To parse or to marshall, that is the question

Jurgen Vinju, **Rodin Aarssen**, Tijs Van Der Storm

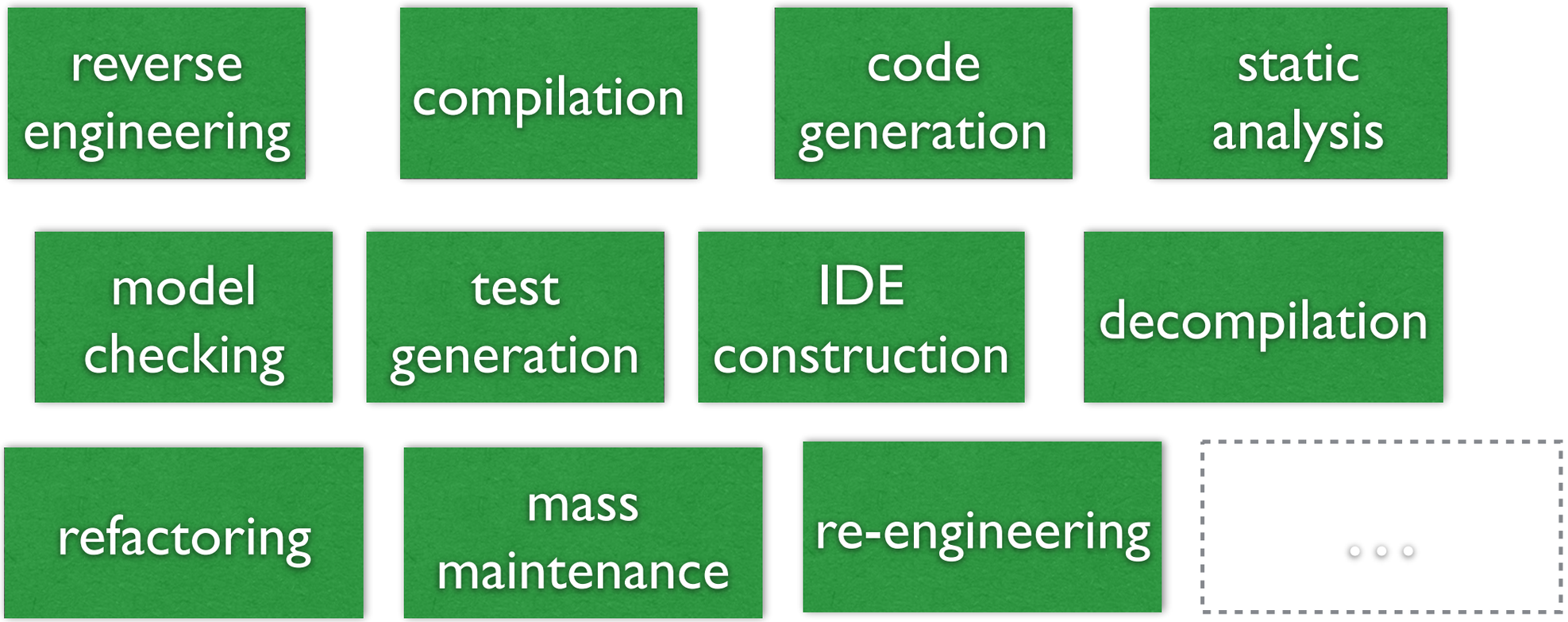
Centrum Wiskunde & Informatica
TU Eindhoven
Swat.engineering BV

What is the agenda?

- Introducing Rascal; *community*
- Reuse of *open compilers* for programming languages; *problems & solutions*, *exchange of thoughts*
- Job opportunities in research software and industrial language engineering; *opportunities*

Context: Rascal MPL

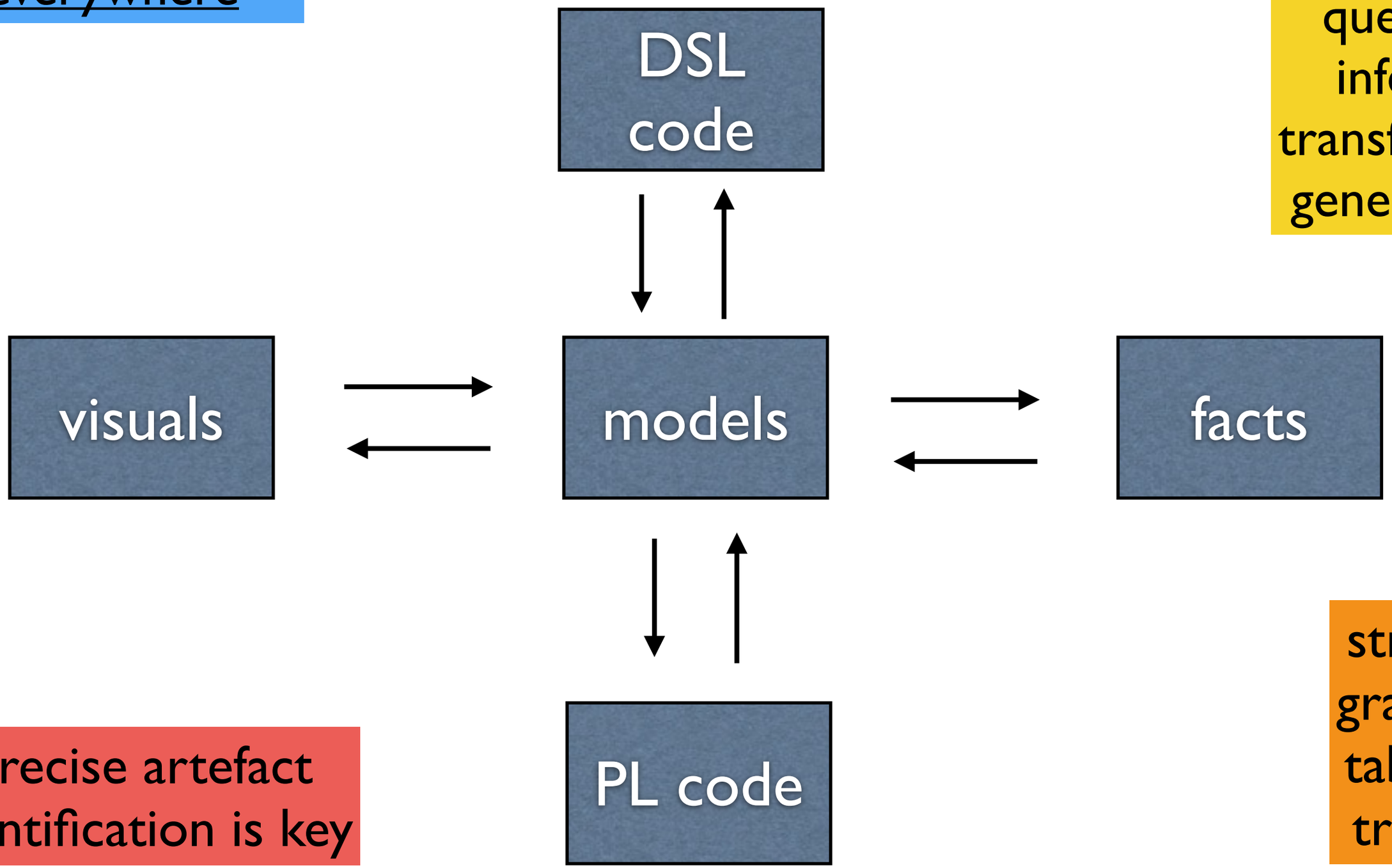
- **Integrated** Metaprogramming Language 2009
  - The successor of ASF+SDF *v1* 1983, 2002 *v2*
- Fuses *all* required features for "metaprogramming"



Rascal is for *everything* meta

diverse languages
<u>everywhere</u>

parse
extract
query
infer
transform
generate



precise artefact
identification is key

string
graphs
tables
trees

Integrated Meta Language

```
module Syntax

extend lang::std::Layout;
extend lang::std::Id;

start syntax Machine = machine: State+ states;
syntax State = @Foldable state: "state" Id name Trans* out;
syntax Trans = trans: Id event ":" Id to;
```

grammars for parsing and ASTs

```
module Analyze

import Syntax;

set[Id] unreachable(Machine m) {
  r = { <q1,q2> | (State)`state <Id q1> <Trans* ts>` <- m.states,
                                  (Trans)`<Id _>: <Id q2>` <- ts }+;
  qs = [ q.name | /State q := m ];
  return { q | q <- qs, q notin r[qs[0]] };
}
```

pattern matching
+ relational calculus
for query

```
module Compile

import Syntax;

str compile(Machine m) =
  "while (true) {
  '  event = input.next();
  '  switch (current) {
  '    <for (q <- m.states) {>
  '    case \"<q.name>\":
  '      <for (t <- q.out) {>
  '      if (event.equals(\"<t.event>\"))
  '        current = \"<t.to>\";
  '      <}>
  '      break;
  '    <}>
  '  }
  '}";
```

templates for
generation

There is a lot more
to discover :-)

open recursion=
no expression problem

URIs are
qualified names

Quickcheck is a language feature

Bridging s*cks

- Most meta-programming activities end up in bridging "stuff";
- **Trap 1**:"*eat your down dog food*", make a DSL for every aspect of meta-programming, then generate all the glue code in between => semantic integration nightmare.
- **Trap 2**:"*best tool for the job*", connect parser generators with database engines, database engines with graph visualizations, etc. => glue code grows out of hand
- Robust separation of concerns; but brittle re-integration.
- **Solution**: language-*integrated* meta-DSL/PL: Rascal.
- **Today**: we explore the **limits** of integration

Quick Demo

DSL with VScode IDE

Problem statement

- For reverse engineering, verification, re-engineering, refactoring, etc, …
- *Parsers* are the **key enabling component**, and then *Name resolution* and *(static) Type resolution* come quickly after.

- For actual programming languages (C++) **years of work**.
  - then there are dialects, versions, and customer extensions, …

- Creation of good (accurate & complete) parsers is **too expensive**

- Where to get **good/excellent** front-ends?

Open compilers to the rescue

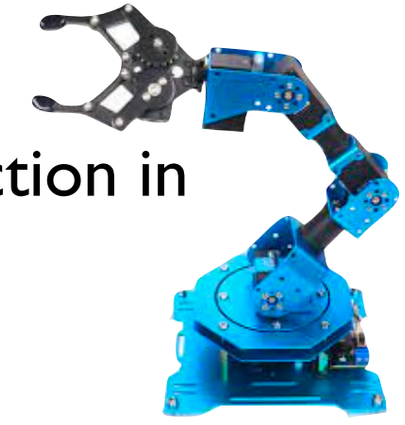
- Eclipse **JDT**: Java Development Toolkit
- **Clang** C/C++ & LLVM intermediate formats
- Eclipse **CDT**: C language Development Toolkit
- AdaCore's **libAdalang**
- **LFortran**: open FORTRAN compiler based on LLVM
- **libAST**: Python's own parser
- **Babylon**: multi-dialect JavaScript parser in Javascript
- owl **ASM**: JVM bytecode parser/generator
- … etc …

"Open" as in **extensible**, not as in open-source

Open Compilers are New

- Before, the **open-source GCC** compiler suite was *alone* & *closed*
- Good parsers were **golden assets** which were not shared easily.
- Parser experts were well-paid, *invisible*, engines of companies

- Eclipse was a huge positive force in open language engineering.
- Their Java compiler was “high fidelity” and complete, and incremental.



- Today we find **many open compilers**. An open compiler nurtures an ecosystem of people and projects around a programming language.
- **Clang** and the **LLVM** set a new standard in open compiler construction in terms of *stable* API, *fidelity*, *completeness*, and *accuracy*.

Open Compilers Gains

- **Complete**: entire language supported
- **Varied**: dealing with dialects and versions
- **Tested**: many users, many human testers
- **Maintained**: language evolution => compiler evolution
- **Supported**: communities on stackoverflow, slack, ..

Open Compilers: Pains

- **Not** automatically **integrated**
- **Lossy & noisy** : abstraction, desugaring, pre-processing, …
- **Ill-defined**:
  - stretched relation to concrete syntax
  - order of nodes undefined/ill-defined
- Complex or over-simplified **AST models**
- No concrete syntax?!?
- **Diverse** host languages and target binaries
- Every open compiler is a “technological space”
- **Is it not easier to just write a grammar?**

Bridged Compilers: Enabling

- "Java AIR" - Java Analysis in Rascal
  - dozens of masters thesis and Ph.D. thesis chapters
  - e.g. Distinguished Paper on Java Reflection at ICSE 2017 (Landman)
  - decades of Software Evolution courses, 1000's students

- "**ClAiR**" - C{++} Language analysis in Rascal
  - Large scale semi-automated migration of C++ test code (Philips Healthcare) [Schuts, Aarssen in SP&E]
- "Lua AIR", "**PHP AIR**", "Python AIR", "**Ada AIR**"
  - Security analysis, architecture conformance,
- But: **lots of work** and **no concrete syntax**..

- Does the **gain** of reusing an open compiler weigh against the **pain** of bridging them?
- That depends on who you are talking to…
- The **authors** of the bridge: mwah.
- The **users** of the bridge: yeah!

show Java, C++ and Ada AST models in VScode

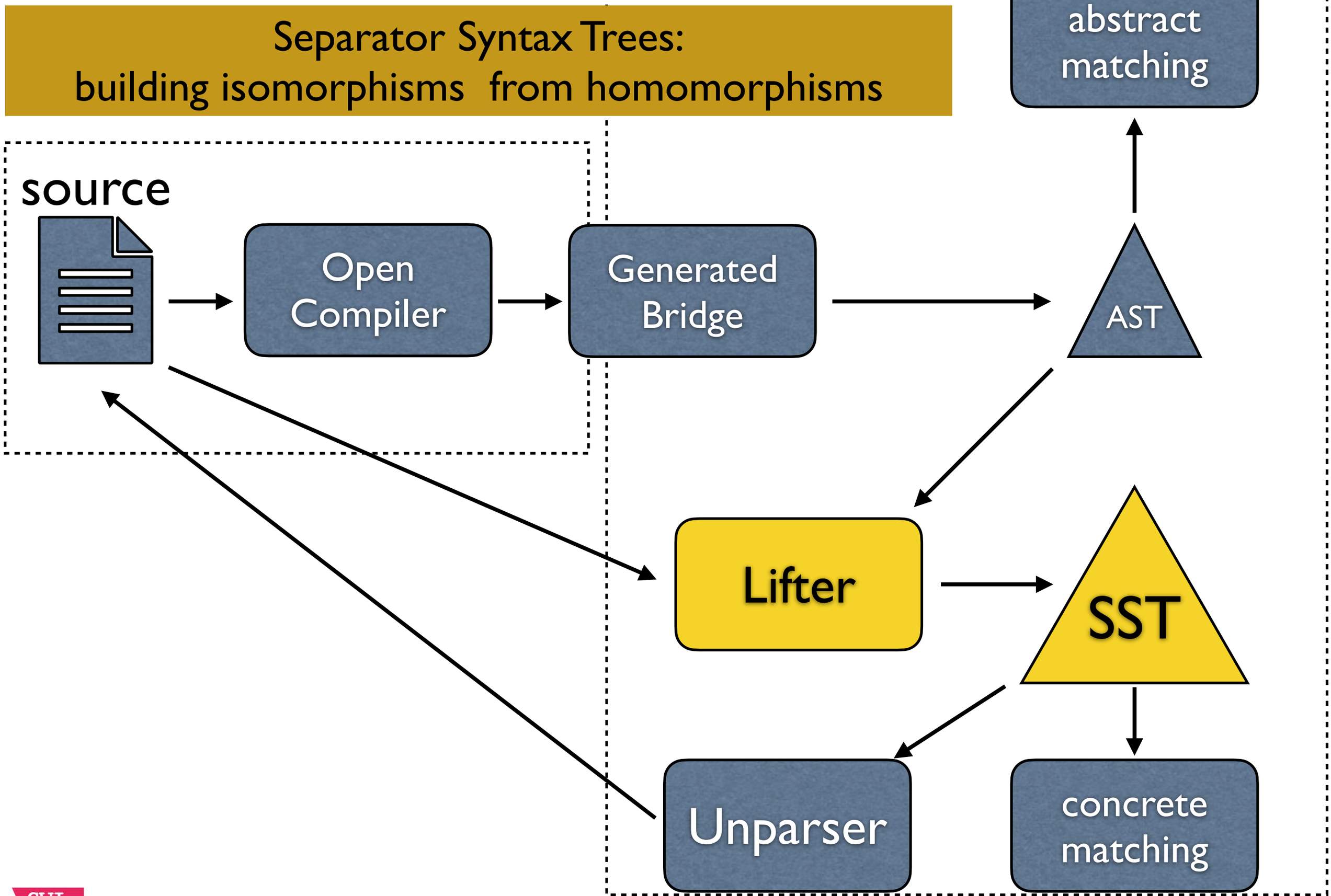
show Java, C++ and Ada AST mappers in VScode

Solutions

- PhD thesis of **Rodin Aarssen:**
  - Automatically deriving AST models and bridges from code
    - Requires a pre-existing Rascal front-end for the host language
    - Or, a reusable language workbench in the host language
  - Lifting ASTs to SSTs "separator syntax trees"
    - **concrete syntax matching and construction**
    - Requires some parser trickery (wrapping, unwrapping)
    - Most requested Rascal feature —*ever*— from all users.

(some other wiring details are elided)

Separator Syntax Trees: building isomorphisms from homomorphisms



Concrete Syntax Value

```
visit (sst) {
  case (Decl)`class <Name c> {
             '  <Decl* pre>
             '  public:
             '  <Decl* between>
             '  <Type t> <Name n>;
             '  <Decl* post>
             '};`
  ⇒ (Decl)`class <Name c> {
           '  <Decl* pre>
           '  public:
           '  <Decl* between>
           '  private:
           '  <Type t> <Name n>;
           '  public:
           '  void <Name setter>(<Type t> val) {
           '    <Name n> = val;
           '  }
           '  <Type t> <Name getter>() {
           '    return <Name n>;
           '  }
           '  <Decl* post>
           '};`
    when !hasPrivateOrProtected(between),
      str name := capitalize("<n>"),
      Name setter := [Name]"get<name>",
      Name getter := [Name]"set<name>"
}
```

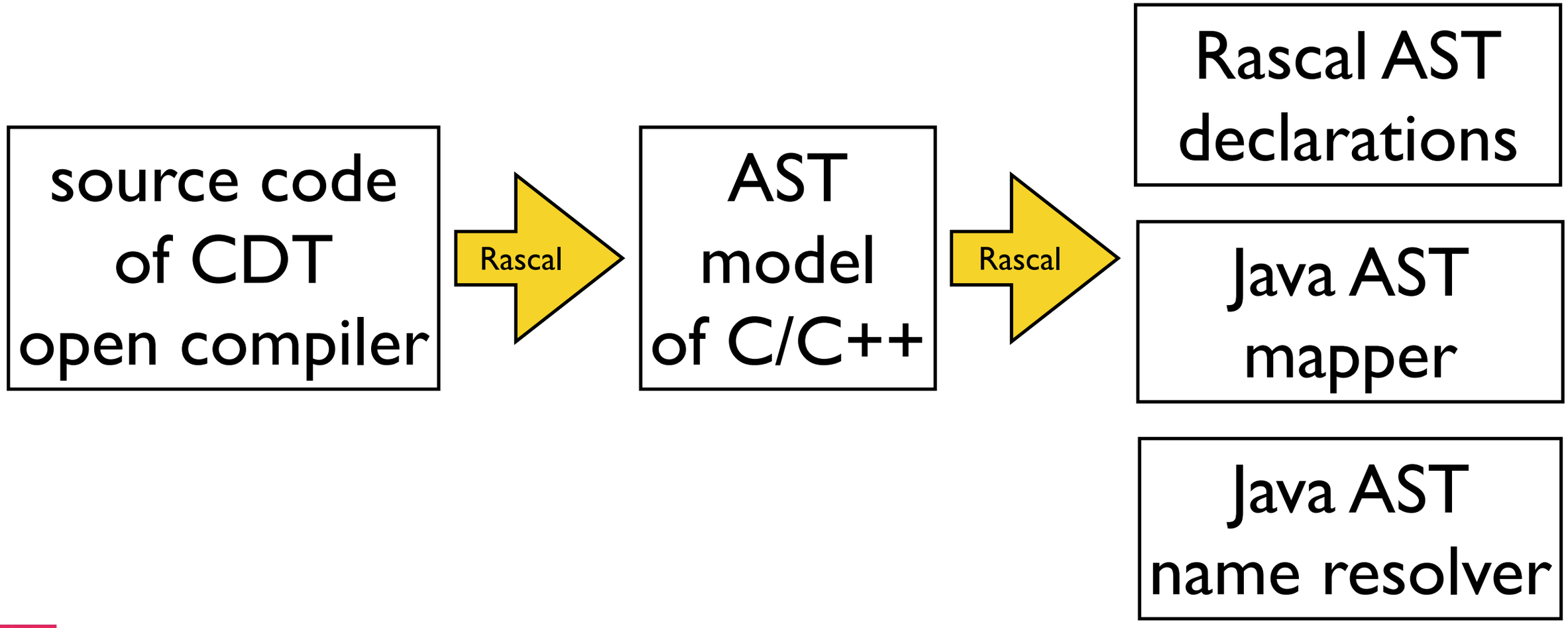
encapsulate field refactoring for C++

imagine this as two humongous prefix tree patterns. It would have filled several slides…

This one you can explain to a C++ programmer, the abstract one.. not.

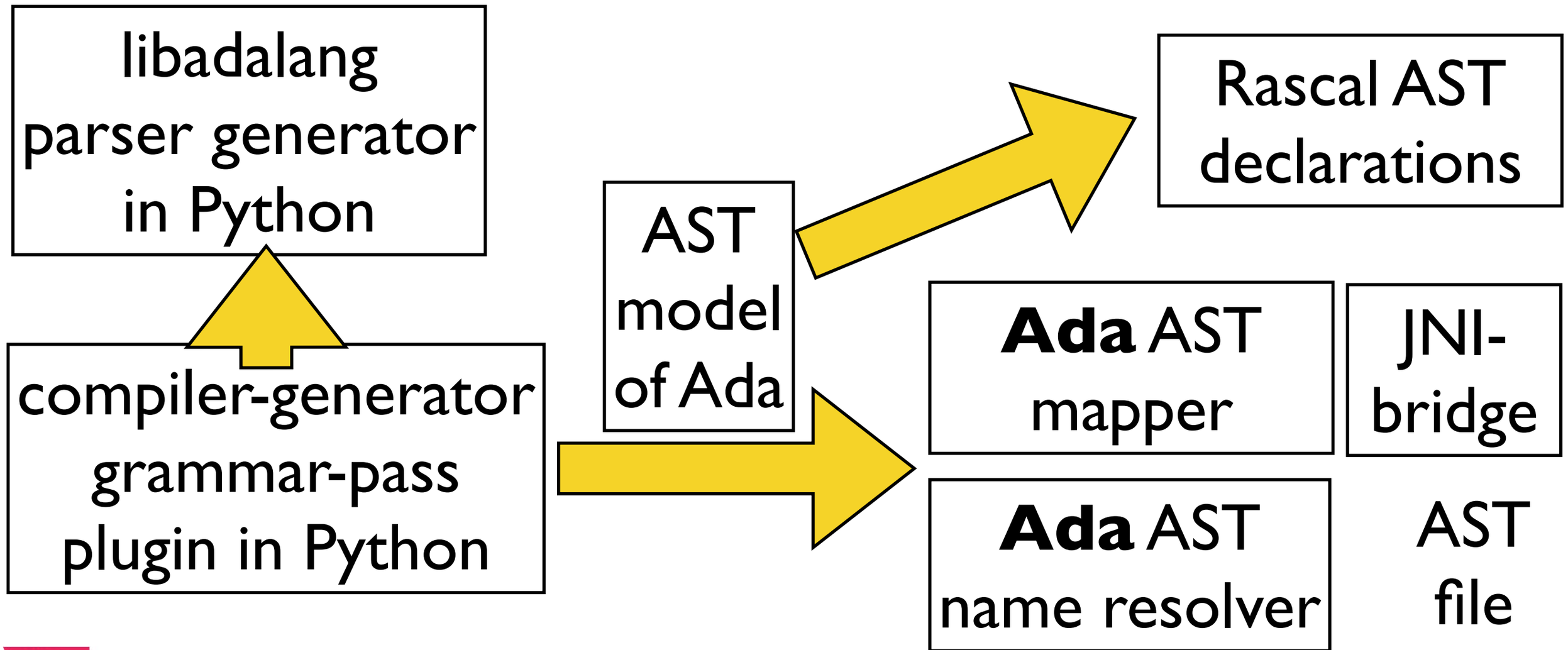
Back to JDT/Clair/Ada-air

- *Ashim Shahi, Bas Basten, YT, et al.* (CWI) wrote "JDT" library by hand
- Rodin Aarssen wrote & generated Clair (CWI, Swat.engineering)
- Damien DeCampos (Paris Saclay, Thales, TNO) wrote/generated Ada-air



Back to JDT/Clair/Ada-air

- *Ashim Shahi, Bas Basten, YT, et al.* (CWI) wrote "JDT" library by hand
- Rodin Aarssen wrote & generated Clair (CWI, Swat.engieering)
- Damien DeCampos (Paris Saclay, Thales, TNO) wrote/generated Ada-air



Bridge Generation

- **Code generation** can automate the mapping from one technological space (open compiler) to another (Rascal), but the generators can not be reused much.
- And you need testing testing testing testing testing, and did I mention testing?

AST specification

- What is a good AST (for Rascal)? How to test an AST mapper?
  - every node has source **location** information
  - **no gaps, no overlap** between siblings
  - parents locations (tightly) **wrap children** locations
    - siblings are ordered from **left to right**
    - all identifiers have resolved (**fully qualified**) URI names
- These are “one-liner” queries in Rascal code, and required for separator syntax trees (concrete syntax) to work well.
- Execute them on a large body (first the standard library!) of example code, open-source code. And then the **real work** begins.

And then? Semantic models!

- Semantic models are derived (*binary*) relations for PL
  - *call graph* **rel**[**loc** caller, **loc** callee]
  - *use-def* **rel**[**loc** use, **loc** def]
  - *scopes* **rel**[**loc** outer, **loc** inner]
  - etc.
- Inspired by UML, FAMIX, KDM, RSF, URL's and RDF
- Java M3 used a lot, others under development
- *Compositional* and language *agnostic*:
  - cross-language, cross-architecture
- Constructed from AST traversal, fact extraction.

Conclusion

- Rascal is an easy general meta-programming language based on grammars and relational calculus and (concrete) string templates, functional/ structured programming, with open recursion.

- Open compilers are **golden**; bridging them is hard work
  - Generating the bridge is necessary, but *bespoke* for every compiler

- Concrete syntax with SST **tips the pain/gain balance.**
  - Concrete syntax is not only for concrete grammars anymore.
  - Only write grammars for PL if there is no open-compiler.
  - Still, always write grammars for DSLs (evolvability)

Caveat: SSTs are not in the `main` branch yet of Rascal.

Community

- **GitHub** http://github.com/usethesource
- **Master** courses at Evolution, Compilers: UvA, TUE, RUG, Bergen, OU, …
- **Industrial** users in software eng, fin-tech, high-tech, government, …
- **Spin-off** Swat.engineering BV: *DSL/PL* {re}{verse}engineering
- *User interfaces*:
  - **VScode IDE**, Language Server Protocol (extension generators)
  - **Jupyter(Lab)** notebooks (kernel generators)
  - **Eclipse IDE** Plugin (plugin generators)
  - **Commandline** read-eval-print loop
  - **HTML5**, elm-like interactions
  - **Maven**, MOJO's for compilation, test running, consol

swat. engineering
control your software

Jobs!

CWI

- Swat.engineering BV, Amsterdam/Almere) Davy Landman
  - **language engineers, sr/jr**
    - "DSLDI" with **Rascal**, VScode
    - Reverse engineering
  - Excellent rewards
- CWI SWAT group (*pending funding proposal*) RASCAL-LAB
  - 5 language engineers in 2023-2028, Rascal, Java, /everything/
  - 500+ components for **empirical software engineering**
  - (inter)national collaboration network