Grammarware is Everywhere
and What to Do about That?

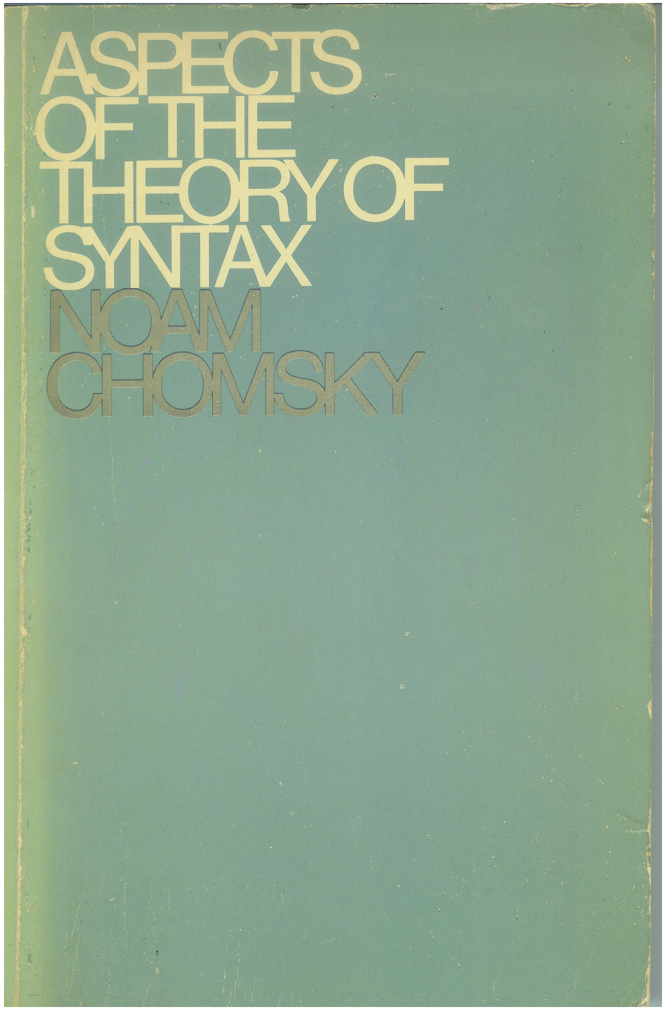
Paul Klint

CWI

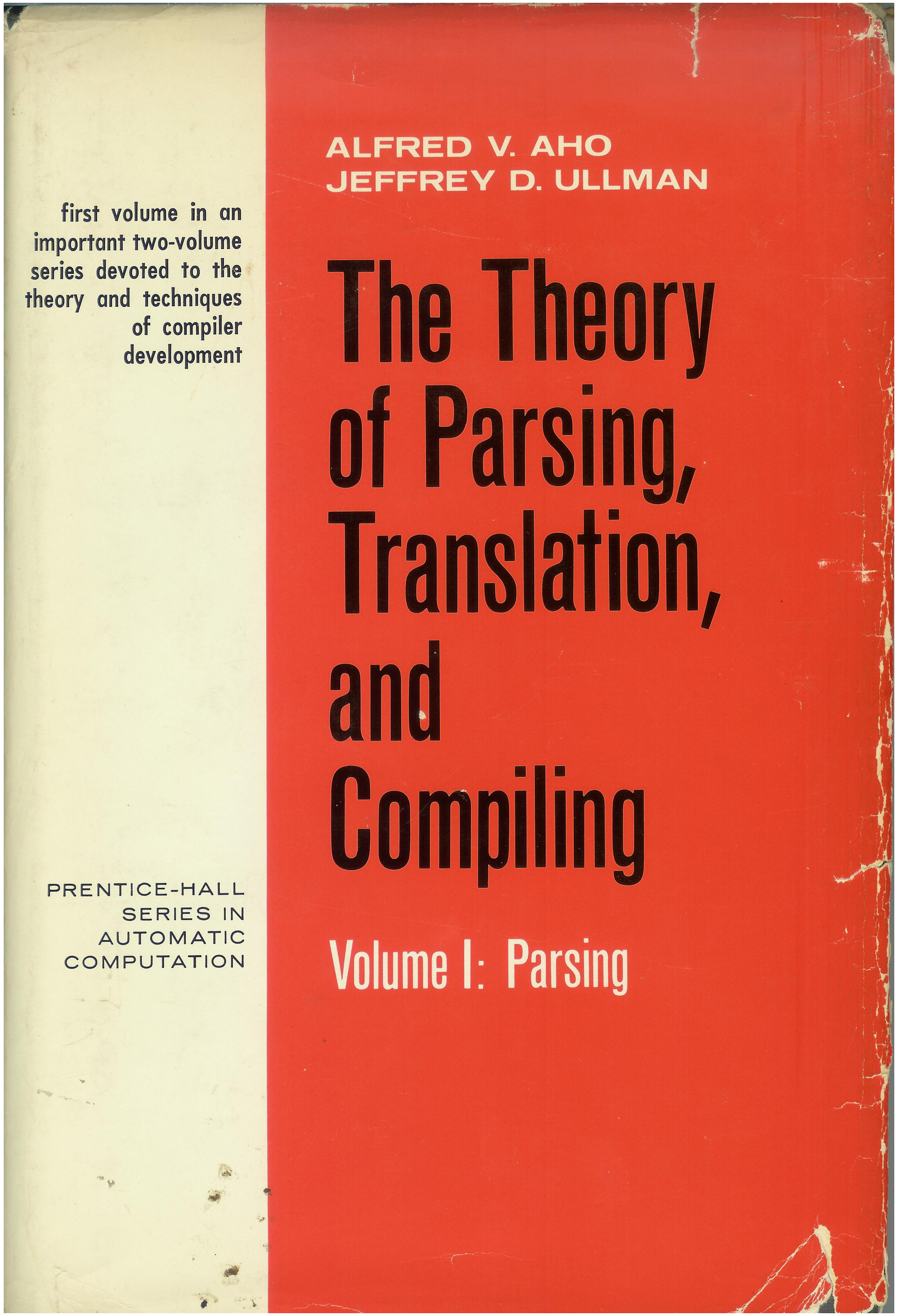
Universiteit van Amsterdam

Institut National de Recherche en Informatique et en Automatique
INRIA
centre de recherche LILLE - NORD EUROPE

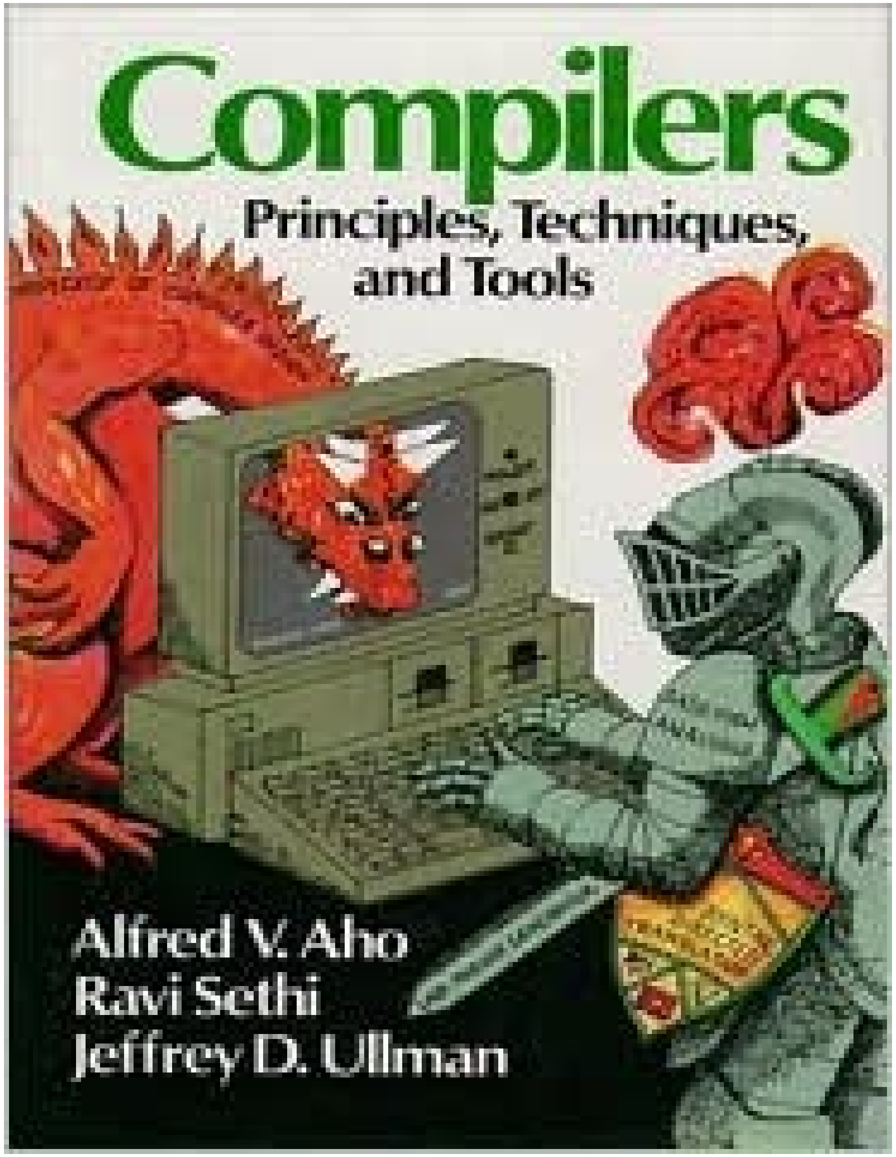
Grammars and Languages are one of the most established areas of Computer Science



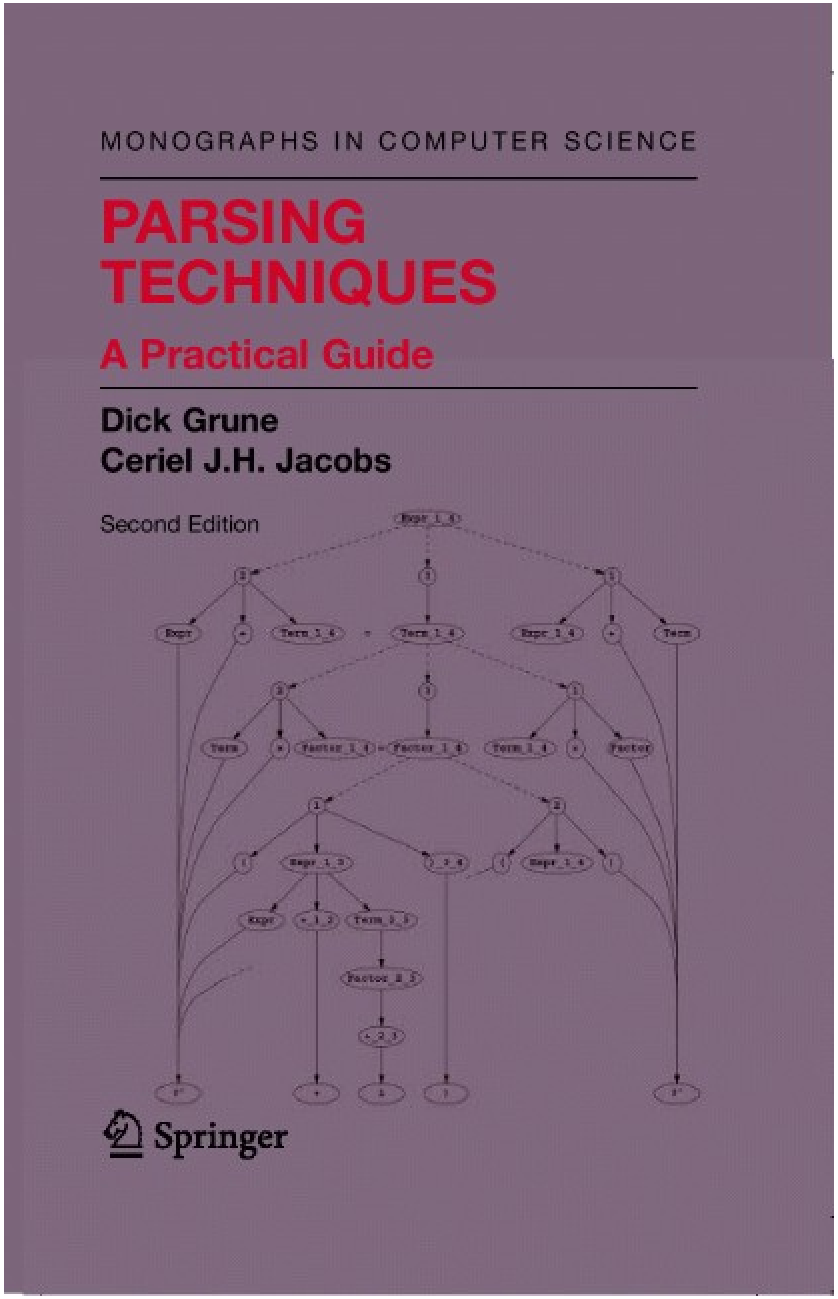
N. Chomsky,
Aspects of the theory of syntax,
1965



A.V. Aho & J.D. Ullman,
The Theory of Parsing,
Translation and Compilimg,
Parts I + II,
1972



A.V. Aho, R. Sethi, J.D. Ullman, Compiler, Principles, Techniques and Tools, 1986



D. Grune, C. Jacobs,
Parsing Techniques,
A Practical Guide,
2008

Is Research on Grammars and Parsing Dead?

Why?

Why Not?

Why?

- The most obvious and visible results have been achieved:
  - Grammar classes
  - Parsing algorithms
  - Complexity
  - Decidability
- The use of parsing in compilers is standard
- Parsing is just one (relatively simple) aspect of compiling

Why Not?

The Use Cases are changing:

- From batch use to interactive use
- From single language to multiple language
- From compiling to understanding and renovation
- From standard language to domain-specific language
- From textual grammar to graphical model
- ...

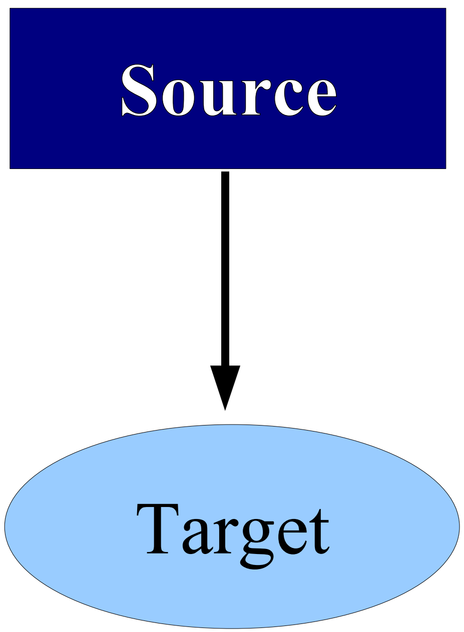
Grammar Use Case #1: Compilation versus Understanding

Compilation is ...

- A well-defined process with well-defined input, output and constraints
- Input: source program in a fixed language with well-defined syntax and semantics
- Output: a fixed target language with well-defined syntax and semantics
- Constraints are known (correctness, performance)
- A batch-like process

Compilation is ...

Single,
well defined,
source



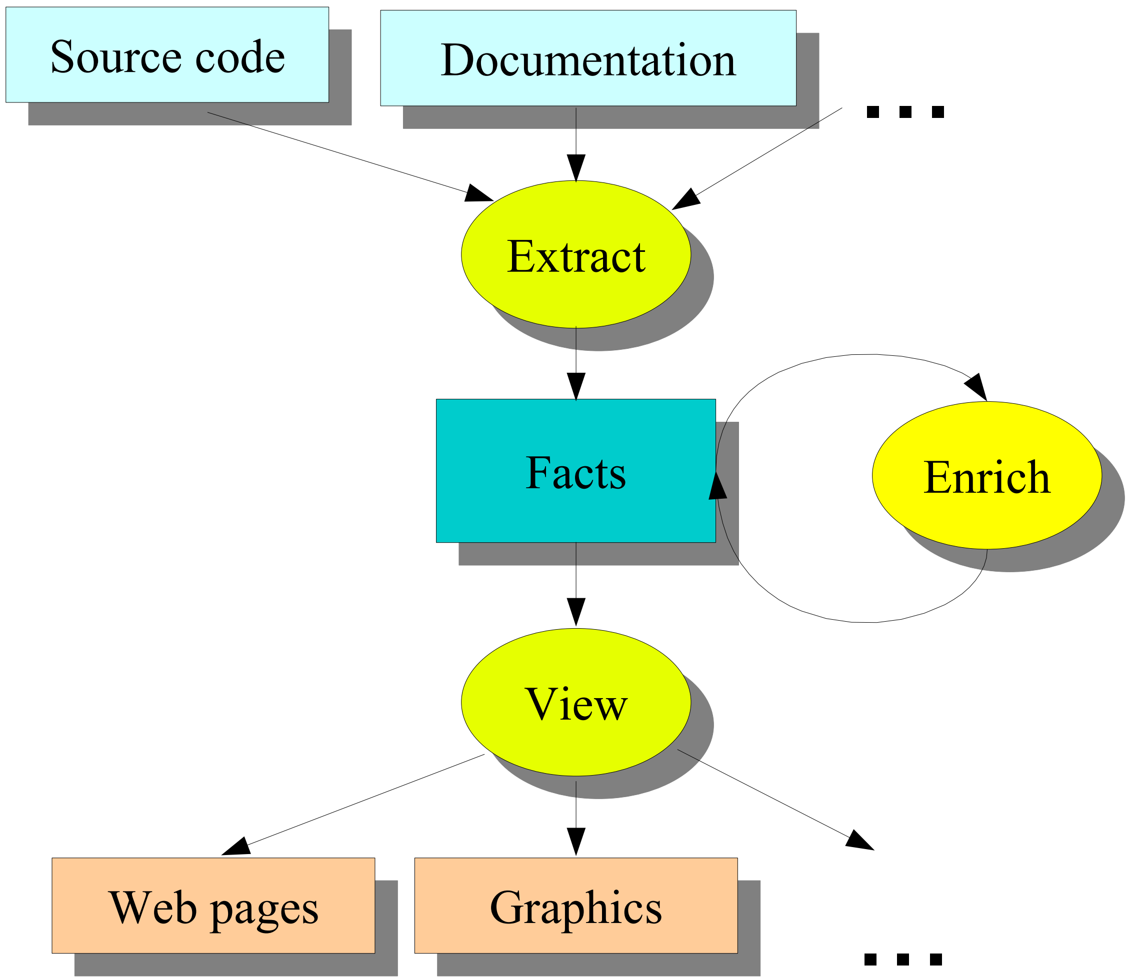
A batch-like process with
clear constraints

Single,
well
defined,
target

Understanding is ...

- An exploration process with as input
  - system artifacts (source, documentation, tests, ...)
  - implicit knowledge of its designers or maintainers
- There is no clear target language
- An interactive process:
  - Extract elementary facts
  - Abstract to get derived facts needed for analysis
  - View derived facts through visualization/browsing

Extract-Enrich-View Paradigm



Examples of understanding problems

- Which programs call each others?
- Which programs use which databases?
- If we change this database record, which programs are affected?
- Which programs are more complex than others?
- How much code clones exist in the code?

Grammar Use Case #1: Summary

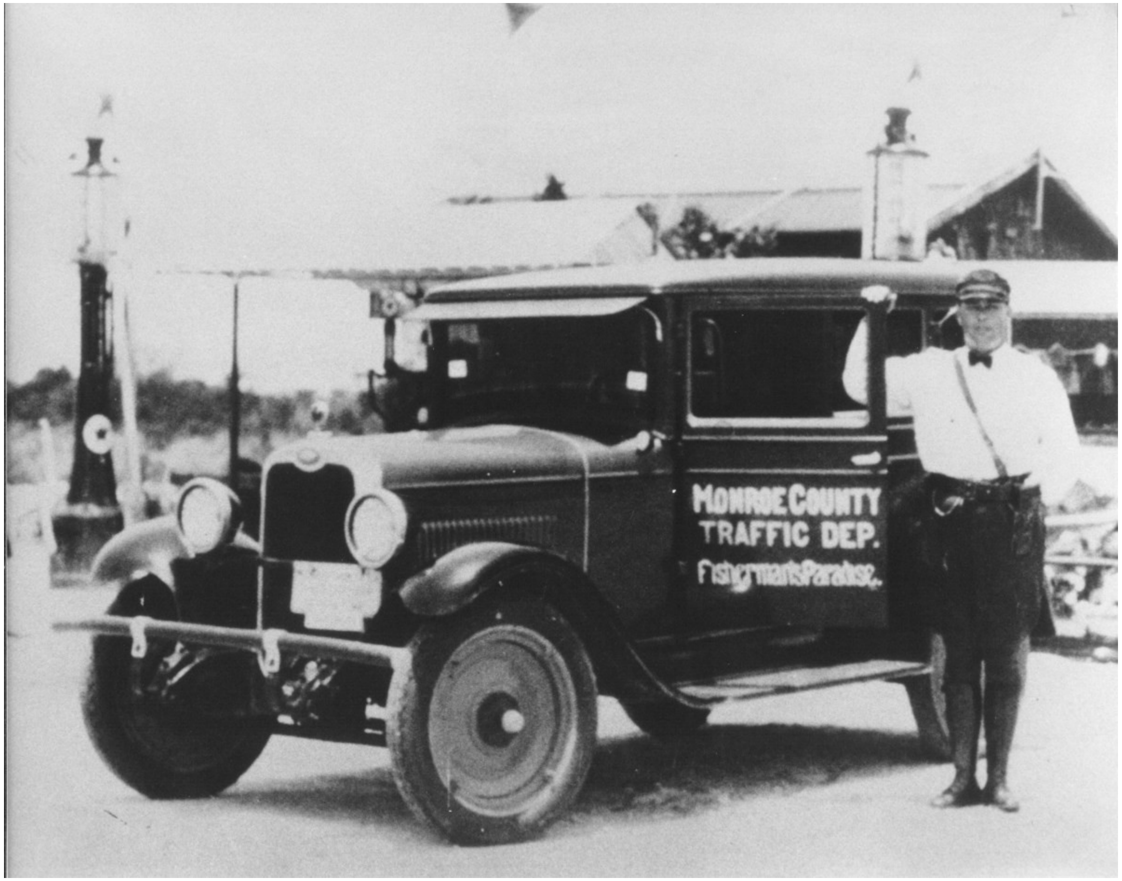
There is a mismatch between

- standard compilation techniques and
- the needs for understanding and restructuring

Grammar Use Case #2: Software Renovation

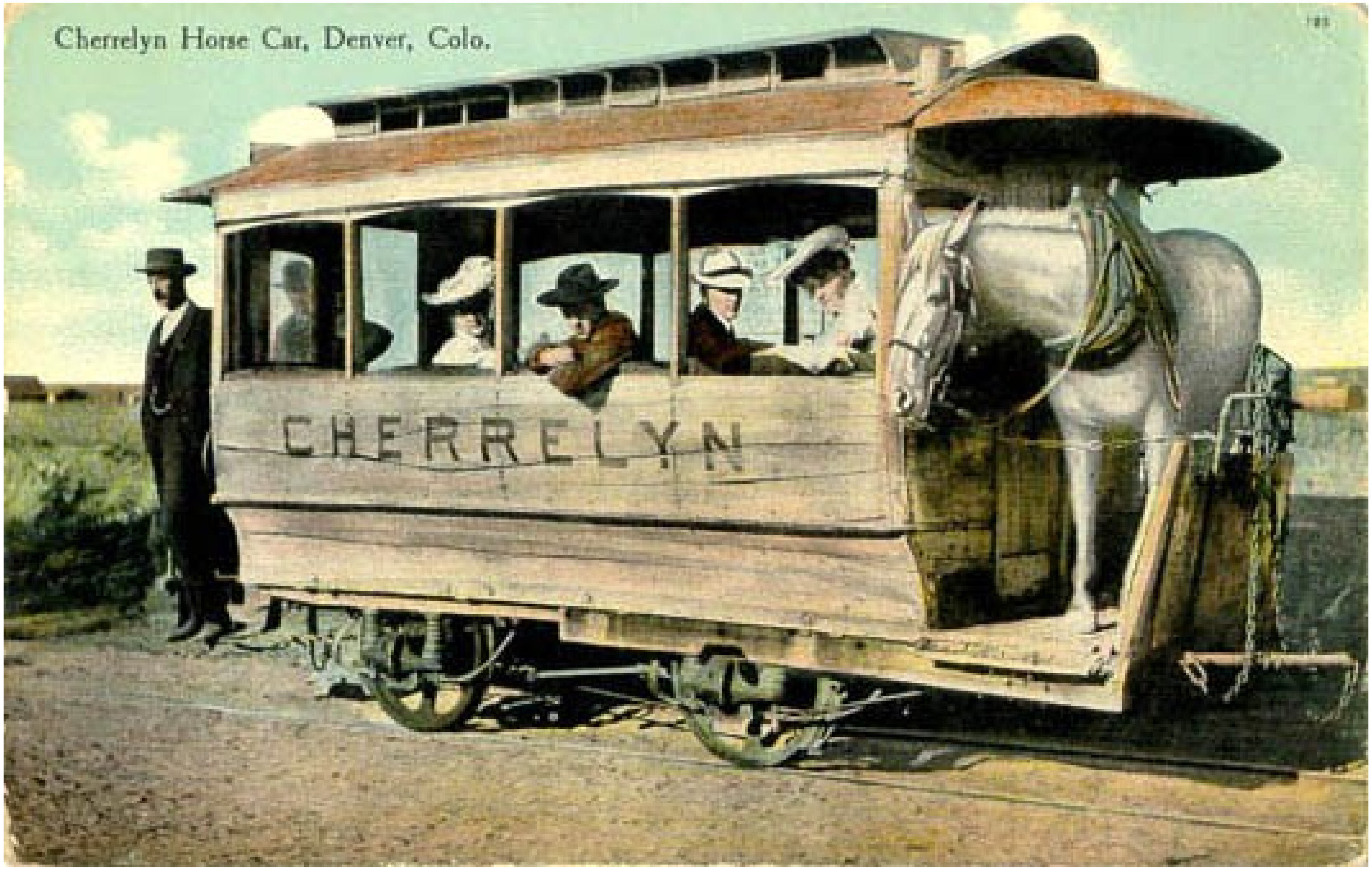
Some cars need maintenance ...

… to replace proven technology ...



1931 Patrol Car

… or proven-technology-inside …



Cherrelyn Horse Car, Denver Colorado

Legacy Software

- The total volume of software is estimated at
  $7 * 10^9$ *function points*
- 1 FP = 128 lines of C or 107 lines of COBOL
- The volume of this volcano is
  - 750 Giga-lines of COBOL code, or
  - 900 Giga-lines of C code

Printed on paper we can wrap planet Earth
9 times!

We need techniques for ...

- Global program understanding
  - Who calls who? What is de module structure?
  - Conformance with software architecture?
- Detailed program analysis
  - Where are buffer overflows?
  - Does lock/unlock occur on every path?
- Program transformation
  - Dialect conversion
  - API upgrades

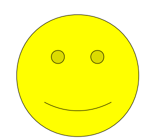
In more detail, we need ...

- Syntax analysis of various language mixtures
- Extracting facts from source code
- Computing with these facts
- Producing reports, visualisations, …
- Transform source code according to given rules

Grammar Use Case #3: Domain-specific Languages


Arctic
Ocean
Greenland
Canada
Atlantic
Ocean

The Inuit

- The Inuit live in Canada and Greenland
- Their language is called Inuktitut: ᐃᓄᒃᑎᑐᑦ
- Inuktitut has many words for the concept *snow*:
  - Fresh snow, Old snow, Frozen snow, Powder snow, etc.
- Inuktitut is (from my nerd's  perspective) a *domain specific language* for describing snow

Domain-specific Languages

- Efficient way to capture domain concepts
- Reduce development time by code generation
- Increase flexibility and infrastructure independence

We need techniques for ...

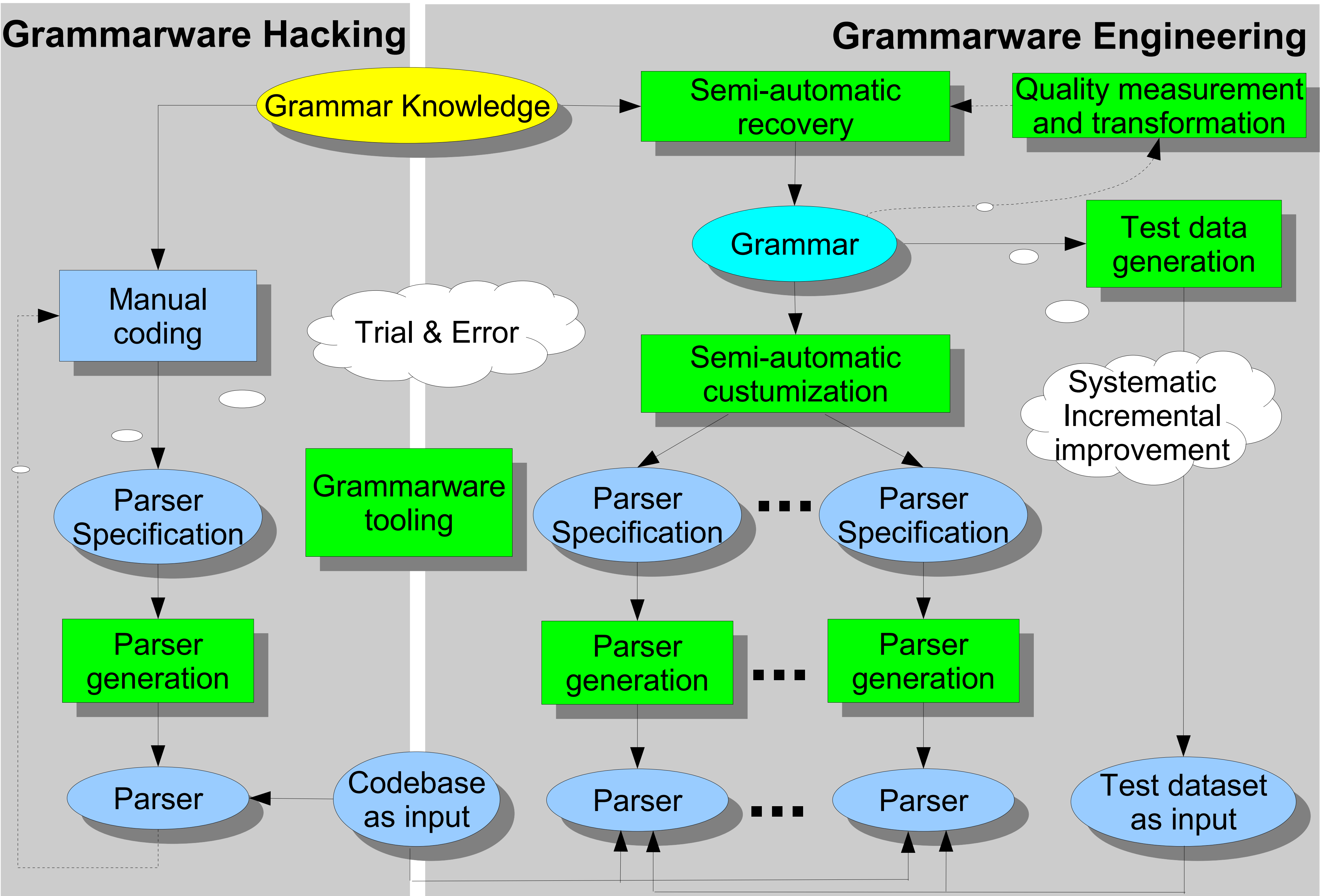
- Analyzing domain concepts
- Defining the syntax of a DSL
- Defining consistency rules on DSL programs
- Definining code generation from DSL program to software infrastructure

Observations based on Use Cases

- Grammars include definitions for
  - Context-free syntax
  - Class dictionaries
  - XML schema's
  - Tree/graph grammars
- Grammars are used for
  - concrete/abstract syntax of programming languages
  - Exchange formats
  - API's

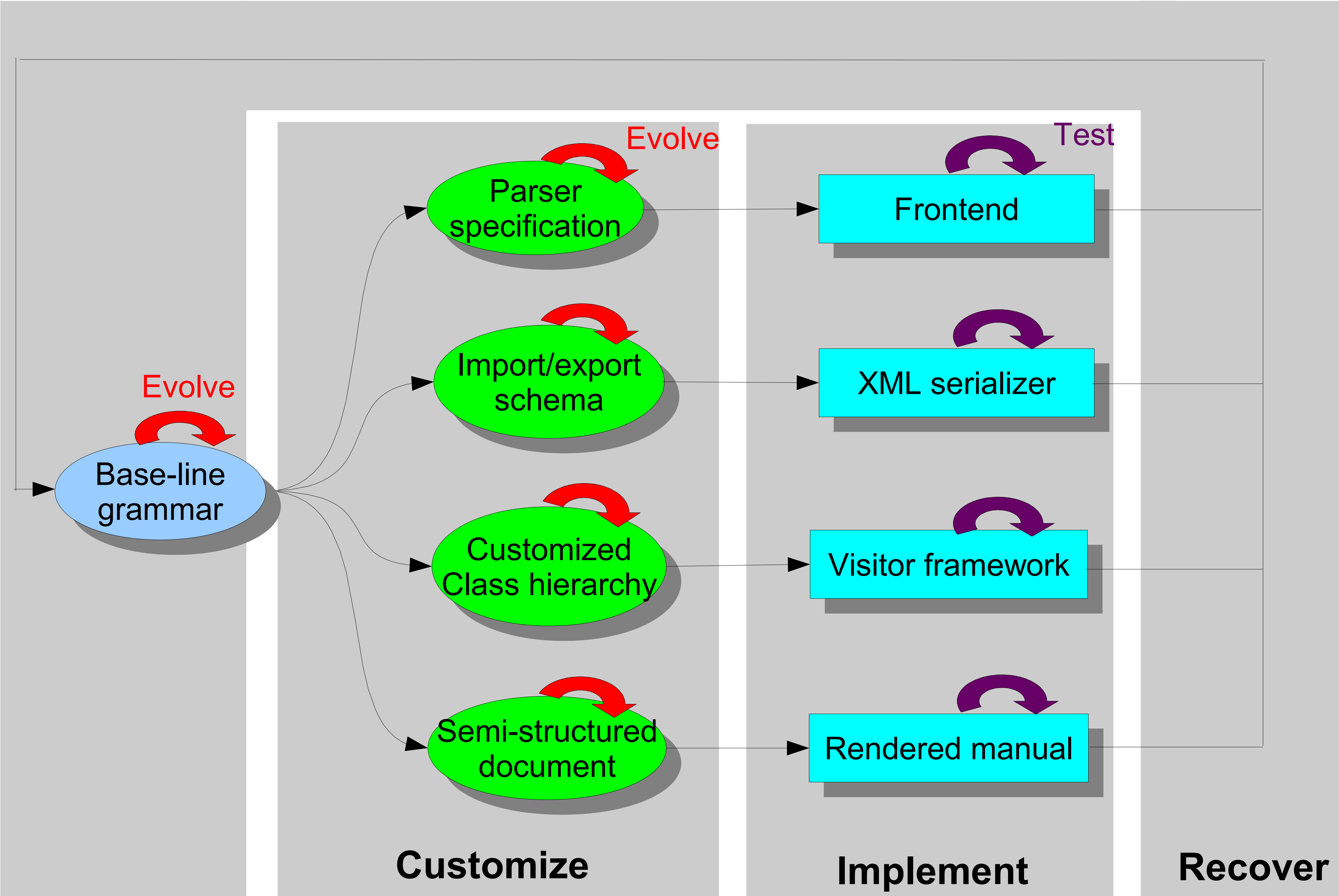
Observations based on Use Cases

- All grammar-dependent software ("grammarware") evolves over time
- For source code development we use software engineering best-practices
- There is no such thing as grammarware engineering
  - But see: P. Klint, R. Laemmel, C. Verhoef, Towards an Engineering Discipline for Grammarware, ACM TOSEM, 14(3) 331—380, 2005.


Grammarware Hacking
Grammarware Engineering
Grammar Knowledge
Semi-automatic recovery
Quality measurement and transformation
Grammar
Test data generation
Manual coding
Trial & Error
Semi-automatic custumization
Systematic Incremental improvement
Parser Specification
Grammarware tooling
Parser Specification
...
Parser Specification
Parser generation
Parser generation
...
Parser generation
Parser
Codebase as input
Parser
...
Parser
Test dataset as input

Grammarware Research Questions

- How to provide modular grammars?
- What is a “good” grammar?
- How to transform grammars (and maintain the link with dependent software)
- How to uncover grammars from grammar-dependent source code?
- How to test grammar-dependent functionality?
- How does the grammarware “lifecycle” look like?


Evolve
Parser specification
Import/export schema
Customized Class hierarchy
Semi-structured document
Test
Frontend
XML serializer
Visitor framework
Rendered manual
Evolve
Base-line grammar
Customize
Implement
Recover

How can we ...

- Describe syntax & semantics of languages?
- Build tools for grammarware engineering?
- Make these tools programming language independent?
  - Generic language technology, also called
  - Language-parametric technology

Generic Language Technology (GLT)

- Goal: Enable the easy creation of language-specific tools and programming environments
- Separate language-specific aspects from generic aspects
- Approach:
  - Find good, reusable, solutions for generic aspects
  - Find ways to define language-specific aspects
  - Find ways to generate tools from language-specific definitions

Generic aspects

- User-interface
  - Text editing, error messages, spell checking
- Program storage
- Version managament
- Documentation & help facilities

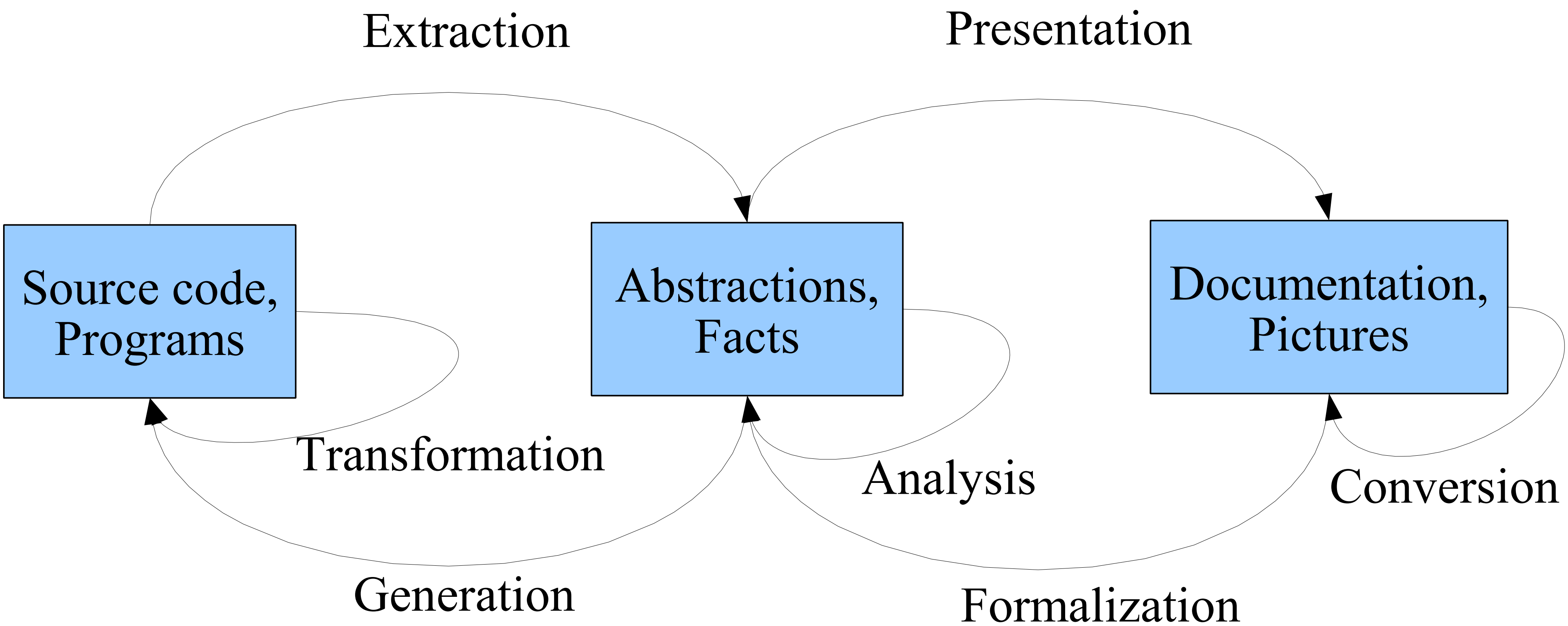
Defining Language Aspects

- **Syntax**: the form of programs
  - Context-free grammar (SDF)
- **Static semantics**: compile-time properties
  - Algebraic specification/rewrite rules (ASF)
  - Relational calculus (RScript)
- **Dynamic semantics**: run-time properties
  - Algebraic specification/rewrite rules

Generate Tools from Definitions

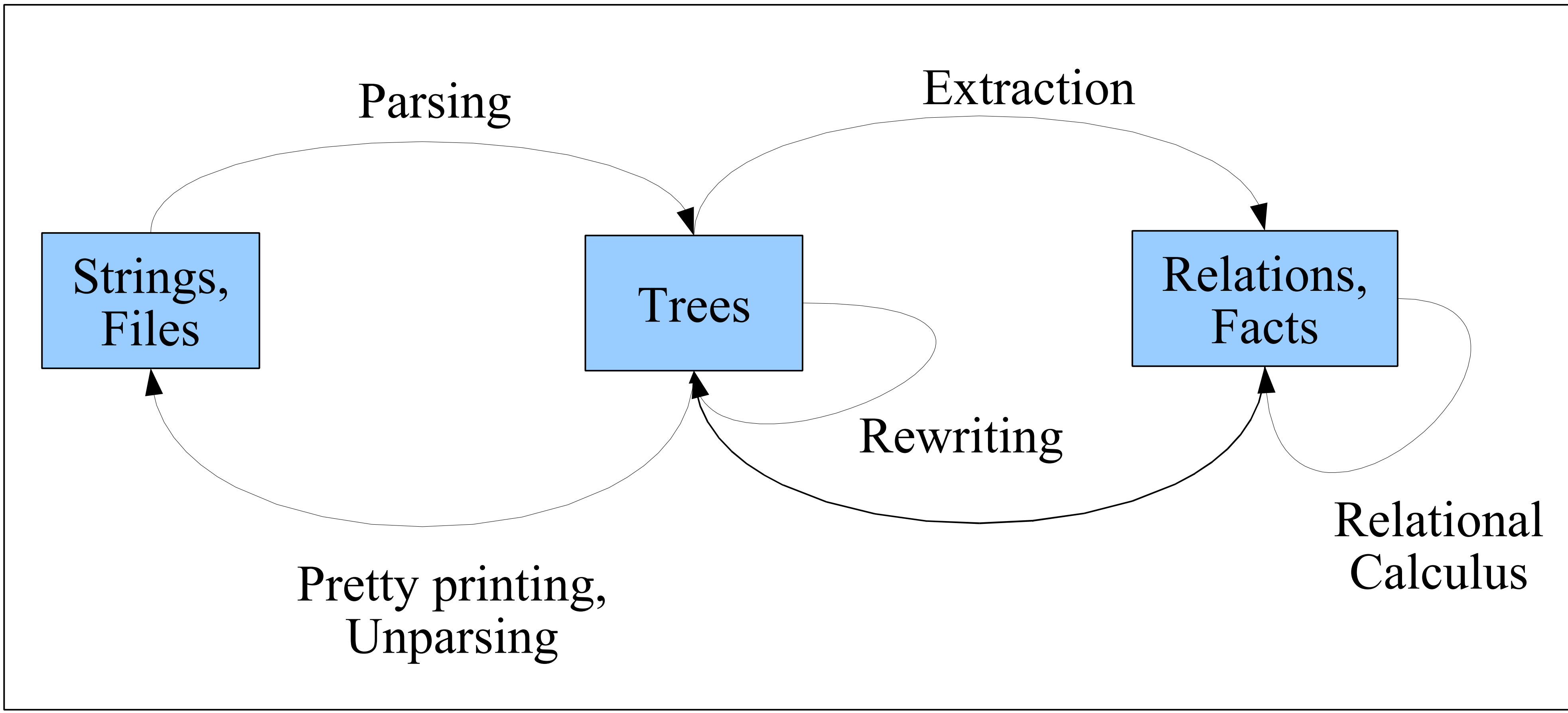
- Context-free syntax
  - Parser generator
- Abstract syntax
  - API generator
- Static semantics
  - Term rewriting compiler
  - Relational Calculator
- Dynamics semantics
  - Term rewriting compiler

Role of GLT



Generic Language Technology helps implementing translations between source code representations

Technology used for GLT



How to connect these technologies?

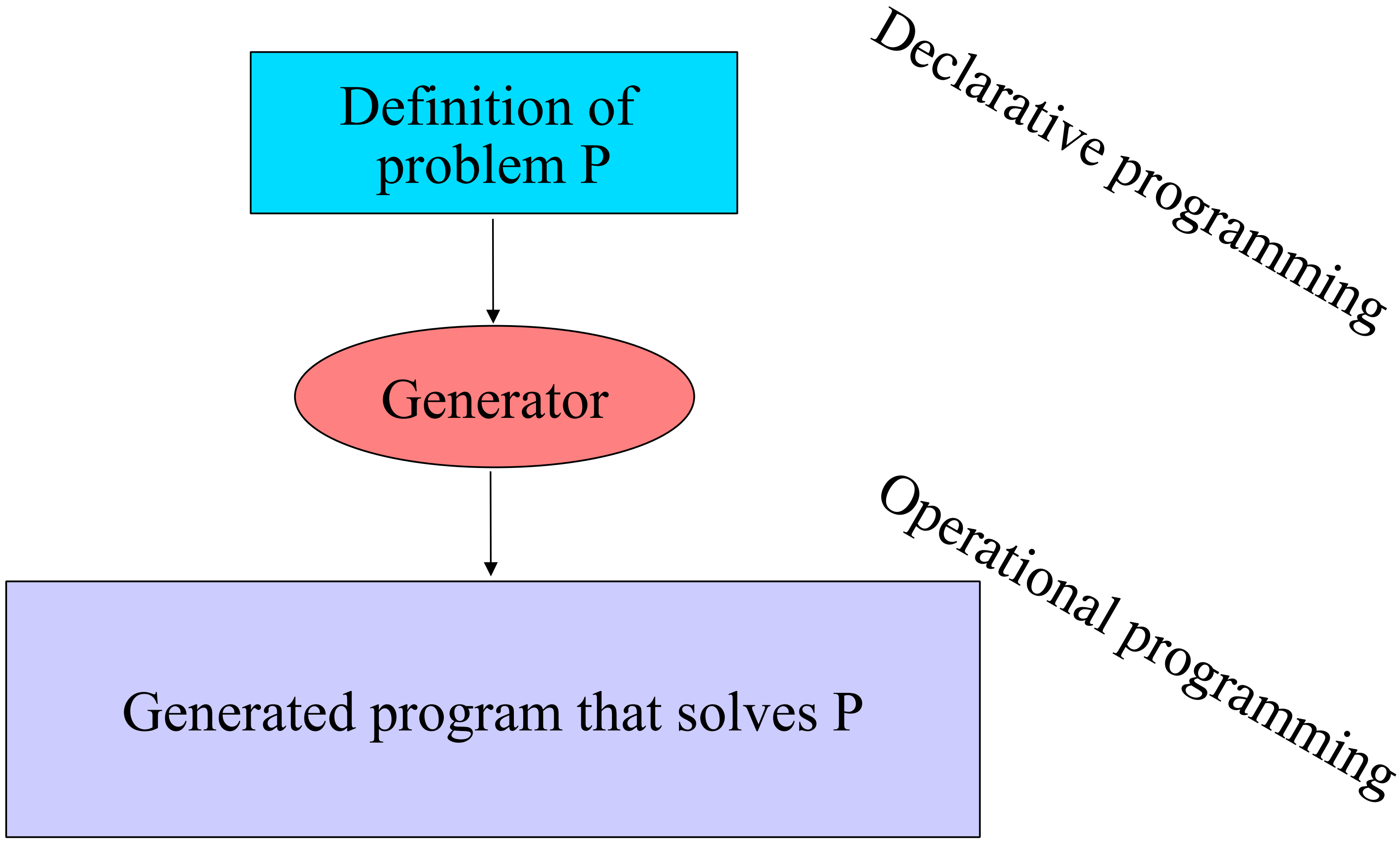
Technology integration: Partial Answers

- Program generators
  - To be discussed next
- Middleware, like the ToolBus
  - not discussed in this presentation
- Integration in a single linguistic framework
  - See discussion on Rascal, at end of talk

Technology integration: Partial Answers

- Program generators
  - To be discussed next
- Middleware, like the ToolBus
  - not discussed in this presentation
- Integration in a single linguistic framework
  - See discussion on Rascal, at end of talk

A Program Generator (PG)?



Examples of Program Generators

- Regular expression matching:
  - Problem: recognize regular expressions $R_1,...,R_n$ in a text
  - Generate: finite automaton
- Web sites
  - Problem: create uniform web site for given content
  - Generate: HTML code with uniform navigation and structure

Examples of Program Generation

- Compiler
  - Input Java program
  - Generates: JVM code
- C preprocessor
  - Input: C program with #include, #define, ... directives
  - Generates: C program with directives replaced.

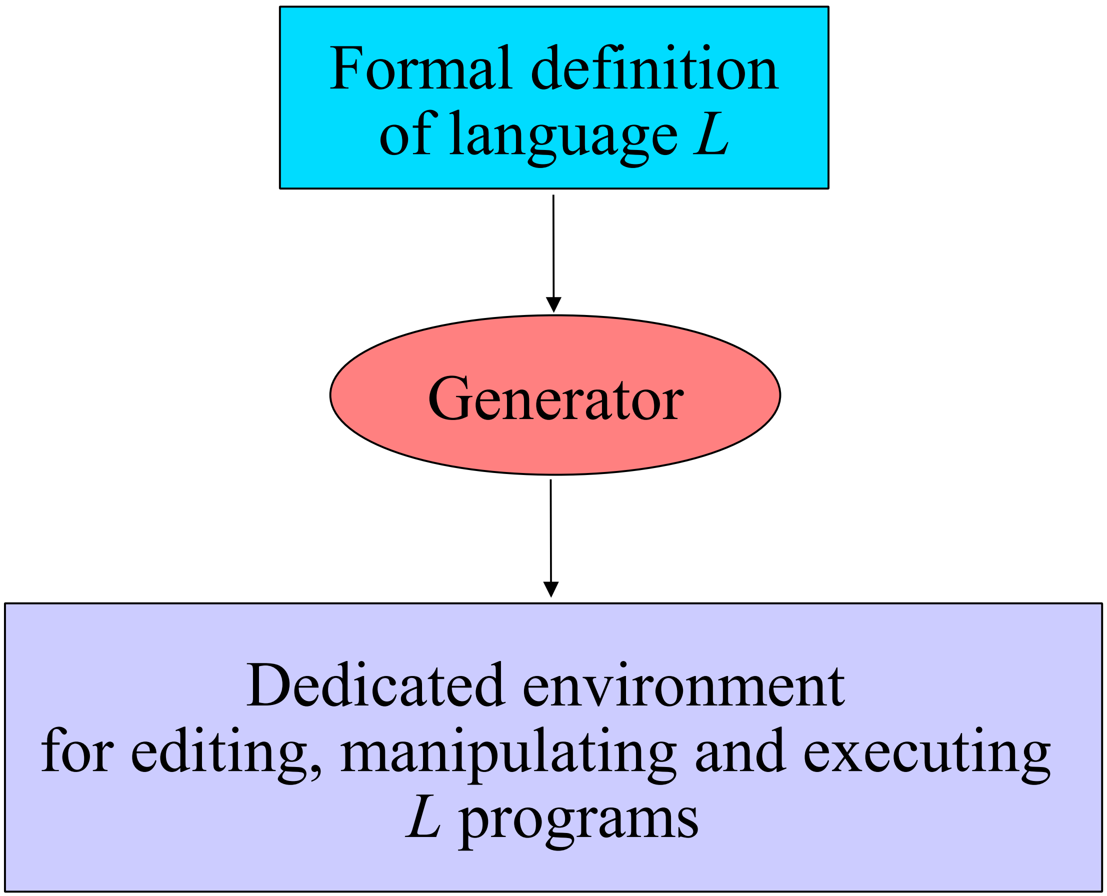
From Program Generator ...

- Problem description is specific and is usually written in a Domain-Specific Language (DSL)
- Generator contains generic algorithms and information about application domain.
- A PG isolates a problem description from its implementation $\Rightarrow$ easier to switch to other implementation methods.
- Improvements/optimizations in the generator are good for all generated programs.

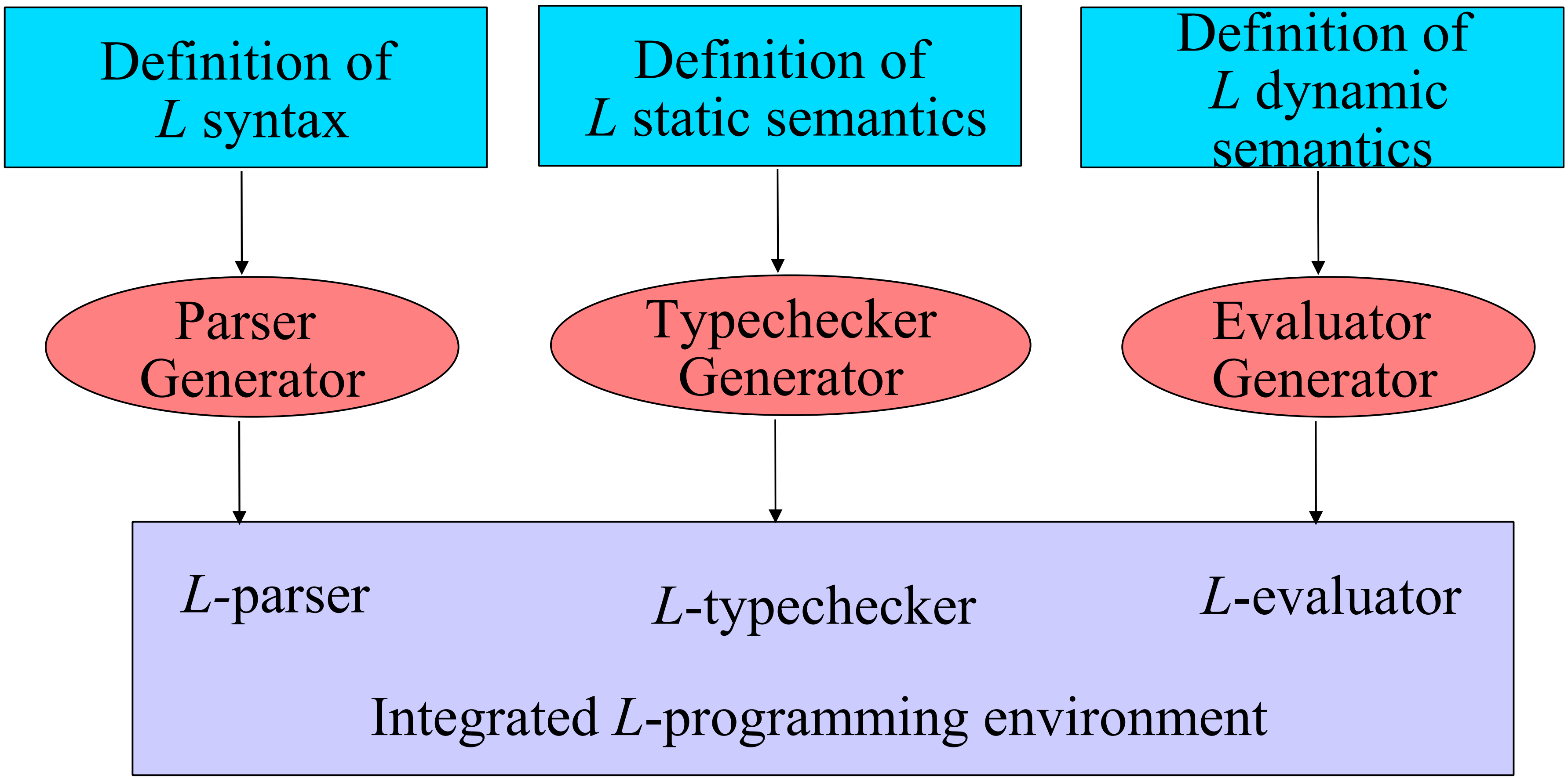
... to Programming Environment Generator (PEG)

- A PEG is a program generator applied in the domain of programming environments
- Input: description of desired language *L*
- Output: (parts of) dedicated *L*-environment
- Advantages:
  - Uniform UI across different languages
  - PEG contains generic, re-usable, implementation knowledge
- Disadvantages: some specializations not easy

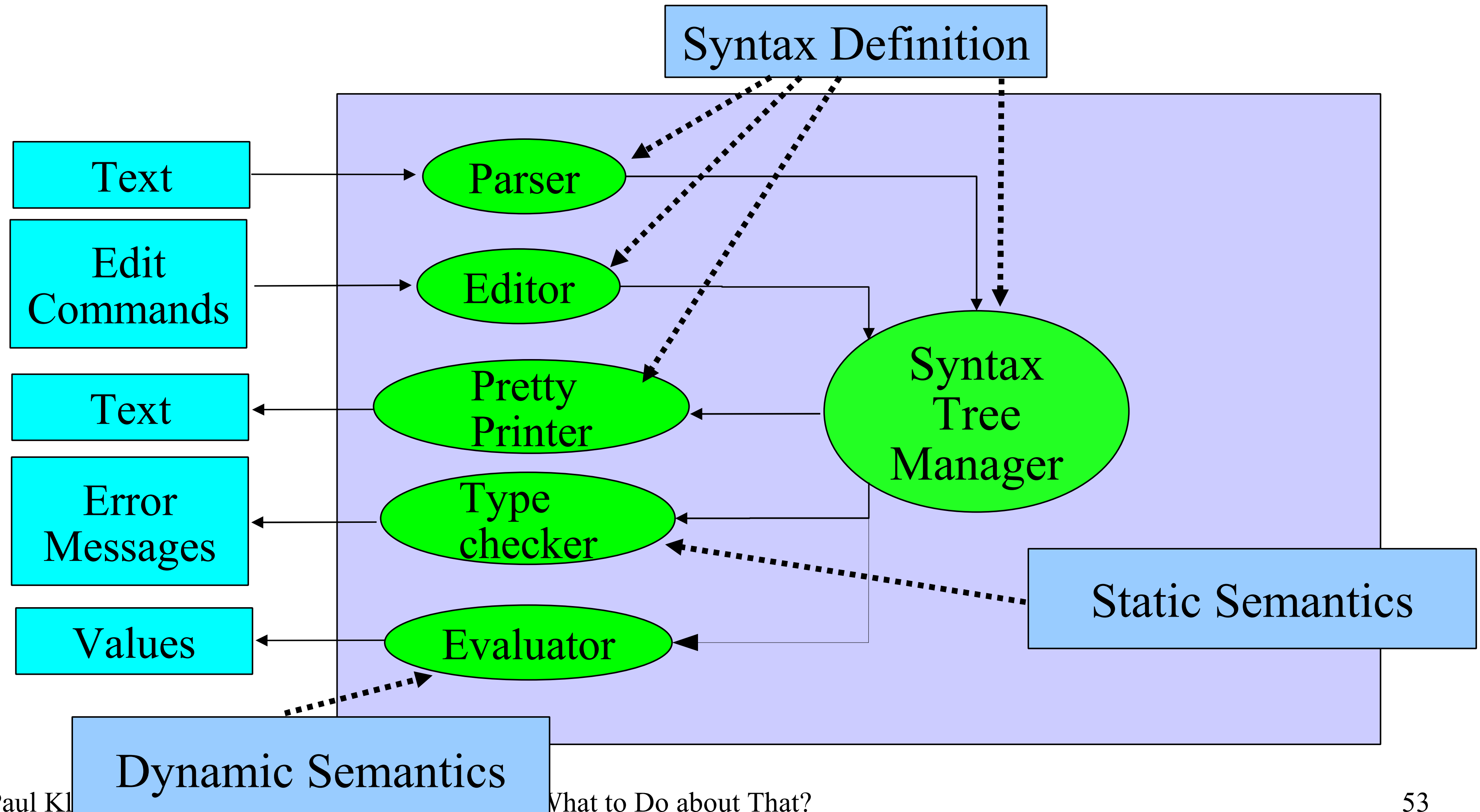
Programming Environment Generator



PEG = collection of program generators



From Definitions to Components



PEG: other definable aspects

- Lexical syntax
- Concrete syntax
- Abstract syntax
- Pretty printing
- Editor behaviour
- Dataflow
- Control flow
- Program Analysis
- Program Queries
- Evaluation rules
- Compilation rules
- User Interface
- Help rules
- ...

A PEG example: ASF+SDF Meta-Environment

- An interactive development environment for generating tools from formal language definitions
- Based on:
  - Full context-free grammars
    - Needed to obtain modular grammar composition
  - Conditional term rewriting
  - Relational calculus

Provides various DSLs

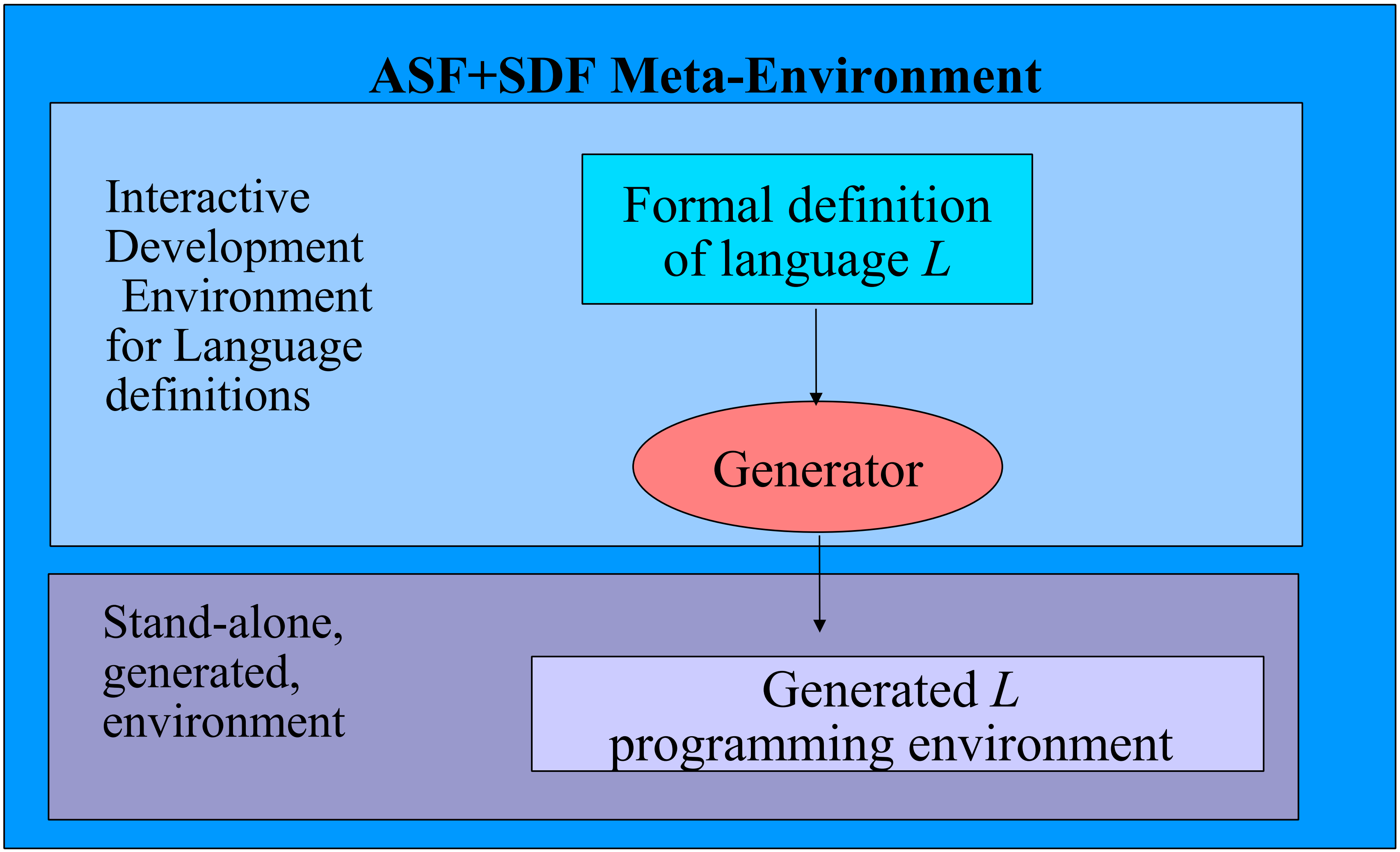
Languages definitions are based on various DSLs:

- Syntax Definition Formalism (SDF)
- Algebraic Specification Formalism (ASF)
- Relational Scripts (RScript)
- Formatting (Pandora)

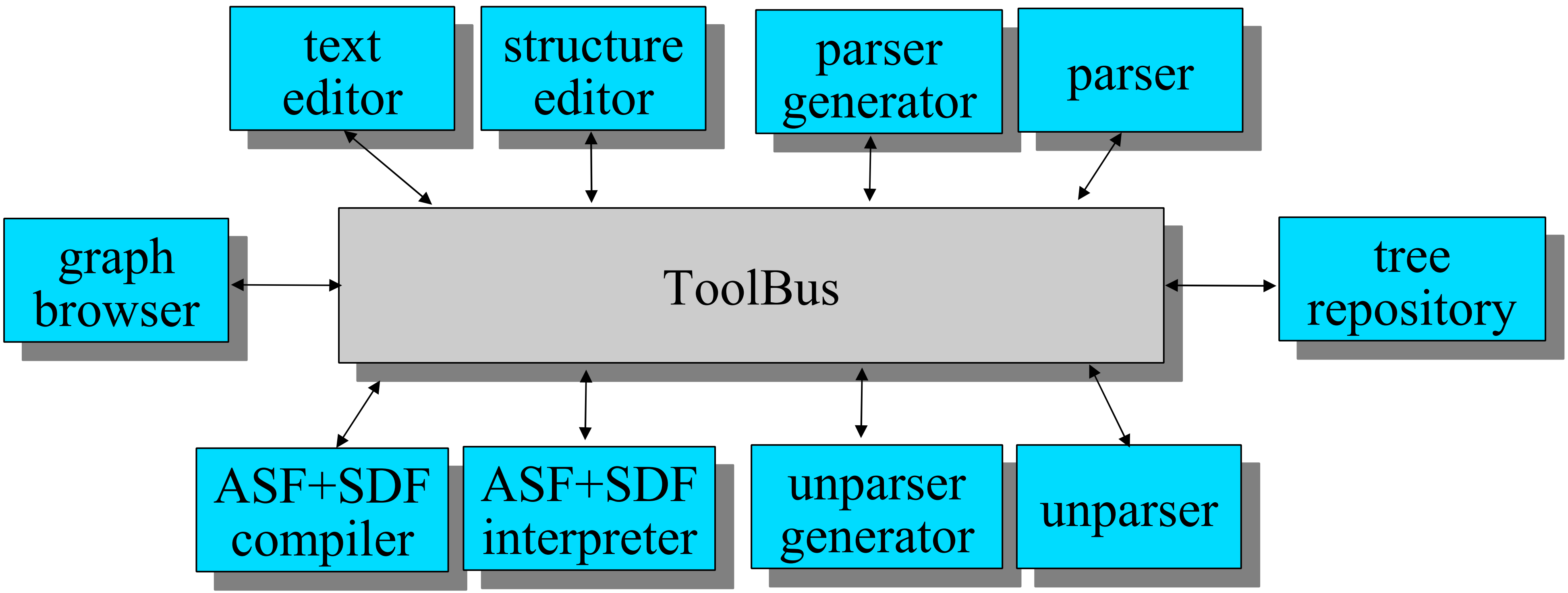
The implementation uses:

- ToolBus Scripts (TScript) for coordination of tools

ASF+SDF Meta-Environment



Architecture of the ASF+SDF MetaEnvironment



ASF+SDF Specifications

- Series of modules that can import each other
- Module can be parameterized; renaming
- Each module consists of two parts:
  - SDF-part defines lexical and context-free syntax, priorities and variables
  - ASF-part defines arbitrary functions, e.g. for typechecking, fact extraction, analysis, evaluation, transformation, ...

ASF+SDF

- One of the most innovative features of ASF+SDF is fully user-definable notation:
  - Instead of a fixed function notation, a function is described by a syntax rule
  - Enables writing rules in concrete syntax and not in abstract syntax (see example below)

ASF+SDF

Surprisingly, these simple techniques scale to large applications. The pattern is always:

- define a syntax (Booleans, numbers, programs in C, Java, Cobol)
- define functions on terms in this syntax (`and`, `plus`, `addEndIf`)
- apply to examples of interest

Example: Cobol transformation

- Cobol 75 has two forms of conditional:
  - "IF" Expr "THEN" Stats "END-IF"?
  - "IF" Expr "THEN" stats "ELSE" Stats "END-IF"?
- *Dangling else* problem:

```
IF expr THEN
    IF expr THEN
      stats
    ELSE
      stats
```

```
IF expr THEN
    IF expr THEN
      stats
ELSE
  stats
```

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

ReCap

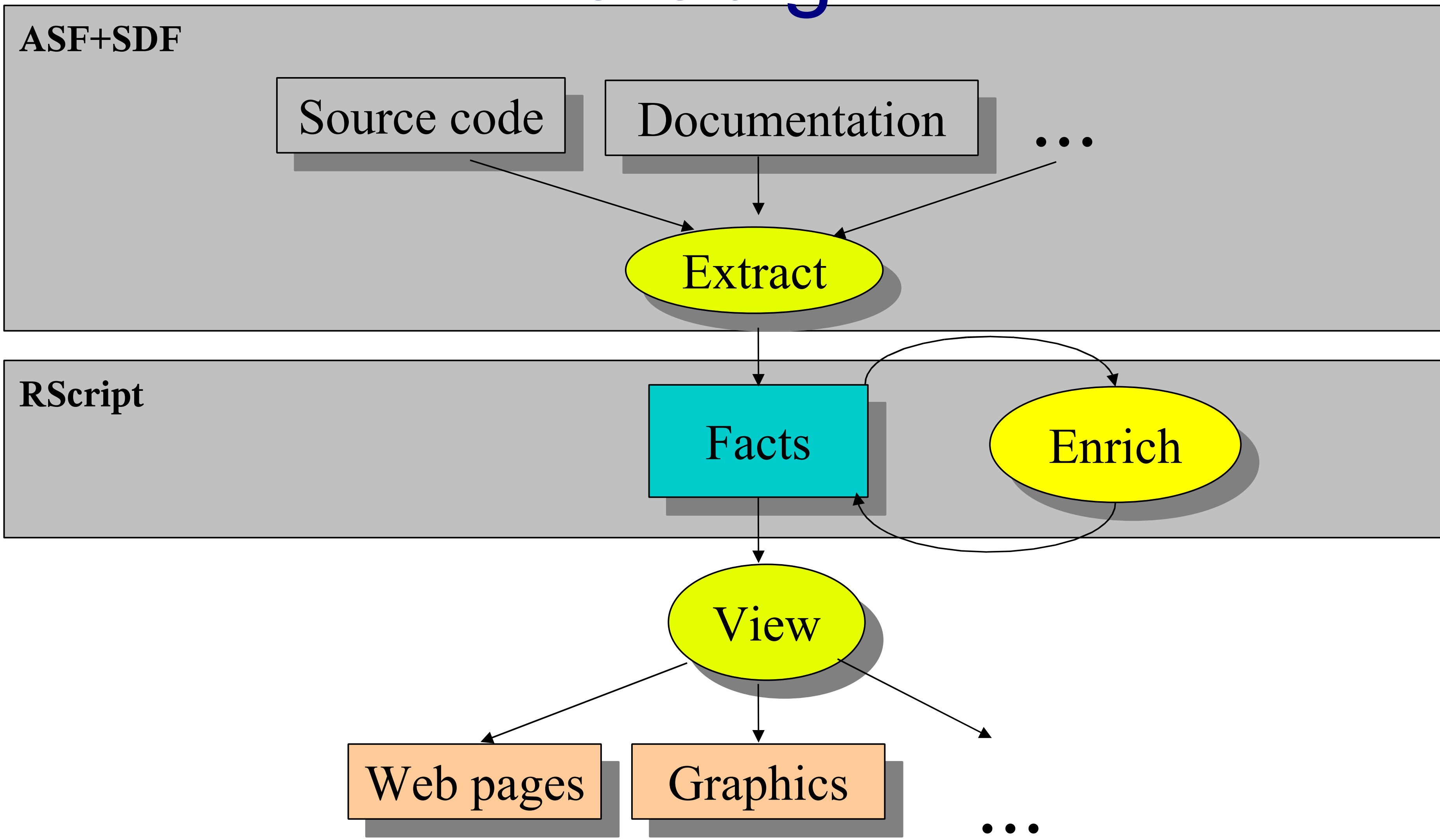
“Everything is a Language”

- ASF+SDF takes care of
  - Syntax definition
  - Transformation
  - Fact extraction

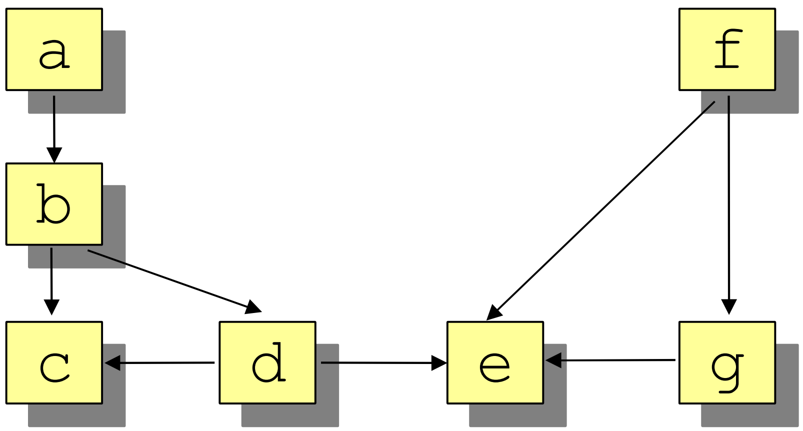
Code Generation

- How about software analysis?
  - Can be done with rewrite rules
  - Relational Calculus adds flexibility & conciseness

Recall: Extract-Enrich-View Paradigm

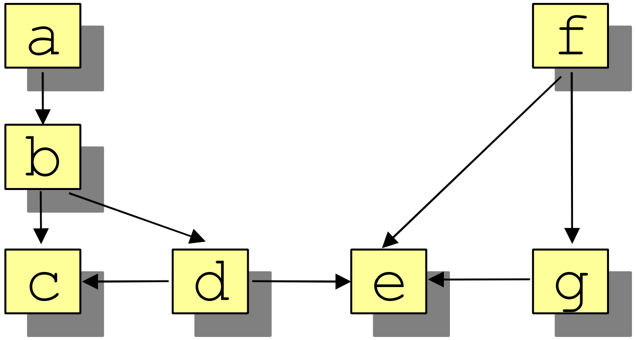


Analyzing the call structure of an application



rel[str, str] calls = {<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">,
<"d","e">, <"f", "e">, <"f", "g">, <"g", "e">}

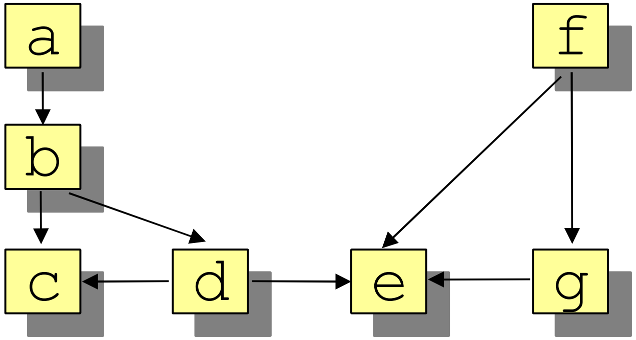
Some questions



- 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)

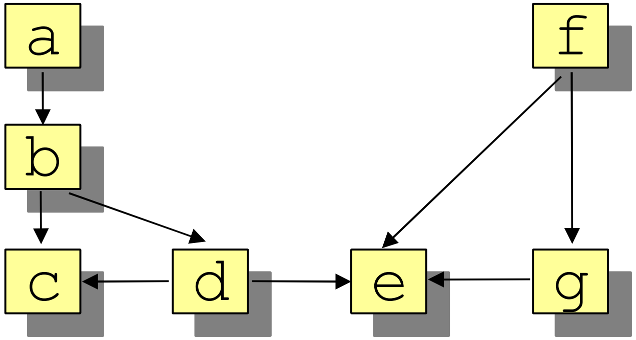
The *leaves* of a relation (viewed as a graph)



Some questions

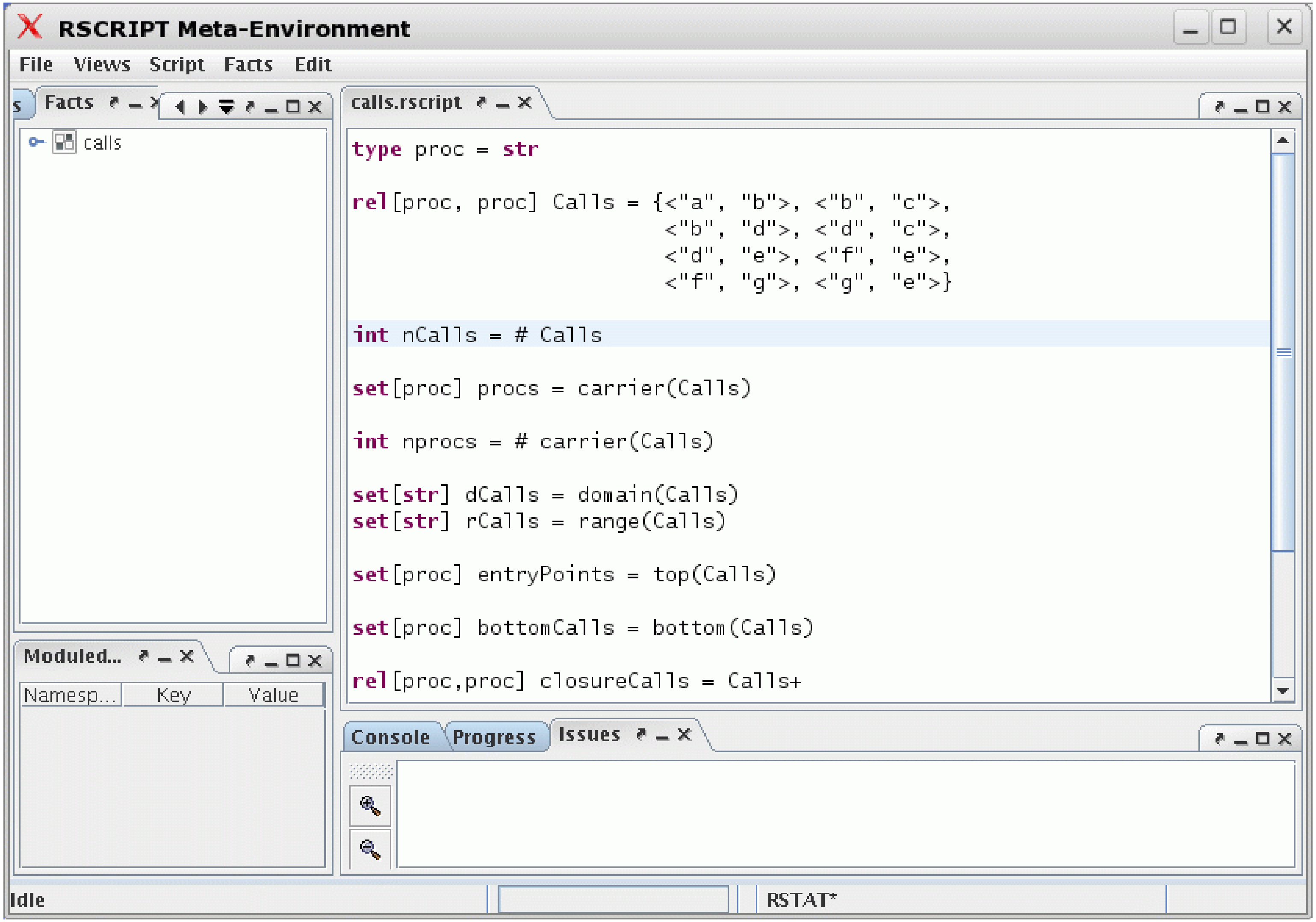
- What are the indirect calls between procedures?
  - rel[str,str] closureCalls = calls+
  - {<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">, <"d","e">, <"f", "e">, <"f", "g">, <"g", "e">, <"a", "c">, <"a", "d">, <"b", "e">, <"a", "e">}
- What are the calls from entry point a?
  - set[str] calledFromA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

Some questions

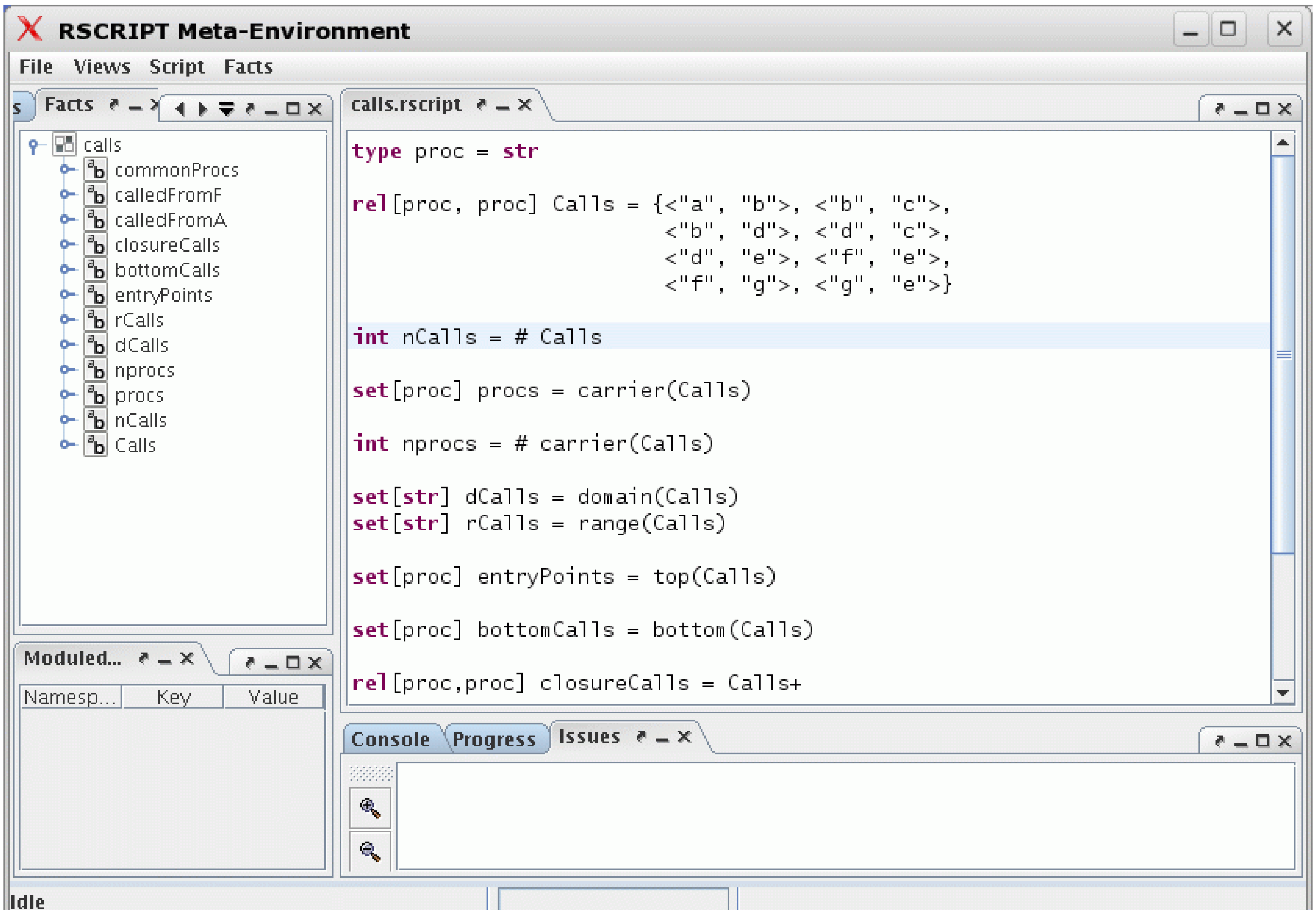


- What are the calls from entry point f?
  - set[str] calledFromF = closureCalls["f"]
  - {"e", "g"}
- What are the common procedures?
  - set[str] commonProcs = calledFromA inter calledFromF
  - {"e"}

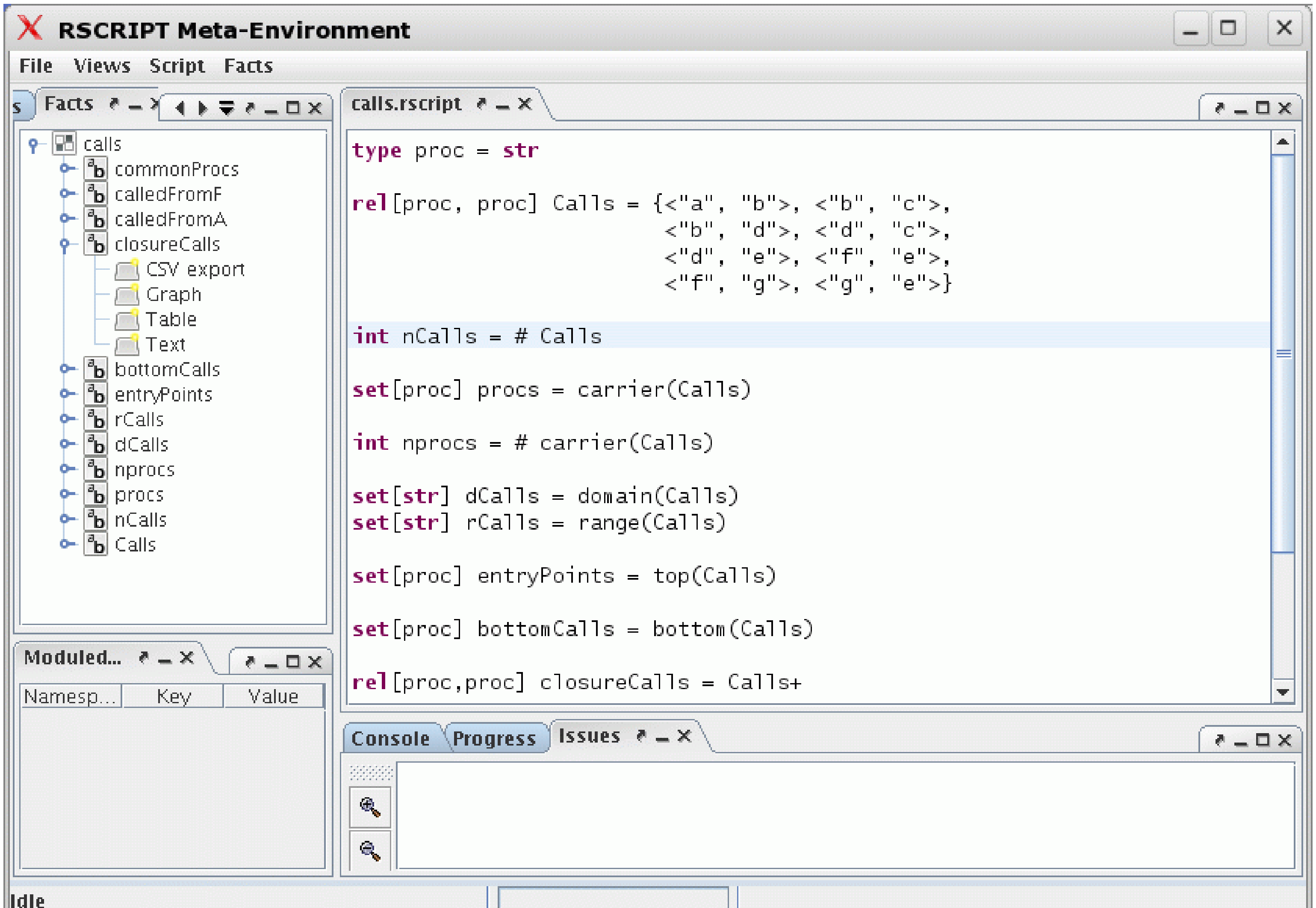
Script -> Run


RSCRIPT Meta-Environment
File Views Script Facts Edit
Facts
calls
calls.rscript
type proc = str
rel[proc, proc] Calls = {<"a", "b">, <"b", "c">,
<"b", "d">, <"d", "c">,
<"d", "e">, <"f", "e">,
<"f", "g">, <"g", "e">}
int nCalls = # Calls
set[proc] procs = carrier(Calls)
int nprocs = # carrier(Calls)
set[str] dCalls = domain(Calls)
set[str] rCalls = range(Calls)
set[proc] entryPoints = top(Calls)
set[proc] bottomCalls = bottom(Calls)
rel[proc,proc] closureCalls = Calls+
Moduled...
Namesp... Key Value
Console Progress Issues
Idle
RSTAT*

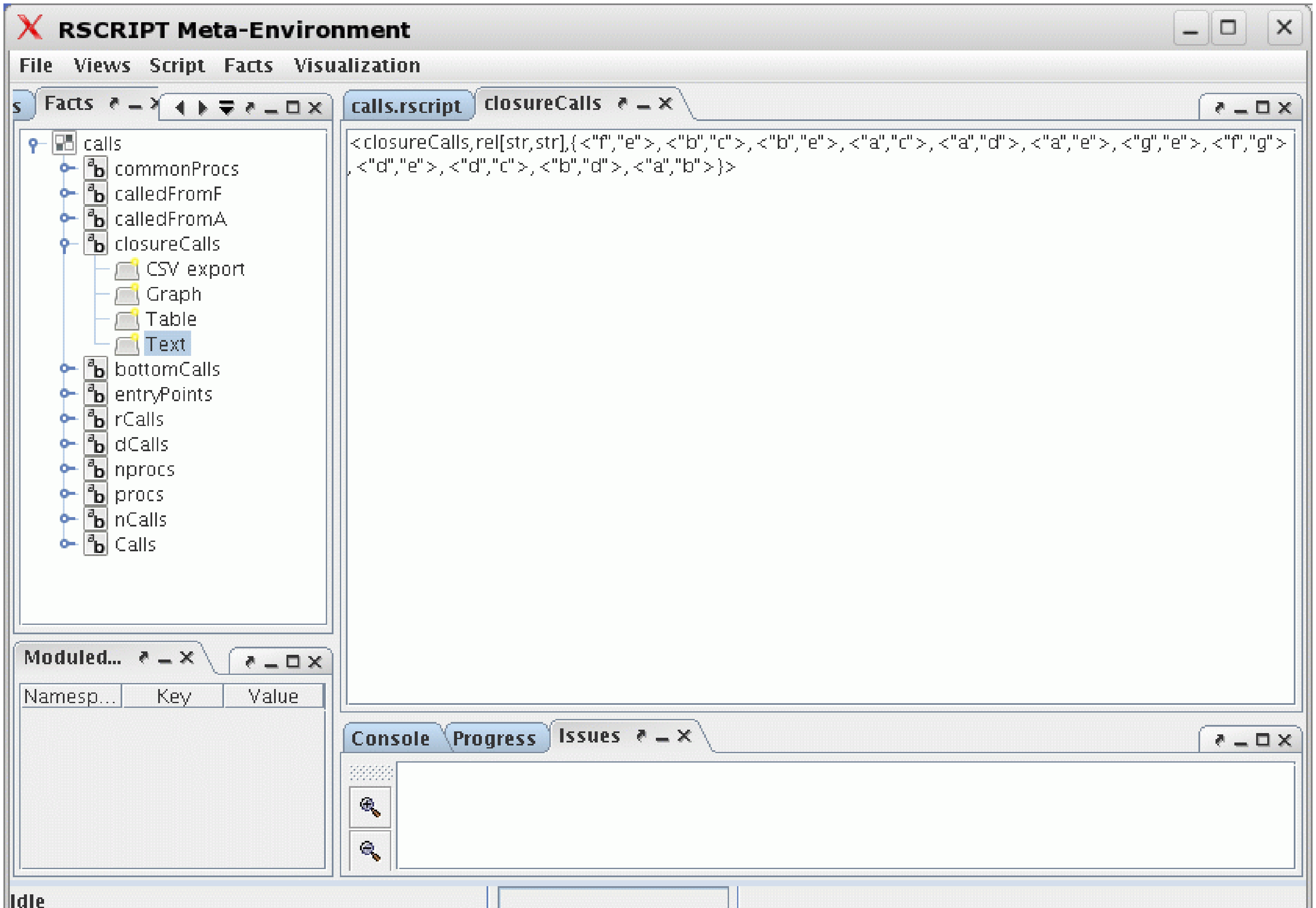
Unfolding the rstore ...



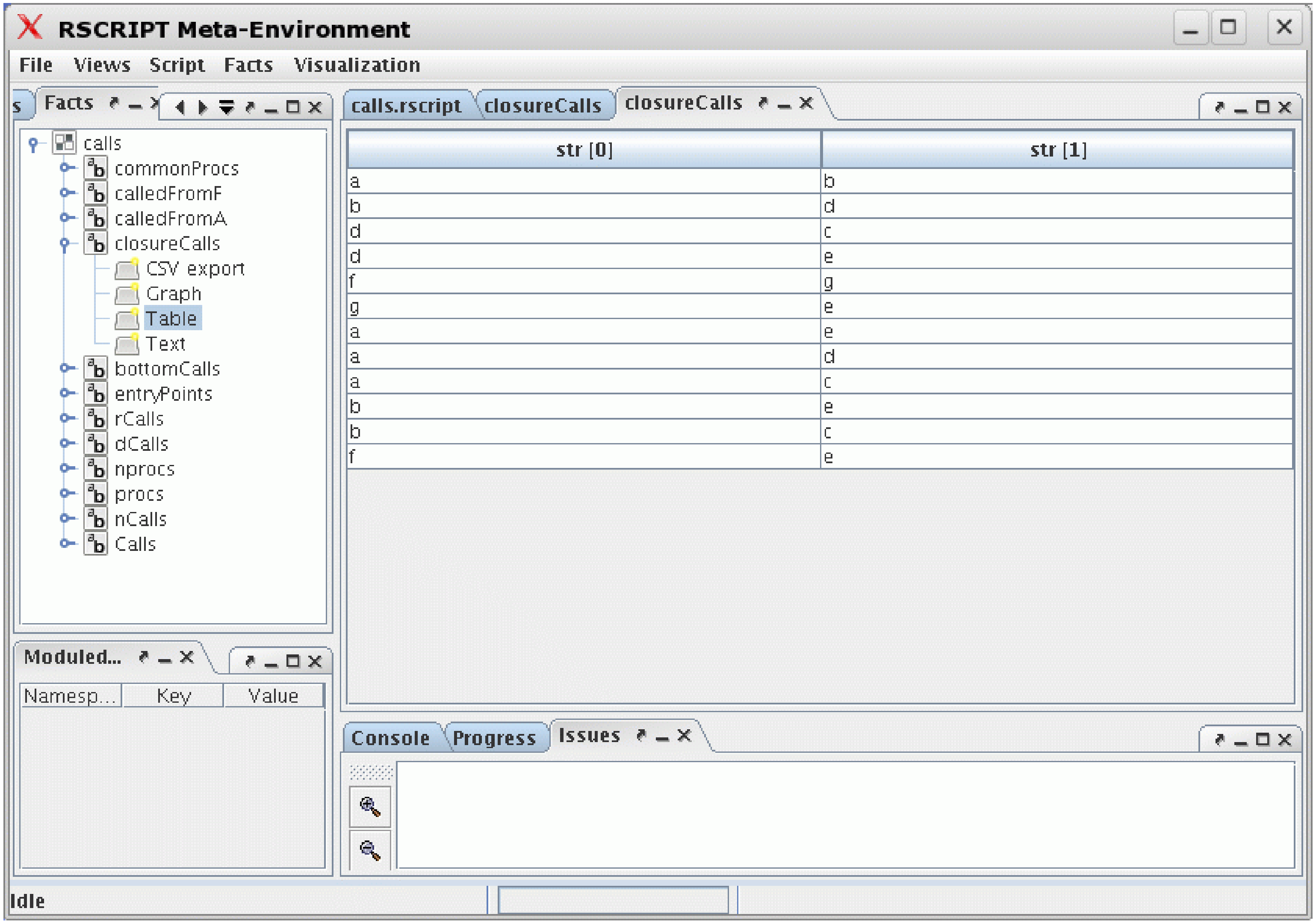
Unfolding closureCalls



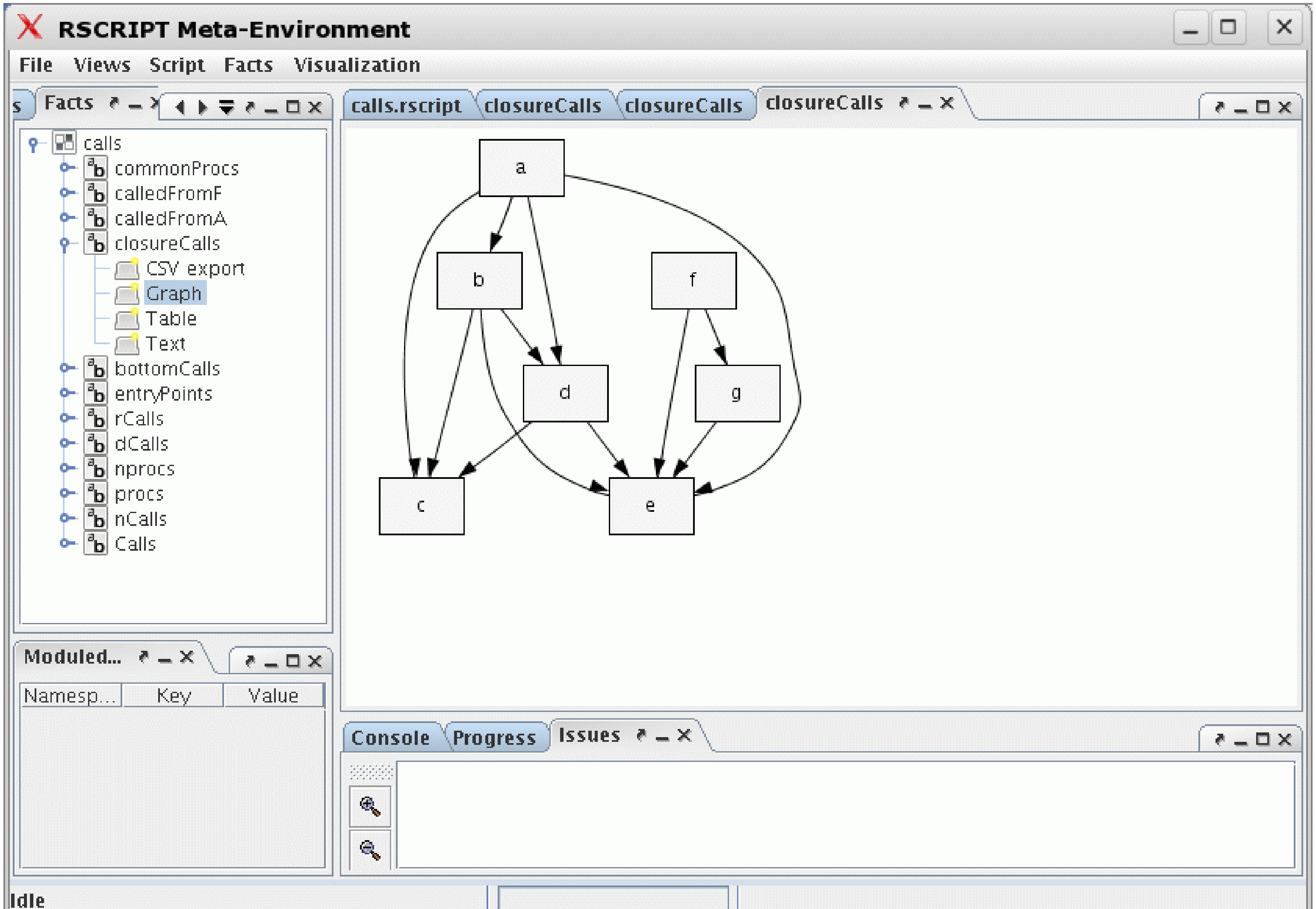
closureCalls as Text



closureCalls as Table



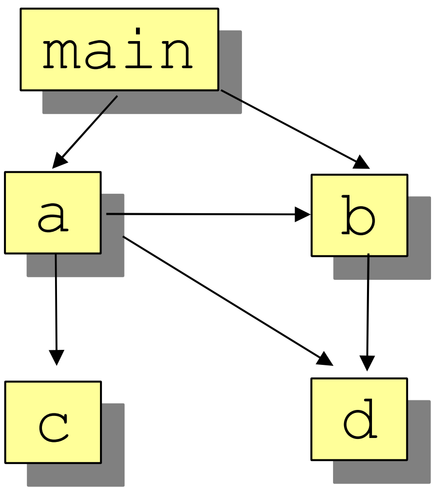
closureCalls as Graph



Component Structure of Application

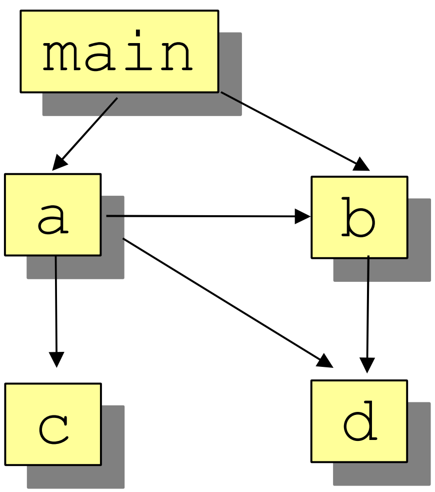
- Suppose, we know:
  - the call relation between procedures (Calls)
  - the component of each procedure (PartOf)
- Question:
  - Can we lift the relation between procedures to a relation between components (ComponentCalls)?
- This is usefull for checking that real code conforms to architectural constraints

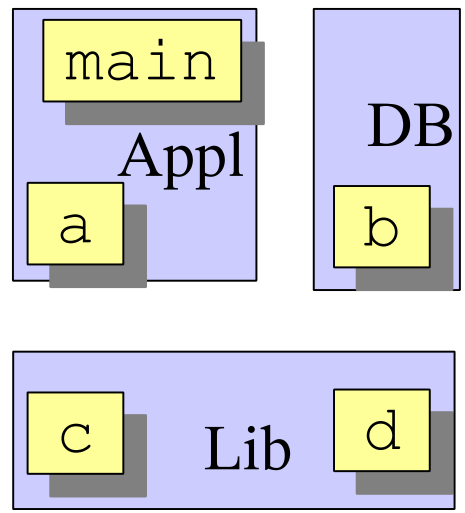
Calls




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

PartOf





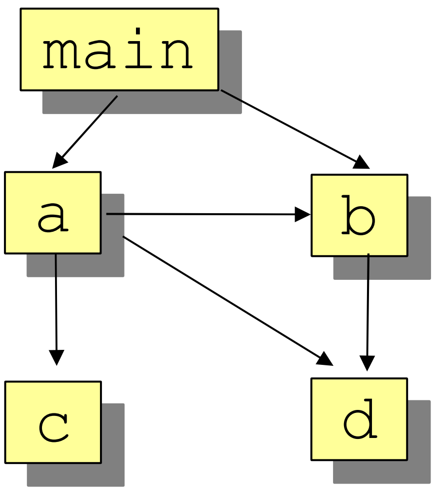
set[comp] Components = {"Appl", "DB", "Lib"}

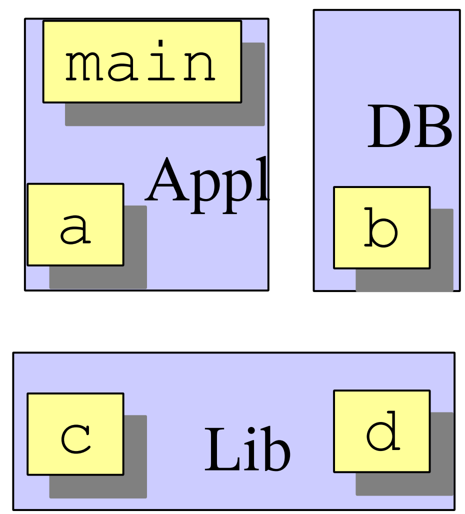
rel[proc, comp] PartOf =

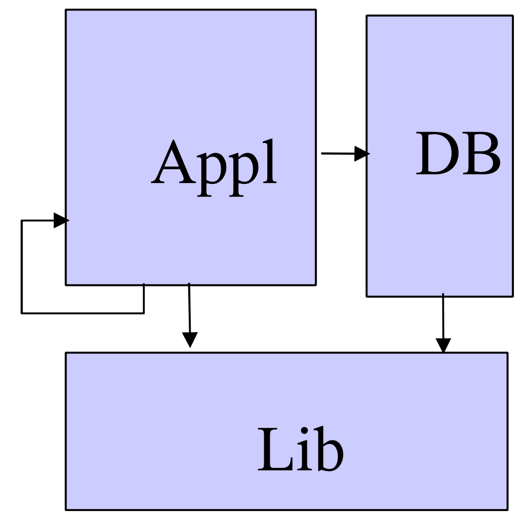
{<"main", "Appl">, <"a", "Appl">, <"b", "DB">,

<"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 good news

- ASF+SDF in use for many analysis & transformation projects
- User-definable syntax + conditional rewrite rules + relational calculus is a good feature set for this domain
- Performance is ok (regular winner of rewrite competions)

The bad News

- Missing features
  - Fact extraction turns out to be *the* bottleneck
  - => DeFacto: annotated grammars
- Mixture of formalisms increases learning curve
- Underlying mechanisms not easy to understand for the average programmer
- We are still struggling with grammarware issues
  - How to develop, test, improve grammars?

Technology integration: Partial Answers

- Program generators
- Middleware, like the ToolBus
  - not discussed in this presentation
- Integration in a single linguistic framework
  - Discussion on Rascal

Technology integration: Partial Answers

- Program generators
- Middleware, like the ToolBus
  - not discussed in this presentation
- Integration in a single linguistic framework
  - Discussion on Rascal

Technology integration: Partial Answers

- Program generators
- Middleware, like the ToolBus
  - not discussed in this presentation
- Integration in a single linguistic framework
  - Discussion on Rascal

Rascal: new scripting language for analysis and transformation

- Small learning curve for Java programmers
- Build on top of Java, easy access to the whole Java infrastructure
- Integration with Eclipse
- Suited for analysis, refactoring & transformation
- First target: simplifying refactorings in Eclipse
- Second target: a grammarware laboratory

Features

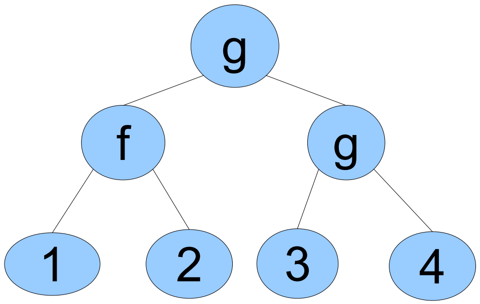
- Full context-free parsing (re-uses SDF)
- Matching (regular, abstract, concrete)
- Rich datatypes (based on Rscript)
- Conditional rewrite rules *and* functions
- Control structures geared to support matching, tree traversal and local backtracking
- Comprehensions (list, set, map)
- Generators in comprehensions can range over abstract and parse tree datatypes

Rascal datatypes

- 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 ("comfort typing")

Manipulating ADTs

```
data NODE =  int N
          |  f(NODE I, NODE J)
          |  g(NODE I, NODE J)
          |  h(NODE I, NODE J)
          ;
```

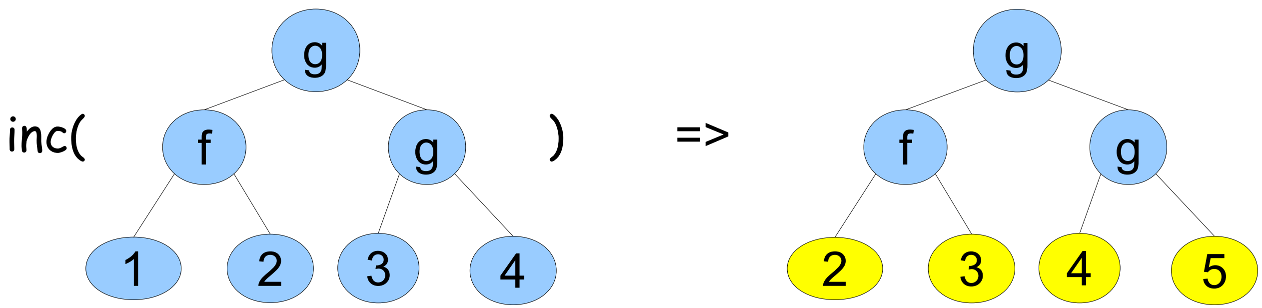


Increment all integer nodes

```
public node inc(NODE 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

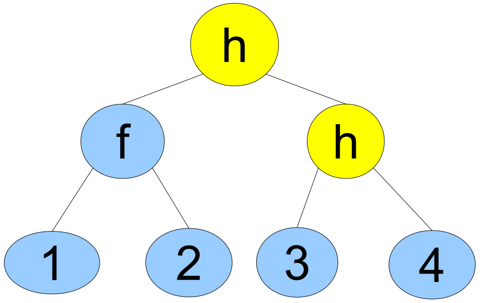


Note

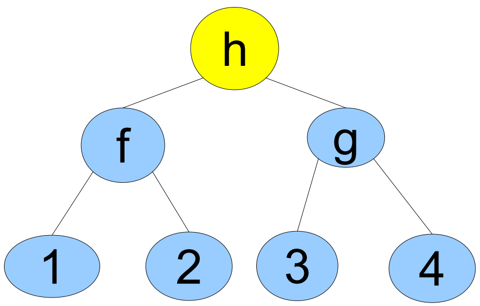
- This code is insensitive to the number of constructors
  - Here: 4
  - In Java or Cobol: hundreds
- Lexical/abstract/concrete matching
- List/set matching
- Visits can be parameterized with a strategy

Full/shallow/deep replacement

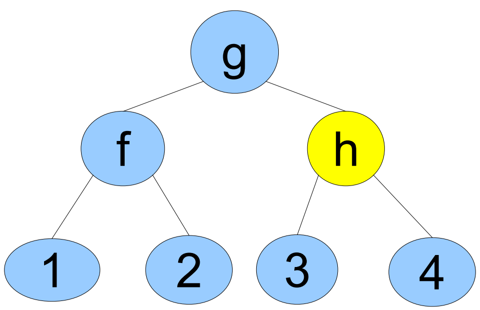
```
public NODE frepl(NODE T) {
   return visit (T) {
      case g(NODE T1, NODE T2) =>  h(T1, T2)
   };
}
```




```
public NODE srepl(NODE T) {
   return top-down-break visit (T) {
      case g(NODE T1, NODE T2) =>  h(T1, T2)
   };
}
```




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



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 := toString(V)
            };
}
```

Import the COBOL grammar

Traverse P and return all occurrences of variables

Put variables that match in result

Variable name matches a date-related heuristic

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};
}
```

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.

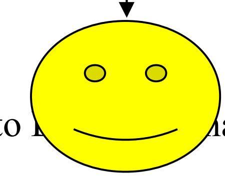
Summary

- Generic Language Technology helps to build tools for language processing quickly
- Programming Environment Generators are an application of GLT
- ASF+SDF Meta-Environment is an Interactive Development Environment for language definitions *and* a Programming Environment Generator
- Rascal: integrated language for analysis & transformation

Software renovation

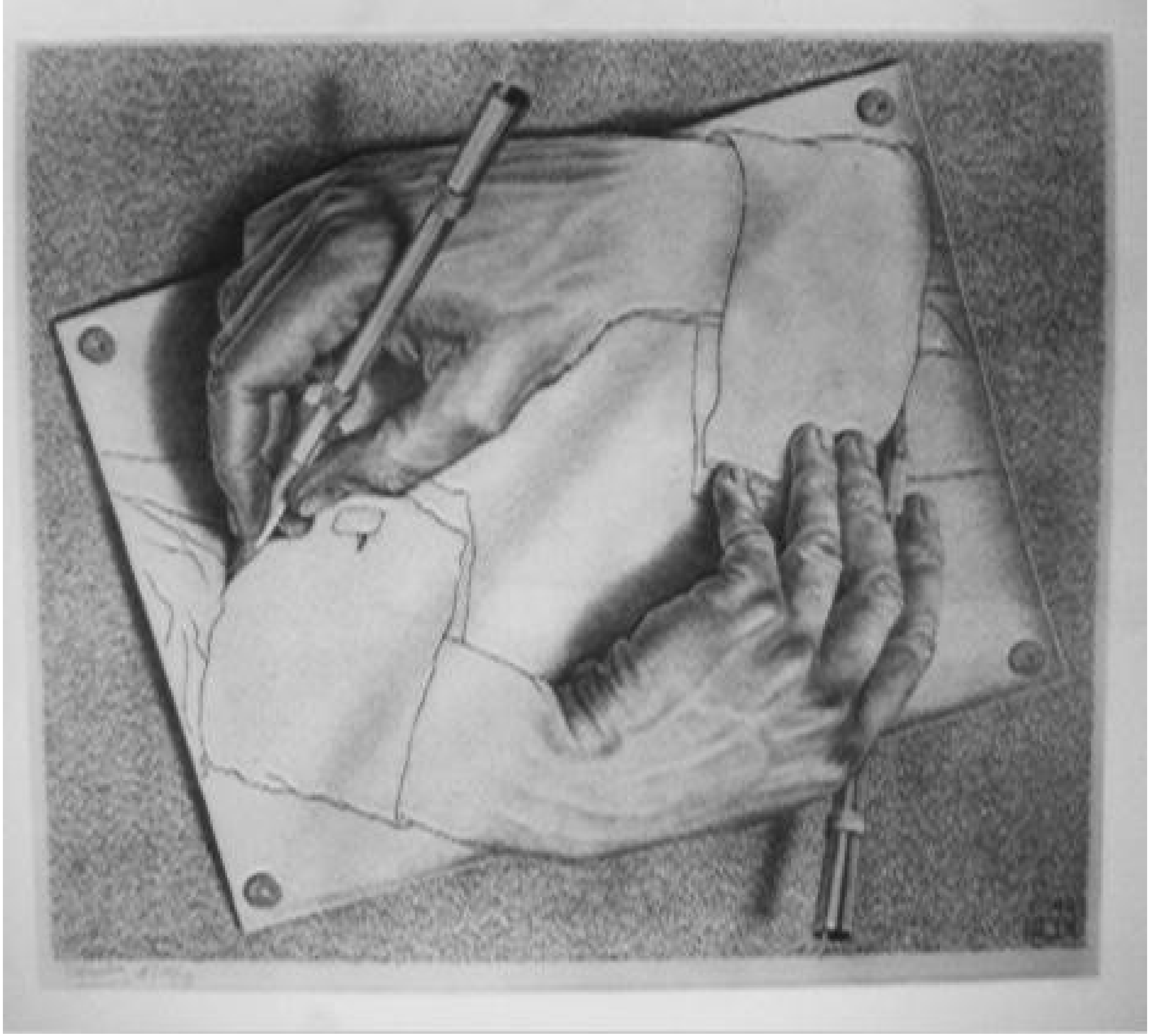
Domain-specific Languages

Generic Language Technology/ASF+SDF



Where are we relative to the grammarware challenges?

Baron Von Münchhausen pulling himself out of the swamp by his hair



M.C. Escher,
Hands drawing
themselves

Related research

- New parsing algorithms
  - Efficient parsing of general CFGs
- Heuristics ambiguity checkers
  - Undecidable, but important for grammar composition
- API refactoring efforts
  - Essential for software evolution
- Grammar metrics
  - How good is a grammar
- Grammar testing

Current Research in our Group

- DeFacto: easy fact extraction by annotating grammars
- Rascal implementation/integration in Eclipse
- Use cases:
  - Refactoring in Eclipse
  - Grammarware: towards a GrammarLab
    - Grammar refactoring
    - Grammar metrics
    - Ambiguity detection

Grammarware Research Questions

- How to provide modular grammars?
- What is a “good” grammar?
- How to transform grammars (and maintain the link with dependent software)
- How to uncover grammars from grammar-dependent source code?
- How to test grammar-dependent functionality?
- How does the grammarware “lifecycle” look like and how can we support it?

Further Reading/Questions

- Technology: www.meta-environment.org
- Home page: www.cwi.nl/~paulk