EASY Meta-Programming with Rascal

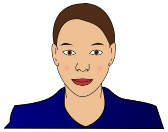
Leveraging the Extract-Analyze-SYnthesize Paradigm

Paul Klint

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

Cast of Our Heros

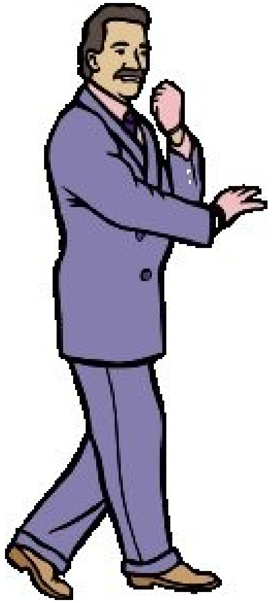
- Alice, system administrator
- Bernd, forensic investigator
- Charlotte, financial engineer
- Daniel, multi-core specialist
- Elisabeth, model-driven engineering specialist



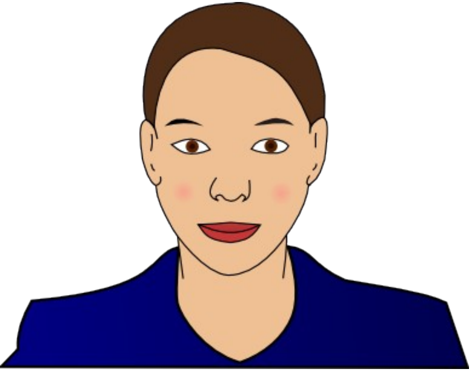
Meet Alice

- 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

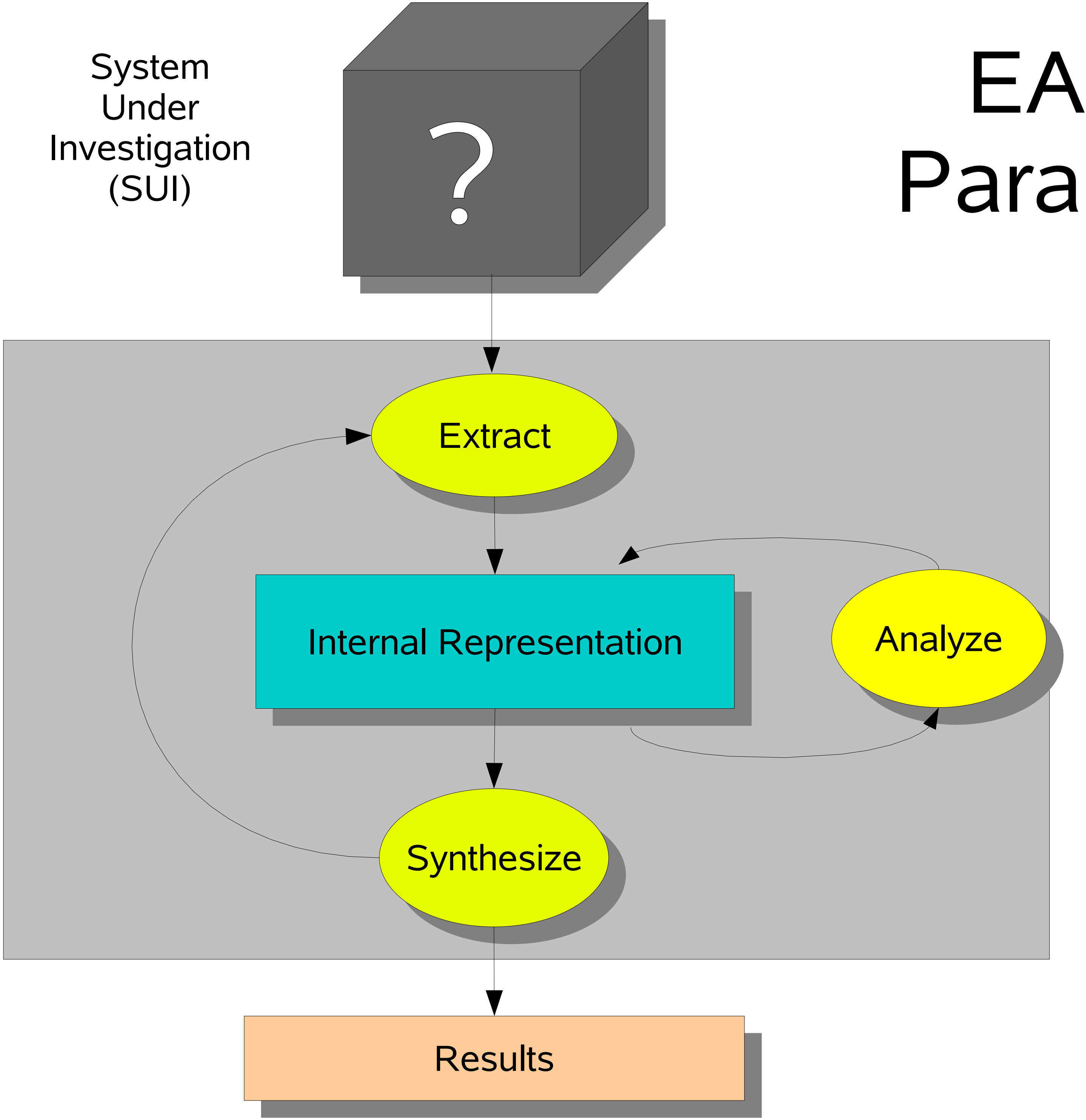
- Elisabeth is software architect at an airplane manufacturer
- Objective: Model reliability of controller software
- Solution:
  - describe software architecture with UML and add reliability annotations
  - Extract reliability information and synthesize input for statistcics tool
  - Generate executable 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

EASY Paradigm



System Under Investigation (SUI)

?

Extract

Internal Representation

Analyze

Synthesize

Results

What tools are available to our heros?

- 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 heros?

- 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-Enviroment
  - 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

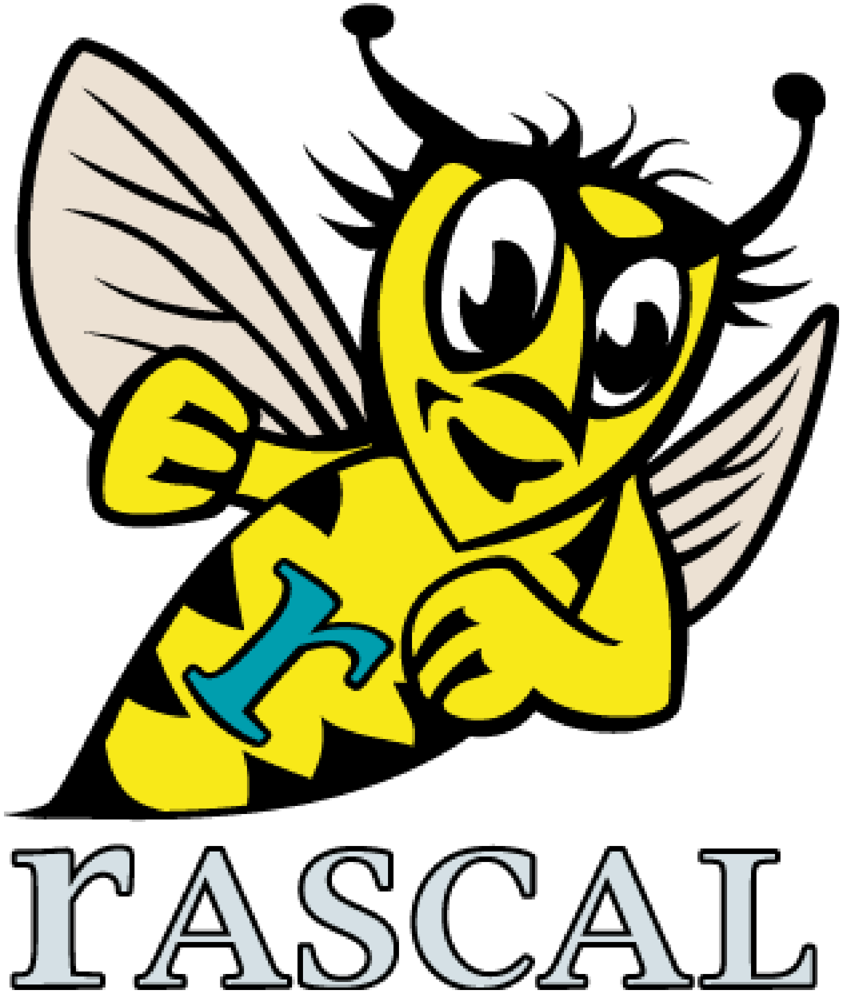
Where applicable?

	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
  - 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
- Static safety
- Generic types
- Local type inference
- Pattern Matching
- Syntax definitions and parsing
- Visiting
- Functions as values
- Familiar syntax
- Eclipse integration

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, this is my first Rascal program
ok
```

Hello (as function in module)

```
module demo::Hello
import IO;
public void hello() {
  println(“Hello, this is my first Rascal program”);
}
```

```
rascal > import demo::Hello;
void: null

rascal> hello();
Hello, this is my first Rascal program
void: null
```

Factorial

```
module demo::Factorial
public int fac(int N){
  return N <= 0 ? 1 : N * fac(N - 1);
}
```

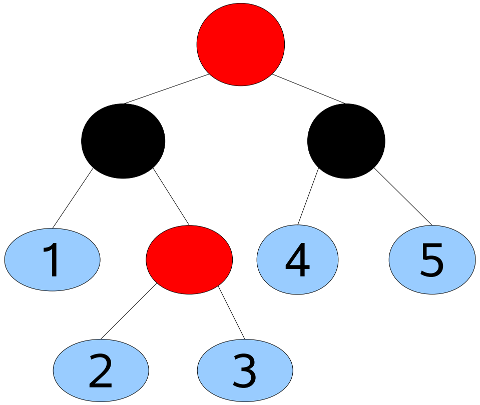
```
rascal> import demo::Factorial;
ok

rascal> fac(47);
int: 258623241511168180642964355153611979996
9197632389120000000000
```

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)));
```



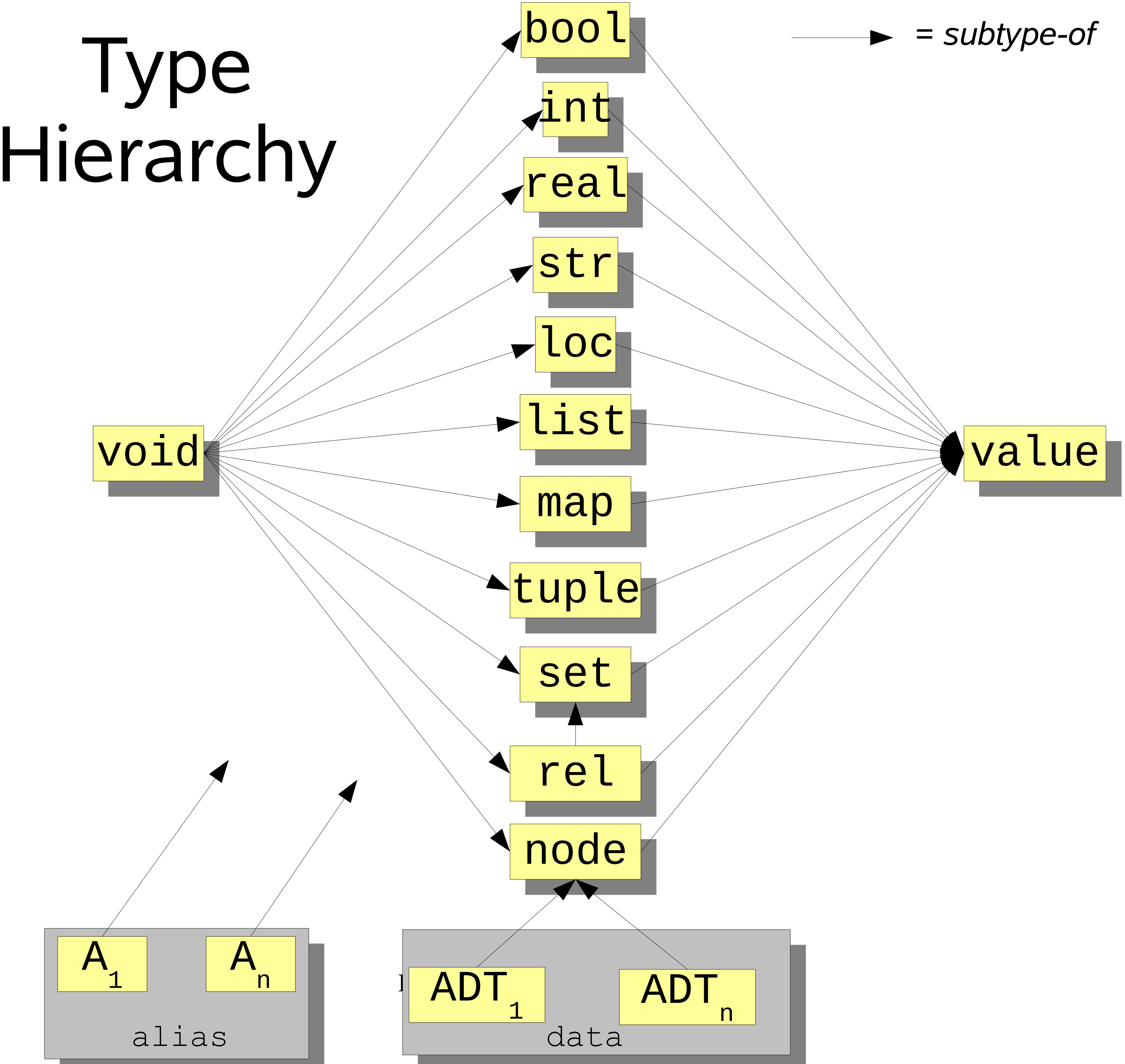
Types and Values

- Atomic: bool, int, real, str, loc (source code location)
- Structured: list, set, map, rel (n-ary relation), abstract data type, parse tree
- Typesystem:
    - Types can be parameterized (polymorphism)
    - All function signatures are explicitly typed
    - Inside function bodies types can be inferred (local type inference)

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`

Type Hierarchy



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
  - Matching concrete code patterns
  - Synthesizing source code

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 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
- Abstract: match data types
- Concrete: match parse trees

Abstract/Concrete patterns support:

- List matching
- Set matching
- Named subpatterns
- Anti-patterns

Enumerators and Tests

- Enumerate the elements in a value:
  - 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!)
- Tests determine properties of a value
- Enumerators and tests are used in comprehensions

Comprehensions

- Comprehensions for lists, sets and maps
- Enumerators generate values and tests filter them

```
rascal> {x * x | int x ← [1 .. 10], x % 3 == 0};
set[int]: {9, 36, 81}

rascal> [ n | leaf(int n) ← rb ];
list[int]: [1,2,3,4,5]
```

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
```

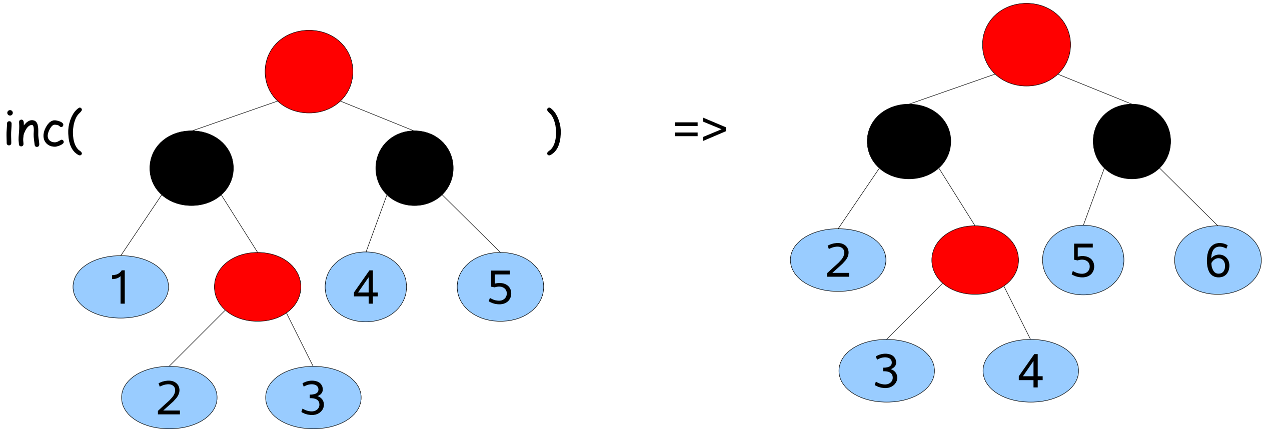
Visiting

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

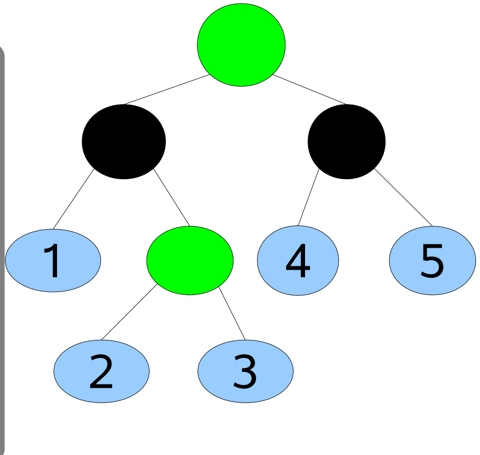


Note

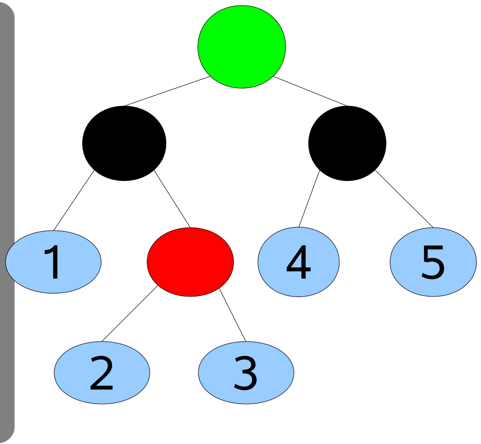
- This code is insensitive to the number of constructors
  - Here: 4
  - In Java or Cobol: hundreds
- Lexical/abstract/concrete matching
- List/set matching
- Visits can be parameterized with a strategy

Full/shallow/deep replacement

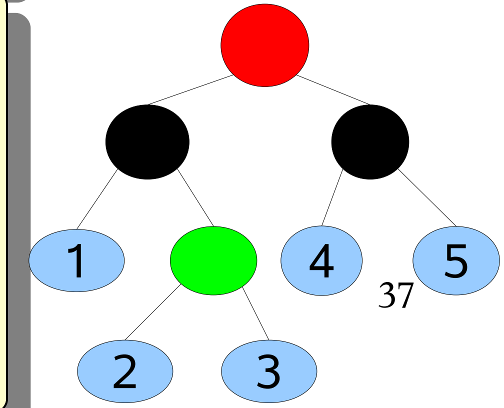
```
public CTree frepl(CTree T) {
    return visit (T) {
        case red(CTree T1, Ctree T2) =>  green(T1, T2)
    };
}
```



```
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)
    };
}
```



Rewrite rules

Counting words in a string

```
public int countLine(str S){
  int count = 0;
  for(/[a-zA-Z0-9]+/: S){
      count += 1;
  }
  return count;
}
```

countLine("'Twas brillig, and the slithy toves") => 6

Finding date-related variables

```
module DateVars
import Cobol;

set[Var] getDateVars(CobolProgram P){

   return {V | Var V : P,

              ^.*(date|dt|year|yr).*$/i
```

Import the COBOL grammar

Traverse P and return all occurrences of variables

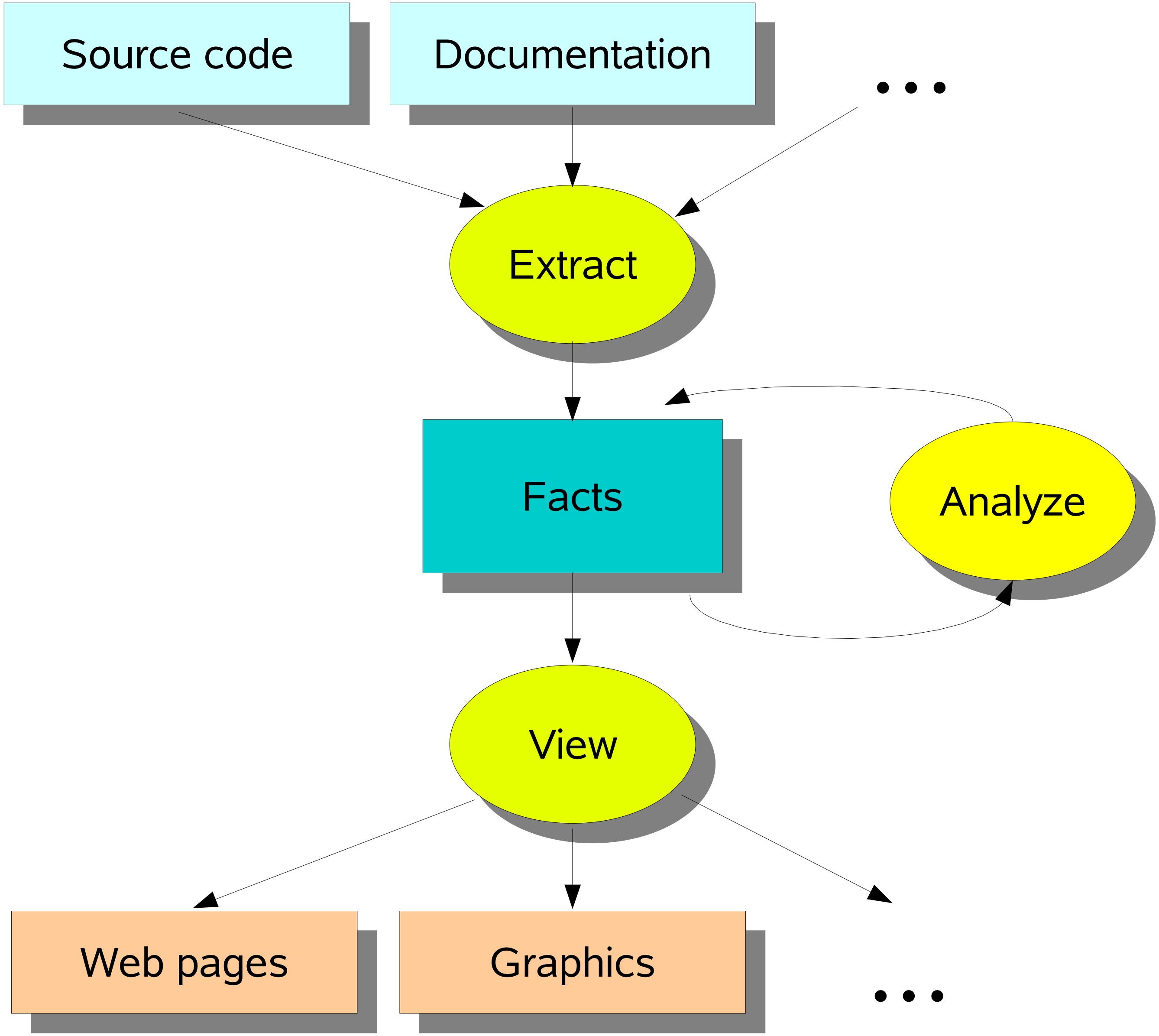
Variable name matches a date-related heuristic

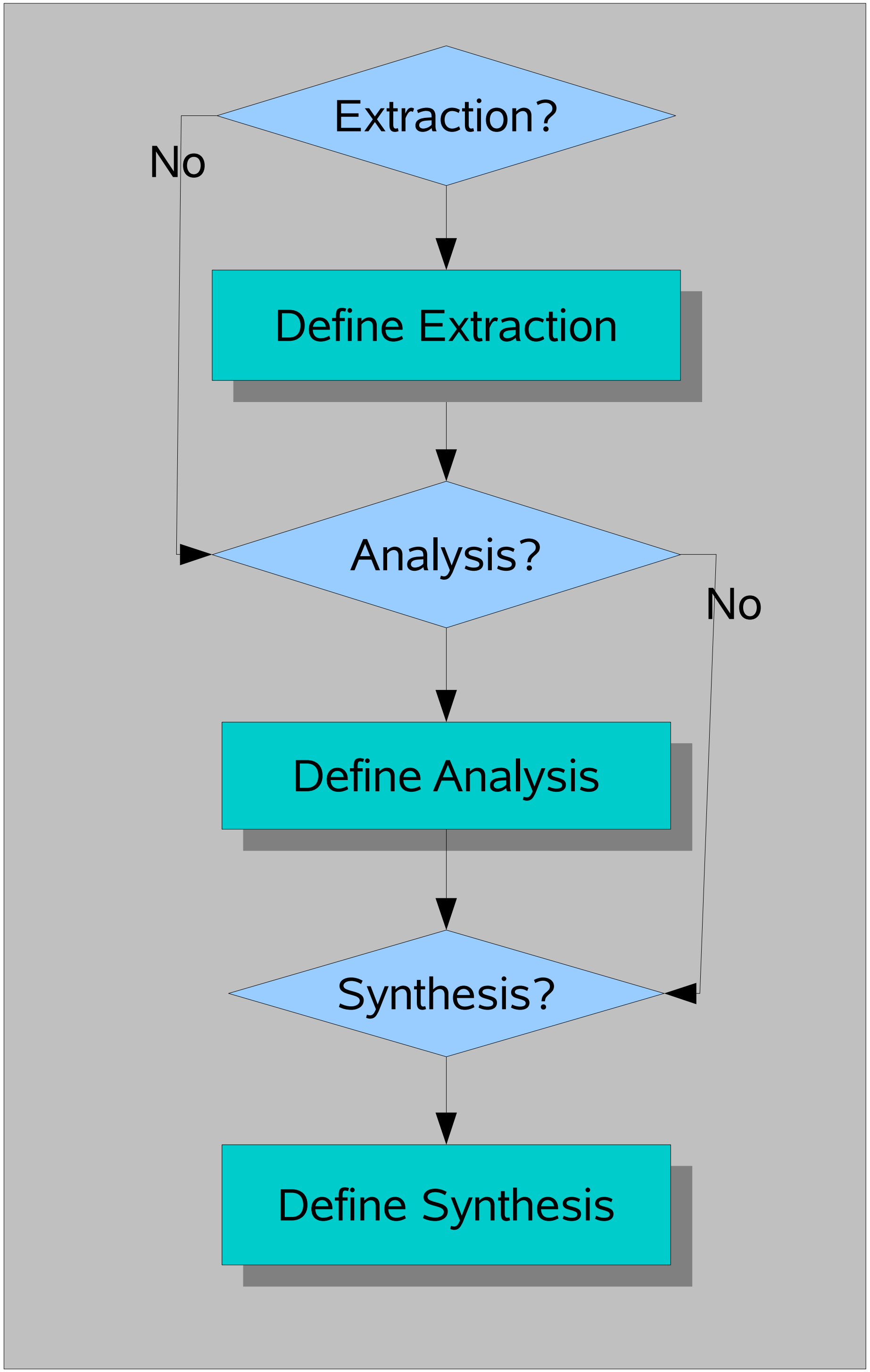
Put variables that match in result

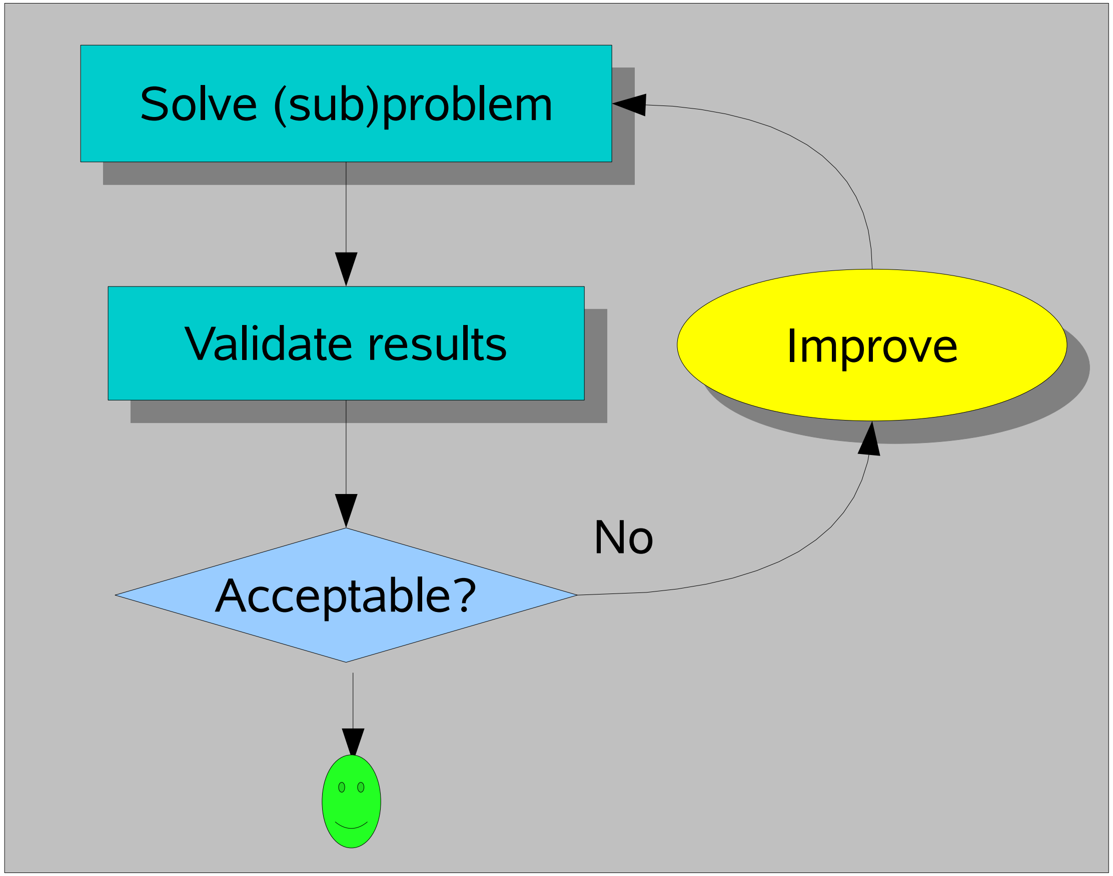
Computing Dominators

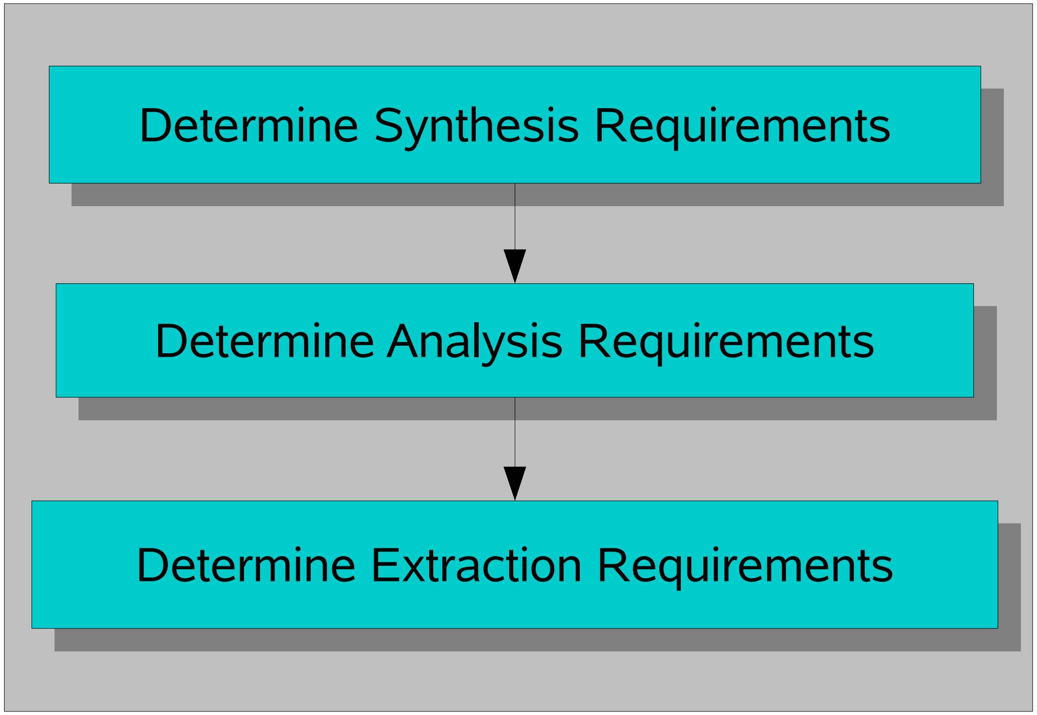
- A node M dominates other nodes S in the flow graph iff all path from the root to a node in S contain M

```
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};
}
```


Source code
Documentation
. . .
Extract
Facts
Analyze
View
Web pages
Graphics
. . .


Extraction?
No
Define Extraction
Analysis?
No
Define Analysis
Synthesis?
Define Synthesis

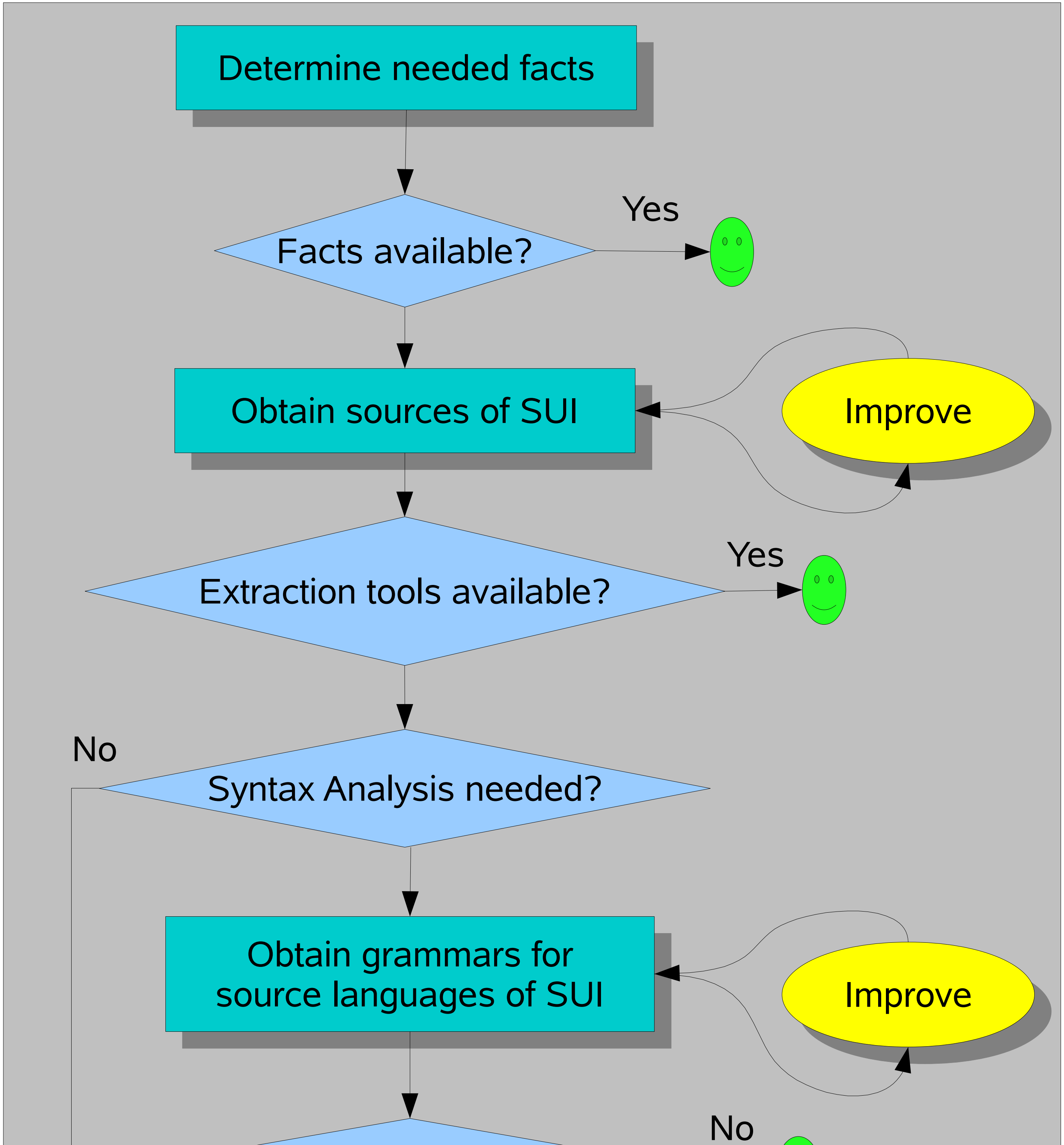

Solve (sub)problem
Validate results
Improve
No
Acceptable?

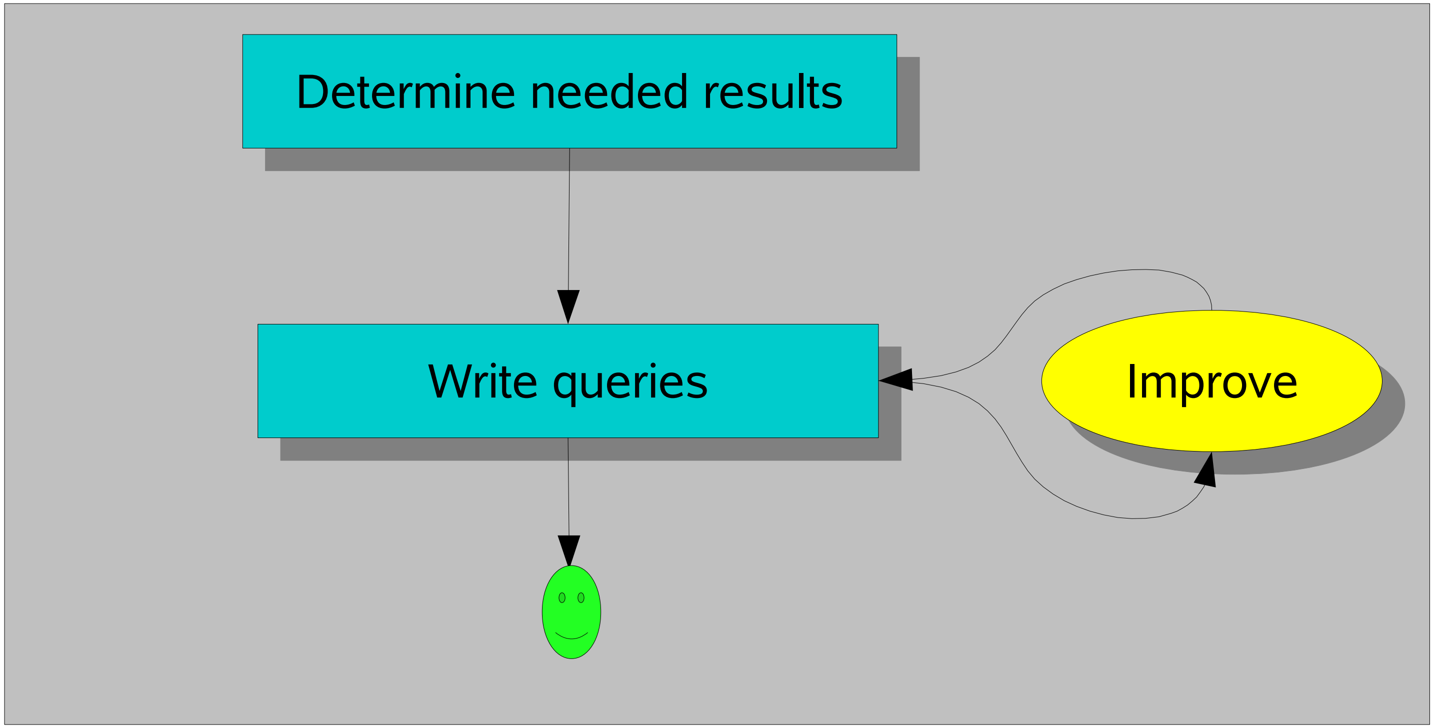


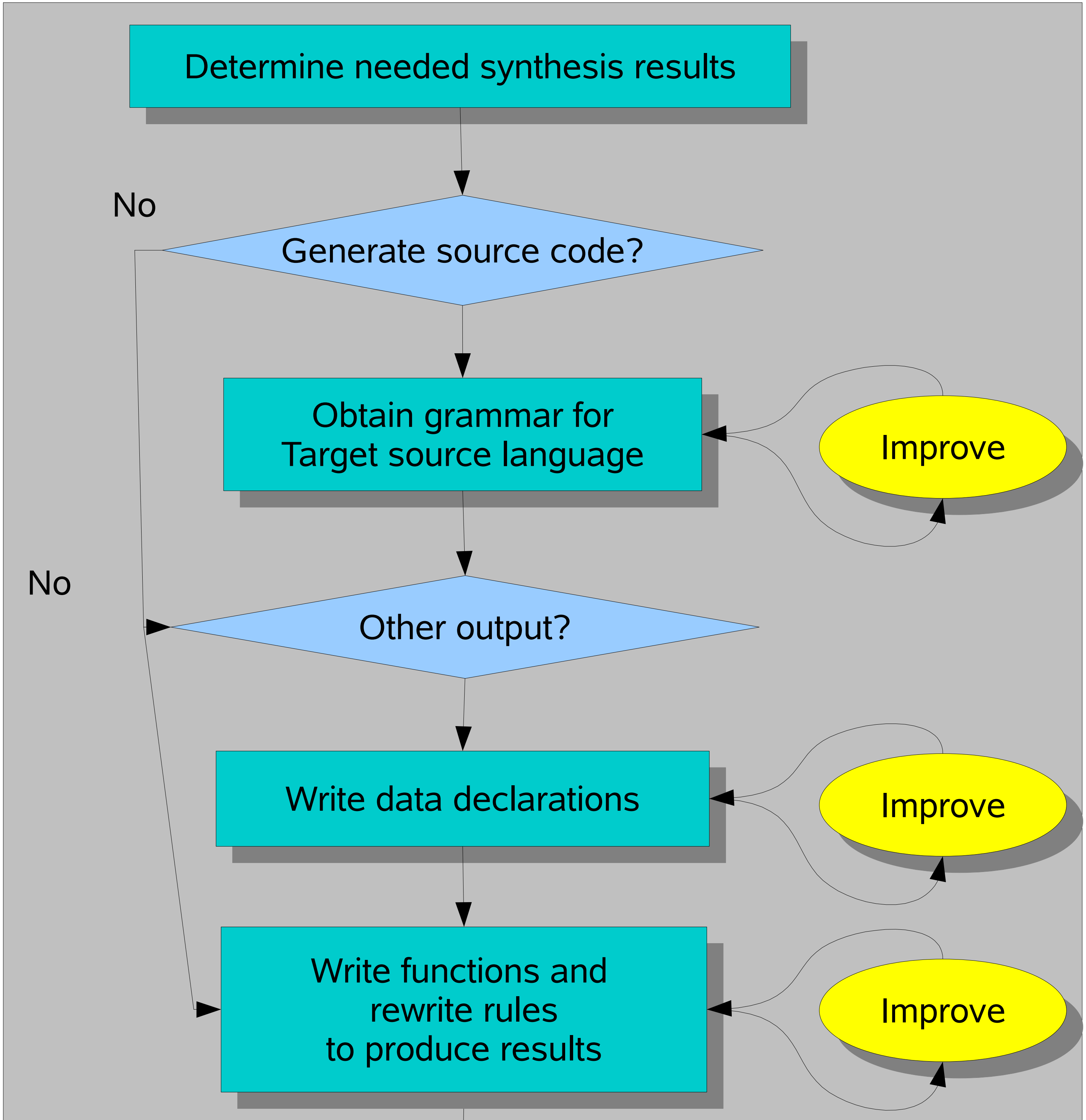
Determine Synthesis Requirements

Determine Analysis Requirements

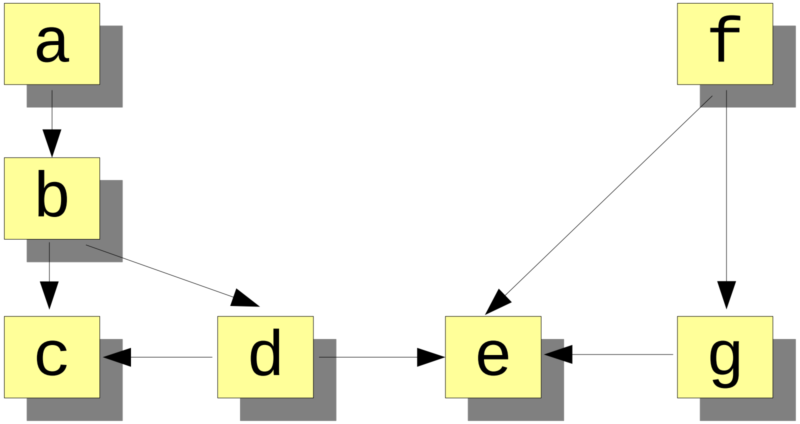
Determine Extraction Requirements


Determine needed facts
Facts available?
Yes
Obtain sources of SUI
Improve
Extraction tools available?
Yes
No
Syntax Analysis needed?
Obtain grammars for source languages of SUI
Improve
No


Determine needed results
Write queries
Improve

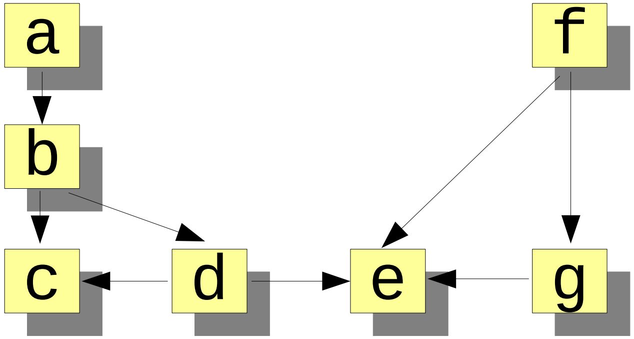

Determine needed synthesis results
No
Generate source code?
Obtain grammar for
Target source language
Improve
No
Other output?
Write data declarations
Improve
Write functions and
rewrite rules
to produce results
Improve

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

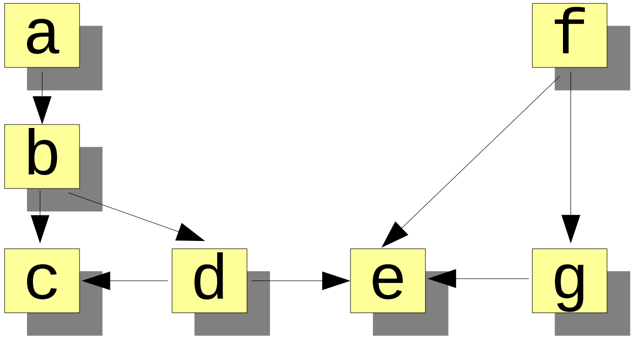


- How many calls are there?
  - int ncalls = size(calls);
  - 8
- How many procedures are there?
  - int nprocs = size(carrier(calls));
  - 7

Number of elements

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

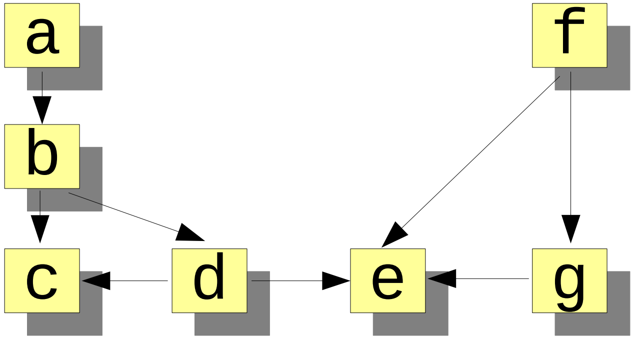
- The roots of a relation viewed as a graph
- top({<1,2>,<1,3>,<2,4>,<3,4>}) yields {1}
- Consists of all elements that occur on the lhs but not on the rhs of a tuple

-
  ```
  set[&T] top(rel[&T, &T] R) =
          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(rel[&T, &T] R) =

  range(R) \ domain(R)

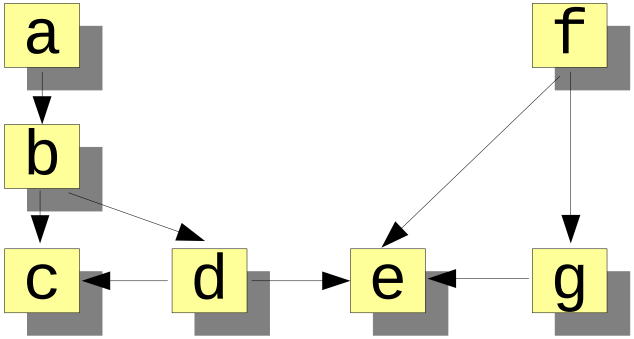


Some questions

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

The image of domain value "a"

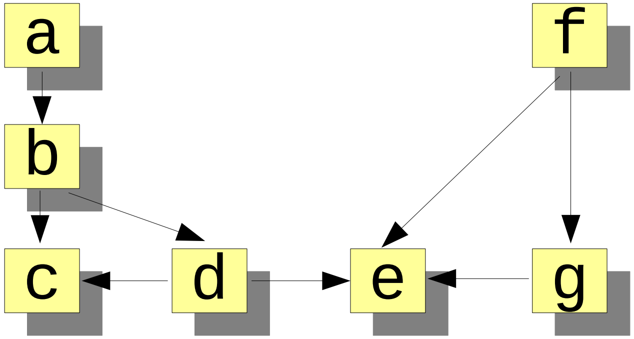
Some questions



- What are the calls to procedure *e*?
  - set[str] callsToE = closureCalls[-,"e"]
  - {"a", "b", "d", "f", "g"}
  -
  -
  - NOTE PAS AAN!!!

The domain of image value "e"

Some questions



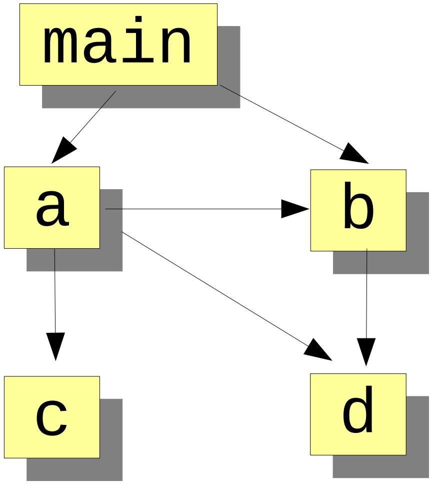
- What are the calls from entry point f?
  - set[str] calledFromF = closureCalls["f"];
  - {"e", "g"}
- What are the common procedures?
  - set[str] commonProcs = calledFromA & calledFromF
  - {"e"}

Intersection

Component Structure of Application

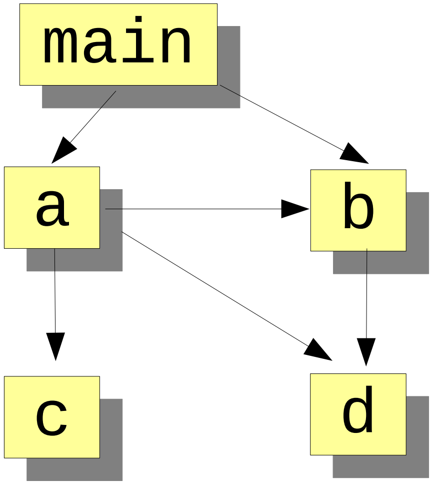
- Suppose, we know:
  - the call relation between procedures (Calls)
  - the component of each procedure (PartOf)
- Question:
  - Can we lift the relation between procedures to a relation between components (ComponentCalls)?
- This is usefull for checking that real code conforms to architectural constraints

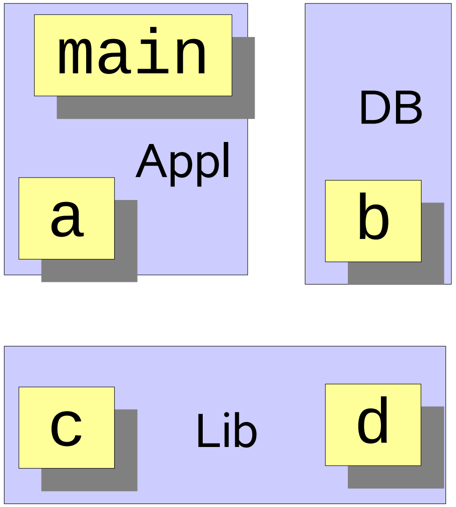
Calls



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

PartOf

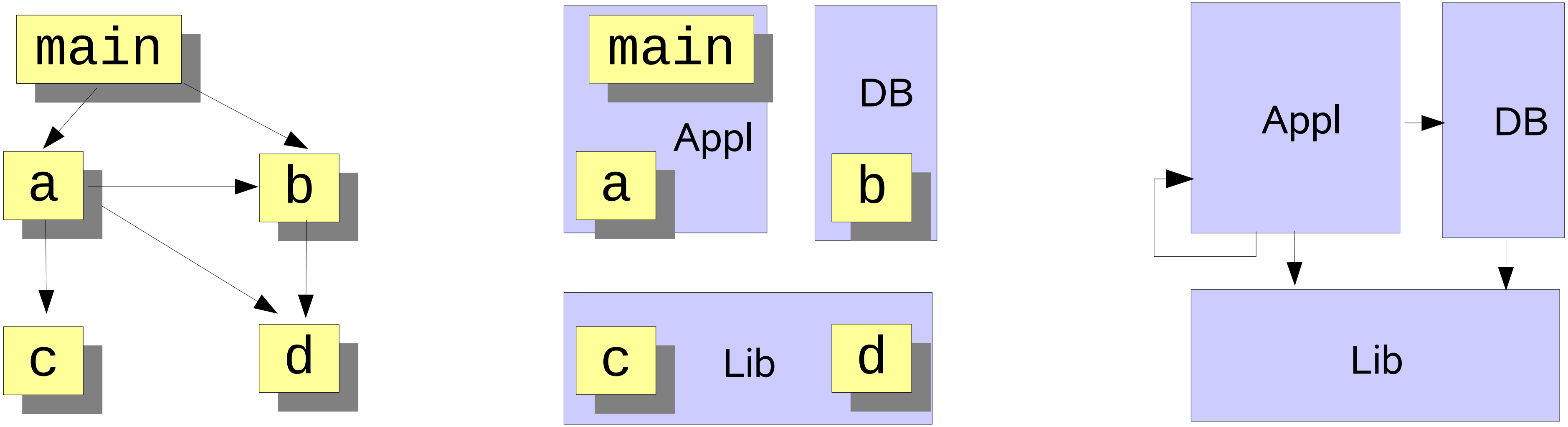




```
set[comp] Components = {"Appl", "DB", "Lib"};

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

lift



```
rel[comp,comp] lift(rel[proc,proc] aCalls,  rel[proc,comp] aPartOf) =
  { <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">}

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
- UnitTest
- Value
- ValueIO
- View (Eclipse only)

Rascal Status

- An interpreter for the core language (currently except parsing and concrete pattern matching) is well underway.
- All the above examples (and many more!) run.
- Full language expected to be implemented mid 2009.

Rascal Implementation

Perspective

More Information

- http://www.meta-environment.org
- Latest version of documentation:
- Download: