

EASY Meta-Programming with Rascal






Leveraging the Extract-Analyze-SYnthesize Paradigm

Paul Klint

*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

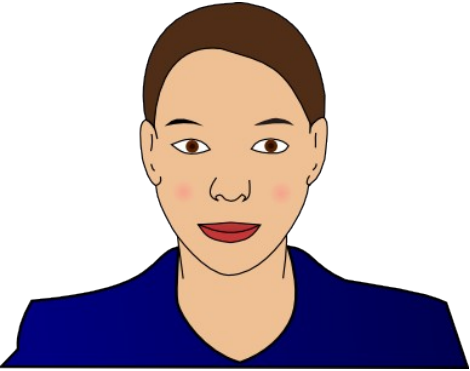


Meet Bernd



- **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?

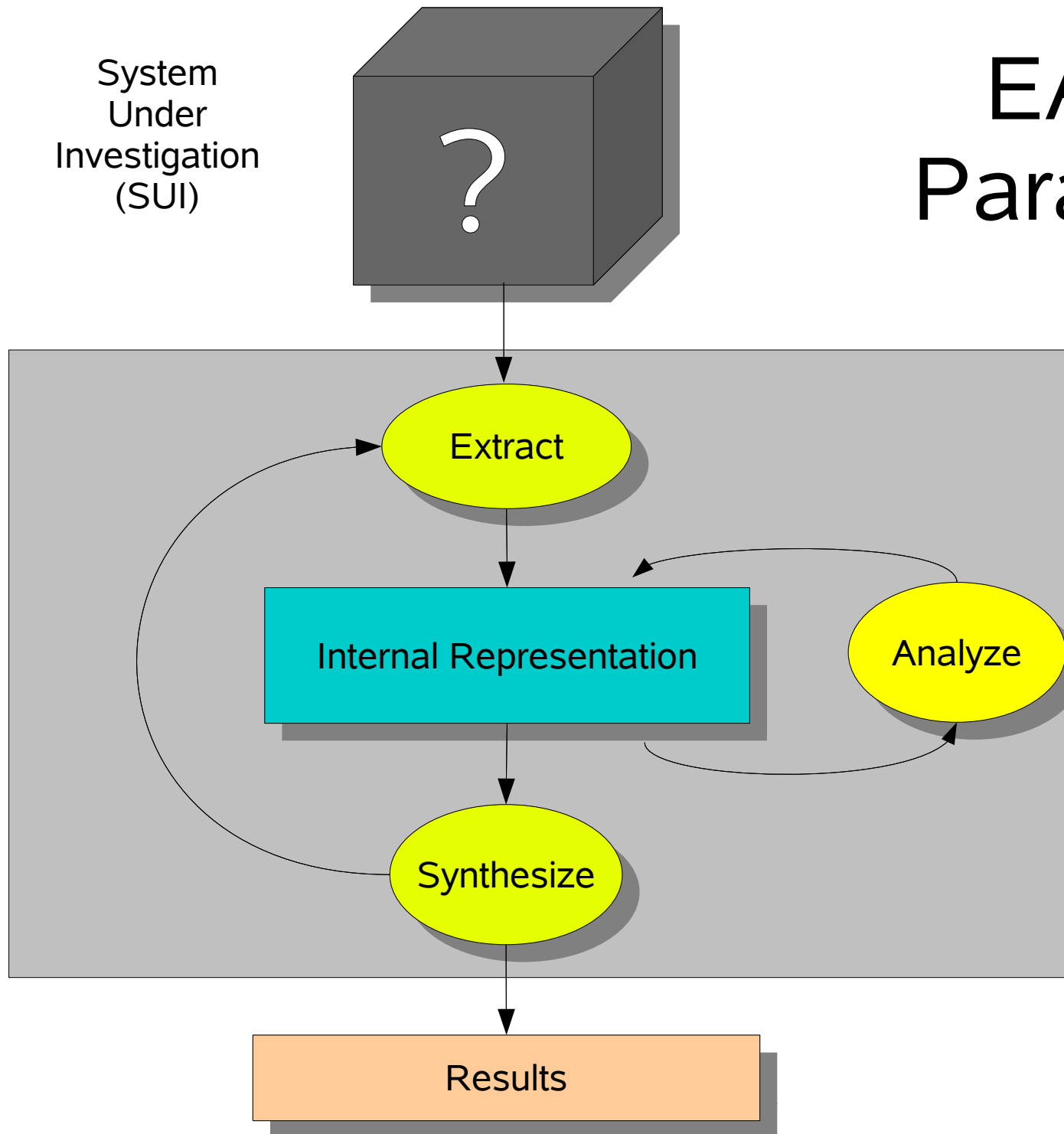
- How to parse source code
- How to extract facts from it
- How to perform computations on these facts
- How to generate new source code
- How to synthesize other information

EASY: Extract-Analyze-SYnthesize Paradigm



System
Under
Investigation
(SUI)

EASY Paradigm



What tools are available to our heroes?

- **Lexical tools:** Grep, Awk, 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, 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
- Many others ...



	Extract	Analyze	Synthesize
Lexical tools	++	+/-	--
Compiler tools	++	+/-	+/-
Attribute grammar tools	++	+/-	--
Relational tools	--	++	--
Transformation tools	--	+/-	++
Rascal	++	++	++

Our Background

- **ASF+SDF Meta-Environment**
 - **SDF**: Syntax Definition Formalism
 - Modular syntax definitions
 - Integrated scanning and parsing
 - Generalized LR parsing
 - **ASF**: Algebraic Specification Formalism
 - Conditional rewrite rules
 - User-defined syntax
- **Rscript**: a relational calculus language
- See <http://www.meta-environment.org>



Where is that background applicable?

	Extract	Analyze	Synthesize
ASF	--	+/-	++
SDF	++	+/-	--
Rscript	--	++	--

Why a new Language?

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



Here comes Rascal to the Rescue

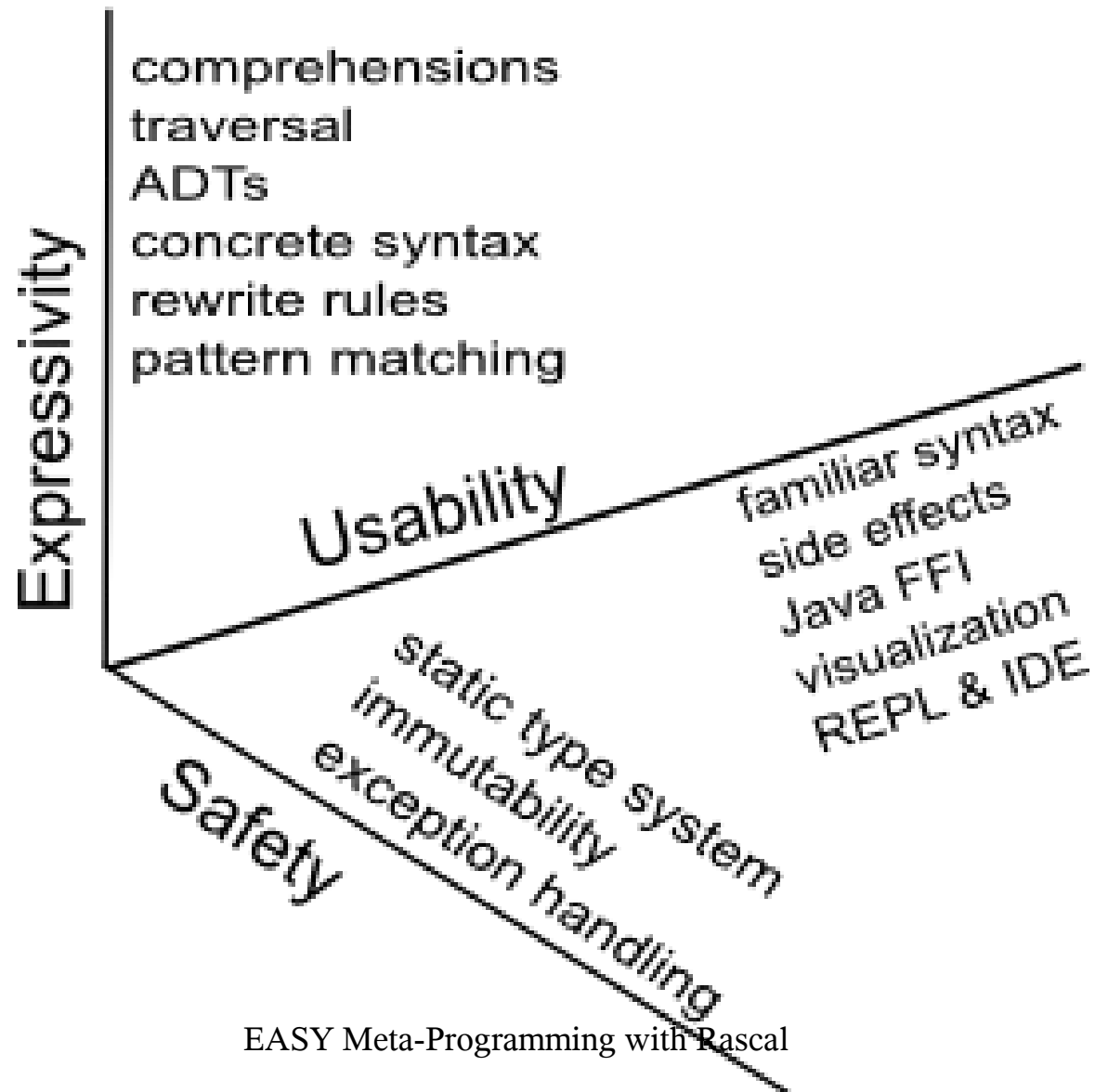


Rascal Elevator Pitch

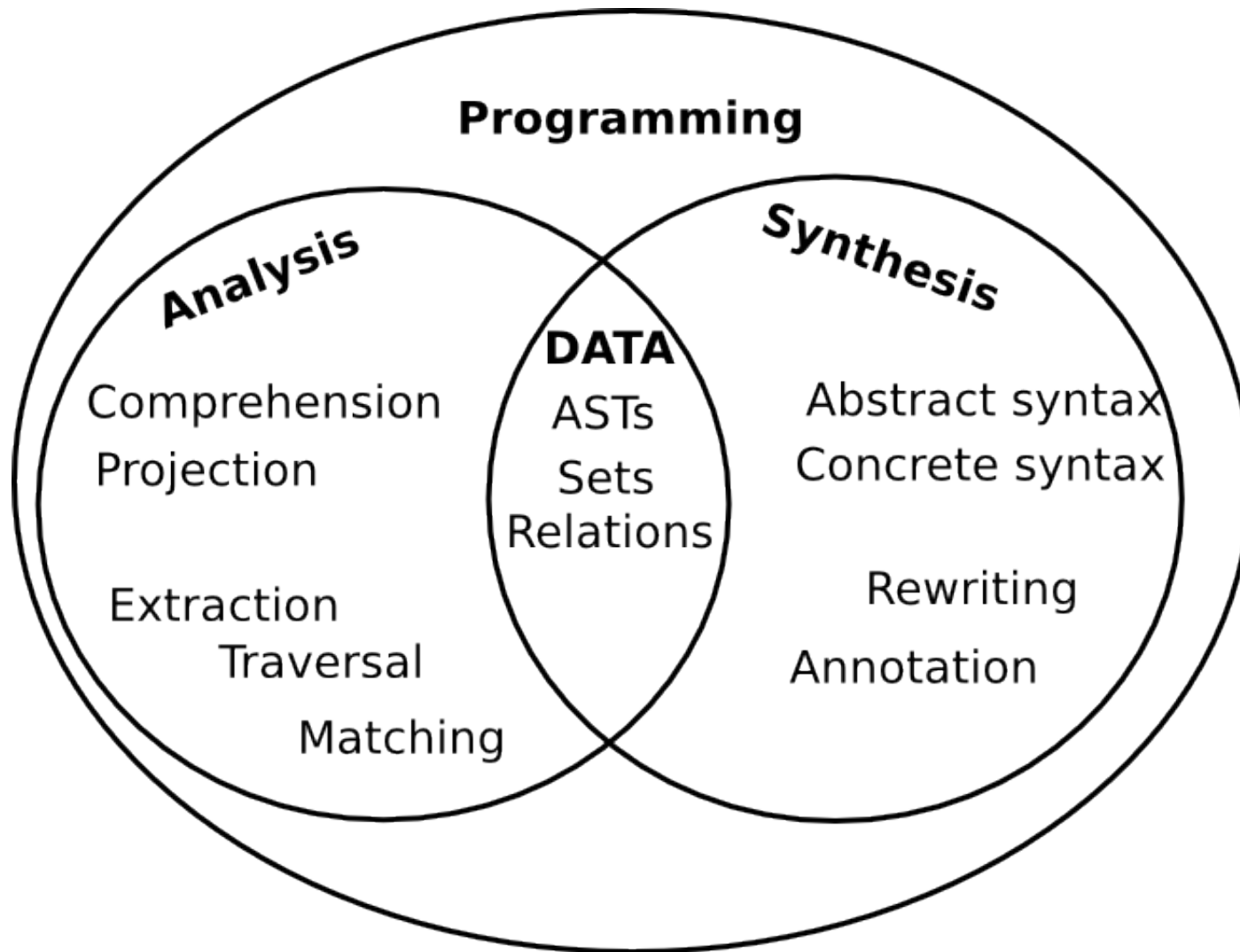
- Sophisticated built-in data types
- Immutable data
- Static safety
- Generic types
- Local type inference
- Pattern Matching
- Syntax definitions and parsing
- Concrete syntax
- Visiting/traversal
- Comprehensions
- Higher-order
- Familiar syntax
- Java and Eclipse integration
- Read-Eval-Print (REPL)



Dimensions of requirements



Bridging analysis and synthesis

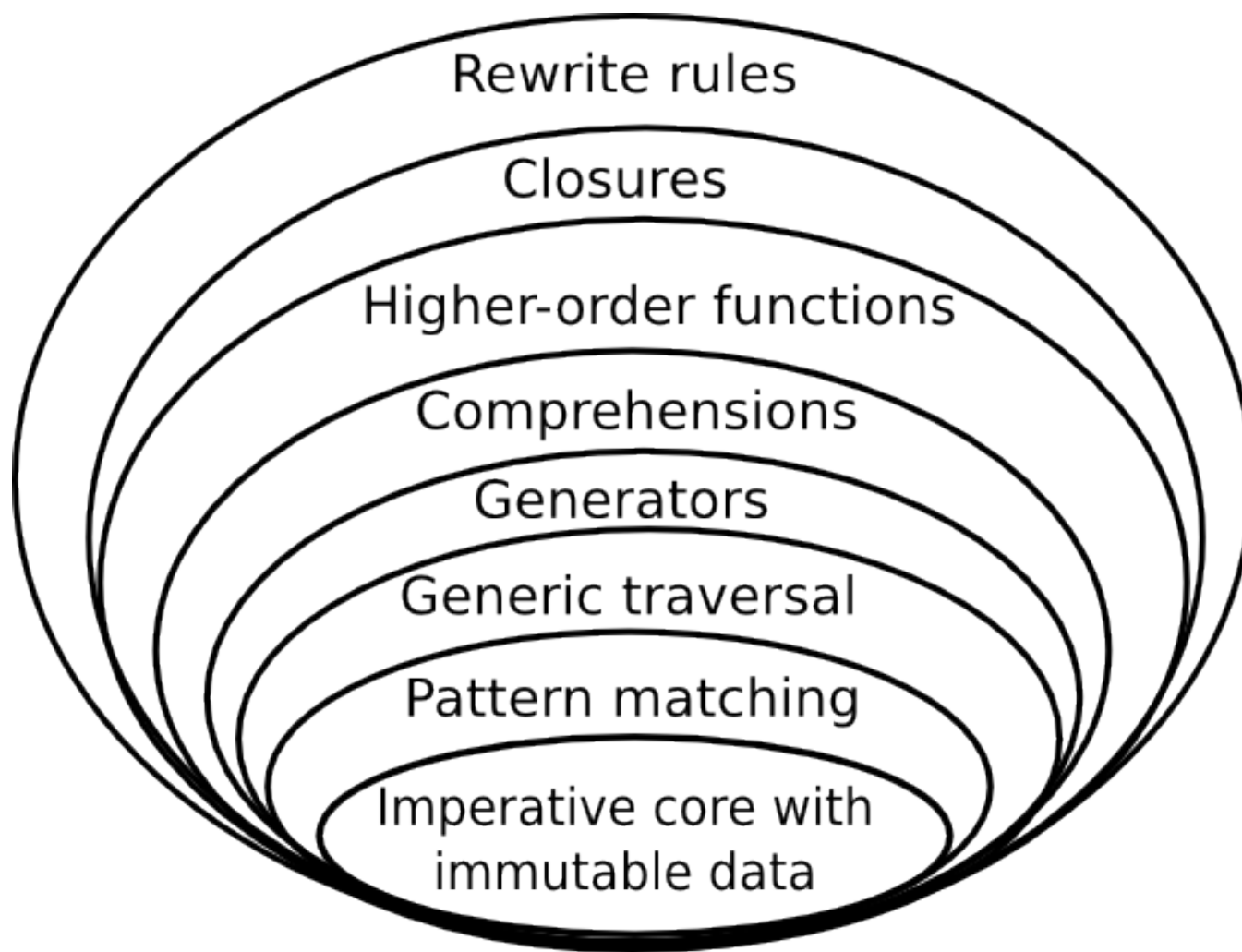


Design Guidelines

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



Rascal's layered design



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



	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
$\text{tuple}[t_1, \dots, t_n]$	<1,2>, <"john", 43, true>
$\text{list}[t]$	[], [1], [1,2,3], [true, 2, "abc"]
$\text{set}[t]$	{}, {1,3,5,7}, {"john", 4.0}
$\text{rel}[t_1, \dots, t_n]$	{<1,10,100>,<2,20,200>}
$\text{map}[t, u]$	(), ("a":1, "b":2,"c":3)
node	f, add(x,y), a("abc",[2,3,4])

User-defined datastructures

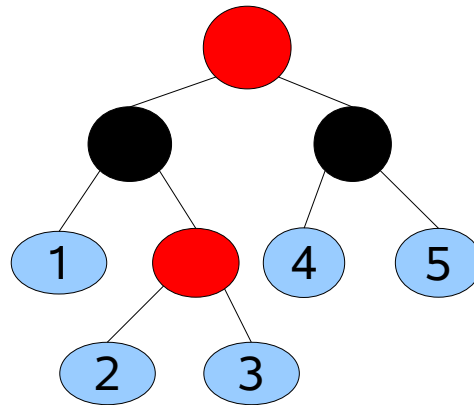
- Named alternatives
 - name acts as constructor
 - can be used in patterns
- Named fields (access/update via . notation)
- All datastructures are a subtype of the standard type *node*
 - Permits very generic operations on data
- Parse trees resulting from parsing source code are represented by the datatype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

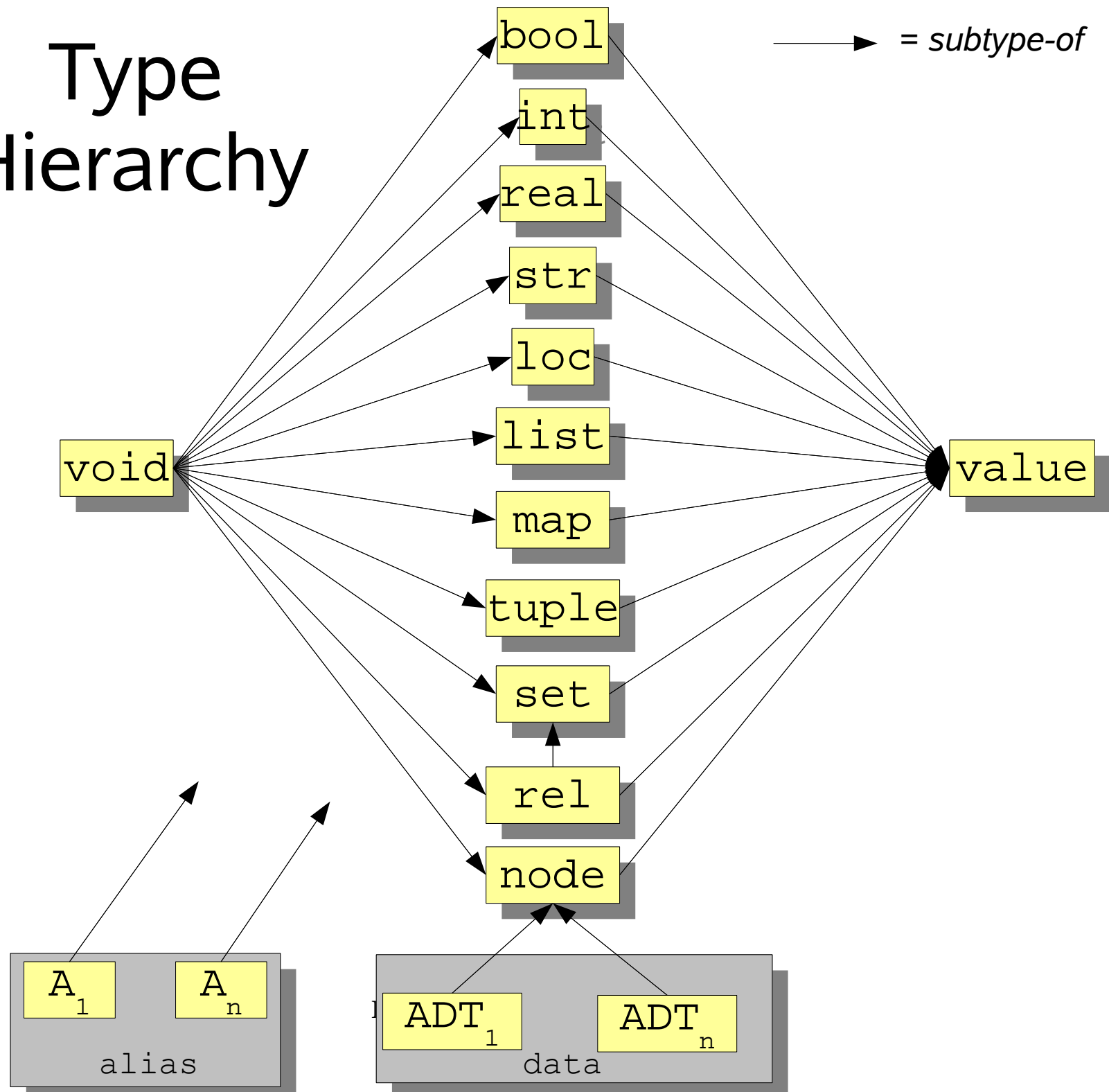


Abstract Syntax

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



Type Hierarchy



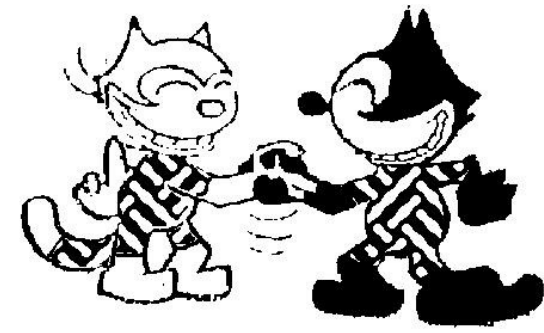
Pattern matching



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

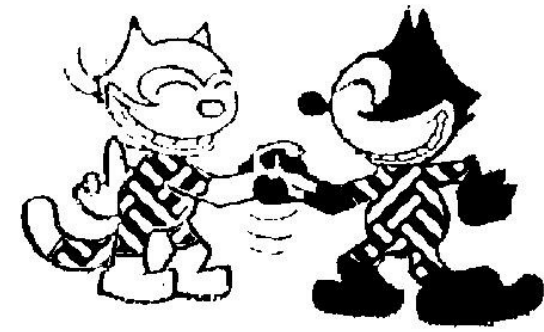


Pattern matching is used in:

- 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

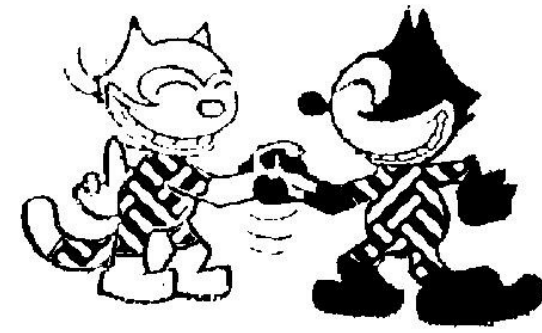
```
whileStat(Exp, Stats*)
```

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



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
```



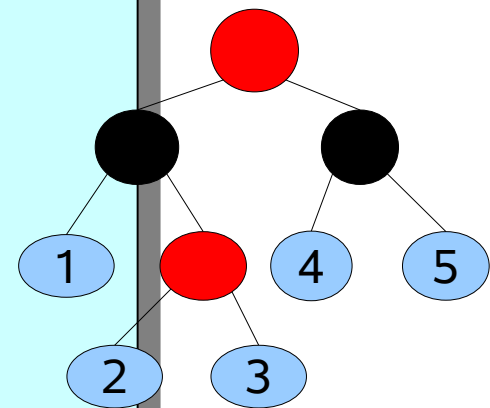
Comprehensions

- 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(id);  
}
```

```
...
```

```
...
```



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

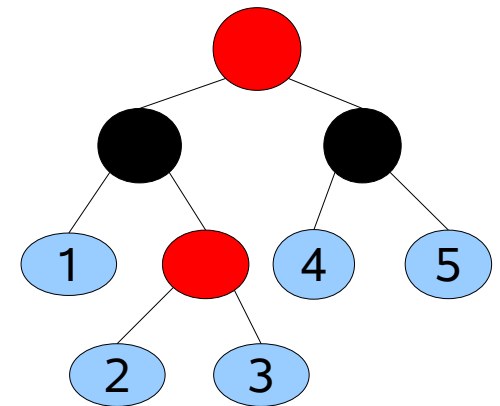
```
switch (P){  
  case whileStat(EXP Exp, Stats*):  
    println("A while statement");  
  case ifStat(Exp, Stats1*, Stat2*):  
    println("An if statement");  
}
```



Visiting

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

Recall the coloured trees (*CTree*):

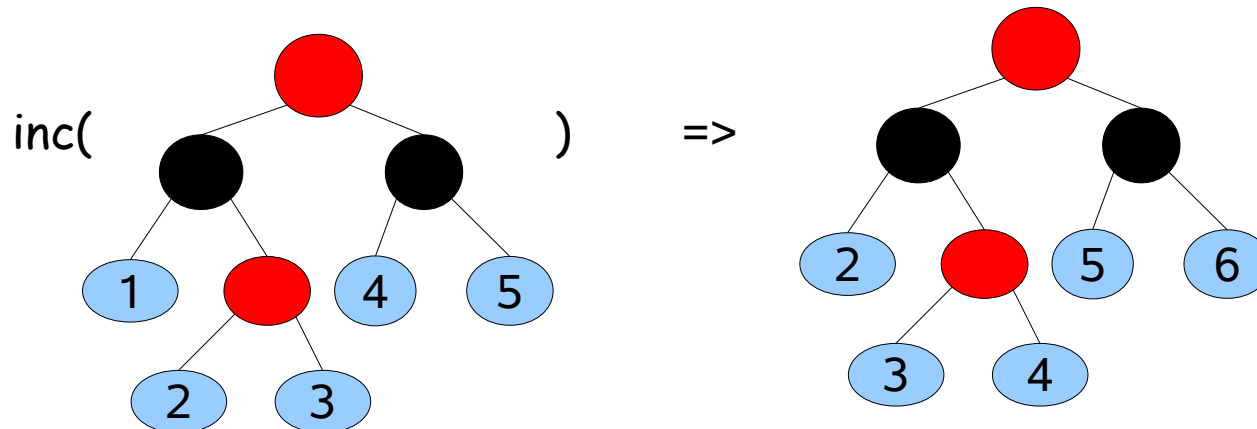


Increment all leaves in a CTree

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

Visit traverses the complete tree and returns modified tree

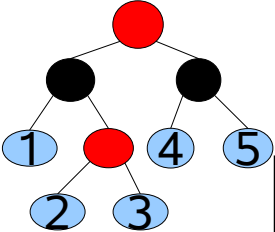
Matching by cases and local subtree replacement



Note

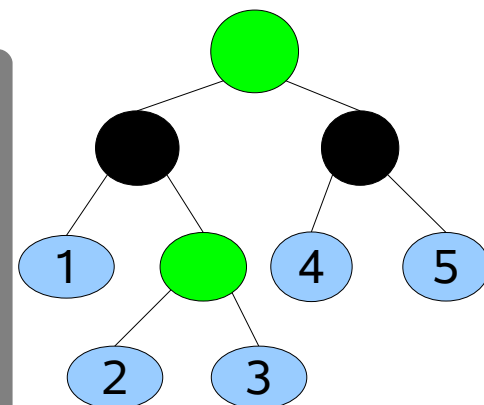
- This code is insensitive to the number of constructors
 - Here 3: leaf, black and red
 - In Java or Cobol: hundreds
- Lexical/abstract/concrete matching
- List/set matching
- Visits can be parameterized with a strategy



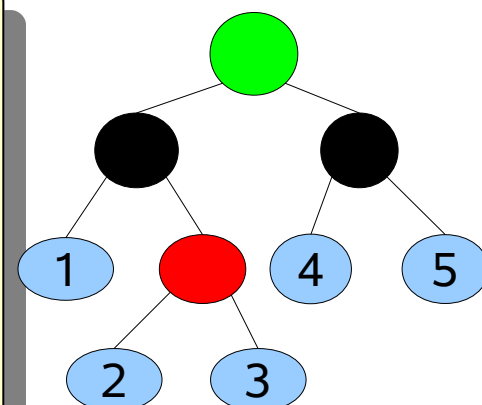


Full/shallow/deep replacement

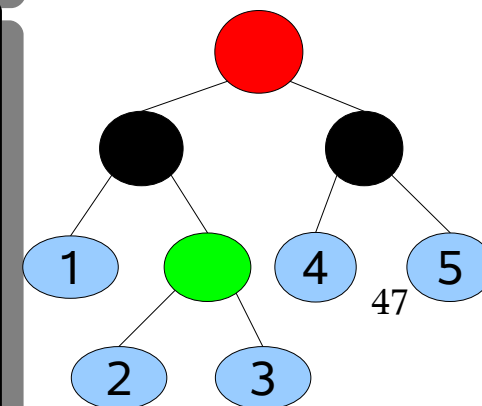
```
public CTree frepl(CTree T) {
    return visit (T) {
        case red(CTree T1, Ctree T2) => green(T1, T2)
    };
}
```



```
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)
    };
}
```



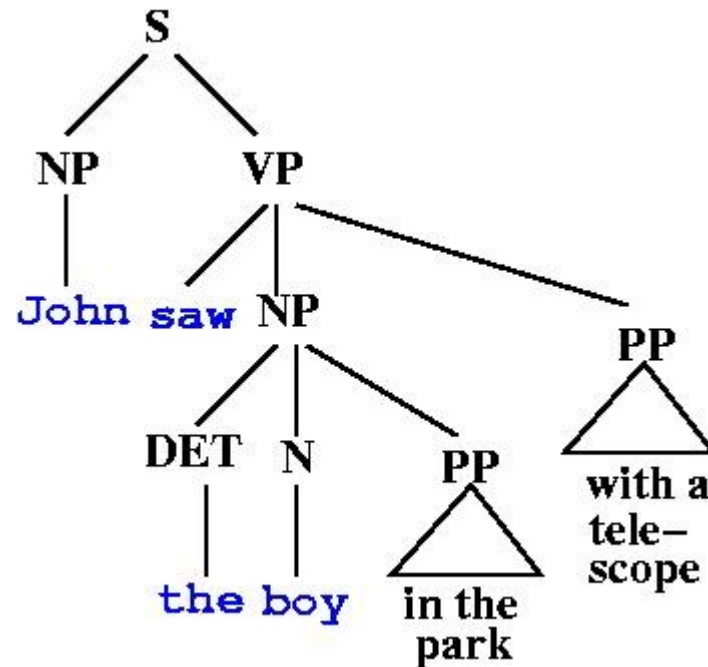
Computing Dominators

- 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};  
}
```



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

Various.rsc

Java.sdf

```
module M
import Various;
import languages::syntax::Java;
```

In *M* we can now use:

Quoted Java fragments : [| ... |]

Unquoted Java fragments
(when unambiguous)

Parse functions for all start symbols



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



Concrete syntax example

```
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> };  
}
```

Syntactic type

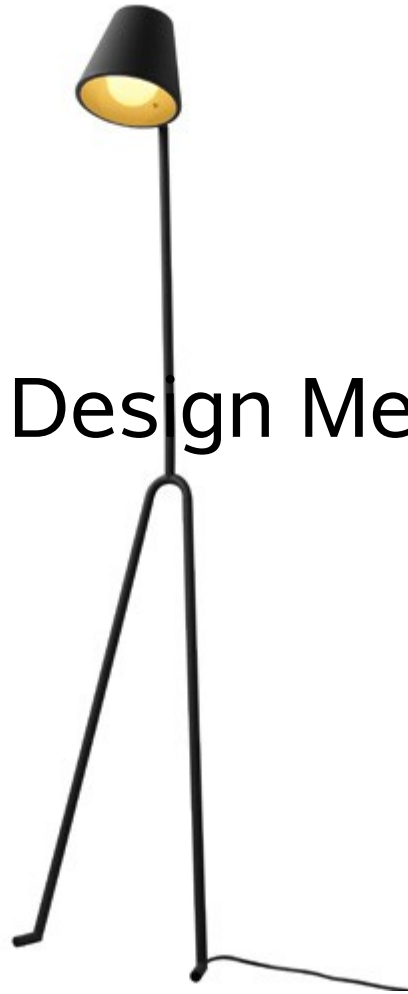
Quoted concrete
syntax expression

Variable
interpolation

Unquoted concrete
syntax expression

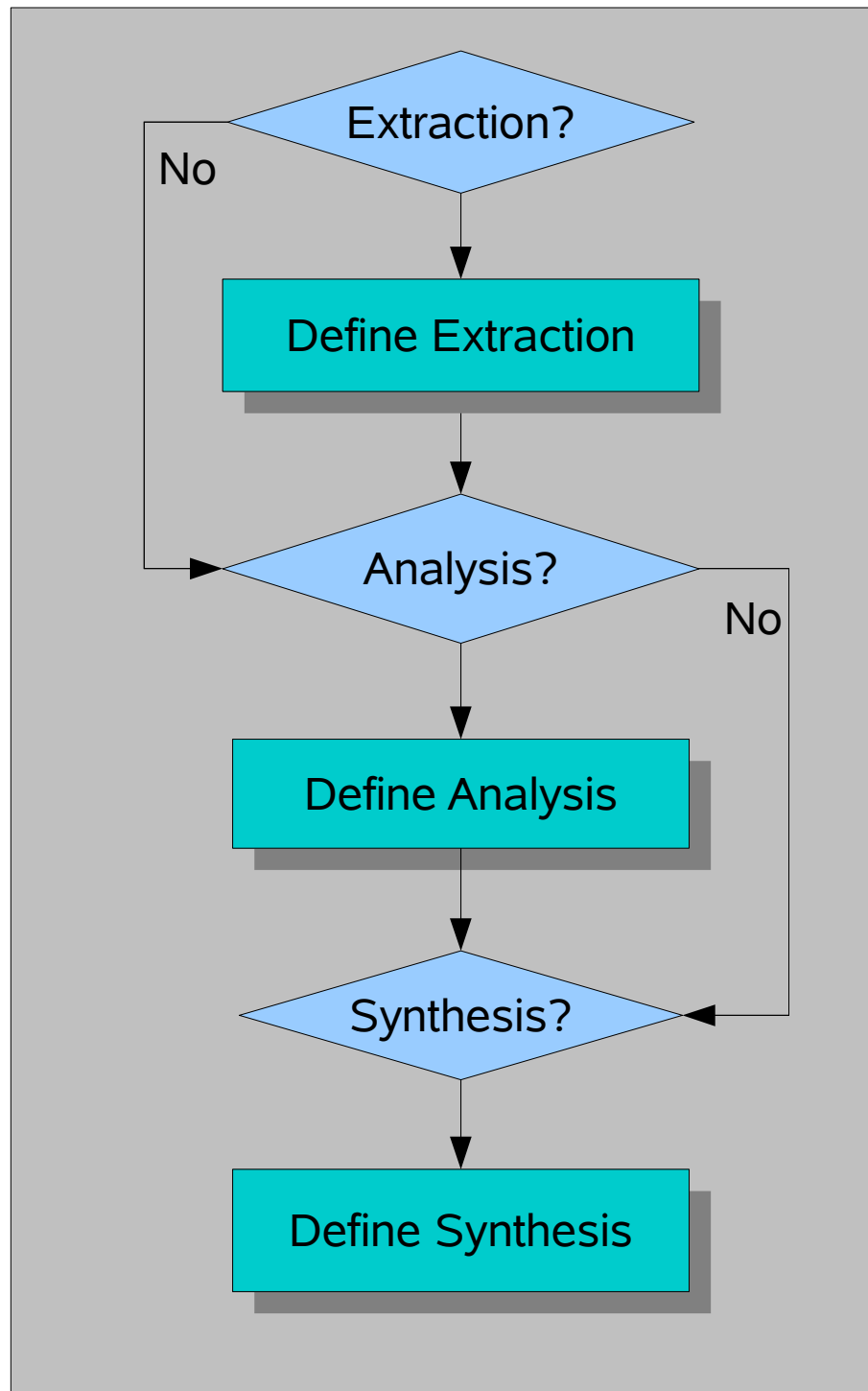


Is there a Rascal Design Methodology?

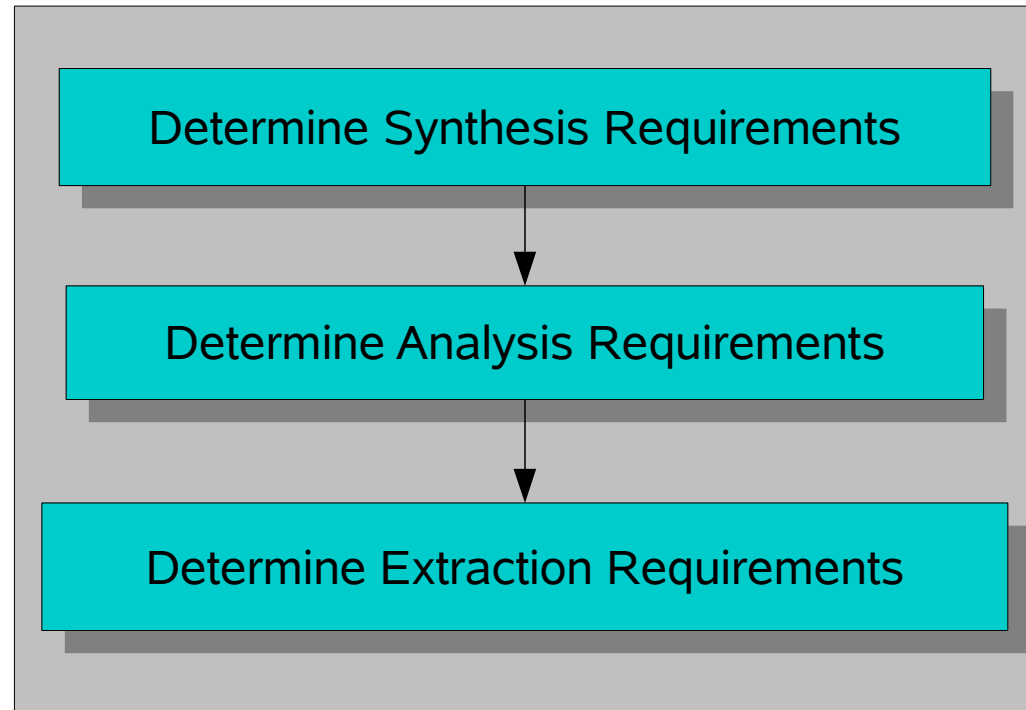


Rascal

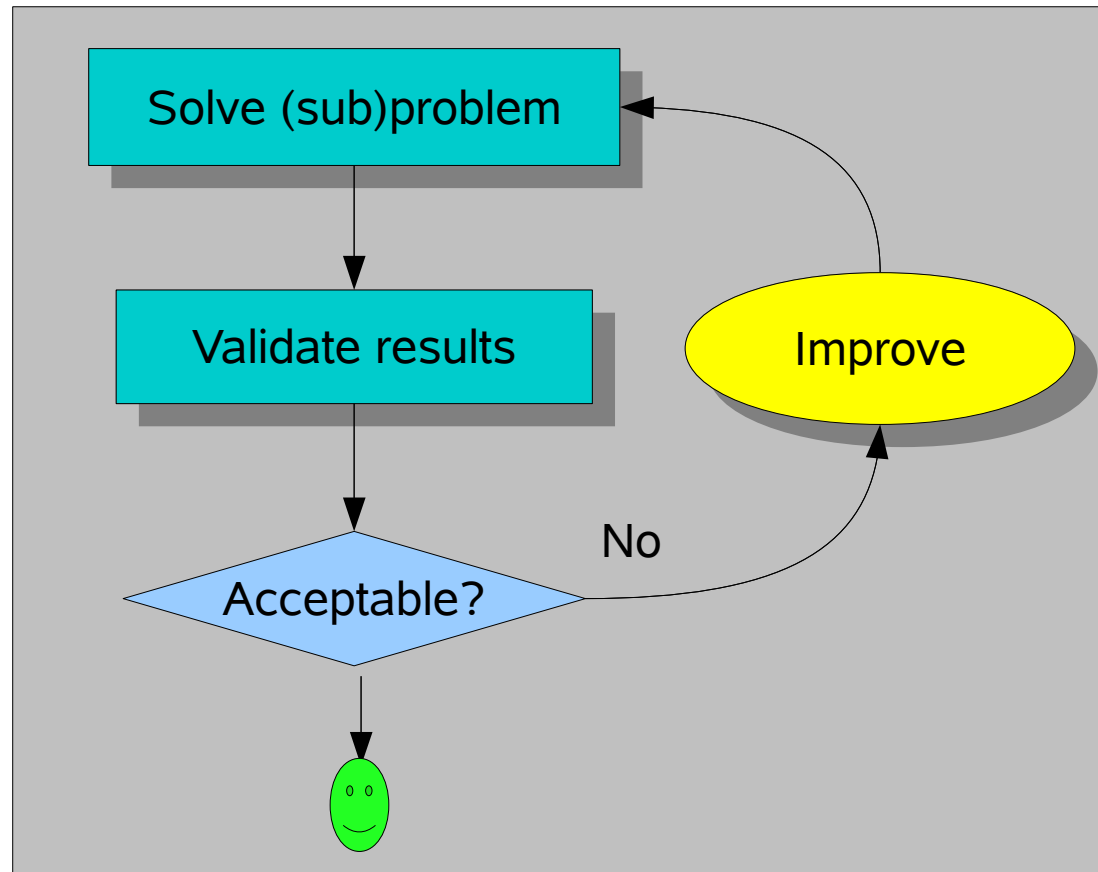
Workflow



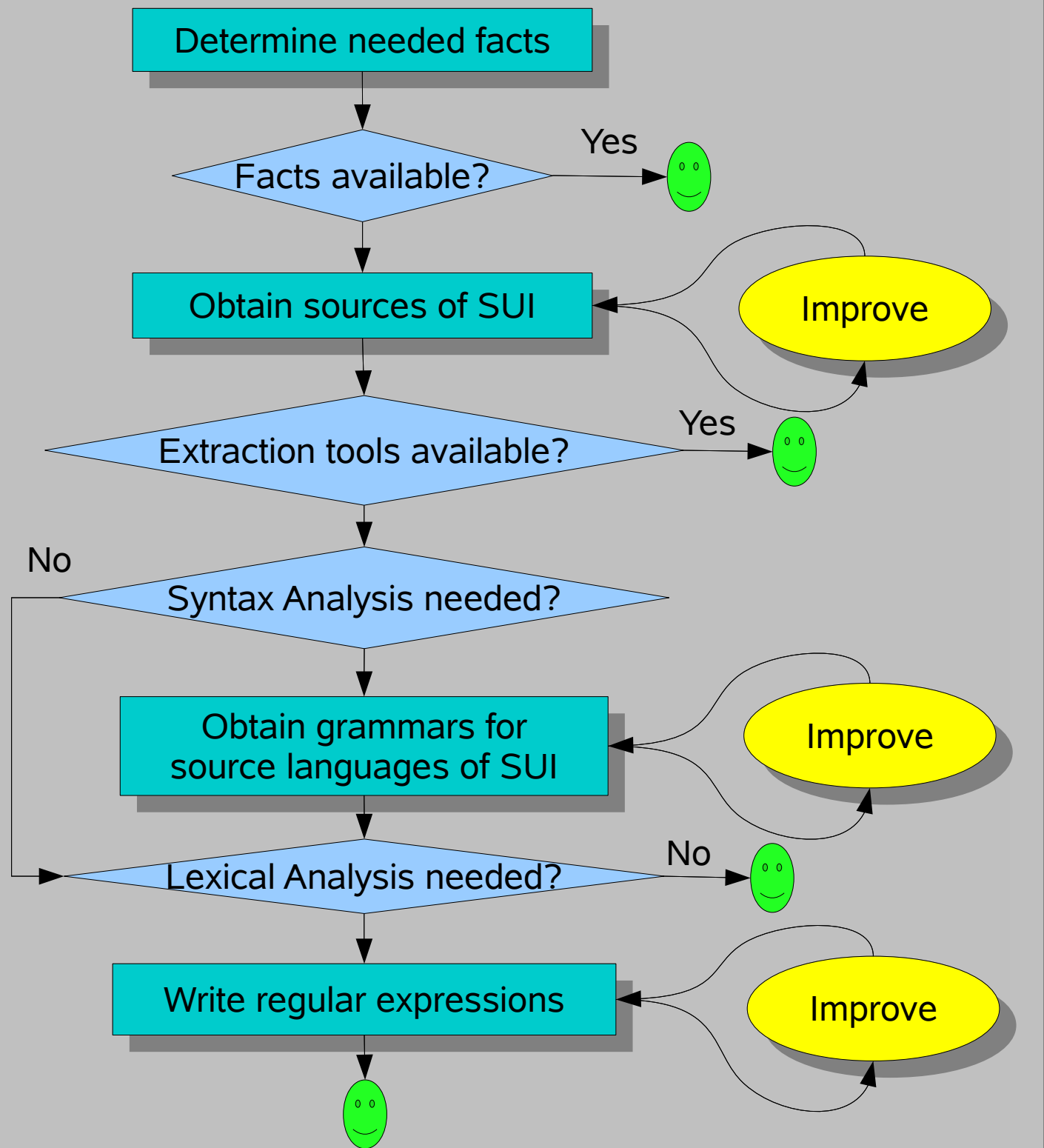
Requirements Analysis



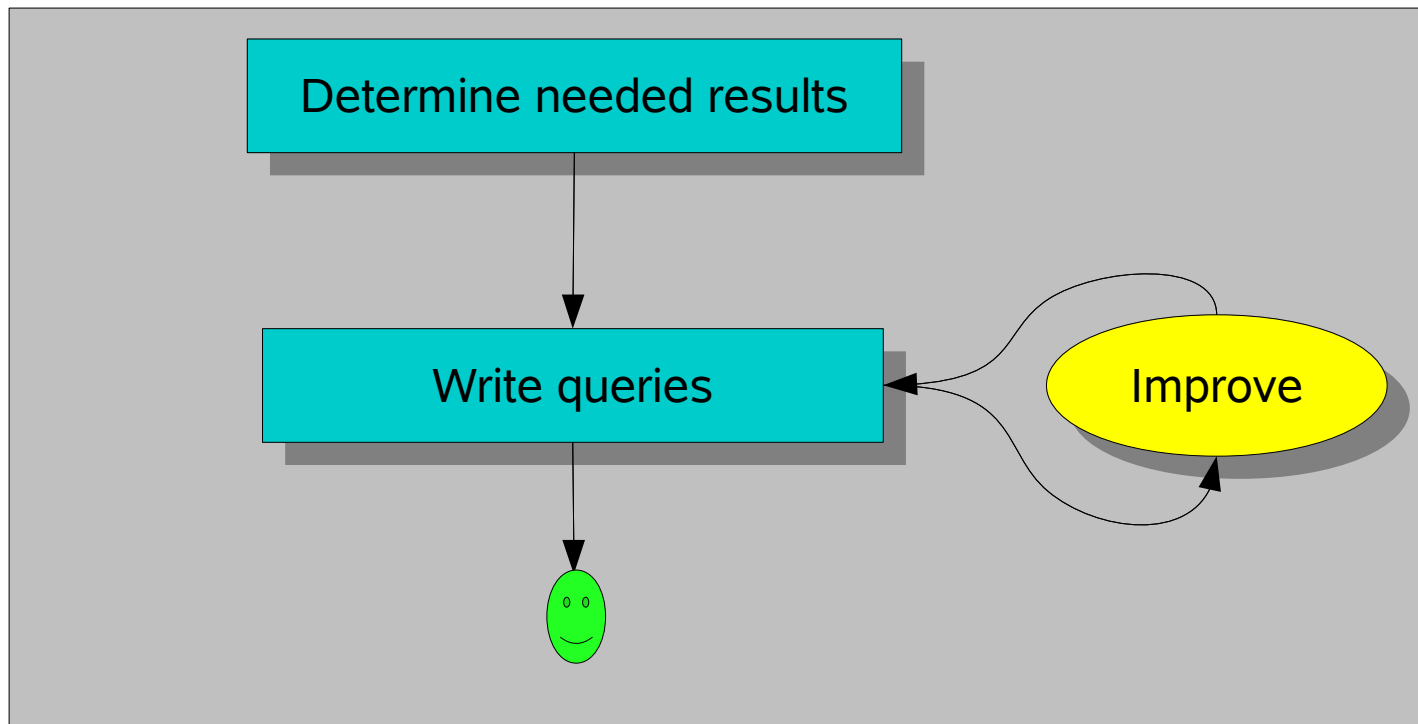
Validation



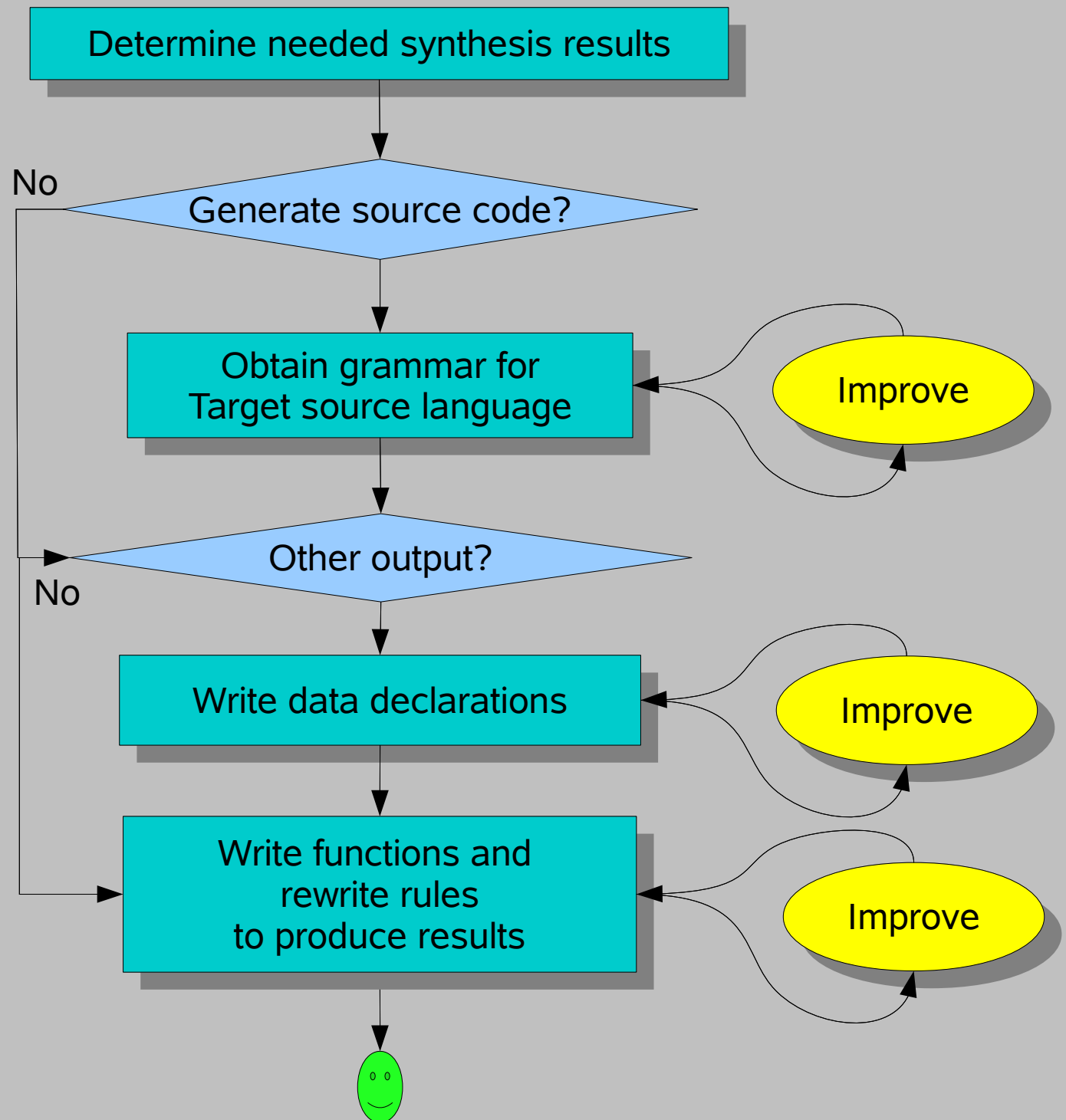
Extraction Workflow



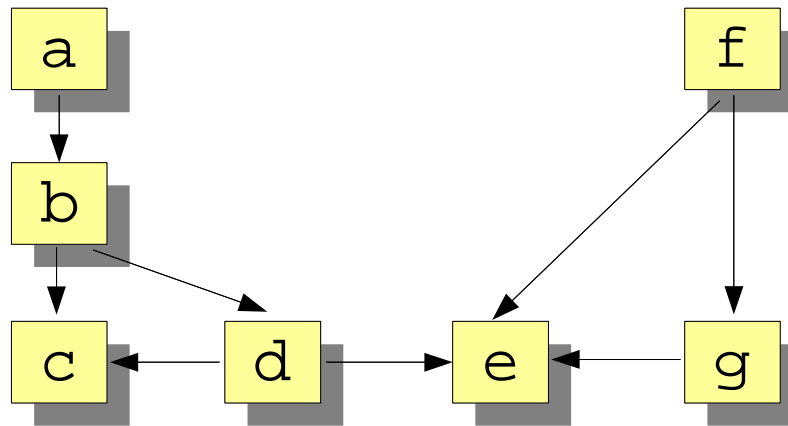
Analysis Workflow



Synthesis Workflow



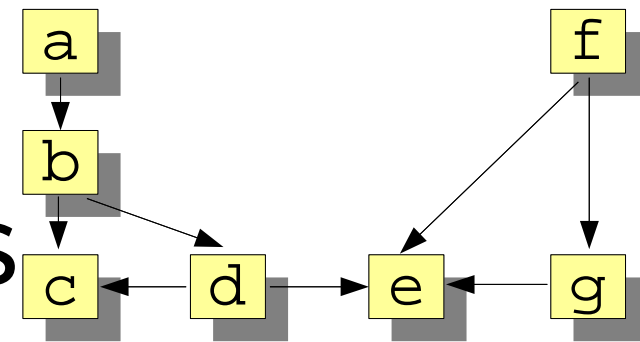
Analyzing the call structure of an application



`rel[str, 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?

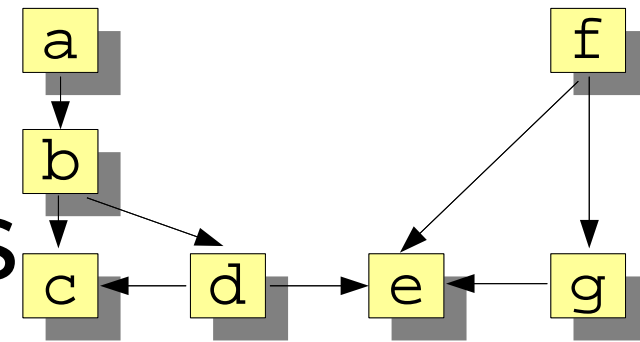
- `int nprocs = size(carrier(calls));`

- 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
- $\text{set}[\&T] \text{ top}(\text{rel}[\&T, \&T] R) = \text{domain}(R) - \text{range}(R)$

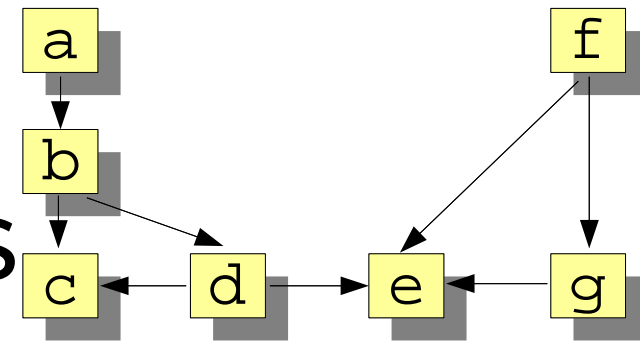


Intermezzo: Bottom

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



Some questions



- What are the indirect calls between procedures?
 - `rel[str,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">}`

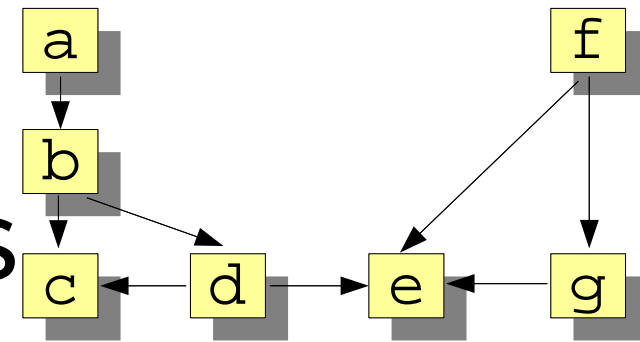
- What are the calls from entry point a?

The image of domain value "a"

- `set[str] calledFromA = closureCalls["a"]`
- `{"b", "c", "d", "e"}`



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

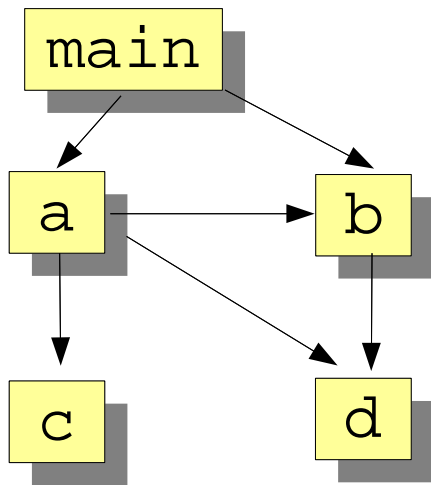


Component Structure of Application

- 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



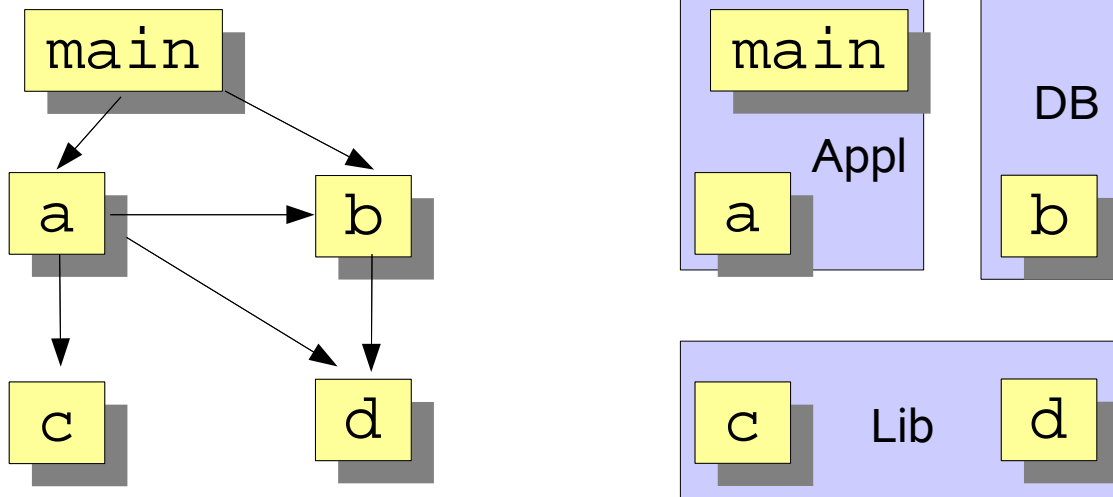
Calls



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



PartOf

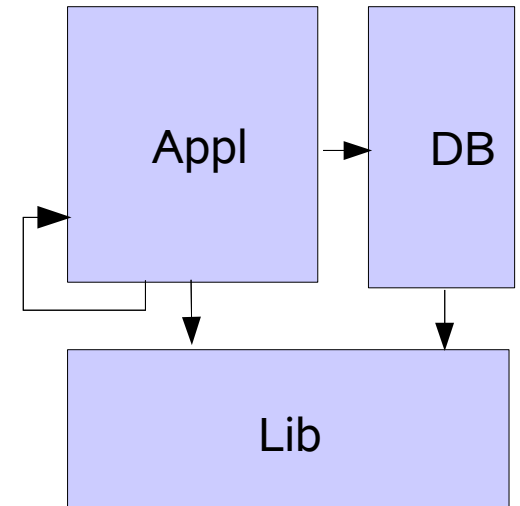
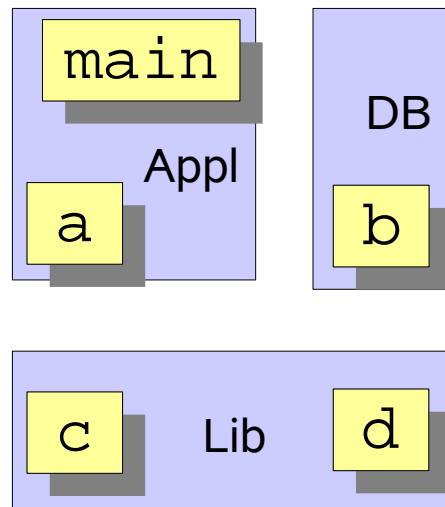
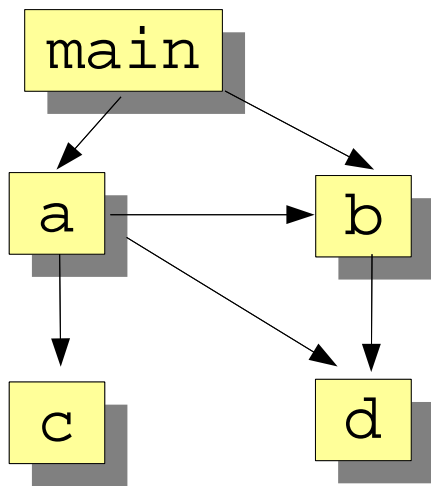


```
set[comp] Components = {"Appl", "DB", "Lib"};
```

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



lift



$\text{rel}[\text{comp}, \text{comp}] \text{ lift}(\text{rel}[\text{proc}, \text{proc}] \text{ aCalls}, \text{rel}[\text{proc}, \text{comp}] \text{ aPartOf}) =$
 $\{ \langle C1, C2 \rangle \mid \langle \text{proc } P1, \text{proc } P2 \rangle \leftarrow \text{aCalls},$
 $\langle \text{comp } C1, \text{comp } C2 \rangle \leftarrow \text{aPartOf}[P1] \times \text{aPartOf}[P2] \};$

$\text{rel}[\text{comp}, \text{comp}] \text{ ComponentCalls} = \text{lift}(\text{Calls2}, \text{PartOf})$

Result: $\{ \langle \text{"DB"}, \text{"Lib"} \rangle, \langle \text{"Appl"}, \text{"Lib"} \rangle, \langle \text{"Appl"}, \text{"DB"} \rangle, \langle \text{"Appl"}, \text{"Appl"} \rangle \}$



The Rascal Standard Library

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

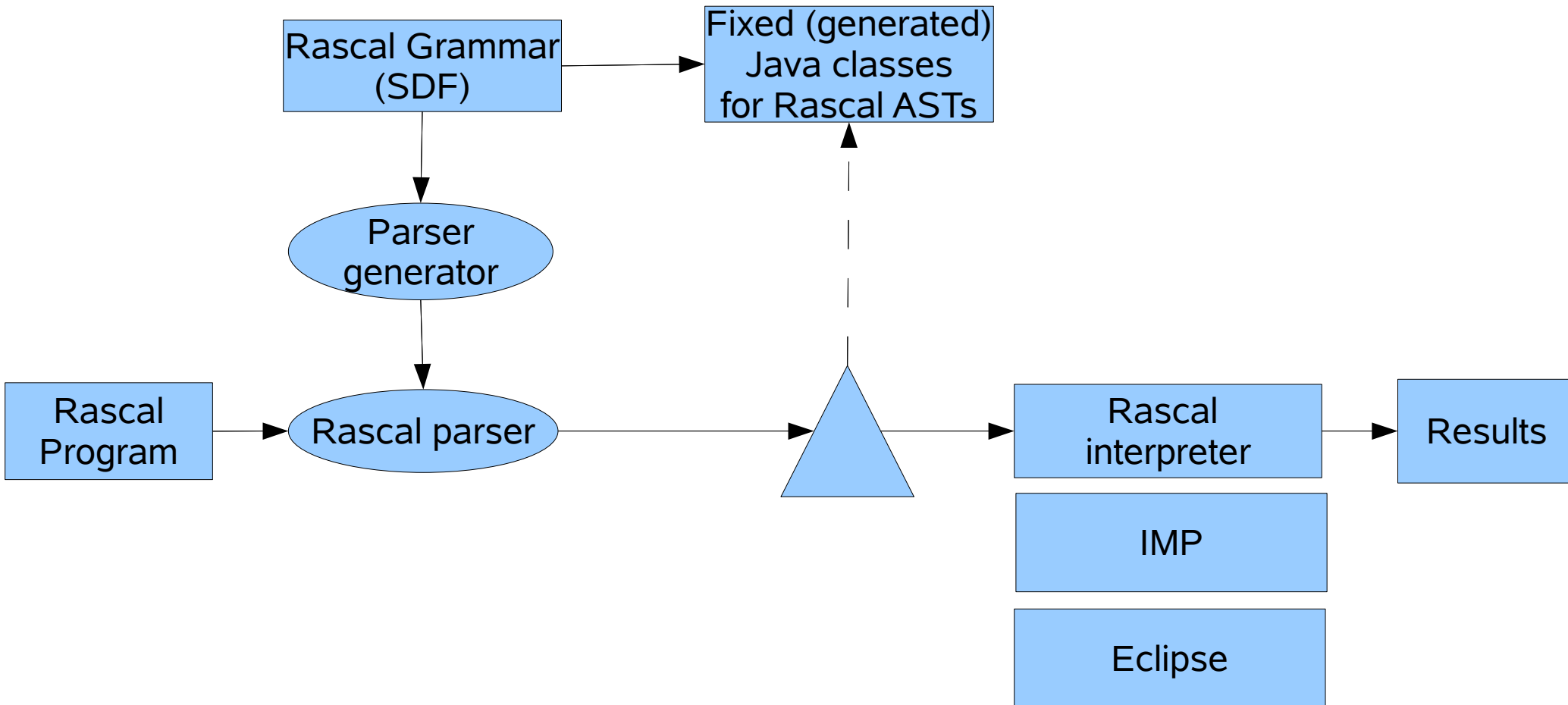


Rascal Status

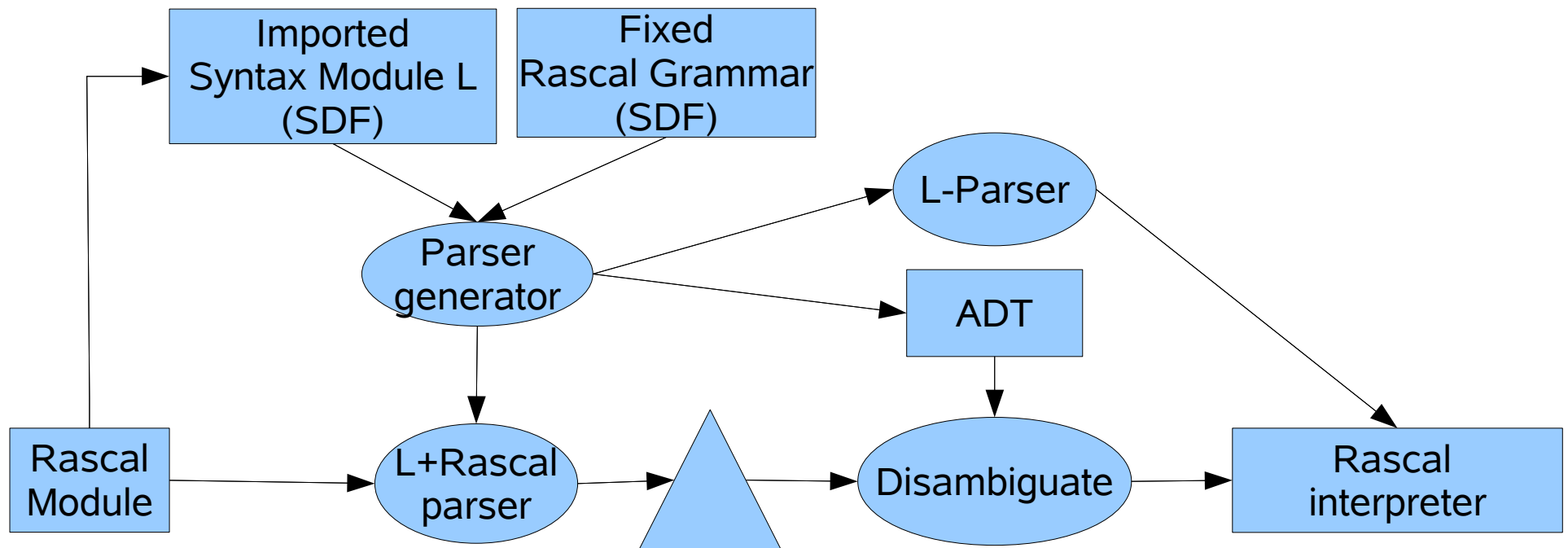
- An interpreter for the core language is well underway.
- All the above examples (and many more!) run.
- Full language expected to be implemented mid 2009.
- Launch: at GTTSE summerschool in Braga Portugal



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



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



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 Rascal implementation:

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



Questions

