Introduction to Generic Language Technology

Mark van den Brand,
Paul Klint,
Jurgen Vinju

CWI
Universiteit van Amsterdam
TU/e technische universiteit eindhoven

How can we ...

- ... build *tools* for software analysis and manipulation?
- ... make the tools *programming language independent* (language-parametric)?
- ... integrate the tools in Interactive Development Environment (IDE)?
- ... a quick general overview, and then an introduction to specific technologies?

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

What is a Language?

- A programming language
  - Assembler, Cobol, PL/I, C, C++, Java, C#, ...
- A Domain-Specific Language (DSL)
  - SQL for queries
  - BibTex for entries in a bibliography
  - Euris for railroad emplacement safety
  - Risla for financial products

Aspects of a Language

- Syntax
  - Textual form of declarations, statements, etc.
- Static Semantics
  - Scope and type of variables, conversions, formal/actual parameters, etc.
  - Queries: who calls who, who uses variable X, ...
- Dynamic Semantics
  - Program execution

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

What is a Programming Environment?

A system that supports the development of programs in order to:

- Increase productivity:
  - Uniform user-interface (UI); integrated tools
  - Increased interaction; early error detection
- Increase quality:
  - Integrated version management
  - Integrated testing

Classical PE

- Text editor only
- Programs stored in files
- Complete recompilation after each change
- Late error detection
- Debugging requires recompilation
- Example:
  - xemacs or vim
  - gcc or javac

Integrated PE (IPE)

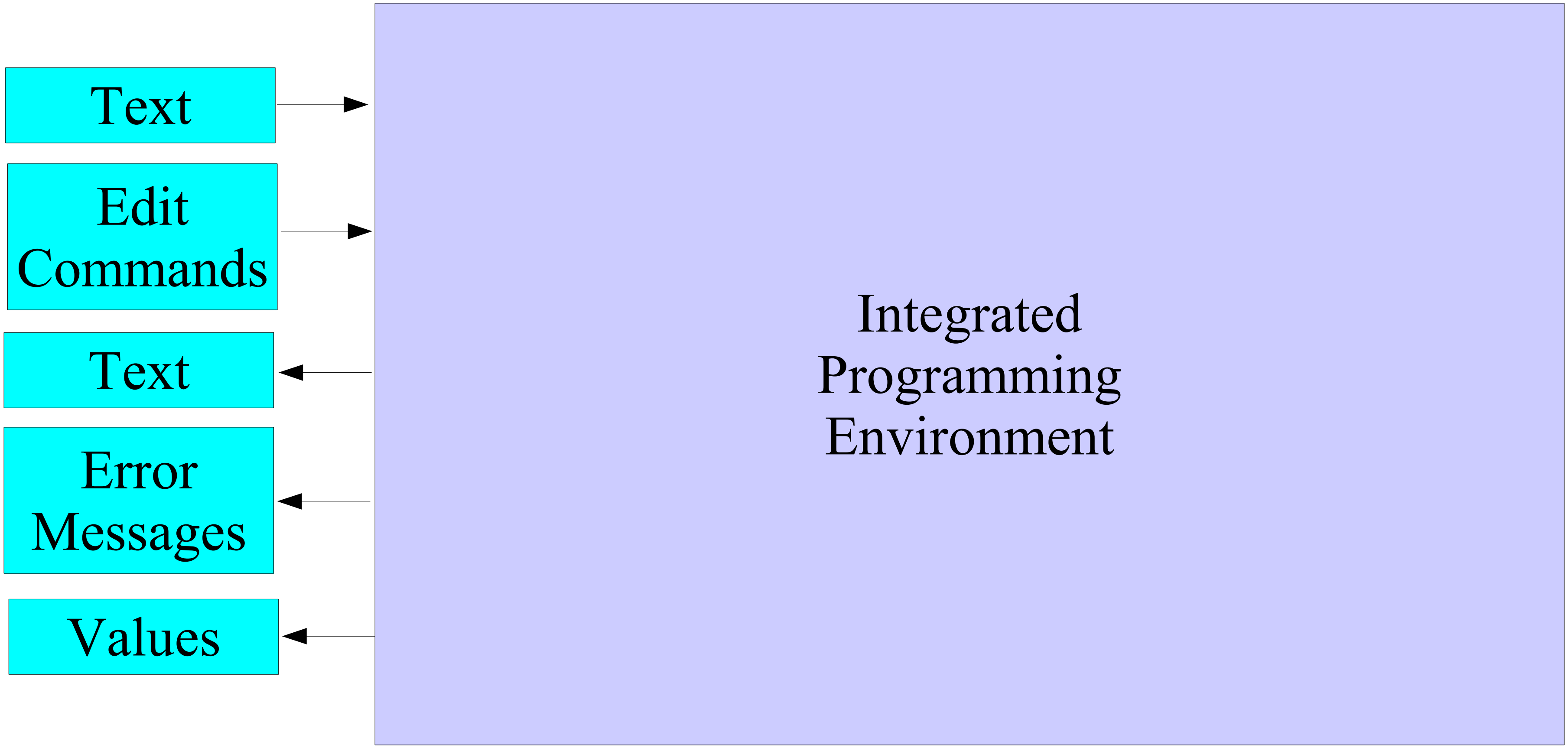
also: Integrated Development Environment (IDE)

- Specialized, syntax-directed, editor for each language
- Common intermediate representation for all tools
- Incremental processing
- Early error detection
  - Syntax errors
  - Undeclared variables
  - Type errors in expressions

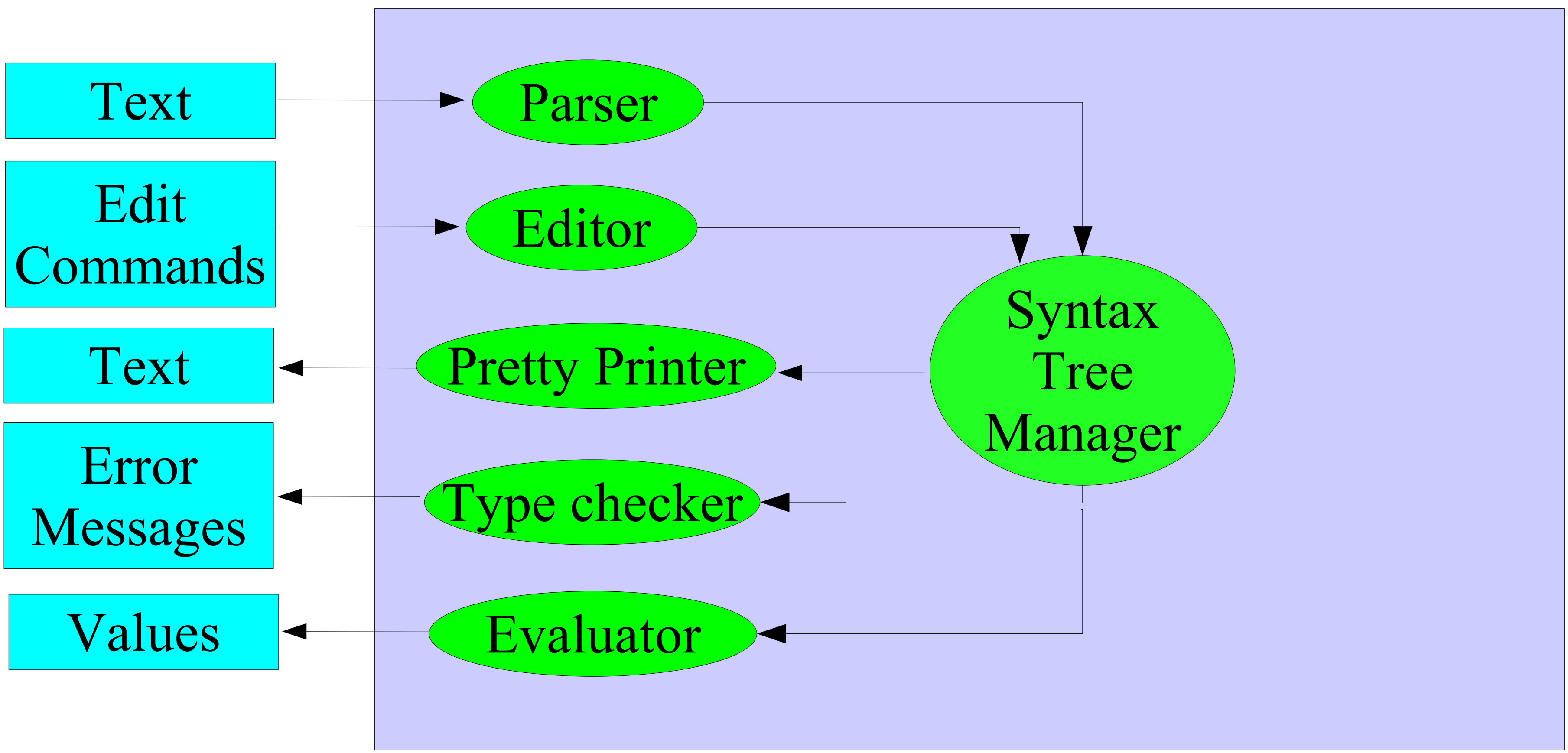
Functionality of a IPE

- Syntax-directed editing/highlighting, pretty printing
- Typechecking
- Restructuring
- Versioning
- Executing, debugging, profiling
- Testing
- Documenting

Simple, External, View of IPE



Simple, Internal, View of IPE



Examples of IPEs

- Eclipse: www.eclipse.org
  - Integrated Development Environment (IDE) for Java
  - Plug-in mechanism for extensions
- MS Visual Studio: msdn.microsoft.com/vstudio
  - IDE for various languages VB, C, C++, C#

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

What is Generic Language Technology?

- 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 editor
- Program storage
- Documentation

Defining Language Aspects

- Syntax
  - Context-free grammar
- Static semantics
  - Algebraic specification/rewrite rules
- Dynamic semantics
  - Algebraic specification/rewrite rules

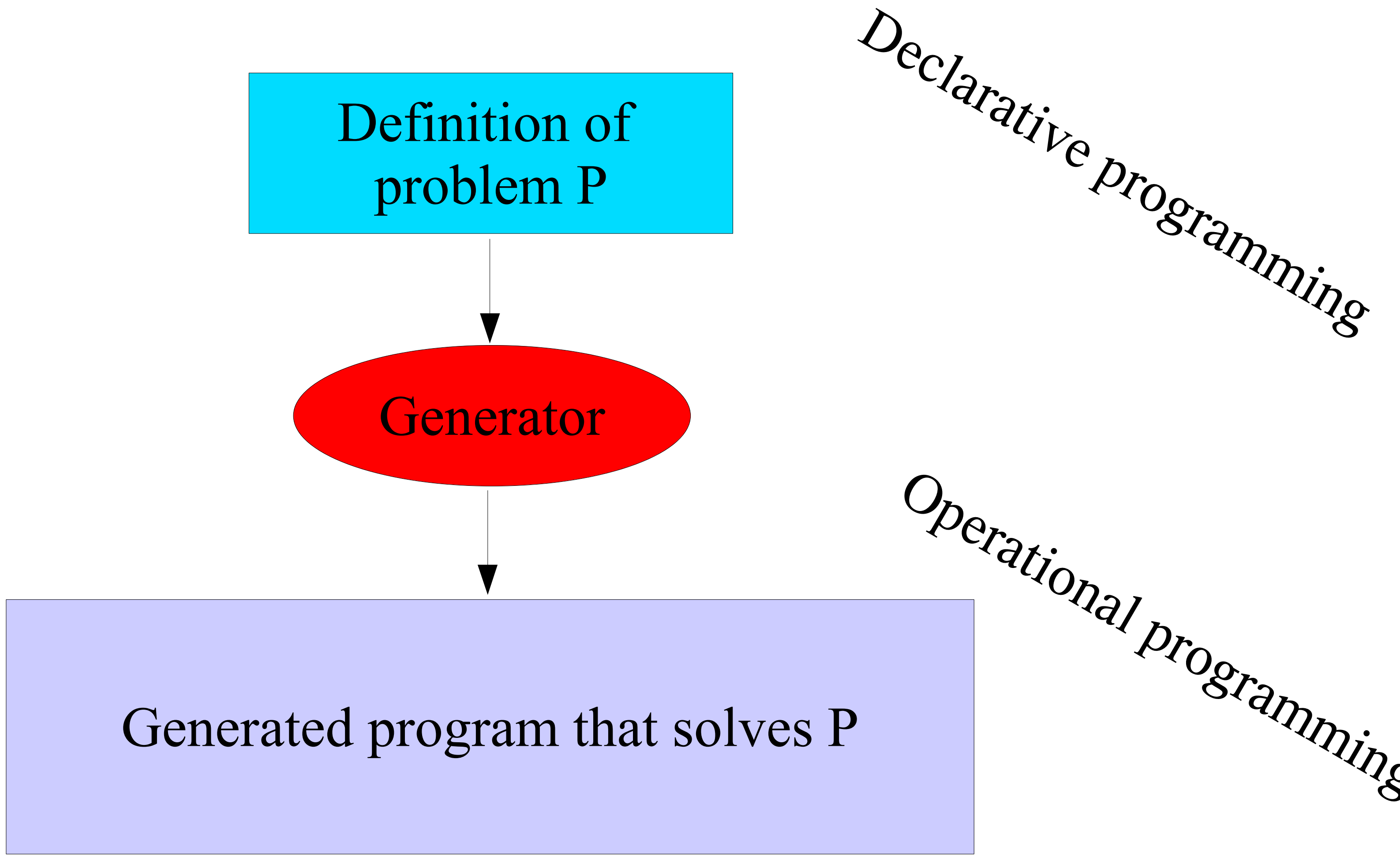
From Definition to Tool

- Syntax
  - Parser generation
- Static semantics
  - Term rewriting
- Dynamic semantics
  - Term rewriting

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

What is a Program Generator?



Examples of Program Generators (1)

- Regular expression matching:
  - Problem: recognize regular expressions R_1, ..., R_n in a text
  - Generates: finite automaton
- Web sites
  - Problem: create uniform web site for set of HTML pages
  - Generate: HTML code with standard layout and site map

Examples of Program Generators (2)

- Generate bibliographic entries; input

```
@article{BJKO00,   author  = {Brand, {M.G.J. van den} and Jong,  {H.A. de}
                              and P. Klint and P. Olivier},
                   title   = {{E}fficient {A}nnotated {T}erms},
                   journal = {Software, Practice \& Experience},
                   year    = {2000},
                   pages   = {259—291},
                   number  = {3},
                   volume  = {30}}
```

generates:

M.G.J. van den Brand, H.A. de Jong, P. Klint and P.A. Olivier, Efficient Annotated Terms, *Software, Practice & Experience*, **30**(3):259—291, 2000

Examples of Program Generators (3)

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

Program Generators (summary)

- 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.

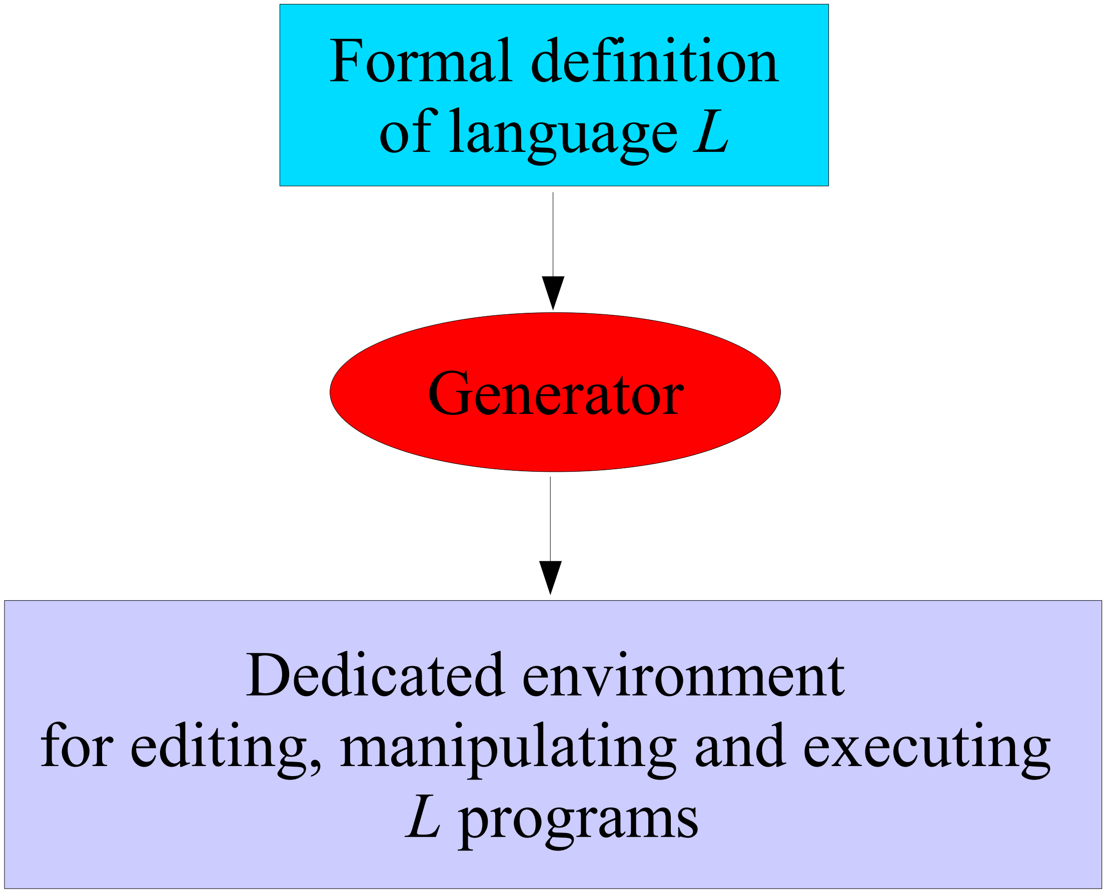
What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

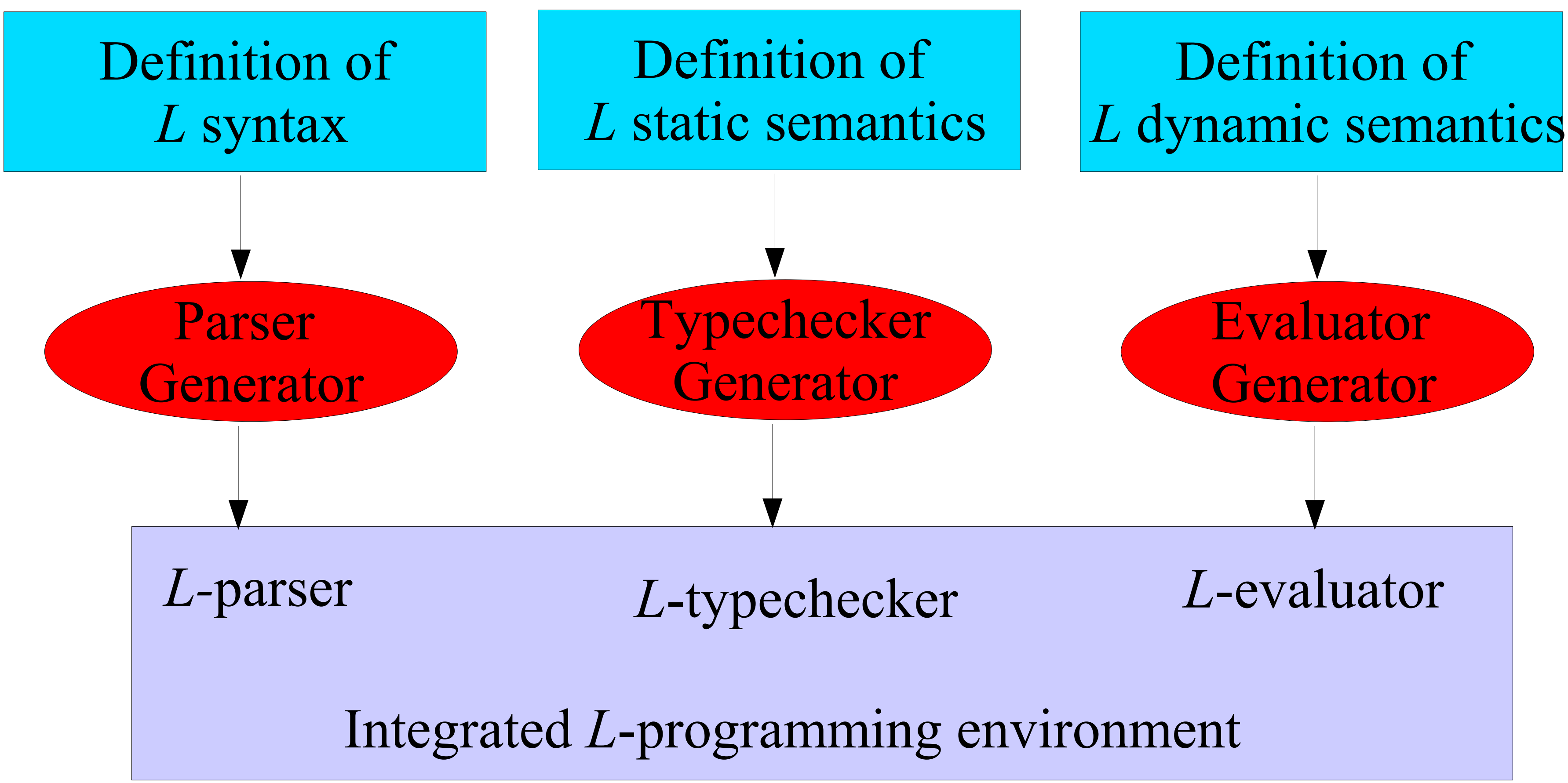
What is a Programming Environment Generator (PEG)?

- A PEG is a program generator applied in the domain of programming environments
- Input: description of a desired language *L*
- Output: (parts of) a dedicated *L* environment
- Advantages:
  - Uniform interface across different languages
  - Generator contains generic, re-usable, implementation knowledge
- Disadvantage: some UI optimizations are hard

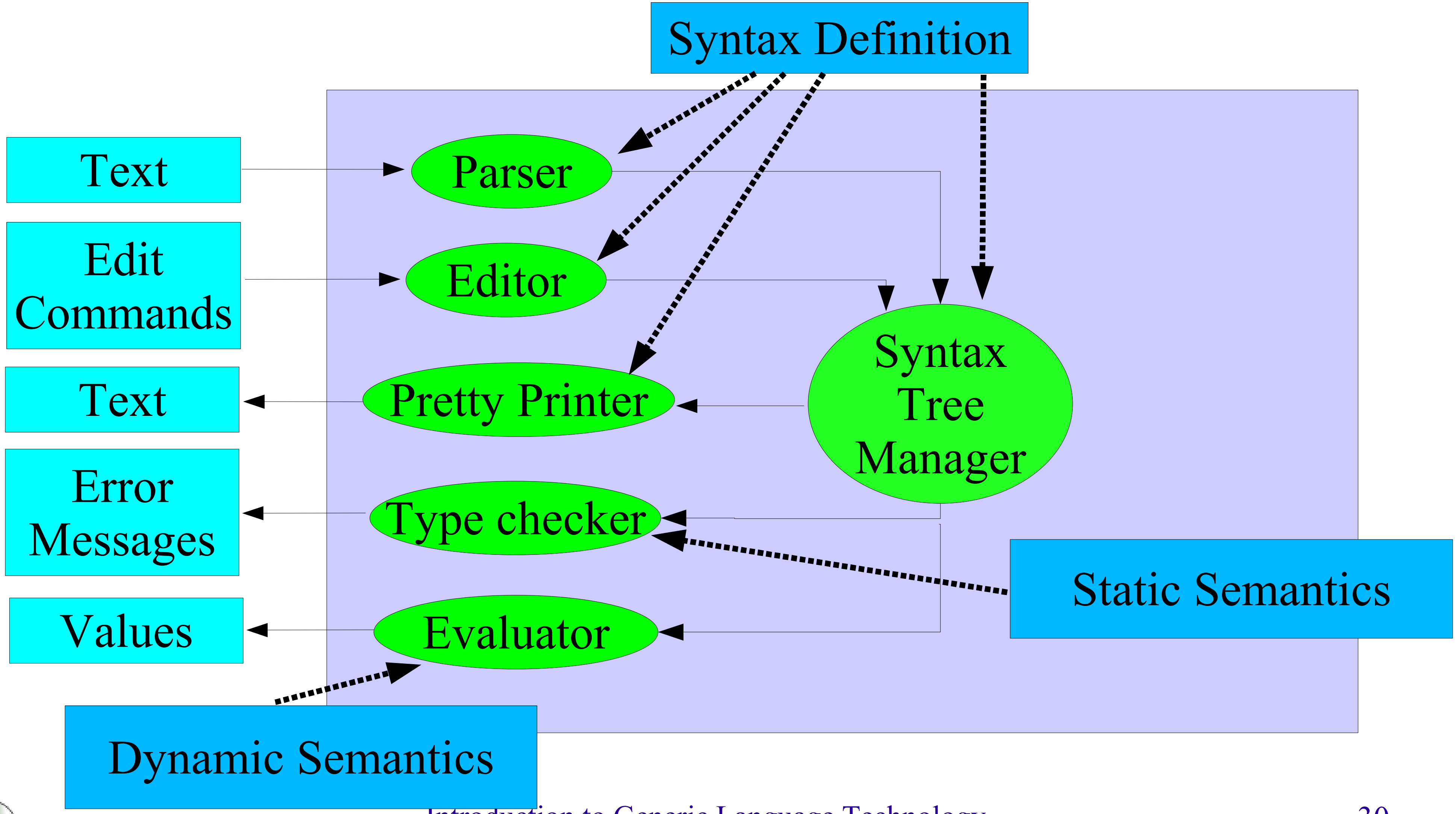
Programming Environment Generator



PEG = collection of program generators



From Definitions to Components



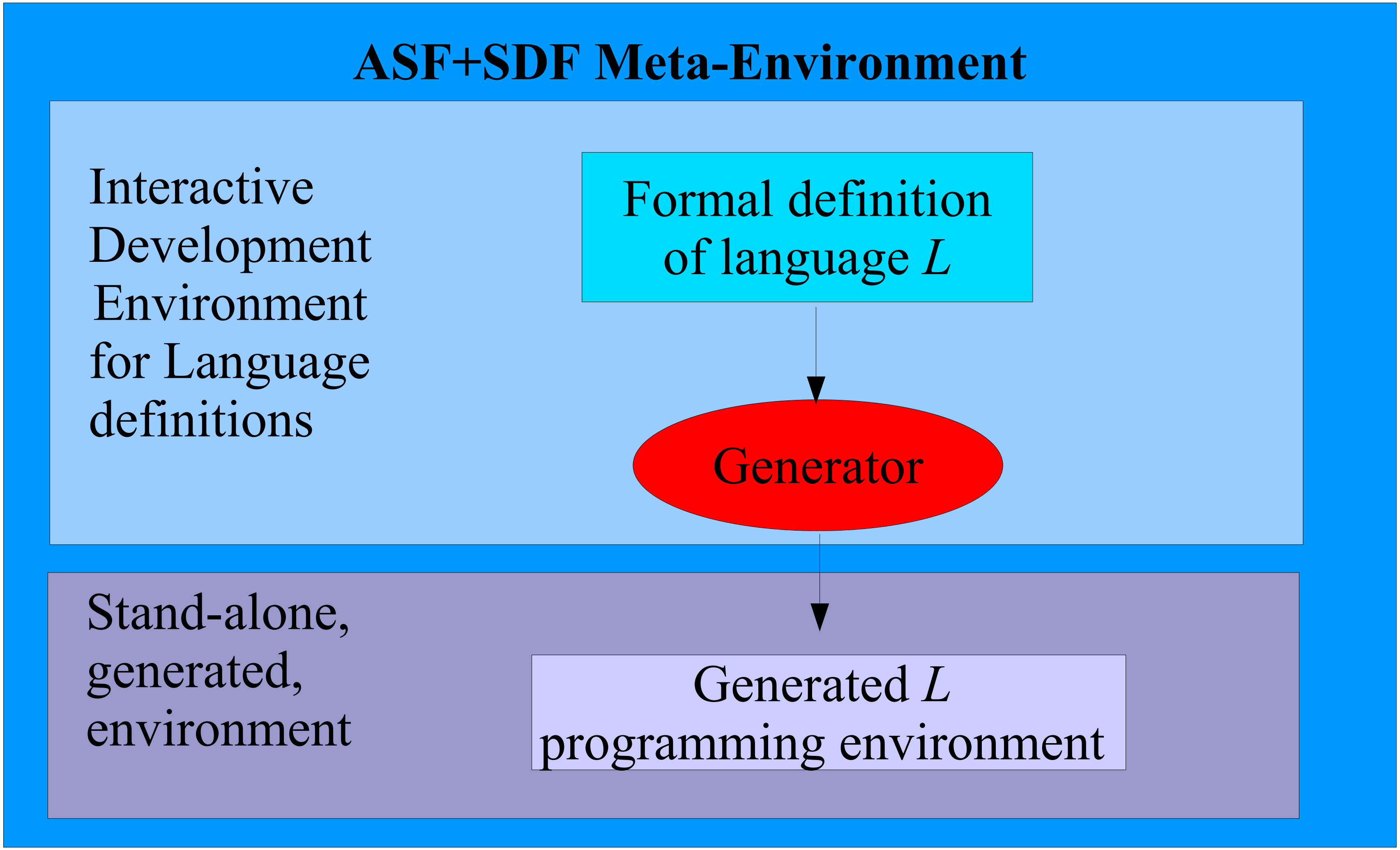
PEG: further definitions

- 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
- ...

ASF+SDF Meta-Environment (1)

- An interactive development environment for generating tools from formal language definitions
- Based on:
  - Full context-free grammars
  - Conditional term rewriting
- Language definitions written in ASF+SDF
  - SDF: Syntax definition Formalism
  - ASF: Algebraic Specification Formalism

ASF+SDF Meta-Environment (2)



ASF+SDF Specifications

- Series of modules
- A module can import other modules
- A module can be parameterized
- 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, analysis, evaluation, transformation, ...

Booleans: syntax

```
module Booleans
 exports
  sorts BOOL
  context-free syntax
     "true"            -> BOOL
     "false"           -> BOOL
     and(BOOL,BOOL) -> BOOL
 variables
    "B" -> BOOL
```

Simplified version of a library module

Booleans: terms

Defines the syntax of the language of Booleans, e.g.

- true
- and(true,false)
- and(and(true,false), and(false, false))

- or, terms with variables (as used in equations):
  - and(true,B)
  - and(and(true,false), and(B, false))

Booleans: semantics

Add semantics for and function:

```
equations
[1] and(true, true) = true
[2] and(true,false) false
[3] and(false,true) = false
[4] and(false, false) = false
```

Alternative:

```
equations
[1] and(true, B) = B
[2] and(false, B) = false
```

Booleans: complete module

```
module Booleans
 exports
   sorts BOOL
   context-free syntax
     true                -> BOOL
     false               -> BOOL
     and(BOOL,BOOL) -> BOOL
   variables
     B -> BOOL
 equations
  [1] and(true, B) = B
  [2] and(false, B) = false
```

Arithmetic (1)

- The successor notation is a well-known device to define numbers and arithmetic:
  - 0 is represented by 0
  - 1 is represented by s(0)
  - 2 is represented by s(s(0))
  - *n* is represented by $s^n(0)$

Arithmetic (2)

```
module Arithmetic
exports
  sorts  INT
  context-free syntax
    "0"              -> INT
    s(INT)           -> INT
    plus(INT, INT)   -> INT
  variables
    "X"              -> INT
    "Y"              -> INT
```

Arithmetic (3)

Add semantics for function `plus`:

```
equations
[p1] plus(0, X) = X
[p2] plus(s(X), Y) = s(plus(X, Y))
```

Or using infix notation:

```
equations
[p1'] 0 + X = X
[p2'] (X+1) + Y = (X + Y) + 1
```

Arithmetic (4)

```
module Arithmetic
 exports
   sorts  INT
   context-free syntax
     "0"                    -> INT
     s(INT)                 -> INT
     plus(INT, INT)         -> INT
   variables
     X                      -> INT
     Y                      -> INT
equations
[p1] plus(0, X) = X
[p2] plus(s(X), Y) = s(plus(X, Y))
```

Arithmetic (5)

Using these rules

we can start computing:

[p1] plus(0, X) = X

[p2] plus(s(X), Y) = s(plus(X, Y))

plus(s(s(0)), s(s(s(0)))) $=^{p2}$

s(plus(s(0), s(s(s(0))))) $=^{p2}$

s(s(plus(0, s(s(s(0)))))) $=^{p1}$

s(s(s(s(s(0)))))

In other words: **2 + 3 = 5**

Term Rewriting (1)

Rewrite rules $\{L_i \rightarrow R_i\}^{r}_{i=1}$ and initial term T_0

Example:

- [p1] plus(0, X) = X
- [p2] plus(s(X), Y) = s(plus(X, Y))

Initial term:

- plus(s(s(0)), s(s(s(0))))

Term Rewriting (2)

Match subterm (redex) of T_j with some L_i and replace by R_i (after variable substitution); this gives T_{j+1}

- Try to apply [p2] plus(s(X), Y) = s(plus(X, Y)) to plus(s(s(0)), s(s(s(0))))
- Match plus(s(s(0)), s(s(s(0)))) with plus(s(X), Y)
- Yields X=s(0) and Y= s(s(s(0)))
- Substitute in r.h.s.: s(plus(s(0), s(s(s(0)))))

Term Rewriting (3)

- We have reached a normal form T_n when no more matches in T_i are possible
- The reduction sequence is: $T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_n$

```
plus(s(s(0)), s(s(s(0)))) ->
s(plus(s(0), s(s(s(0)))) ->
s(s(plus(0, s(s(s(0)))) ->
s(s(s(s(s(0)))))
```

Term Rewriting (4)

- The order in which a redex is selected may differ
- We use innermost selection
- There is more to term rewriting:
  - Lists and list matching
  - Conditional rules
  - Default rules
  - Traversal functions
  - ...

ASF+SDF (summary)

Surprisingly, these trivial examples scale to large applications. The pattern is always:

- define a syntax (Booleans, numbers, COBOL programs)
- define functions on terms in this syntax (and, plus, eliminate-goto's)
- apply to examples of interest

We will hear later about other interesting features of ASF+SDF

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

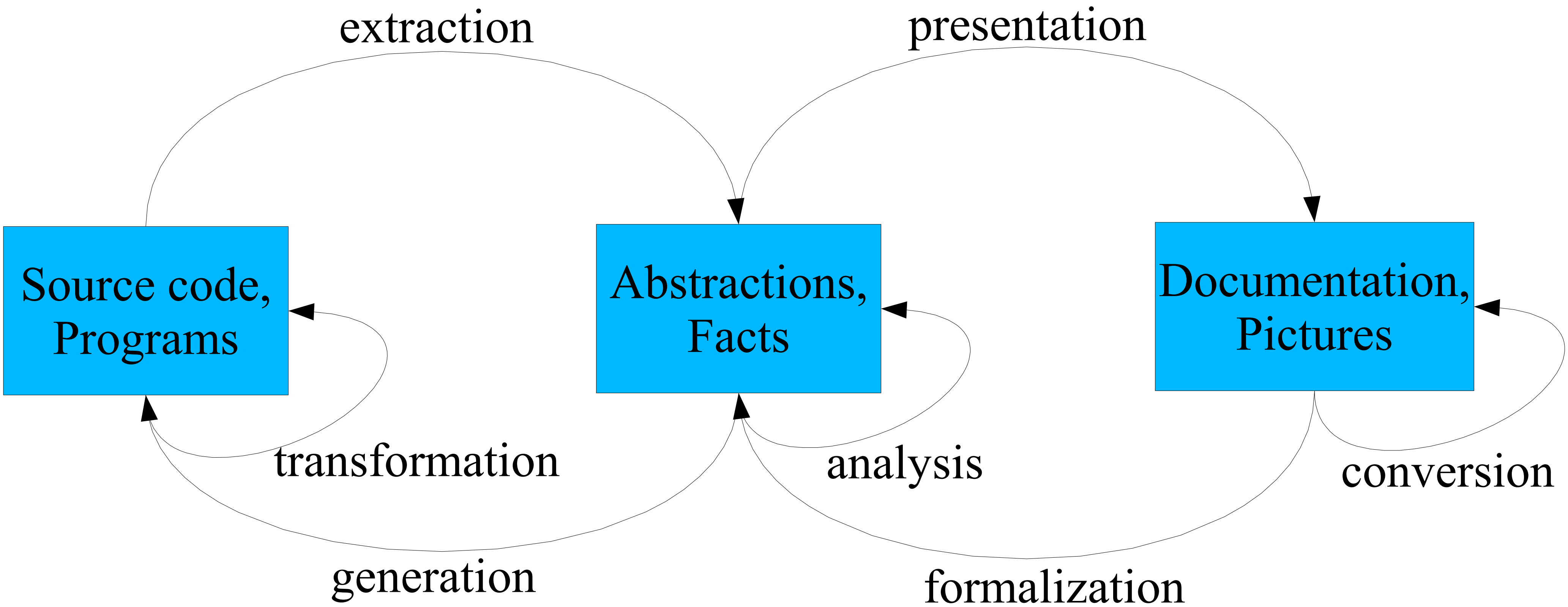
What are applications of GLT? (1)

- Domain-specific languages
  - RISLA (financial software, Fortis, ING)
  - EURIS (railroad safety, Dutch Rail)
- Software renovation
  - Analysis of telephone software (Ericsson)
  - Analysis and transformation of COBOL systems
  - Analysis of Java systems (code smells)

What are applications of GLT? (2)

- Code generation from UML
- Java verification
- Tools for various specification languages: CHI, Elan, Action Semantics, LOTOS, muCRL, ...
- Various tools of the Meta-Environment: parser generator, compiler, checkers, ...

What are applications of GLT (3)



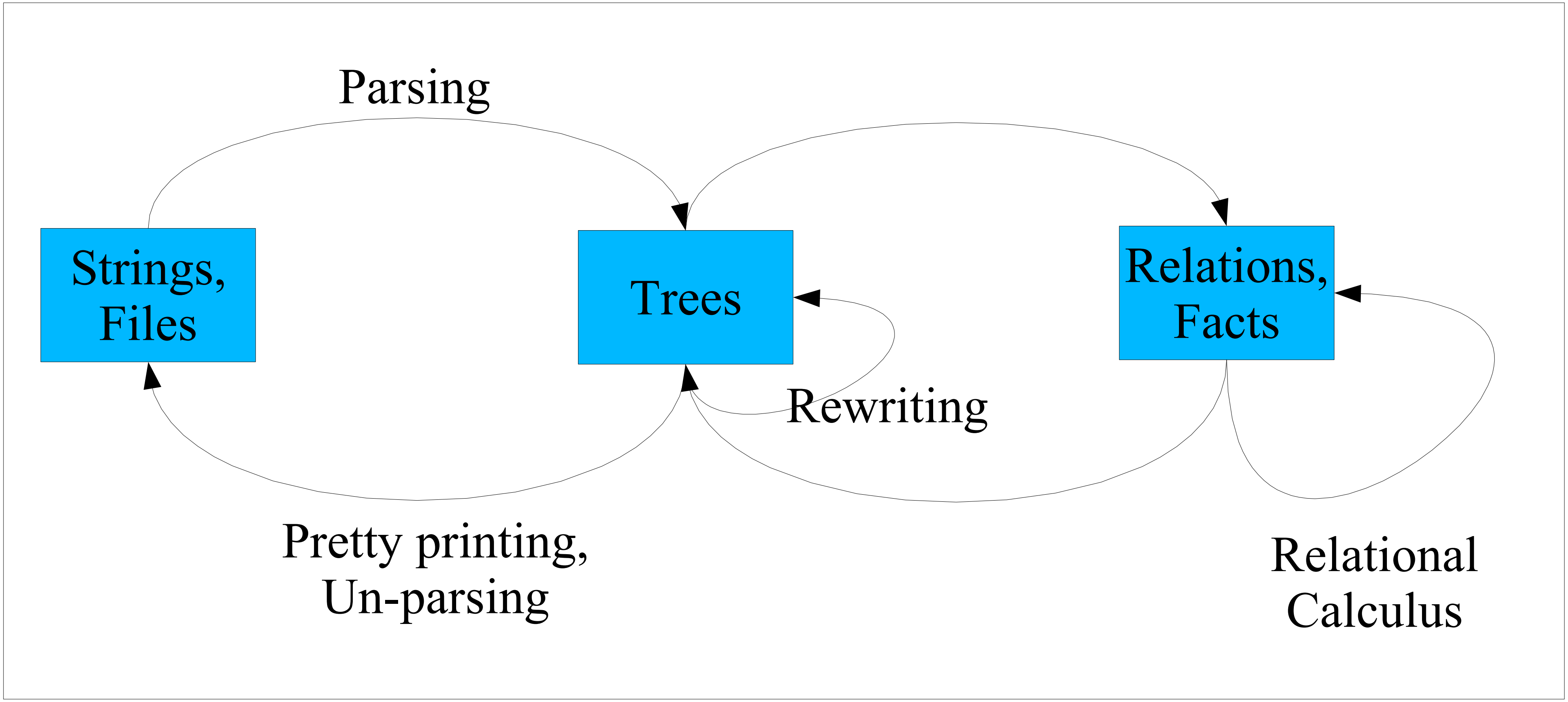
Generic Language Technology helps implementing translations between source code representations

What ...

- ... is a language?
- ... is a Programming Environment (PE)?
- ... is Generic Language Technology (GLT)?
- ... is a Program Generator?
- ... is a Programming Environment Generator?
- ... are the applications areas of GLT?
- ... technology is used for GLT?

What technology is used for GLT?

ToolBus/ATerm middleware



What Technology is used for GLT?

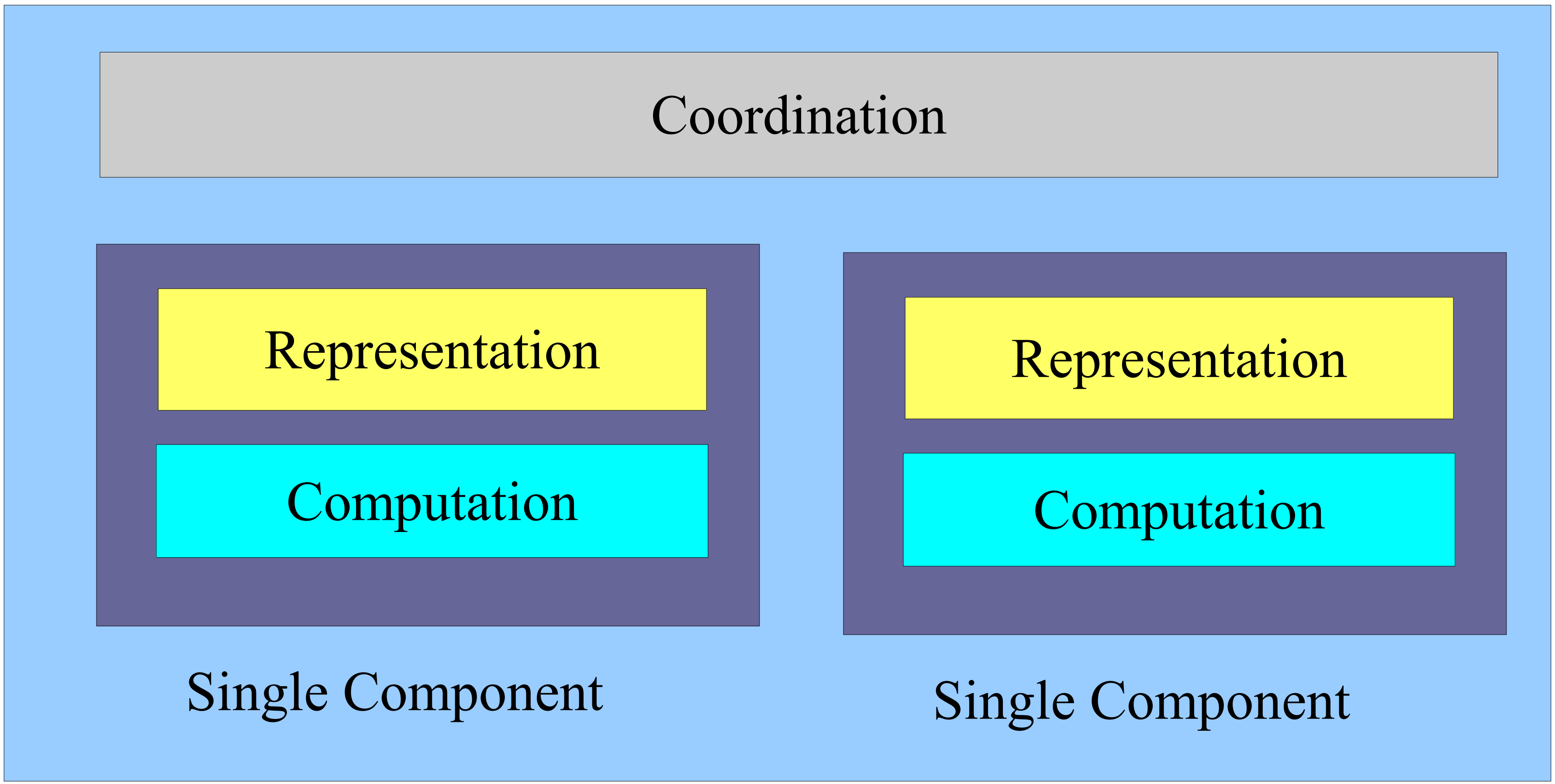
- ToolBus: a software coordination architecture used for connecting tools
- ATerms: Annotated Terms used to exchange data between tools
- SGLR: Scannerless Generalized LR parsing
- Conditional term rewriting and efficient compilation techniques

Coordination, Representation & Computation

- Coordination: the way in which program and system parts interact (procedure calls, RMI, ...)
- Representation: language and machine neutral data exchanged between components
- Computation: program code that carries out a specialized task

A rigorous separation of coordination from computation is the key to flexible and reusable systems

Architectural Layers

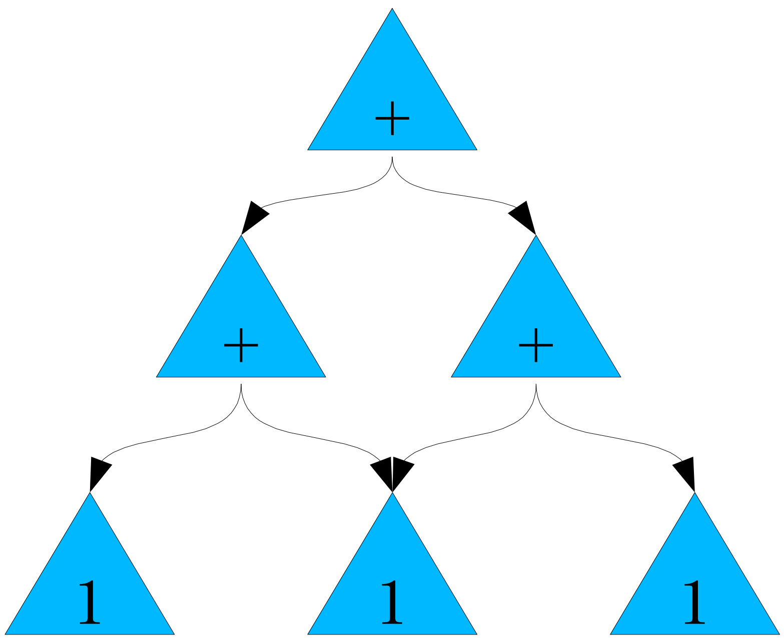


Cooperating Components

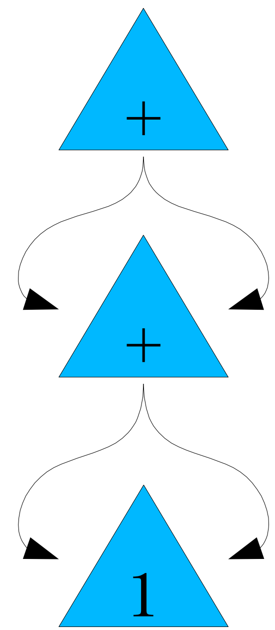
Generic Representation
Annotated Terms (ATerms)

- Applicative, prefix terms
- Maximal subterm sharing ($\Rightarrow$ DAG)
  - cheap equality test, efficient rewriting
  - automatic generational garbage collection
- Annotations (text coordinates, dataflow info, ...)
- Very concise, binary, sharing preserving encoding
- Language & machine independent exchange format

ATerms
Sharing

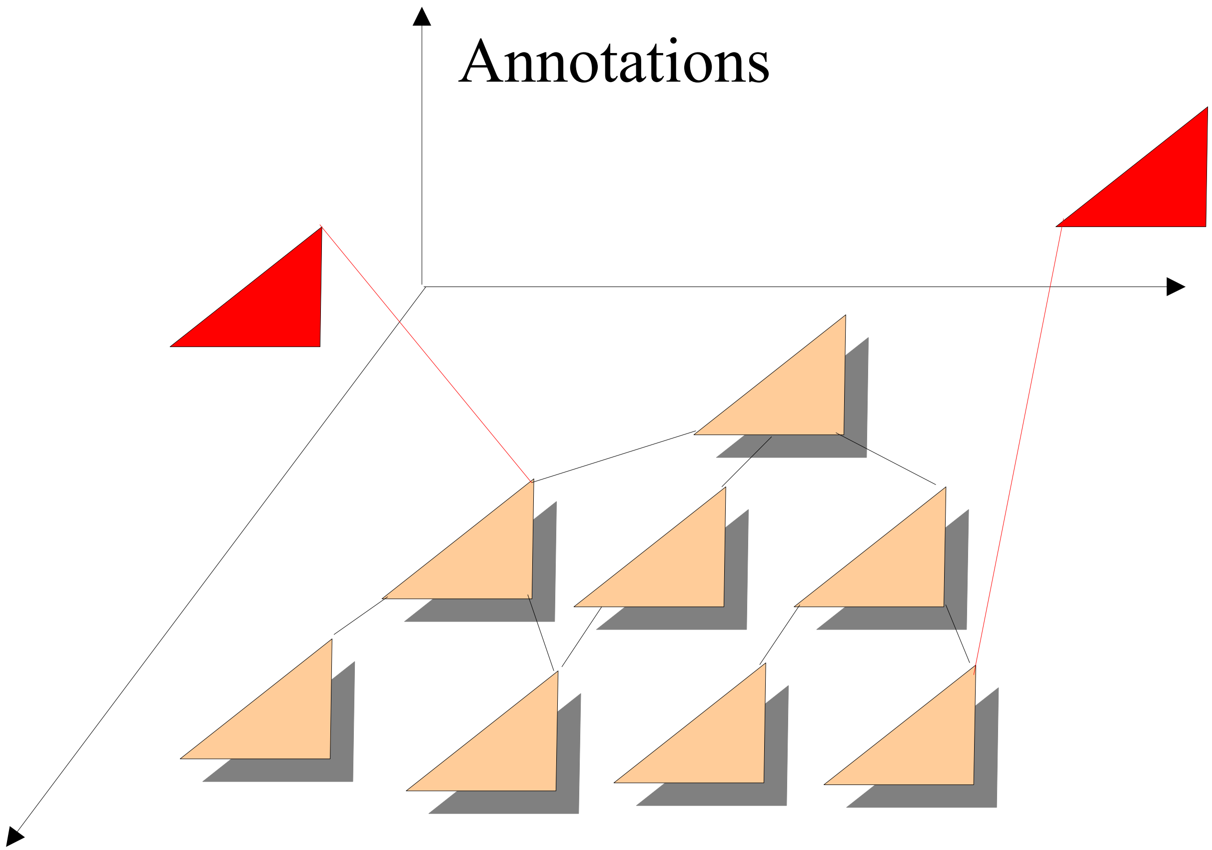


6 nodes



3 nodes

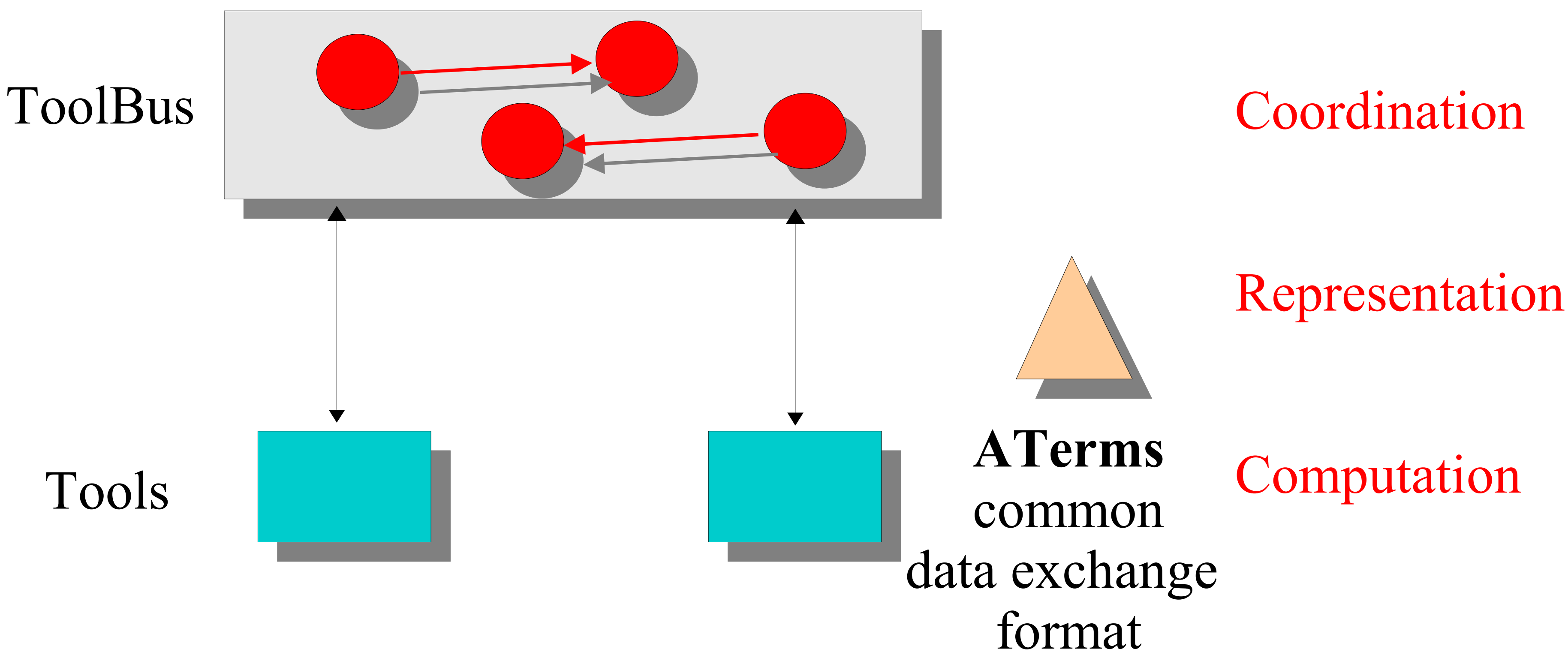
ATerms
Term and Annotations



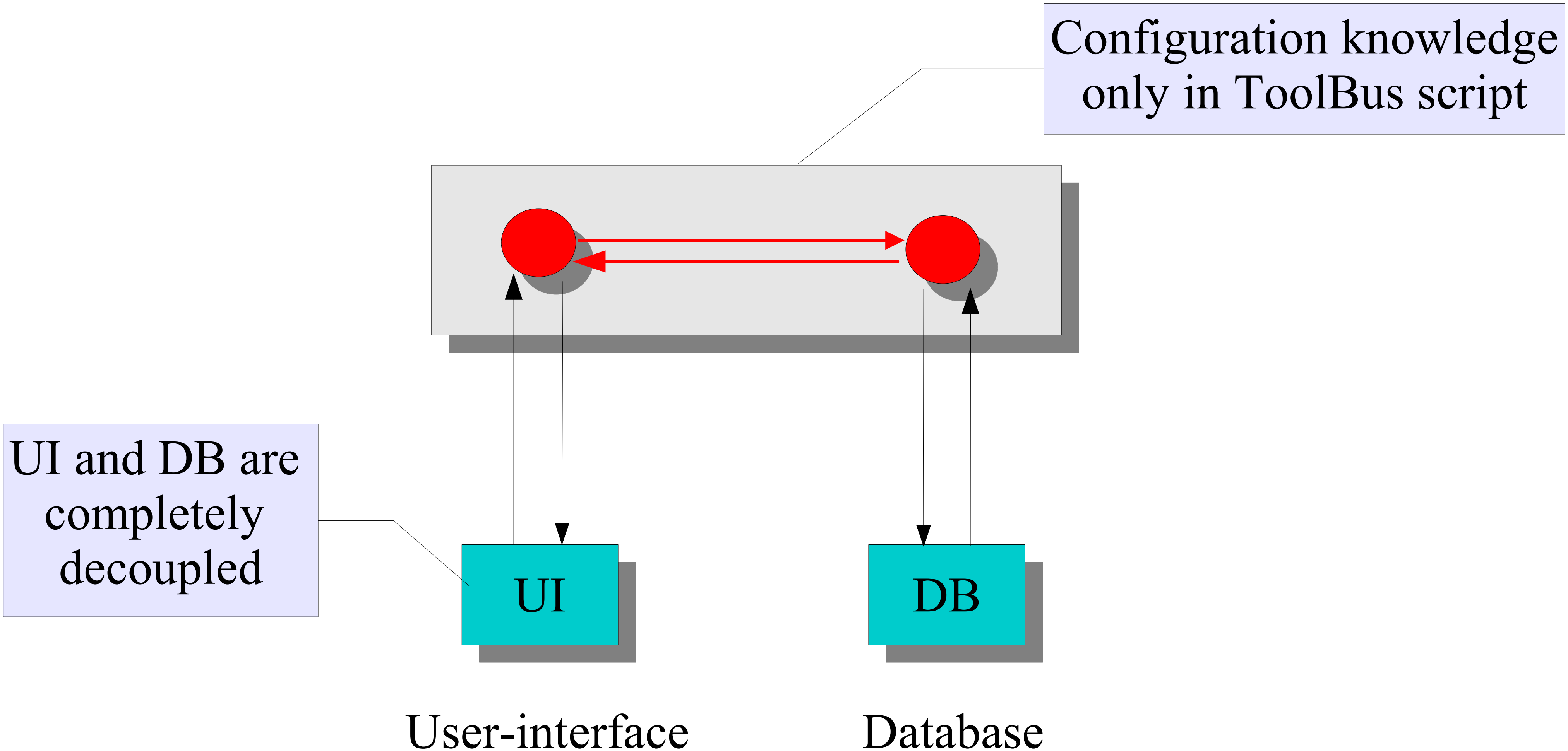
The ToolBus Architecture (1)

- Goals: integrate tools written in different languages running on different machines
- A programmable software bus
- Scripts describe the cooperation of tools
- Scripts are based on Process Algebra

The ToolBus Architecture (2)



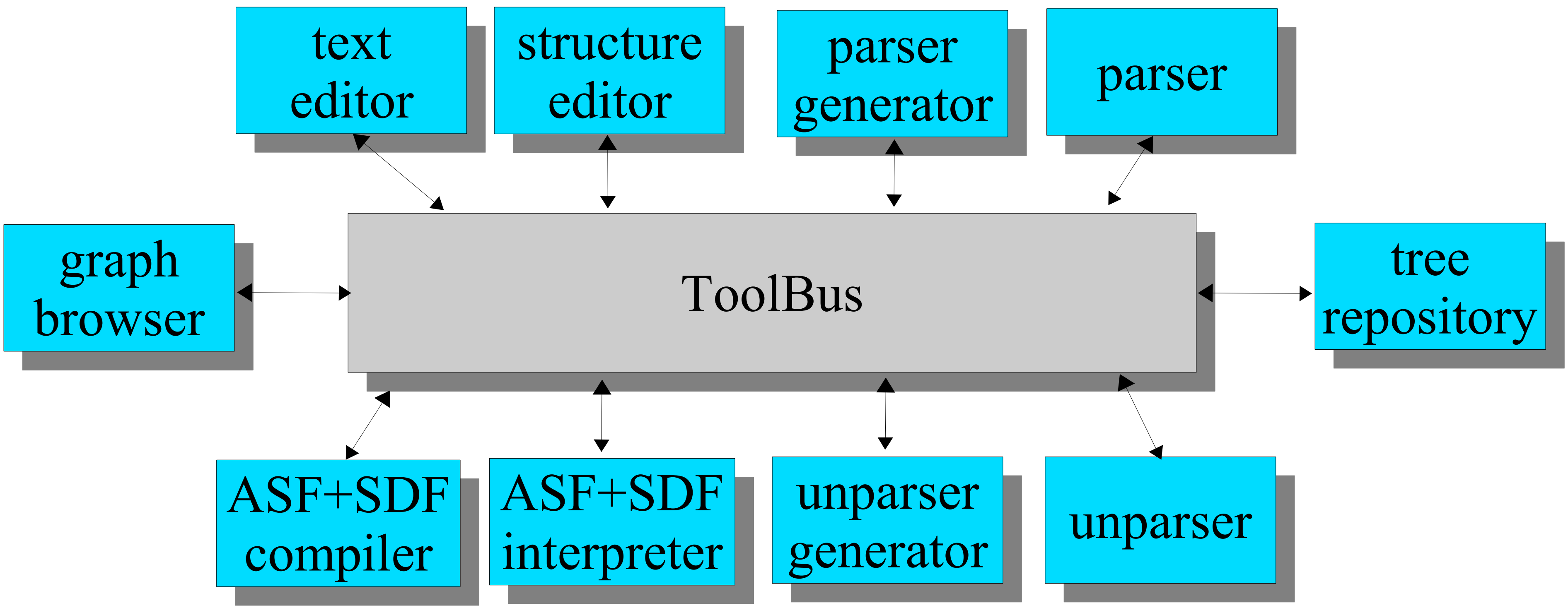
A typical scenario



ToolBus scripts

- Send, receive message (handshaking)
- Send/receive notes (broadcasting)
- $P_1 + P_2$ $P_1 . P_2$ $P_1 || P_2$ $P_1 * P_2$
- :=, if then else
- Absolute/relative delay, timeout
- Dynamic process creation
- Execution, connection & termination of tools

Architecture of the ASF+SDF MetaEnvironment



What Technology is used for GLT?

- ToolBus: a software coordination architecture used for connecting tools
- ATerms: Annotated Terms used to exchange data between tools
- SGLR: Scannerless Generalized LR parsing
- Conditional term rewriting and efficient compilation techniques

Scannerless Generalized LR Parsing (1)

- Scannerless: in a traditional compiler lexical syntax is implemented by a scanner and context-free syntax by a parser. SGLR: scanner and parser are integrated
  - makes resulting parser more expressive
  - simplifies the implementation

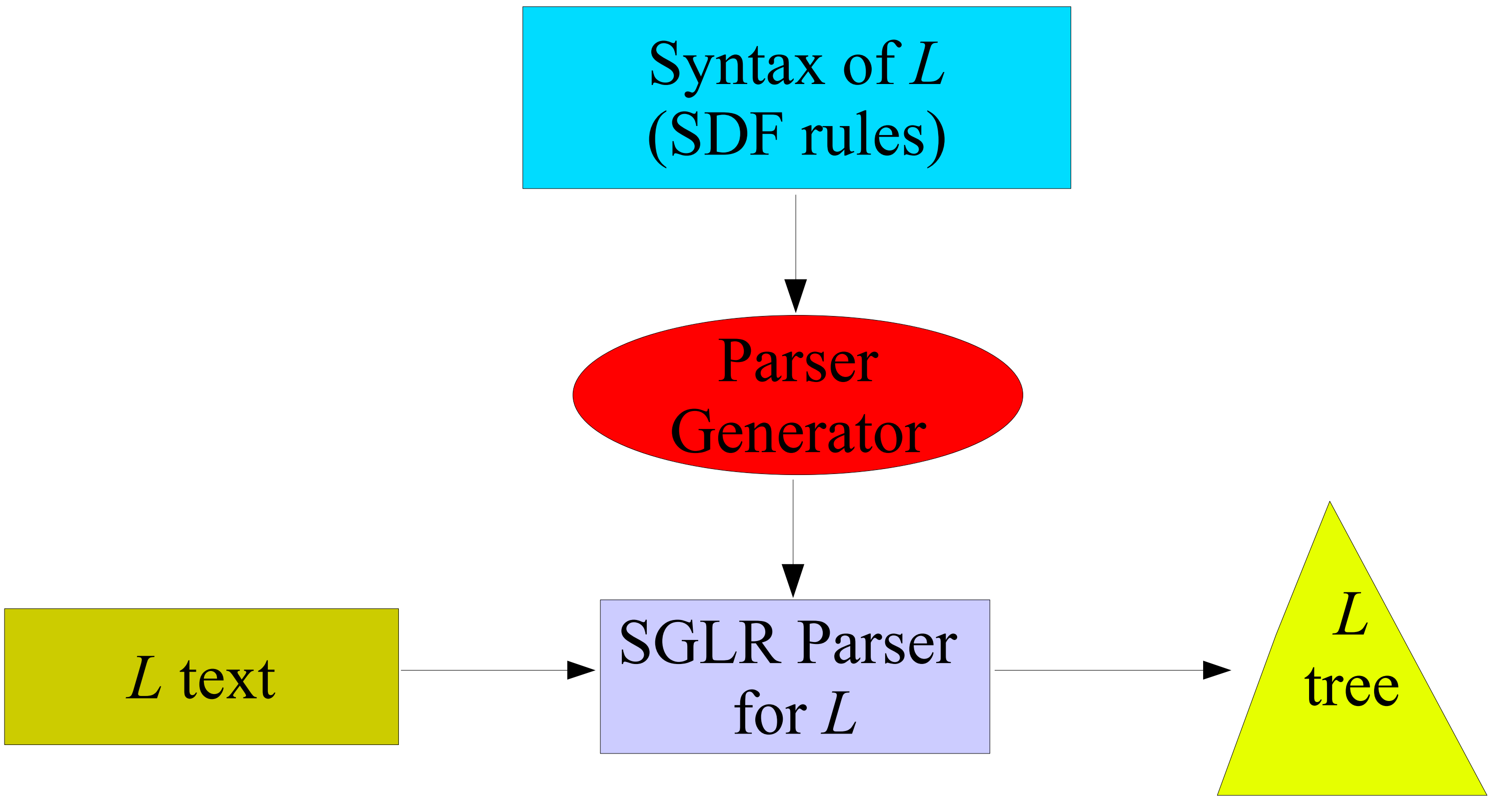
Scannerless Generalized LR Parsing (2)

- LR: left-to-right (bottom-up) parsing as used by Yacc and Bison.
- Generalized: extends the class of accepted grammars to all context-free grammars
  - Context-free grammars are closed under composition (as opposed to, e.g., LR grammars)
  - Enables modular grammars
  - Important for large grammars and language dialects

Scannerless Generalized LR Parsing (3)

- An LR-based parser generator does not allow conflicts: (shift/reduce, reduce/reduce)
- Key ideas in SGLR:
  - split a concurrent parse when a conflict occurs
  - merge concurrent parses as soon as possible
  - an ambiguity node represents alternative parses
- It is undecidable whether a context-free grammar is ambiguous, but heuristics might help.

Parsing Architecture



What Technology is used for GLT?

- ToolBus: a software coordination architecture used for connecting tools
- ATerms: Annotated Terms used to exchange data between tools
- SGLR: Scannerless Generalized LR parsing
- Conditional term rewriting and efficient compilation techniques

Conditional Term Rewriting

- Collect rules with same outermost symbol and generate one C function for them
- Generate a finite automaton for the matching of left-hand sides
- Use ATerms to represent terms:
  - maximal subterm sharing
  - structural equality can be implemented by pointer comparison!

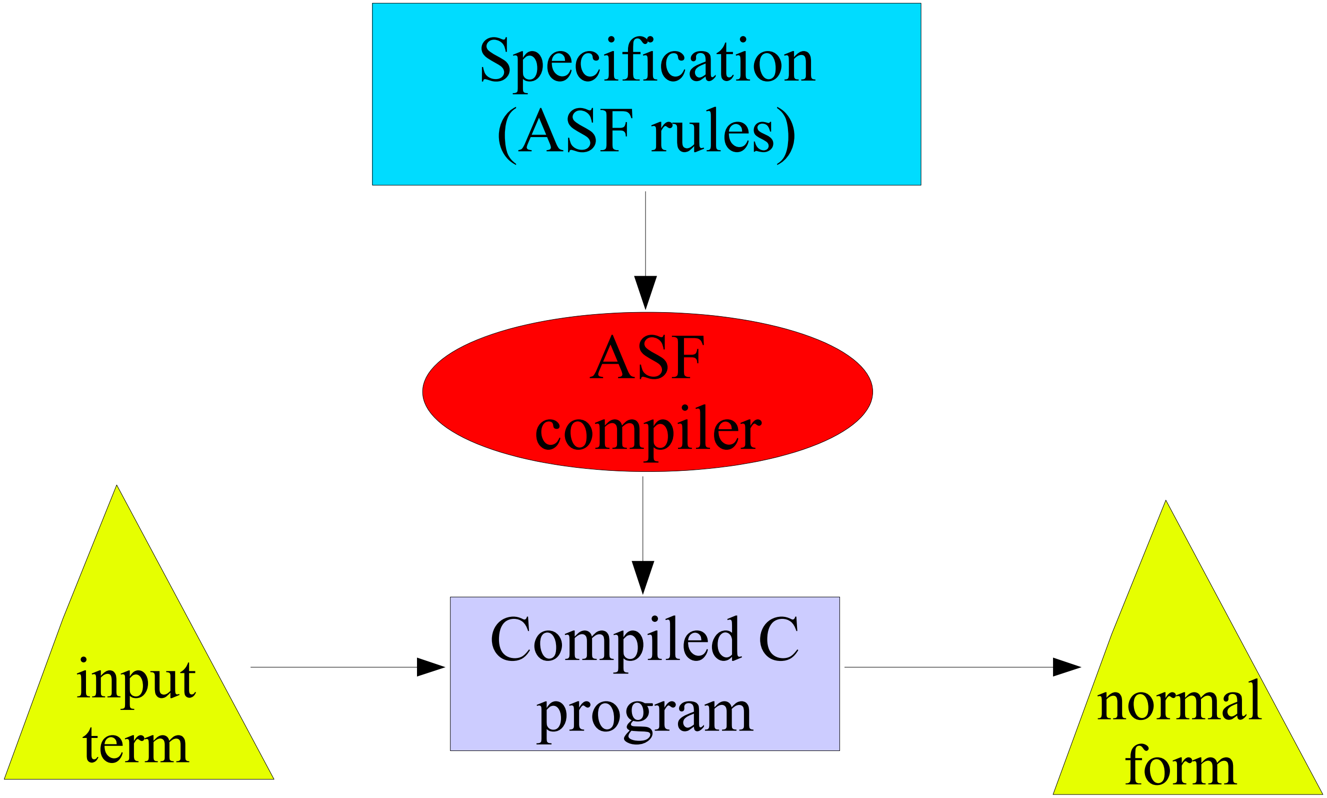
Compilation to C

```
f(a,b,c) = g(a)
f(X,b,d) = g(X)
```

Compiles to

```
ATerm f(ATerm arg0, ATerm arg1, ATerm arg2) {
  if term_equal(arg0,a) {
    if term_equal(arg1,b) {
      if term_equal(arg2,c) {
        return g(a);
      }
    }
  }
  if term_equal(arg1,b) {
    if term_equal(arg2,d) {
      return g(arg0);
    }
  }
  return make_nf3(fsym, arg0, arg1, arg2)
}
```

Rewriting Architecture



Effects of sharing

Application	Time (sec) sharing/no sharing	Memory (Mb) sharing/no sharing
ASF+SDF compiler	45/155	27/134
Java servlet generator	12/50	10/34
Typesetter	10/49	5/5
SDF normalizer	8/28	8/11
Pico interpreter	20/80	4/4

Wrap up

- Summary of GLT
- Current work on applications
- Current work on technology
- Further reading

Summary

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

Current work on Applications (1)

- Verification of JavaCard
- Detection and visualization of code smells in Java
- Transformation of formulae in Abramowitz and Stegun, Handbook of Mathematical Functions, from LaTeX to MathML and Mathematica
- Using relational calculus for software analysis
- Cobol transformations
- Design of DSL for ASML's chip manufacturing machines

Current work on Applications (2)

- ELAN Environment (Nancy)
- Action Semantics Environment (Aarhus)
- CHI environment (Eindhoven)
- C++ restructuring (Bell Labs)
- Connection with Eclipse

Current work on Technology

- Generic/generated IPE/GUI features
- Redesign/implementation of ToolBus
  - Reimplement in Java
  - Connections with RMI, Corba, .NET, Eclipse
- Grammar engineering
- Smoother coupling between term rewriting and relational calculus

Further reading (1)
technology

- J. Heering and P. Klint, Rewriting-Based Languages and Systems, Chapter 15 in Terese, Term Rewriting Systems, Cambridge University Press, 2003
- M.G.J. van den Brand, J. Heering, P. Klint and P.A. Olivier, Compiling language definitions: The ASF+SDF compiler. ACM Transactions on Programming Languages and Systems, 24 (4):334-368, July 2002
- M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser, The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. in: R. Wilhelm (ed). Proceedings of Compiler Construction (CC'01), LNCS 2027, 365--370, 2001.
- M.G.J. van den Brand, H.A. de Jong, P. Klint and P.A. Olivier, Efficient Annotated Terms, Software, Practice & Experience, 30(3):259--291, 2000
- J. A. Bergstra and P. Klint, The discrete time ToolBus -- a software coordination architecture, Science of Computer Programming 31(2-3):205-229, 1998

Further reading (2)
application areas

- A. van Deursen, P. Klint and J. Visser. Domain-Specific Languages: An Annotated Bibliography ACM SIGPLAN Notices 35(6):26-36, June 2000.
- M.G.J. van den Brand, P. Klint and C. Verhoef, Reverse Engineering and System Renovation: an Annotated Bibliography, ACM Software Engineering Notes, 22(1): 42--57, January 1997
- M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, E.A. van der Meulen, Industrial Applications of ASF+SDF, In M. Wirsing and M. Nivat (eds) Proceedings of Algebraic Methodology and Software Technology (AMAST'96), LNCS Vol. 1101, 9-18, 1996
- See: www.meta-environment.org
- See: Home pages of the authors