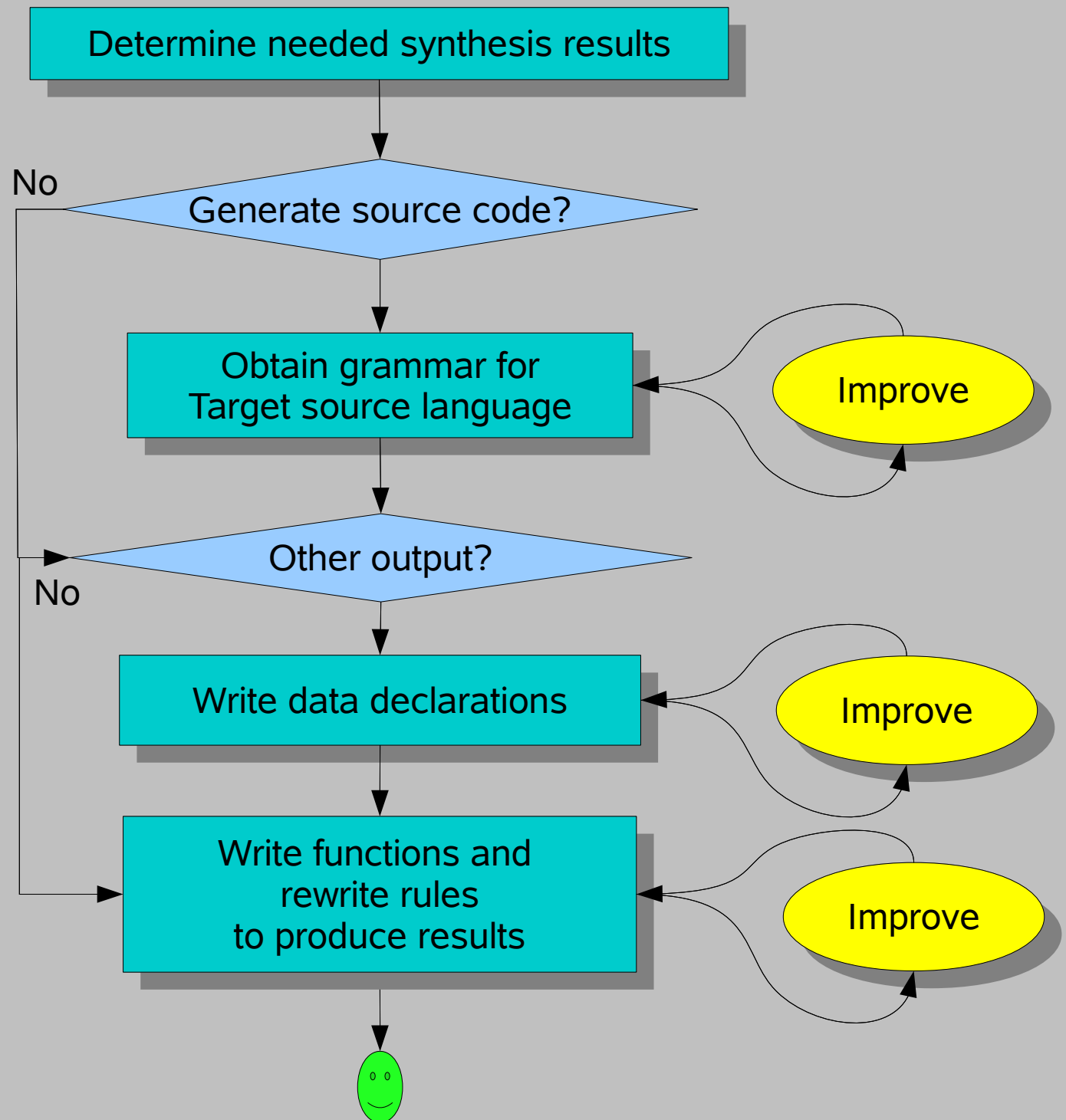
# Extraction Workflow



# Analysis Workflow



# Synthesis Workflow



# The Rascal Standard Library

- Benchmark
- Boolean
- Exception
- Graph
- Integer
- IO
- JDT (Eclipse only)
- Labelled Graph
- List
- Location
- Map
- Node
- Real
- Relation
- RSF
- Resource (Eclipse only)
- Set
- String
- Tuple
- ValueIO
- View (Eclipse only)

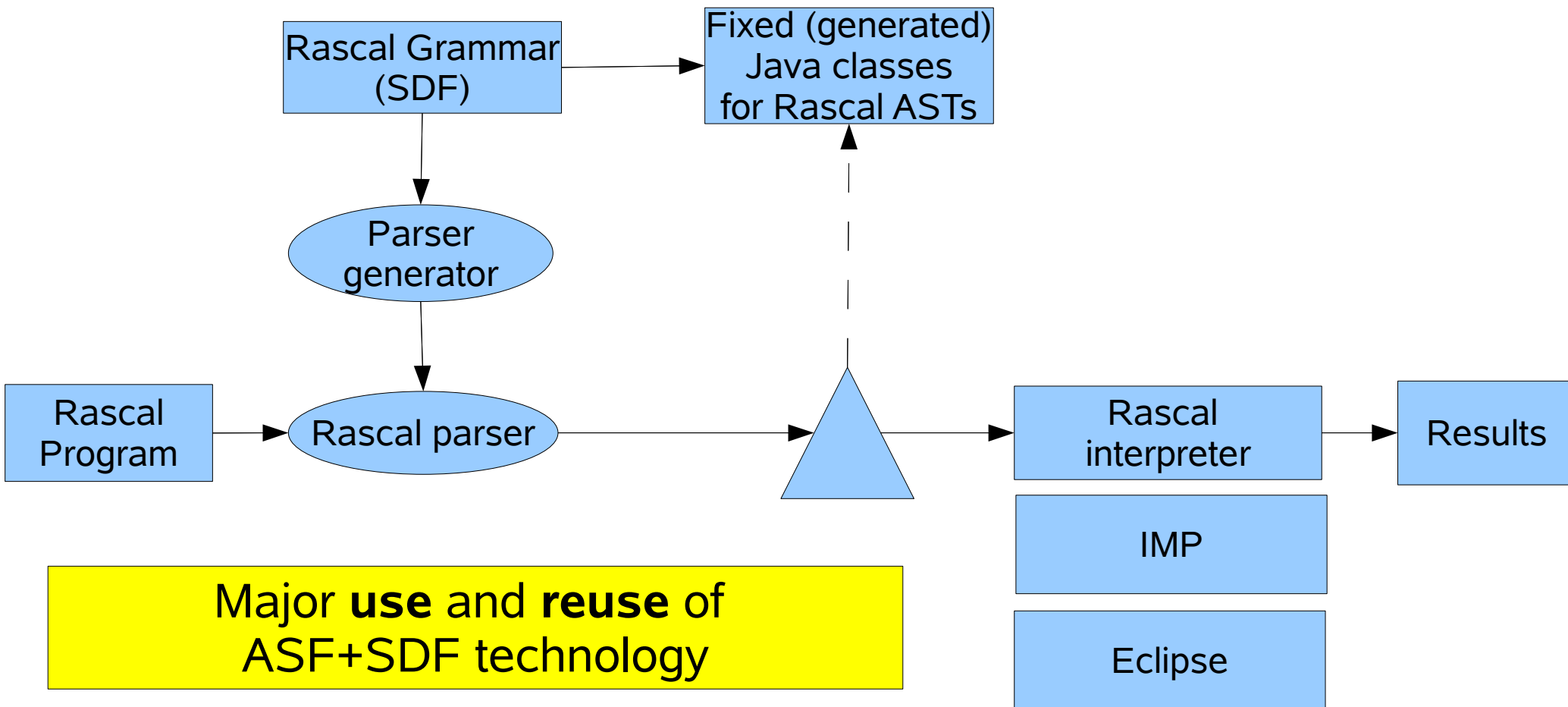


# Rascal Status

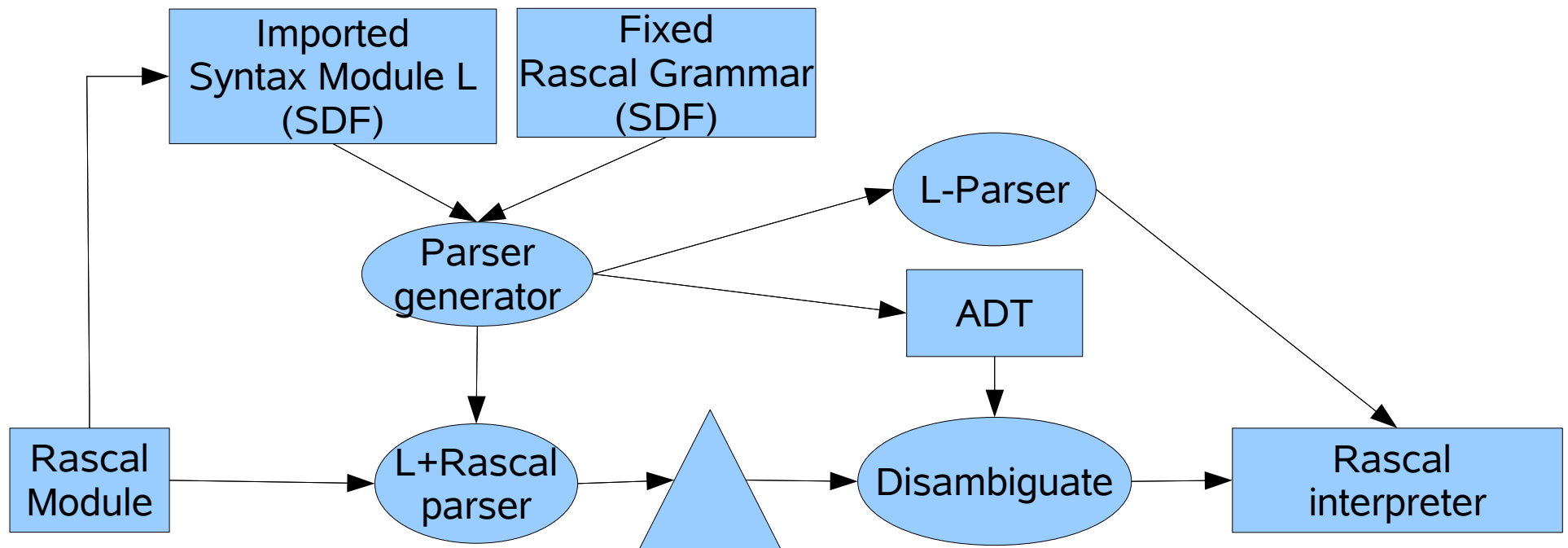
- An interpreter for the core language is well underway.
- All the above examples (and many more!) run.
- Standalone version and Eclipse version
- Eclipse version: debugger and JDT integration
- Launch of  $\alpha$ -version at GTTSE summerschool in Braga Portugal, july 2009
- Linux, Windows and Mac OS versions



# Rascal Implementation



# Implementation SDF modules



# About disambiguation

- We use a number of fixed heuristics
- In the future we will also use type information from the Rascal program itself
- The user can always manually disambiguate using
  - Quotes `[| ... |]`
  - Typed quotes *Type*`[| ... |]`





# Rascal Implementation (some metrics)

- PDB (23 Kloc)
- Rascal (92 KLoc)
  - incl 7 Kloc tests (2200 tests)
  - incl. 18 Kloc generated ASTs
- Rascal-eclipse (32 KLoc)
  - incl. Debugger
- Total, circa 147 KLoc





- Rascal library that uses JDT from Eclipse and enables Java analysis and transformation
- Parsing library
- De Facto: extraction by grammar annotation
- Various graph algorithms
- Bisimulation algorithms
- Concept analysis
- Automata extraction/generation



# Longer-term Perspective

- Rascal enables the creation and execution of fact analysis and transformation tools and provides meta-programming for the software composition domain
- Applications: refactoring, renovation, model transformation?
- Transition path from ASF+SDF to Rascal?
- Expressivity: on the right track
- **Challenge: performance, performance, performance**



# Information

General information:

<http://www.meta-environment.org>

Latest version of Rascal  
documentation:

[http://www.meta-environment.org/doc/books/analysis/rascal-manual/rascal-manual.\[html|pdf\]](http://www.meta-environment.org/doc/books/analysis/rascal-manual/rascal-manual.[html|pdf])

Download  $\alpha$ -version of Rascal implementation:

<http://www.meta-environment.org/Meta-Environment/Rascal>



# Join Us in Creating Something Beautiful



- Feedback  $\alpha$ -version
- Criticism on design
- Suggest additions
- Case studies
- Tool support
- Tutorials