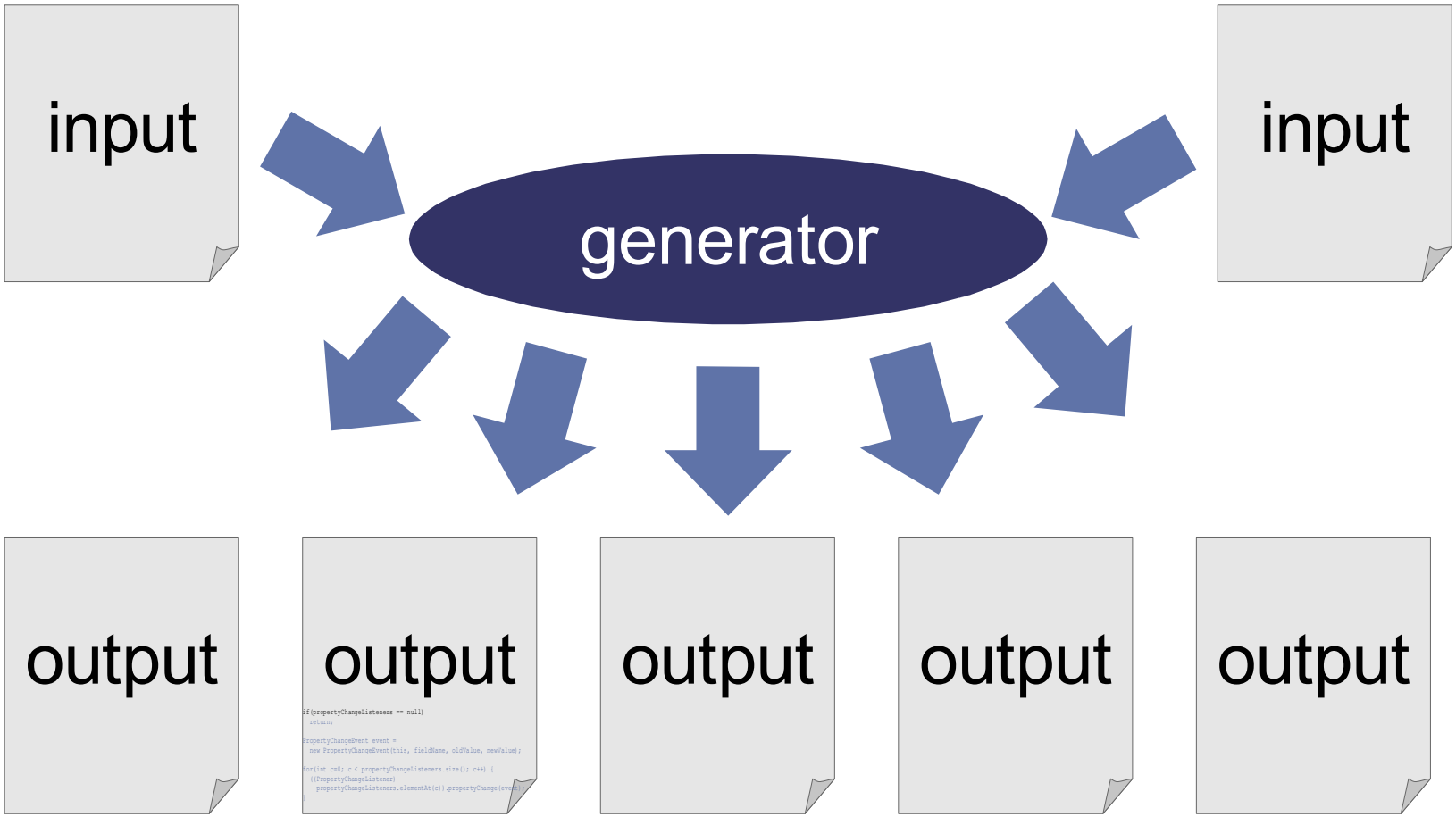
Towards Intelligent Coding Assistance for Code Generation in Modeling Frameworks

Martin Bravenboer

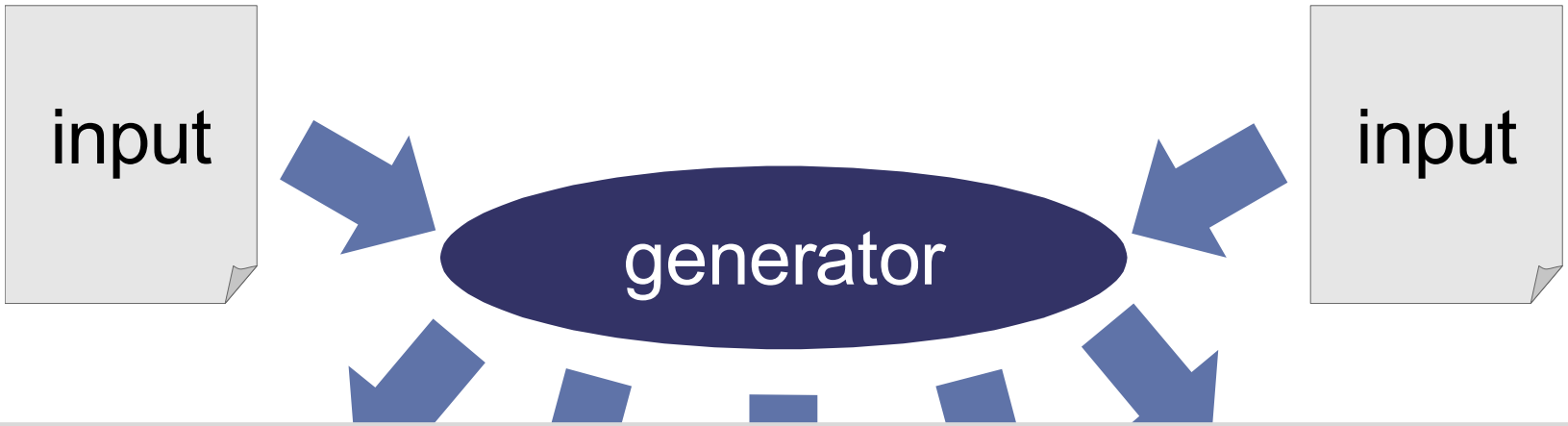
Software Engineering Research Group
Department of Software Technology
Delft University of Technology
The Netherlands

November 1, 2007

Meta Programming: Implementing Code Generators

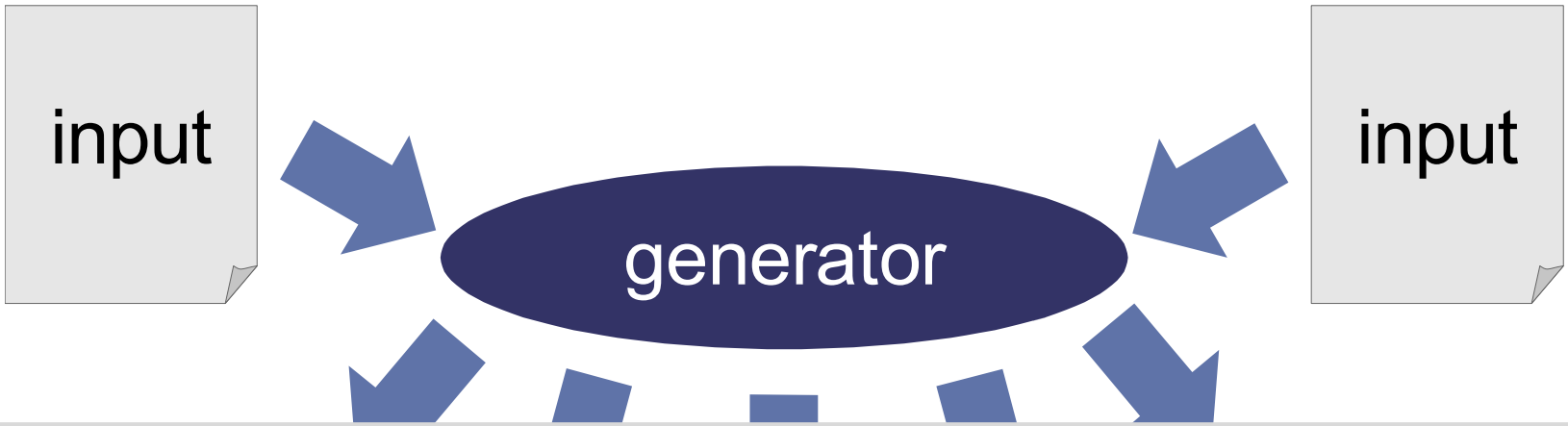


Meta Programming: Implementing Code Generators 1




```
if(propertyChangeListeners == null)
  return;

PropertyChangeEvent event =
  new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
  ((PropertyChangeListener)
      propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

Meta Programming: Implementing Code Generators 1




```
if(propertyChangeListeners == null)
  return;

PropertyChangeEvent event =
  new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
  ((PropertyChangeListener)
      propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

The Topic for Today

Writing code generation templates

```
<<FOREACH ... AS class>>
<<FILE "samples/petclinic/" + class.name + ".java">>
package samples.petclinic;

publc class <<class.name>> {
  private long id;

  public void setId(long id) {
    this.id = id;
  }

  public long getId() {
    return id;
  }
}
<<ENDFILE>>
<<ENDFOREACH>>
```

Part I

Problems with Code Generation Tools

Problem: Syntax Errors in Templates


```
<<FOREACH ... AS class>>
<<FILE "samples/petclinic/" + class.name + ".java">>
package samples.petclinic;

publc class <<class.name>> {
  private long id;

  public void setId(long id) {
    this.id = id;
  }

  public long getId() {
    return id;
  }
}
<<ENDFILE>>
<<ENDFOREACH>>
```

Problem: Syntax Errors in Templates

Why not just test the generator with an example input?

Errors can be input specific:

```
<<IF opposite != null && opposite.isMultivalued()>>
  <many-to-many
     column="<<columnName()>>"
     class="<<type.fqn()>>"
     unique="<<isUnique()>>"
     <<IF unique_Key() != null>>
       unique-key="<<unique_Key()>>"
     <<ENDIF>>/>
<<ELSE>>
  <one-to-many class="<<type.fqn()>>"/>
<<ENDIF>>
```

. . . and of course we don't have code coverage tools.

Problem: Semantic Errors in Templates

Not only syntactic problems: type errors

```
public Collection getChildren(String linkId) {
   Collection returnChildren = new ArrayList();

   <xsl:call-template name="children">
        <xsl:with-param name="lid" select="$lid"/>
        <xsl:with-param name="superTypeLid" select="$superTypeLid"/>
        <xsl:with-param name="classPreFix" select="$classPreFix"/>
        <xsl:with-param name="first" select="true()"/>
        <xsl:with-param name="output">genericGetChildren</xsl:with-param>
   </xsl:call-template>

   return returnChildren;
}
```

```
<xsl:when test="$output = 'genericGetChildren'">
if (linkId.equals(LID_CHILD_LINK_<.../>)) {
  returnChildren = this.getChildren_<.../>();
}
</xsl:when>
```

Problem: Semantic Errors in Templates


```
<xsl:for-each select="//association[
      @target-lid = $lid and @resource-link = 'false'
      and ../@instantiable = 'true']">
if (this.getParent_<xsl:value-of
                 select="translate(../@name, '-', '_')"/>() != null) {
  parent = this.getParent_<xsl:value-of
                              select="translate(../@name, '-', '_')"/>();
}
</xsl:for-each>
```

```
<xsl:for-each select="//association[
      @target-lid = $lid and @resource-link = 'false'
      and ../@instantiable = 'true']">
public <xsl:value-of select="$classPreFix"/><xsl:value-of
                              select="translate(../@name, '-', '_')"/>
  getParent_<xsl:value-of select="translate(../@name, '-', '_')"/>() {
    return this.parent_<xsl:value-of
                              select="translate(../@name, '-', '_')"/>;
}
</xsl:for-each>
```

Problem: No Syntax Highlighting


```
«FOREACH eAllContents.typeSelect(uml::Class).select(e|e.isAbstract==true) AS class»
«FILE "org/springframework/samples/petclinic/" + class.name + ".java"»
package org.springframework.samples.petclinic;

public class «class.name» {

    private long id;

    public void setId(long id) {
        this.id = id;
    }

    public long getId() {
        return id;
    }
}
«ENDFILE»
«ENDFOREACH»
```

Problem: No Intelligent Error Highlighting


```
public rw_PakketResponse getElementrw_Pakket(..., LoggingToken log) {
  logger.debug("WS getAPPParent");

  rw_PakketParentResponse response = new rw_PakketParentResponse();

  if ((security = null) || (log = null)) {
    response.addFout(ServiceFout.SOF_DEFAULT_FOUT);
    return response;
  }

  // Log de aanroep bij de Sofia Management Info Service
  LoggingContext ctx =
    LoggingContext.newInstance(log.getSessieId(), log.getRequestId());

  ContractServiceBSImpl service = new ContractServiceBSImpl();
  response.setArrayrw_PakketOutputDTO(elementrw_PakketOutputDTOs);

  logger.debug("WS end getAPPParent");
  return response;
}
```

Lack of static guarantees

- no guarantee the generated program compiles
- compiler will report problem in terms of generated code

Limited editing support for metaprogrammer

- syntax highlighting
- coding style / formatter
- code completion
- code folding
- error reporting
- debugging

Wait, isn't that obvious?

Solution: object language aware template compiler

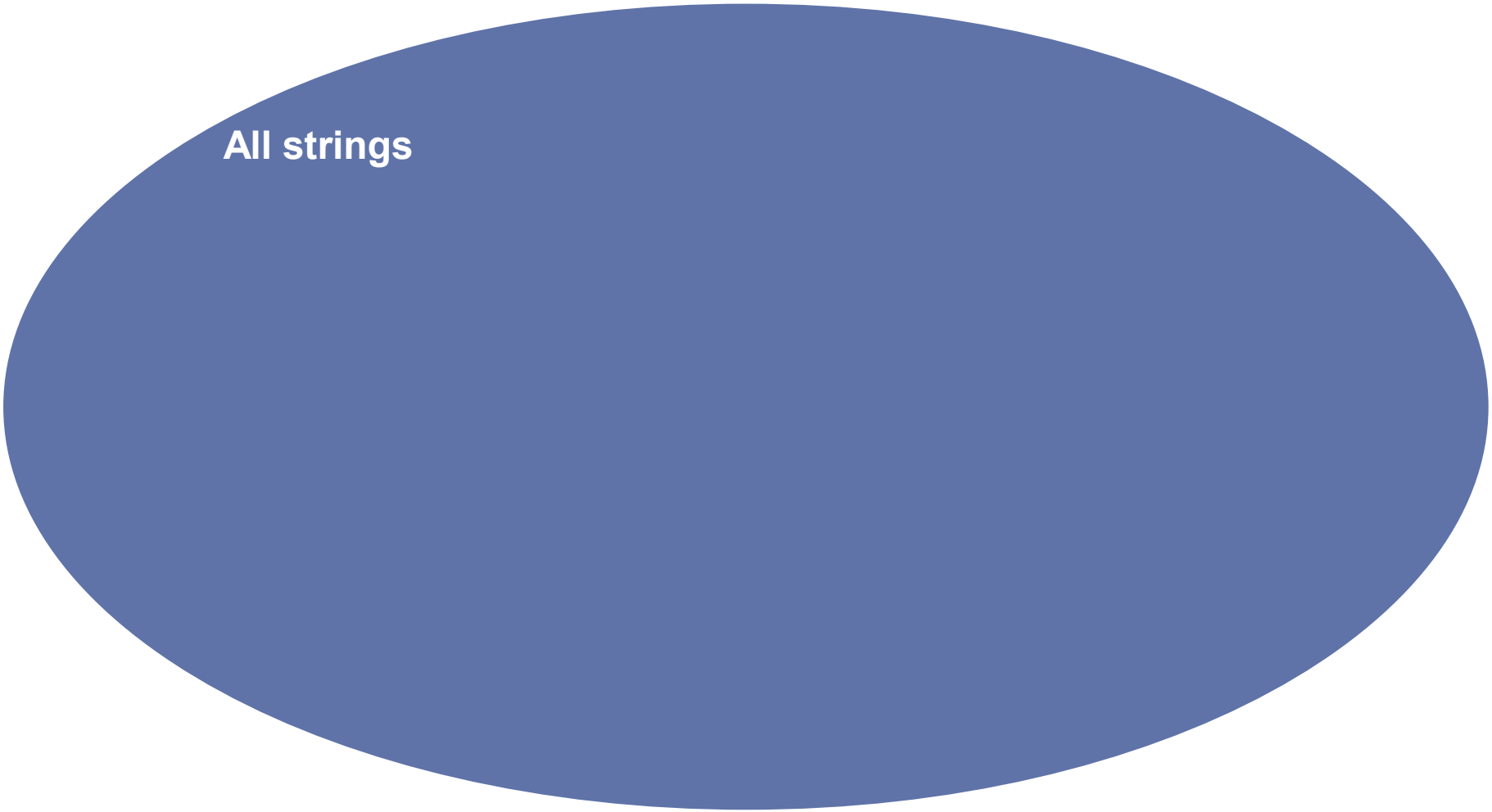
yeah, but . . .

- difficult to realize: current IDEs are mostly realized by brute force manpower
- no spectacular advancement in understanding of how to implement interactive source editors
- move to parsing source code in editor

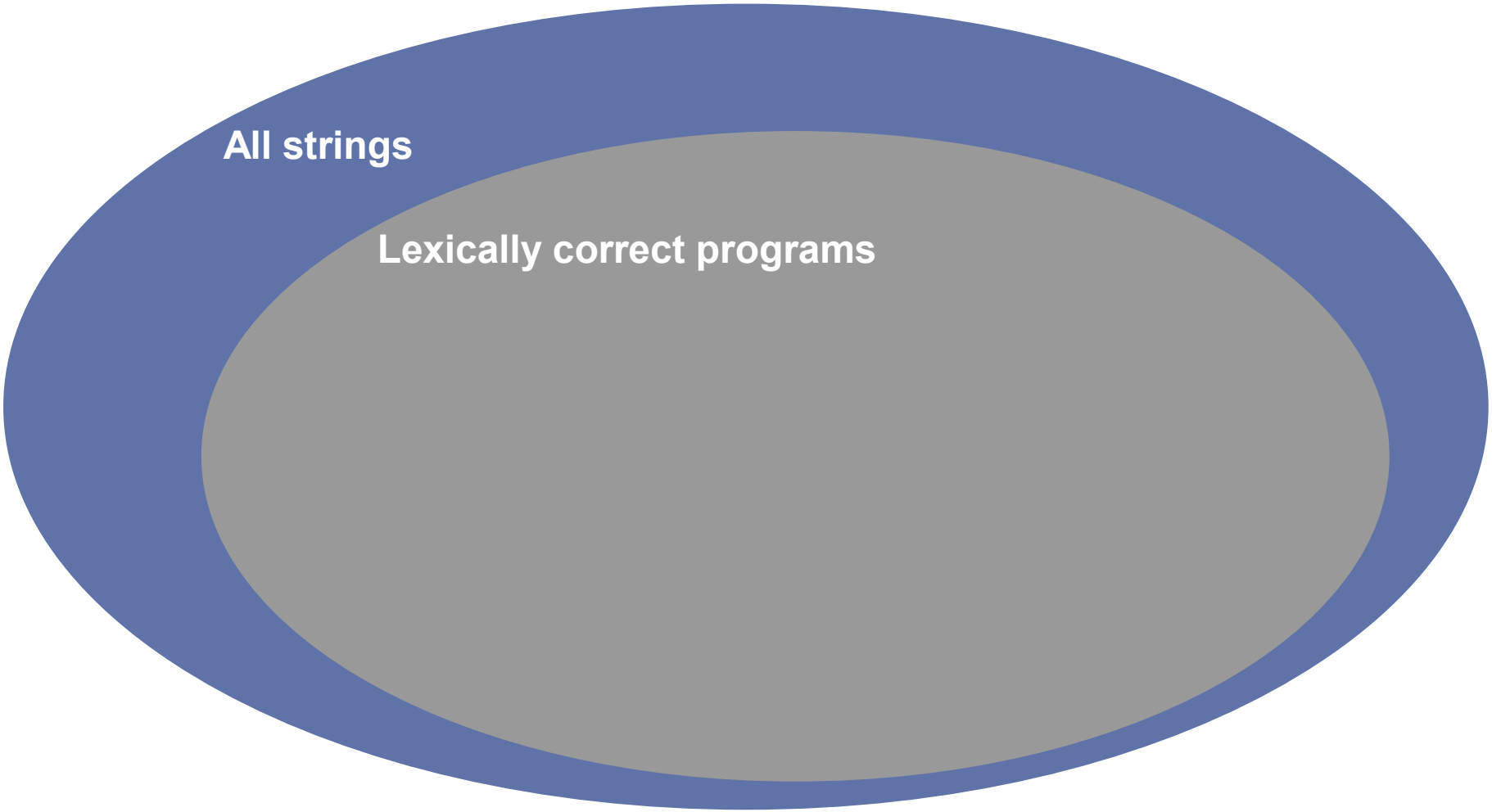
Part II

Overview of Related Work in Research

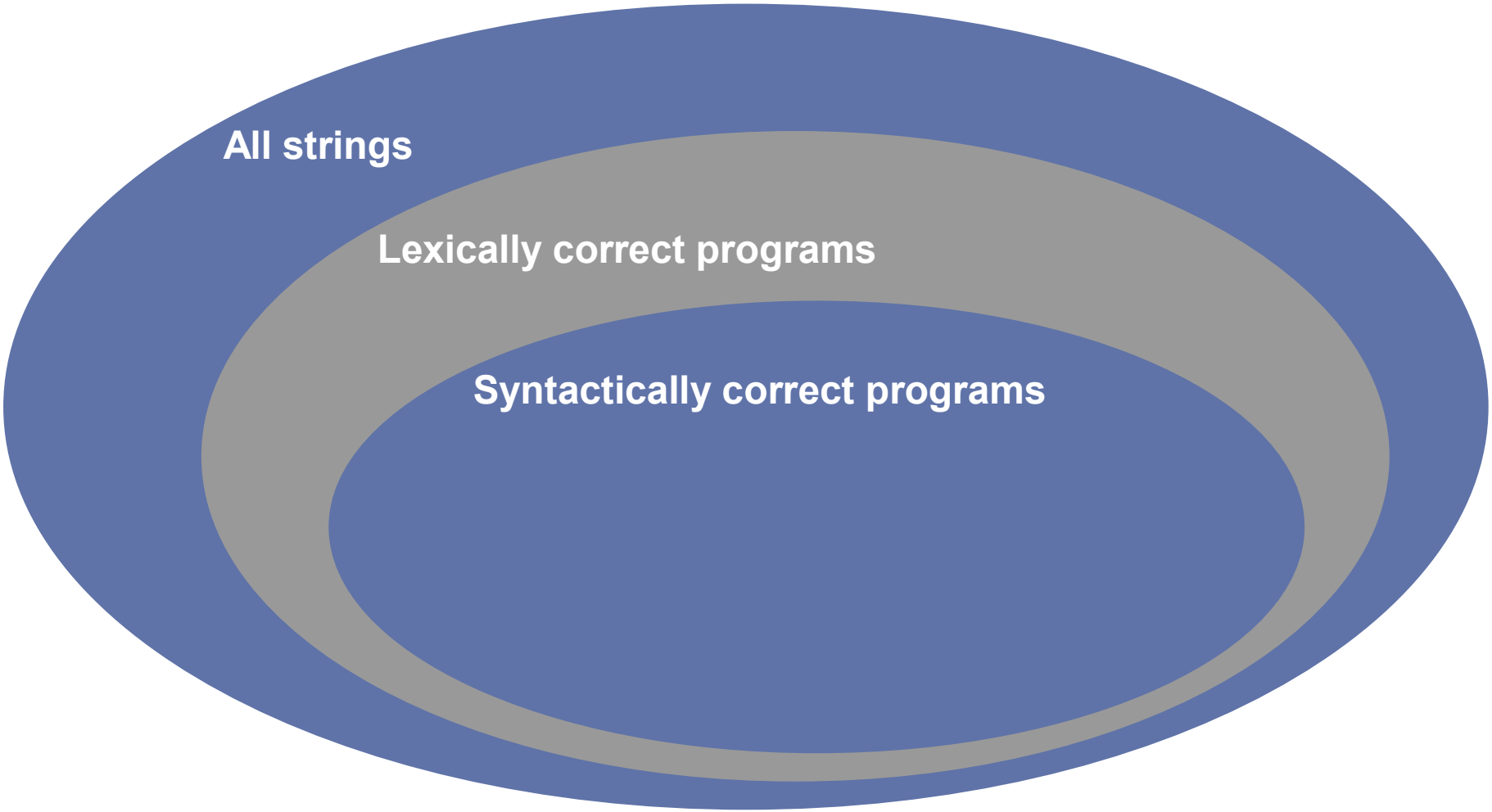
Problem: Programming Errors in Templates



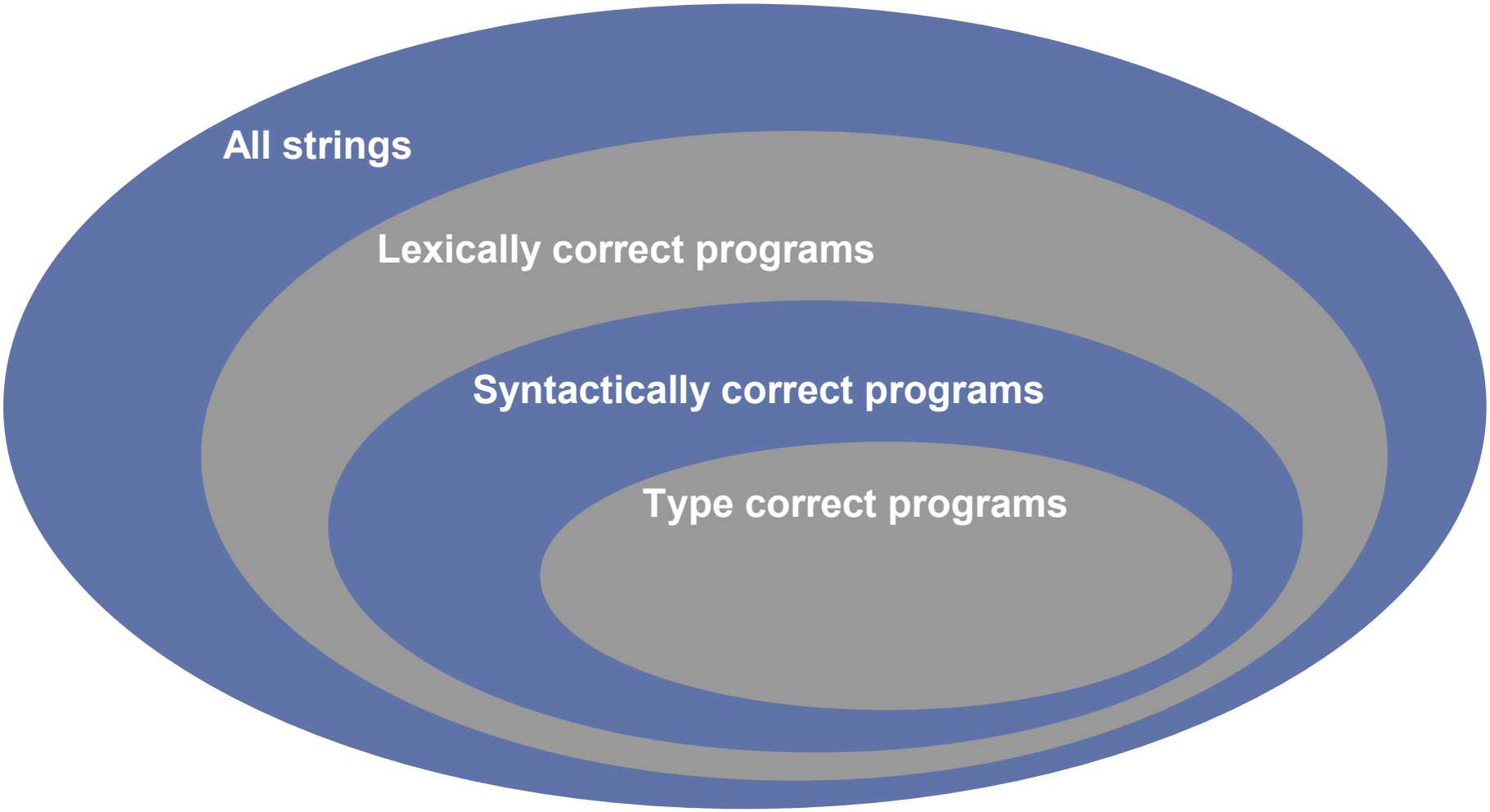
Problem: Programming Errors in Templates



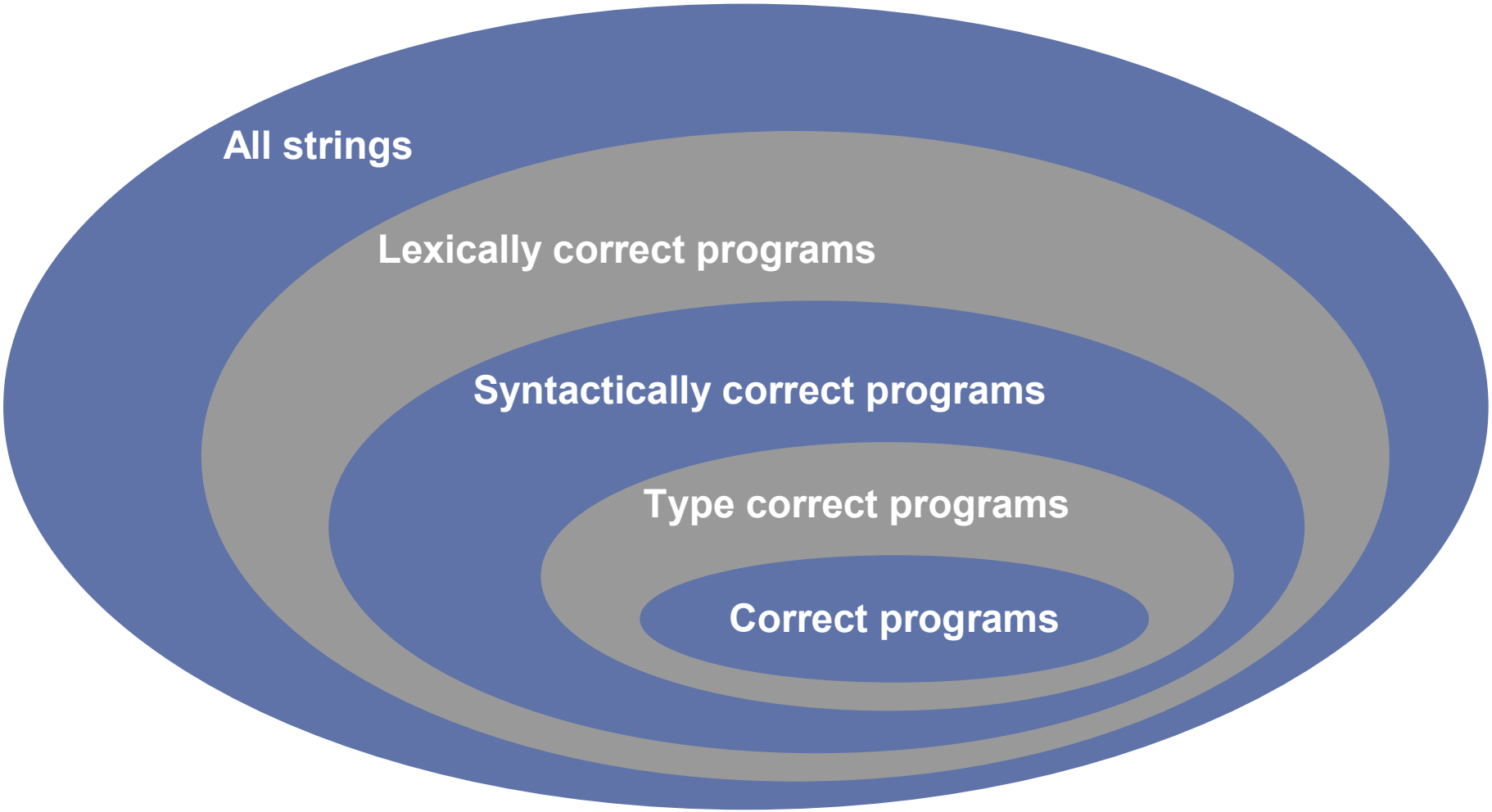
Problem: Programming Errors in Templates



Problem: Programming Errors in Templates



Problem: Programming Errors in Templates



Parsing of code templates

- for static guarantees
- to avoid tricky code generation errors
- issues: parsing combinations of languages

Parsing of code templates

- for static guarantees
- to avoid tricky code generation errors
- issues: parsing combinations of languages

Interactively parsing code templates

- for interactive coding assistance
- isssues: performance, error reporting and recovery

Parsing of code templates

- for static guarantees
- to avoid tricky code generation errors
- issues: parsing combinations of languages

Interactively parsing code templates

- for interactive coding assistance
- isssues: performance, error reporting and recovery

Semantic analysis of code templates

- for coding assistance
- for static guarantees
- issues: insufficient context information, false positives

Syntax-Safe Program Generation

- Concrete syntax
  - ASF+SDF MetaEnvironment, Stratego/XT, Repleo
  - DMS, JTS, Meta-AspectJ
- Abstract syntax

Statically Safe Program Generation

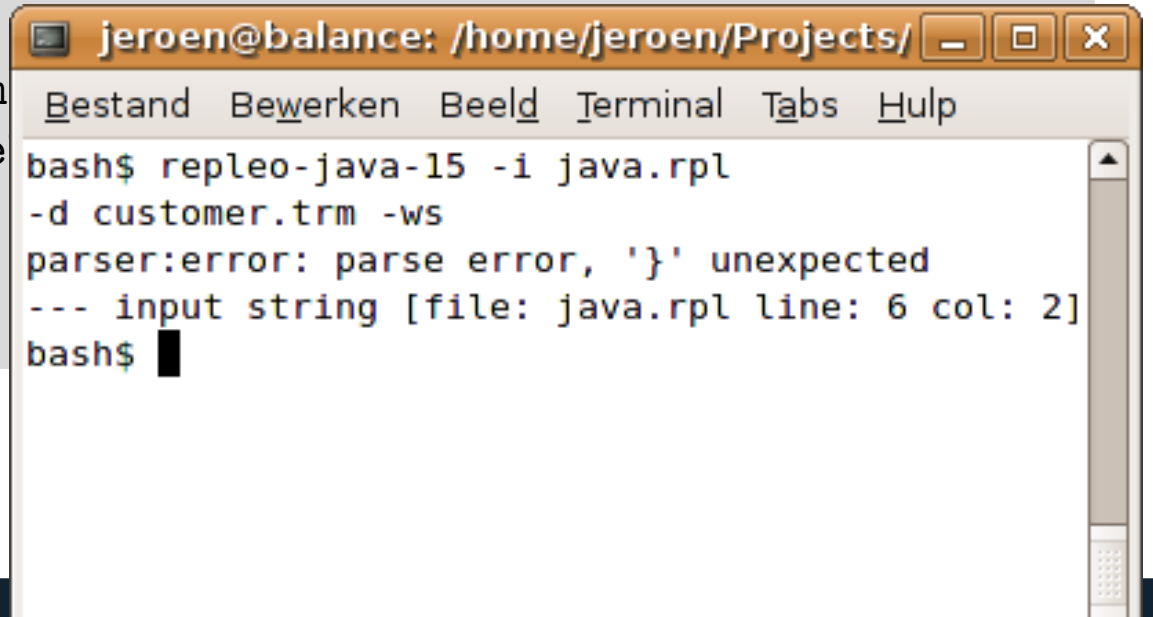
- SafeGen, MJ
- Generalized Algebraic Data Types (GADT)

IDE Plugin Development Tools

- Eclipse IDE Meta-tooling Platform (aka Safari)
- Project Schliemann (NetBeans)

Repleo template engine

```
public class <%class/name%> {
  <%foreach class/attribute do%>
  private <%type%>  <%name%>;
  public <%type%> <%get||name%>(){
    return <%name%>
  }

  public void <%set||nam
    this.<%name%>=<%name
  }
  <%od%>
}
```



SafeGen

```
#defgen makeDelegator ( input(Class c) => !Abstract(c) )
{
  #foreach( Class c : input(c) )
  {
    public class Delegator extends #[c]
    {
      #foreach(Method m : MethodOf(m, c) & !Private(m))
      {
        #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] )
        {
          return super.#[m](#[m.ArgNames]);
        }
      }
    }
  }
}
```

Part III

Metaprogramming with Concrete Object Syntax

Metaprogramming with Concrete Object Syntax 16

- *Meta*programming:
  - Generating, transforming, or analyzing *object* programs
- Metaprograms should use concrete syntax of object language
  - Implications for metalanguage implementation
- General architecture for extending metalanguages
  - Modular syntax definition
  - Meta explosion and assimilation
  - By default available in Stratego/XT
- Running example
  - Metalanguage: Stratego
  - Object language: Tiger
  - Meta program: instrumentation of Tiger program

```
String vName = "propertyChangeListeners";
jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");
jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");
jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

```
String vName = "propertyChangeListeners"
jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");
jsc.add("PropertyChangeEvent event = new
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue,
jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Uses the Java syntax: the syntax of the domain.

No syntactic checks of the generated code.

Escaping to the meta language is difficult.

Code generator tries to do some pretty printing.

```
<<FILE "...." .... ".java">>
  if(<<listeners>> == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < <<listeners>>.size(); c++) {
    ((PropertyChangeListener)
        <<listeners>>.elementAt(c)).propertyChange(event);
  }
<<ENDFILE>>
```

```
<<FILE "...." .... ".java">>
  if(<<listeners>> == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < <<listeners>>.size();
    ((PropertyChangeListener)
        <<listeners>>.elementAt(c)).propertyChange(event);
  }
<<ENDFILE>>
```

Uses the Java syntax: the syntax of the domain.

No syntactic checks of the generated code.

Easy to escape to the meta language

```
VariableDeclarationFragment fragment =
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List<Expression> args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

```
VariableD                                    =
    _ast.newVariableDeclarationFragment(
fragment.setName(_ast.newSimpleName("eve
ClassInstanceCreation newi = _ast.newCla
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List<Expression> args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

Extremely verbose and unclear: 90 lines of code!

Does not correspond to the structure of the code to be generated.

Syntactically checked by meta compiler and further processing is possible.

Don't worry about the layout.

```
String x = "propertyChangeListeners";

List<Statement> stms = stmt* |[
  if(#(id)[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #(id)[x].size(); c++) {
    ((PropertyChangeListener)
      #(id)[x].elementAt(c)).propertyChange(event);
  }
]|;
```

Quotation:

|[Java (Object)]|

```
String x = "propertyChangeListeners";

List<Statement> stms = stmt* |[
  if(#(id)[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #(id)[x].size(); c++) {
    ((PropertyChangeListener)
      #(id)[x].elementAt(c)).propertyChange(event);
  }
]|;
```

Anti-Quotation:

#[Java (Meta)]

```
String x = "propertyChangeListeners";

List<Statement> stms = stmt* |[
  if(#(id)[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #(id)[x].size(); c++) {
    ((PropertyChangeListener)
      #(id)[x].elementAt(c)).propertyChange(event);
  }
]|;
```

```
String x = "propertyChangeListeners";

List<Statement> stms = stmt* |[
  if(#(id)[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #(id)[x].size(); c++) {
    ((PropertyChangeListener)
      #(id)[x].elementAt(c)).propertyChange(event);
  }
]|;
```

Uses the syntax of the domain: Java.

Syntax of the generated code is checked and further processing is possible.

Separate pretty-printer: don't worry about the layout.

Support for interaction between the generated code and the meta language.

Syntax of Stratego (meta language)
is **extended** with
syntax of Tiger (object language)

Recipe

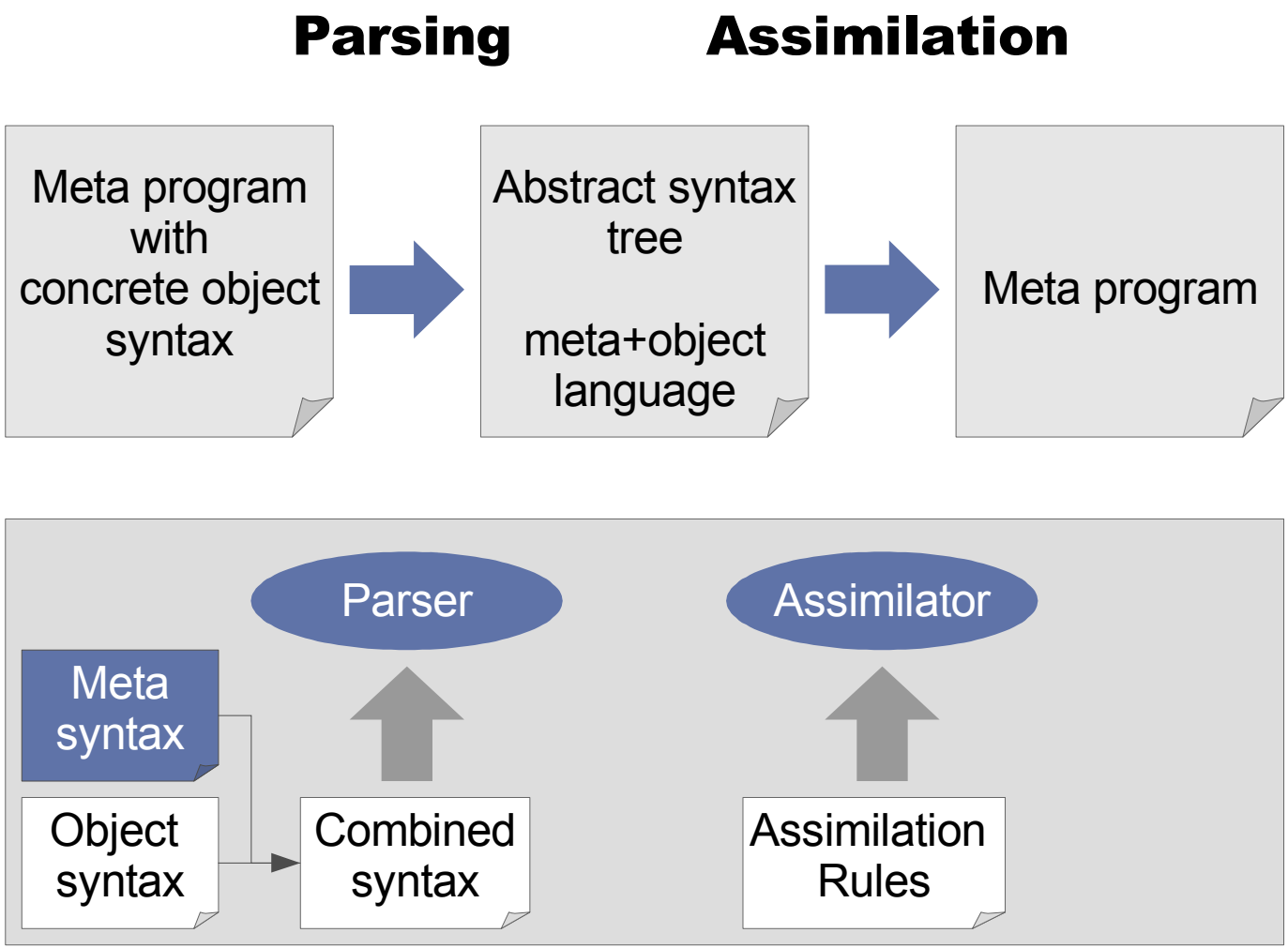
Combine syntax of meta- and object-languages

Parse meta-program with combined syntax

Feed parse tree to meta-language compiler

Ingredients

- *M*-compile: meta-language compiler
- SDF: Modular syntax definition formalism
- SGLR: Scannerless Generalized LR parser


Parsing
Assimilation
Meta program with concrete object syntax
Abstract syntax tree
meta+object language
Meta program
Parser
Assimilator
Meta syntax
Object syntax
Combined syntax
Assimilation Rules

Small example program

```
let x = 3 in x * x end
```

```
Let(
  [ VarDec("x", Int("3")) ]
. [ Mul(Var("x"), Var("x")) ]
)
```

SDF syntax definition

```
module Tiger
exports
  context-free syntax
    "let" Dec* "in" {Exp ";"}* "end" -> Exp {cons("Let")}
    Id "=" Exp  -> Dec {cons("VarDec")}
    Id          -> Exp {cons("Var")}
    Int         -> Exp {cons("Int")}
    Exp "*" Exp -> Exp {cons("Mul"), left}
    Id ":=" Exp -> Exp {cons("Assign")}
    "(" {Exp ";"}* ")" -> Exp {cons("Seq")}
  lexical syntax
    [A-Za-z]+ -> Id
    [0-9]+    -> Int
```

```
regular tree grammar
productions
    Exp -> Int(<string>)
         | Var(<string>)
         | Mul(Exp, Exp)
         | Assign(<string>, Exp)
         | Seq(Exp*)
         | Let(Dec*, Exp*)
    Dec -> VarDec(Id, Exp)
```

Stategо rewrite rule for Tiger

```
a := let b = 3 in b * b end   ->   let b = 3 in a := b * b end
```

```
Assign(x, Let(d*,e*)) -> Let(d*, [Assign(x,Seq(e*))])
```

```
|[ x := let d* in e* end ]| -> |[ let d* in x := (e*) end ]|
```

Implementation: Syntax of Meta Language (Stratego) 25

Statego rewrite rule for Tiger

```
Assign(x, Let(d*,e*)) -> Let(d*, [Assign(x,Seq(e*))])
```

SDF syntax definition for Stratego

```
module Stratego
exports
  context-free syntax
    Id                          -> Term {cons("Var")}
    "[" {Term ","}* "]"         -> Term {cons("List")}
    Id "(" {Term ","}* ")" -> Term {cons("Op")}
    Term "->" Term              -> Rule {cons("Rule")}

  lexical syntax
    [A-Za-z]+ "*"? -> Id
```

Implementation: Syntax of Meta Language (Stratego) 26

Stategо rewrite rule for Tiger

```
Assign(x, Let(d*,e*)) -> Let(d*, [Assign(x,Seq(e*))])
```

Stategо rewrite rule for Tiger, after parsing

```
Rule(
  Op("Assign",[
    Var("x"),
    Op("Let",[Var("d*"),Var("e*")])]
  )
, Op("Let",[
    Var("d*")
  , List([
      Op("Assign",[Var("x"), Op("Seq",[Var("e*")])])
    ])
  ])
)
```

```
module Stratego+Tiger
imports TigerMix[Tiger]
imports StrategoMix[Stratego]
exports
  context-free syntax
    "|[" Dec[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}
    "|[" Exp[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}

    "${" Term[[Stratego]] "}" -> Exp[[Tiger]]         {cons("FromTerm")}
    "${" Term[[Stratego]] "}" -> {Exp[[Tiger]] ";"}+ {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id[[Tiger]]
    [e][0-9]*      -> Exp[[Tiger]]
    "e"[0-9]* "*"  -> {Exp[[Tiger]] ";"}+
```

Qualify nonterminals to unintended avoid clashes

```
module Stratego+Tiger
imports TigerMix[Tiger]
imports StrategoMix[Stratego]
exports
  context-free syntax
    "|[" Dec[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}
    "|[" Exp[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}

    "${" Term[[Stratego]] "}" -> Exp[[Tiger]]         {cons("FromTerm")}
    "${" Term[[Stratego]] "}" -> {Exp[[Tiger]] ";"}+ {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id[[Tiger]]
    [e][0-9]*      -> Exp[[Tiger]]
    "e"[0-9]* "*"  -> {Exp[[Tiger]] ";"}+
```

```
module Stratego+Tiger
imports TigerMix[Tiger]
imports StrategoMix[Stratego]
exports
  context-free syntax
    "|[" Dec[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}
    "|[" Exp[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}

    "${" Term[[Stratego]] "}" -> Exp[[Tiger]]         {cons("FromTerm")}
    "${" Term[[Stratego]] "}" -> {Exp[[Tiger]] ";"}+ {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id[[Tiger]]
    [e][0-9]*      -> Exp[[Tiger]]
    "e"[0-9]* "*"  -> {Exp[[Tiger]] ";"}+
```

Quotation: inject object expressions in meta expressions

Anti-quotation: inject meta expressions into object expressions

```
module Stratego+Tiger
imports TigerMix[Tiger]
imports StrategoMix[Stratego]
exports
  context-free syntax
    "|[" Dec[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}
    "|[" Exp[[Tiger]]    "]|" -> Term[[Stratego]] {cons("ToTerm")}

    "${" Term[[Stratego]] "}" -> Exp[[Tiger]]         {cons("FromTerm")}
    "${" Term[[Stratego]] "}" -> {Exp[[Tiger]] ";"}+ {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id[[Tiger]]
    [e][0-9]*      -> Exp[[Tiger]]
    "e"[0-9]* "*"  -> {Exp[[Tiger]] ";"}+
```

Declare meta-variables for object sorts

Meta Variables

```
TraceProcedure :
  |[ function f(x*) = e ]|
    ->
  |[ function f(x*) = (enterfun(s); e; exitfun(s)) ]|
  where
    s := f
```

meta-variables

object identifiers

Distinguish meta-variables by anti-quotation

```
TraceProcedure :
  |[ function ${f}(${xs}) = ${e} ]| ->
  |[ function ${f}(${xs}) = (enter(${s}); ${e}; exit(${s})) ]|
  where
    s := f
```

Splice code generated by meta-computation into object code

```
TraceProcedure :
  |[ function ${f}(${xs}) = ${e} ]| ->
  |[ function ${f}(${xs}) =
       (print(${f + " entry"});
        ${e};
        print(${f + " exit"});
       ) ]|
```

```
|[ x := let d* in ${e*} end ]| -> |[ let d* in x := (${e*}) end ]|
```

```
Rule(ToTerm(Assign(meta-var("x"),
                   Let(meta-var("d*"),FromTerm(Var("e*"))))),
     ToTerm(Let(meta-var("d*"),
                [Assign(meta-var("x"),
                        Seq(FromTerm(Var("e*"))))])))
```

```
Rule(Op("Assign",[Var("x"),
                  Op("Let",[Var("d*"),Var("e*")])]),
     Op("Let",[Var("d*"),
               List([Op("Assign",[Var("x"),
                                  Op("Seq",[Var("e*")])])])]
     ]))
```

```
Assign(x, Let(d*,e*)) -> Let(d*, [Assign(x,Seq(e*))])
```

Extend meta language with object language syntax

- combine arbitrary meta and object language
- meta-language does not need to be designed for this purpose
- limitation: language should have context-free syntax

Modular syntax definition is essential

- reuse existing syntax definitions – no changed needed

Scannerless Generalized LR parsing

- Only context-free grammars closed under composition
- No compositionality: regular grammars, LR, LL

Part IV

Ambiguities

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(#[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #[x].size(); c++) {
    ((PropertyChangeListener)
      #[x].elementAt(c)).propertyChange(event);
  }
]|;
```

```
String x = "propertyChangeListeners";

List<Statement> stms = stmt* |[
  if(#(id)[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #(id)[x].size(); c++) {
    ((PropertyChangeListener)
      #(id)[x].elementAt(c)).propertyChange(event);
  }
]|;
```

Ambiguities in Concrete Object Syntax


```
String x = "propertyChangeListeners";

List<Statement> stms = stmt* |[
  if(#(id)[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #(id)[x].size(); c++) {
    ((PropertyChangeListener)
      #(id)[x].elementAt(c)).propertyChange(event);
  }
]|;
```

Syntactic clutter of explicit typing.

Intimate knowledge of syntactic structure required.

Limited number of quotations and anti-quotations are supported.

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(#[x] == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < #[x].size(); c++) {
    ((PropertyChangeListener)
      #[x].elementAt(c)).propertyChange(event);
  }
]|;
```

Ambiguities: Quotations

- Single quotation symbol
- Multiple object non-terminals

Example:

- ⟦ `class Foo {}` ⟧

TypeDeclaration

```
package bar;

class Foo {}
```

CompilationUnit

```
class Foo {}
```

Statement

```
class Bar {
  void fred() {
    class Foo {}
  }
}
```

- Single anti-quotation symbol
- Multiple object non-terminals
- Syntactical category of meta code unknown

Examples:

- `|[ return #[x]; ]|`
  - x can be `Expression`
  - x can be `SimpleName`
  - x can be `String` (identifier)
- `|[ foo(#[xs]) ]|`
  - xs can be `Expression`
  - xs can be `List<Expression>`
  - ...

```
⟦
  class Foo() {
    #[type] getBar() {
      ...
    }
  }
⟧
```

- *type* can be `Type, Id`
- *type* can be `BodyDeclaration`
- *type* can be `List<BodyDeclaration>`
- *type* can be `Modifier`
- *type* can be `List<Modifier>`

Ambiguities: Current Solutions

Use Different (Anti-)Quotation Symbols

- JTS, DMS, Stratego, Template Haskell, Jumbo
- Redundant
- Support for (anti-)quotations irregular

Type-based Disambiguation by Context-Sensitive Parsing

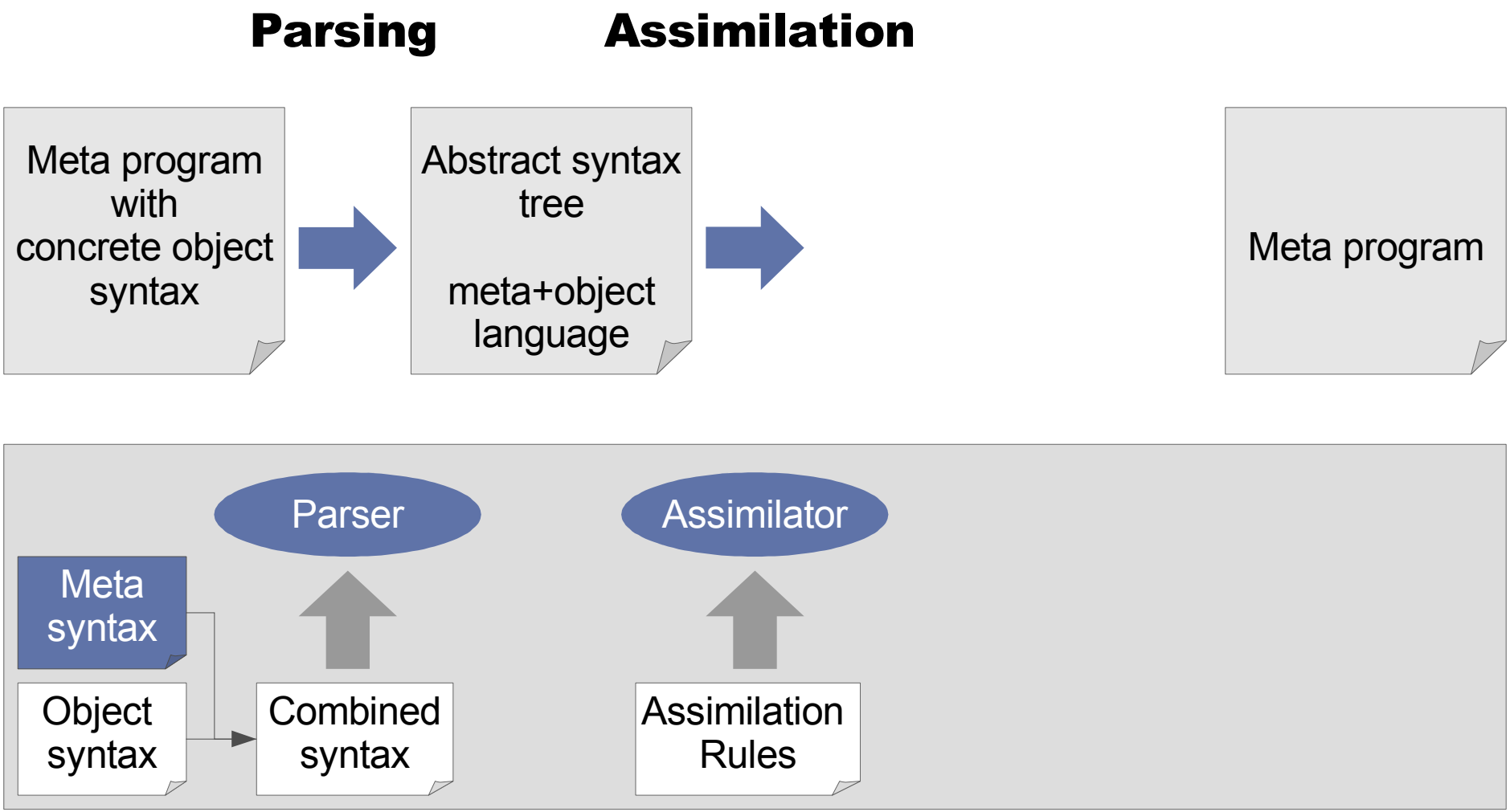
- Meta-AspectJ, Aasa (ML)
- Excellent usability
- Parsing and type-checking tangled
- Object language specific, no multiple object languages

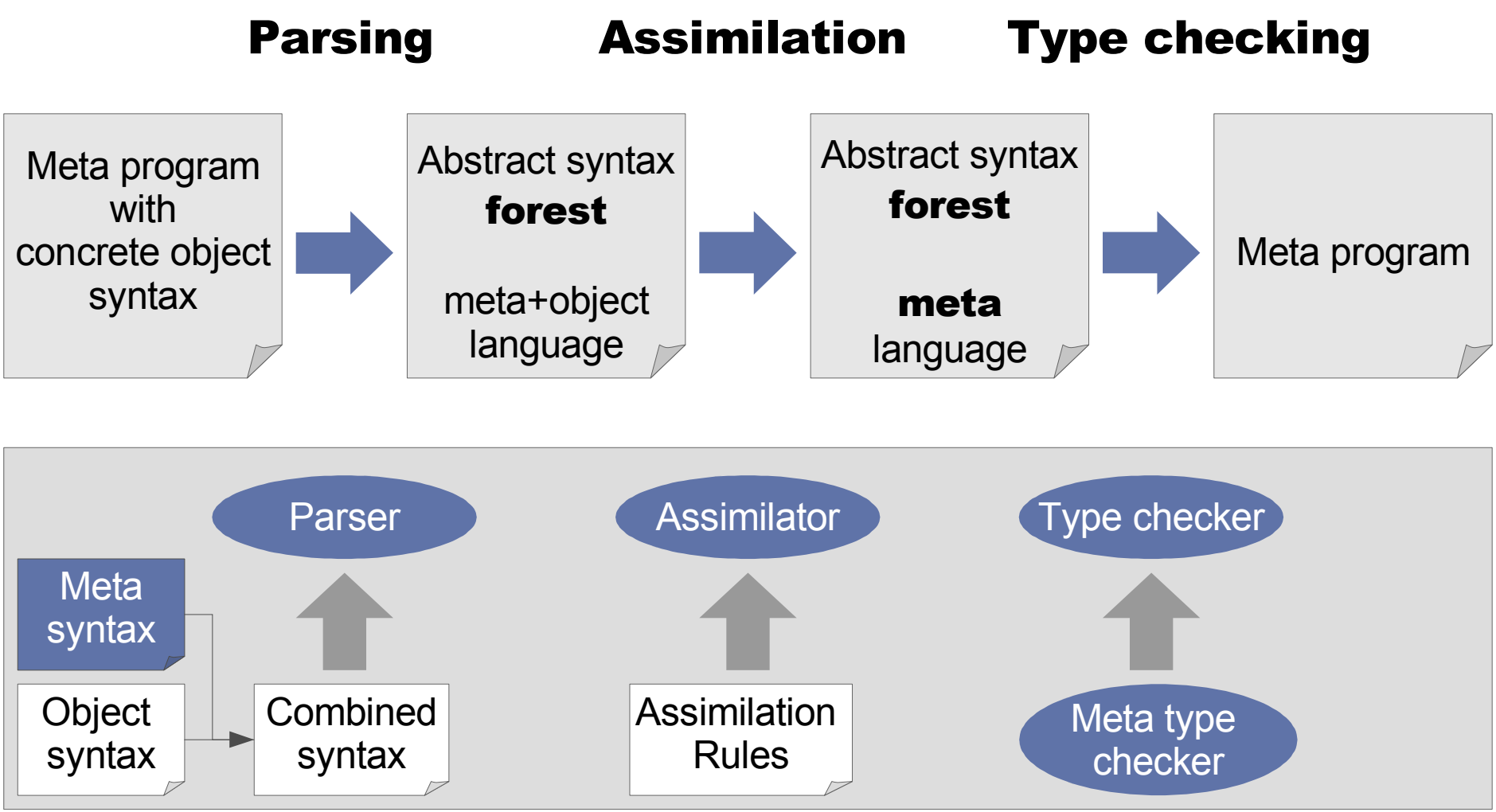
We need a *general* architecture for type-based disambiguation.

- **Introduce type-based disambiguation**
  - Single quotation and anti-quotation symbol
- **Preserve modular syntax definition**
  - Fully automatic parser generation
- **Preserve modular assimilation**
  - Embedding multiple object languages

Define only syntactical embedding and assimilation rules

Type-based Disambiguation




Parsing
Assimilation
Type checking
Meta program with concrete object syntax
Abstract syntax forest
meta+object language
Abstract syntax forest
meta language
Meta program
Parser
Assimilator
Type checker
Meta syntax
Object syntax
Combined syntax
Assimilation Rules
Meta type checker

Example: Syntax Definition and Parsing


```
context-free syntax
  "|[" CompilationUnit "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" TypeDec         "]|" -> MetaExpr {cons("ToMetaExpr")}
  "|[" ClassBodyDec*   "]|" -> MetaExpr {cons("ToMetaExpr")}

context-free syntax
  "#[" MetaExpr "]"-> ID   {cons("FromMetaExpr")}
  "#[" MetaExpr "]"-> Expr {cons("FromMetaExpr")}
```

```
|[ package bar; class Foo {} ]|
```

```
ToMetaExpr(
  CompilationUnit(
    Some(PackageDec([], PackageName([Id("bar")]))), []
  , [ ClassDec(
        ClassDecHead([], Id("Foo"), None, None, None)
      , ClassBody([])
      )
    ]))
```

Example: Syntax Definition and Parsing


```
dec = ⟦ class Foo {} ⟧
```

```
Assign(ExprName(Id("dec")),
  amb([
    ToMetaExpr(  CompilationUnit(... ClassDec(... Id("Foo")...) ...) )



  , ToMetaExpr(  ClassDec(... Id("Foo") ...)   )


  , ToMetaExpr([ ClassDec(... Id("Foo") ...) ] )


  , ToMetaExpr(  ClassDecStm(ClassDec(... Id("Foo") ...)   ))

  , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)   )])

  ]))
```

Example: Syntax Definition and Parsing


```
dec = ⟦ class Foo {} ⟧
```

```
Assign(ExprName(Id("dec")),
  amb([
    ToMetaExpr(  CompilationUnit(... ClassDec(..
  , ToMetaExpr(  ClassDec(... Id("Foo") ...)   )
  , ToMetaExpr([ ClassDec(... Id("Foo") ...) ] )
  , ToMetaExpr(  ClassDecStm(ClassDec(... Id("Foo") ...)   ))
  , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)   )])
  ]))
```

Ambiguities are preserved by the GLR parser

All possible parses, represented by ambiguity nodes

Example: Assimilation


```
dec = ⟦ class Foo {} ⟧
```

```
Assign(ExprName(Id("dec")),
  amb([
    {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
       TypeDeclaration class_0 = _ast.newTypeDeclaration();
       class_0.setName(_ast.newSimpleName("Foo"));  ... | cu_0 |}

  , ToMetaExpr(  ClassDec(... Id("Foo") ...)   )


  , ToMetaExpr([ ClassDec(... Id("Foo") ...) ] )


  , ToMetaExpr(  ClassDecStm(ClassDec(... Id("Foo") ...)   ))

  , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)   )])

  ]))
```

Example: Assimilation


```
dec = ‖[ class Foo {} ]‖
```

```
Assign(ExprName(Id("dec")),
  amb([
    {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
       TypeDeclaration class_0 = _ast.newTypeDeclaration();
       class_0.setName(_ast.newSimpleName("Foo"));  ... | cu_0 |}

  , {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
       class_1.setName(_ast.newSimpleName("Foo"));  ... | class_1 |}

  , ToMetaExpr([ ClassDec(... Id("Foo") ...) ] )


  , ToMetaExpr(  ClassDecStm(ClassDec(... Id("Foo") ...)   ))

  , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)   )])

  ]))
```

Example: Assimilation


```
dec = ⟦ class Foo {} ⟧
```

```
Assign(ExprName(Id("dec")),
  amb([
    {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
       TypeDeclaration class_0 = _ast.newTypeDeclaration();
       class_0.setName(_ast.newSimpleName("Foo"));  ... | cu_0 |}

  , {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
       class_1.setName(_ast.newSimpleName("Foo"));  ... | class_1 |}

  , {| List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();
       decs_0.add( ... );  ... | decs_0 |}

  , ToMetaExpr(  ClassDecStm(ClassDec(... Id("Foo") ...)   ))

  , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)   )])

  ]))
```

Example: Assimilation

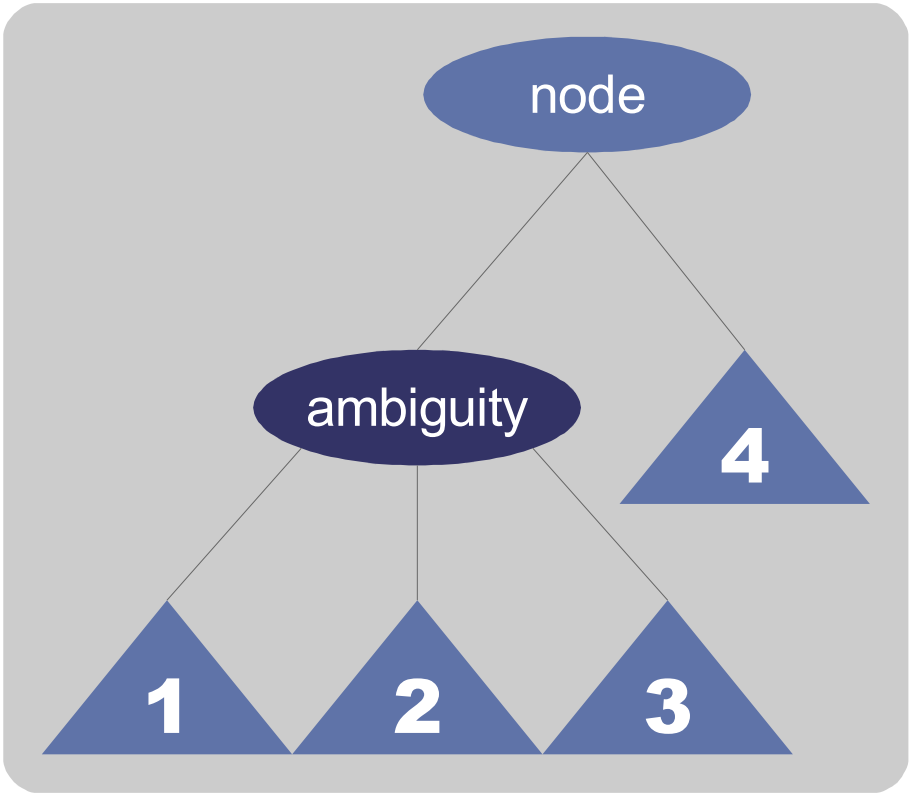

```
dec = ⟦ class Foo {} ⟧
```

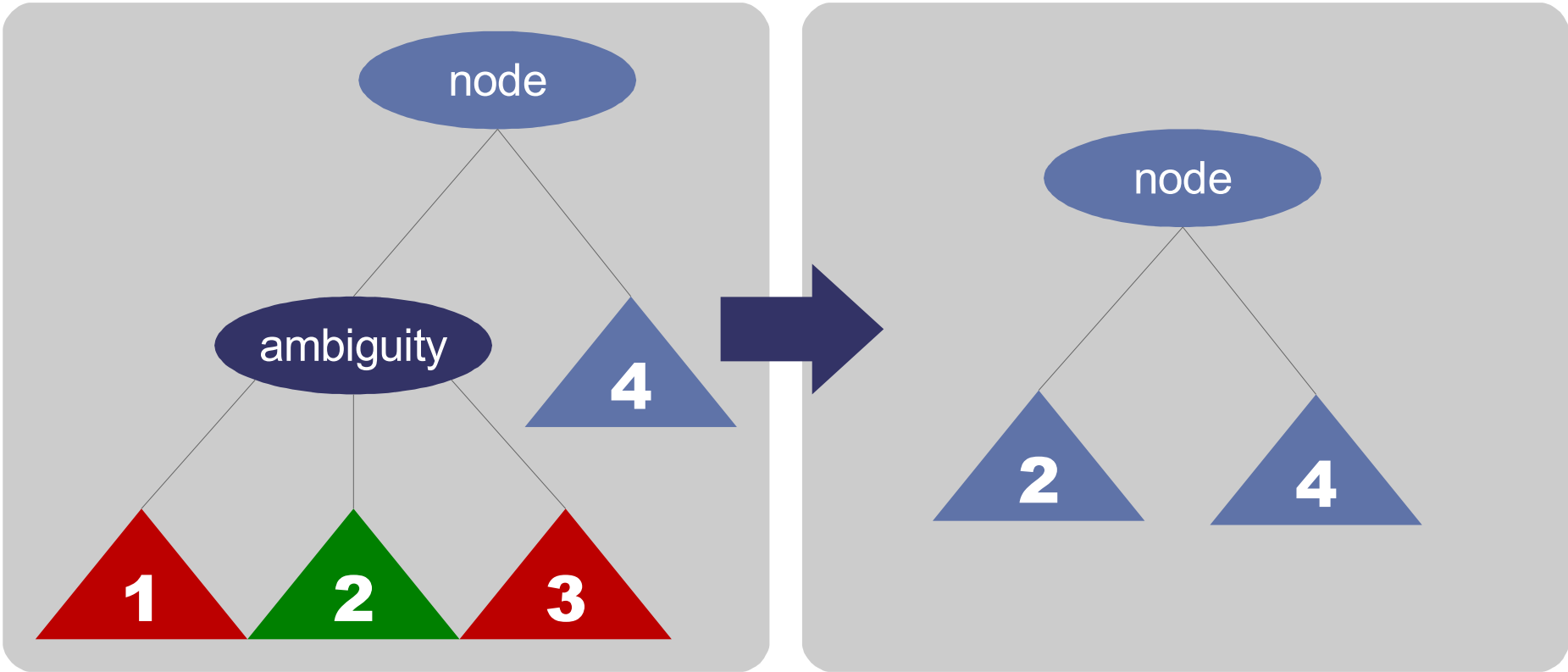
```
Assign(ExprName(Id("dec")),
  amb([
    {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
       TypeDeclaration class_0 = _ast.newTypeDeclaration();
       class_0.setName(_ast.newSimpleName("Foo"));  ... | cu_0 |}

  , {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
       class_1.setName(_ast.newSimpleName("Foo"));  ... | class_1 |}

  , {| List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();
       decs_0.add( ... );  ... | decs_0 |}

  , _ast.newTypeDeclaratonStatement(...);

  , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...)   )])

  ]))
```

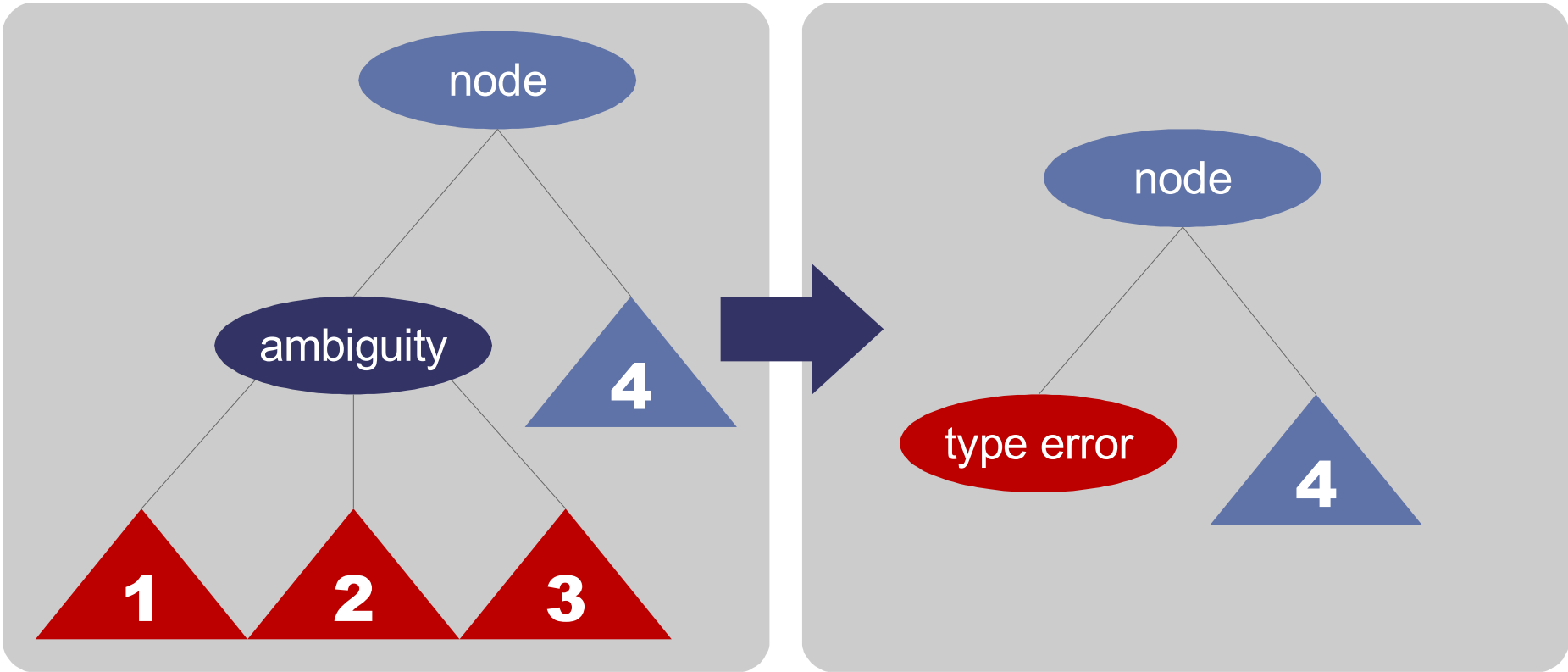
Example: Assimilation

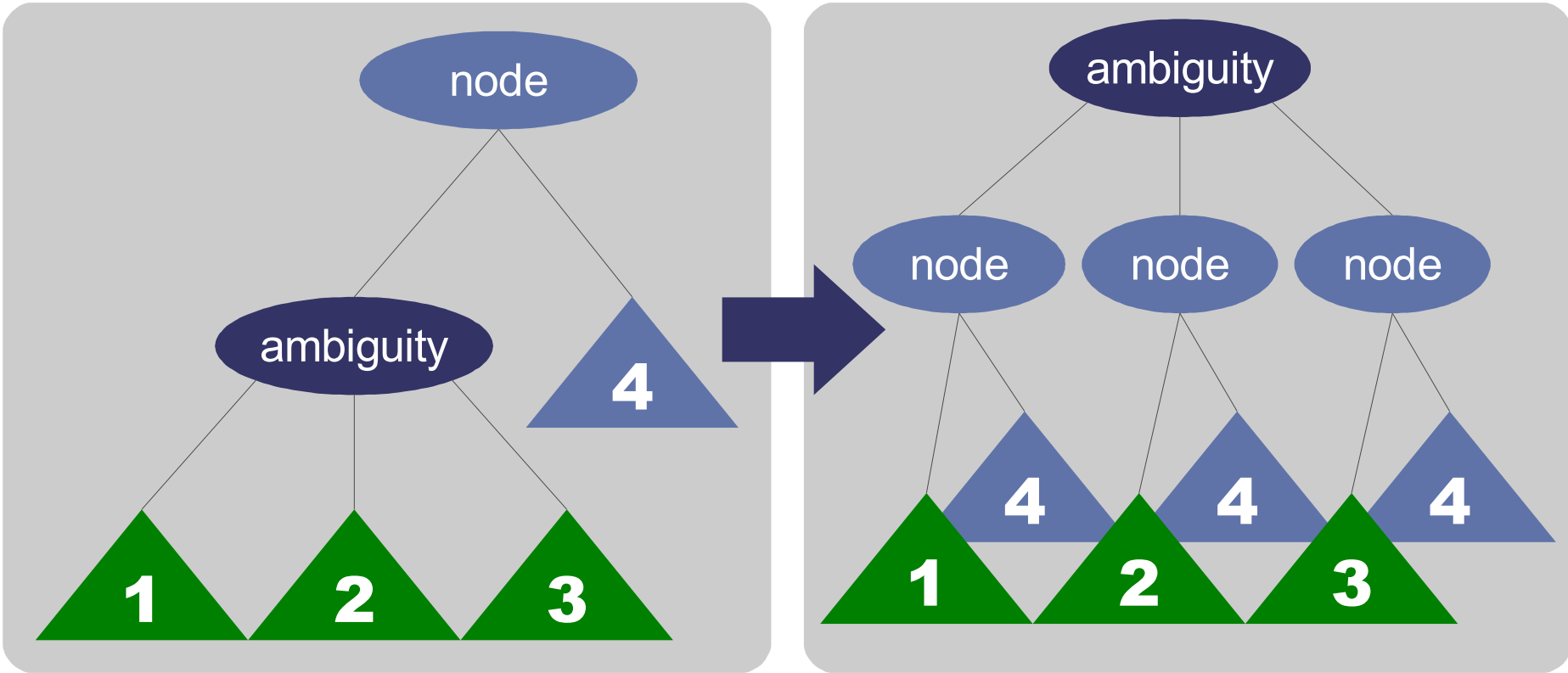

```
dec = ⟦ class Foo {} ⟧
```

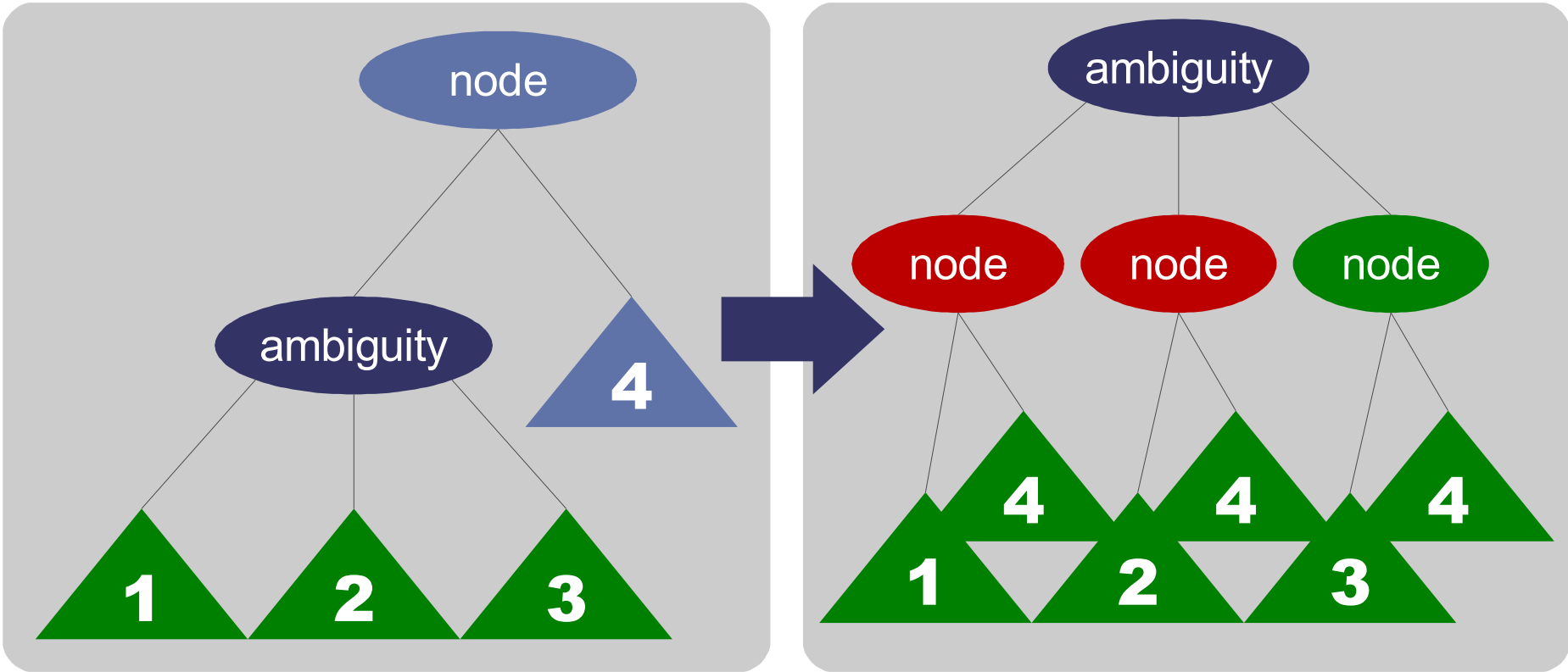
```
Assign(ExprName(Id("dec")),
  amb([
    {| CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
       TypeDeclaration class_0 = _ast.newTypeDeclaration();
       class_0.setName(_ast.newSimpleName("Foo"));  ... | cu_0 |}

  , {| TypeDeclaration class_1 = _ast.newTypeDeclaration();
       class_1.setName(_ast.newSimpleName("Foo"));  ... | class_1 |}

  , {| List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();
       decs_0.add( ... );  ... | decs_0 |}

  , _ast.newTypeDeclaratonStatement(...);

  , {| List<Statement> stms_0 = new ArrayList<Statement>();
       stms_0.add(_ast.newTypeDeclaratonStatement(...)); ... | stms_0 |}
  ]))
```


node
ambiguity
4
1
2
3


node
ambiguity
4
1
2
3
node
2
4


node
ambiguity
4
1
2
3
node
type error
4


node
ambiguity
4
1
2
3
ambiguity
node
node
node
4
4
4
1
2
3


node
ambiguity
4
1
2
3
ambiguity
node
node
node
4
4
4
1
2
3

Example: Type-based Disambiguation


```
dec = ⟦ class Foo {} ⟧
```

```
Assign(ExprName(Id("dec")), amb([
    CompilationUnit
  , TypeDeclaration
  , List<BodyDeclaration>
  , TypeDeclaratonStatement
  , List<Statement>
]))
```

```
amb([
  Assign(ExprName(Id("dec")), CompilationUnit)
, Assign(ExprName(Id("dec")), TypeDeclaration)
, Assign(ExprName(Id("dec")), List<BodyDeclaration>)
, Assign(ExprName(Id("dec")), TypeDeclaratonStatement)
, Assign(ExprName(Id("dec")), List<Statement>)
])
```

Example: Type-based Disambiguation


```
dec = ‖[ class Foo {} ]‖
```

```
Assign(ExprName(Id("dec")), amb([
    CompilationUnit
  , TypeDeclaration
  , List<BodyDeclaration>
  , TypeDeclaratonStatement
  , List<Statement>
]))
```

```
amb([
  Assign(ExprName(Id("dec")), CompilationUnit)
, Assign(ExprName(Id("dec")), TypeDeclaration)
, Assign(ExprName(Id("dec")), List<BodyDeclaration>)
, Assign(ExprName(Id("dec")), TypeDeclaratonStatement)
, Assign(ExprName(Id("dec")), List<Statement>)
])
```

Example: Anti-Quotation


```
Expression expr = ...
Statement stmt = |[ return #[expr]; ]|
```

```
Return(
  Some(
    amb([
      FromMetaExpr(ExprName(Id("expr")))
    , ExprName([ FromMetaExpr(ExprName(Id("expr"))) ])
    , ExprName([ Id(FromMetaExpr(ExprName(Id("expr")))) ])
    ])))
```

```
Expression expr = ...
Statement stmt = {|
  ReturnStatement return_1 = _ast.newReturnStatement();
  return_1.setExpression(amb([expr, _ast.newSimpleName(expr)]));
  | return_1 |};
```

Example: Anti-Quotation


```
Expression expr = ...
Statement stmt = |[ return #[expr]; ]|
```

```
Return(
  Some(
    amb([
      FromMetaExpr(ExprName(Id("expr")))
    , ExprName([ FromMetaExpr(ExprName(Id("expr"))) ])
    , ExprName([ Id(FromMetaExpr(ExprName(Id("expr")))) ])
    ])))
```

```
Expression expr = ...
Statement stmt = {|
  ReturnStatement return_1 = _ast.newReturnStatement();
  return_1.setExpression(amb([expr, _ast.newSimpleName(expr)]));
  | return_1 |};
```

Example: Anti-Quotation


```
Expression expr = ...
Statement stmt = |[ return #[expr]; ]|
```

```
Return(
  Some(
    amb([
      FromMetaExpr(ExprName(Id("expr")))
    , ExprName([ FromMetaExpr(ExprName(Id("expr"))) ])
    , ExprName([ Id(FromMetaExpr(ExprName(Id("expr")))) ])
    ])))
```

```
Expression expr = ...
Statement stmt = {|
  ReturnStatement return_1 = _ast.newReturnStatement();
  return_1.setExpression(expr);
  | return_1 |};
```

- Code generators do not need to know the exact type of every AST node.
- no need to determine the exact category statically, or even dynamically: it's ok for a value to belong to multiple syntactical categories.
- Therefore, we associate a set of syntactic categories to every node and preserve ambiguities.
- Fragments that do not are programming errors.

Parse first, solve ambiguities later.

Enablers

- Generalized-LR parsing produces all alternatives
- Separate type checking on ambiguous program

Results

- Modular syntactical embedding and assimilation
- Object language independent disambiguation
- Extensible type checker
- General architecture extended with type-based disambiguation

Part V

The Parsing Techniques

Conventional parsing techniques do not support this.

- LR/LL parsers do not compose
- Scanners (regular grammars) do not compose

Restricted classes of context-free grammars not closed under composition.

What is the problem?

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem?

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

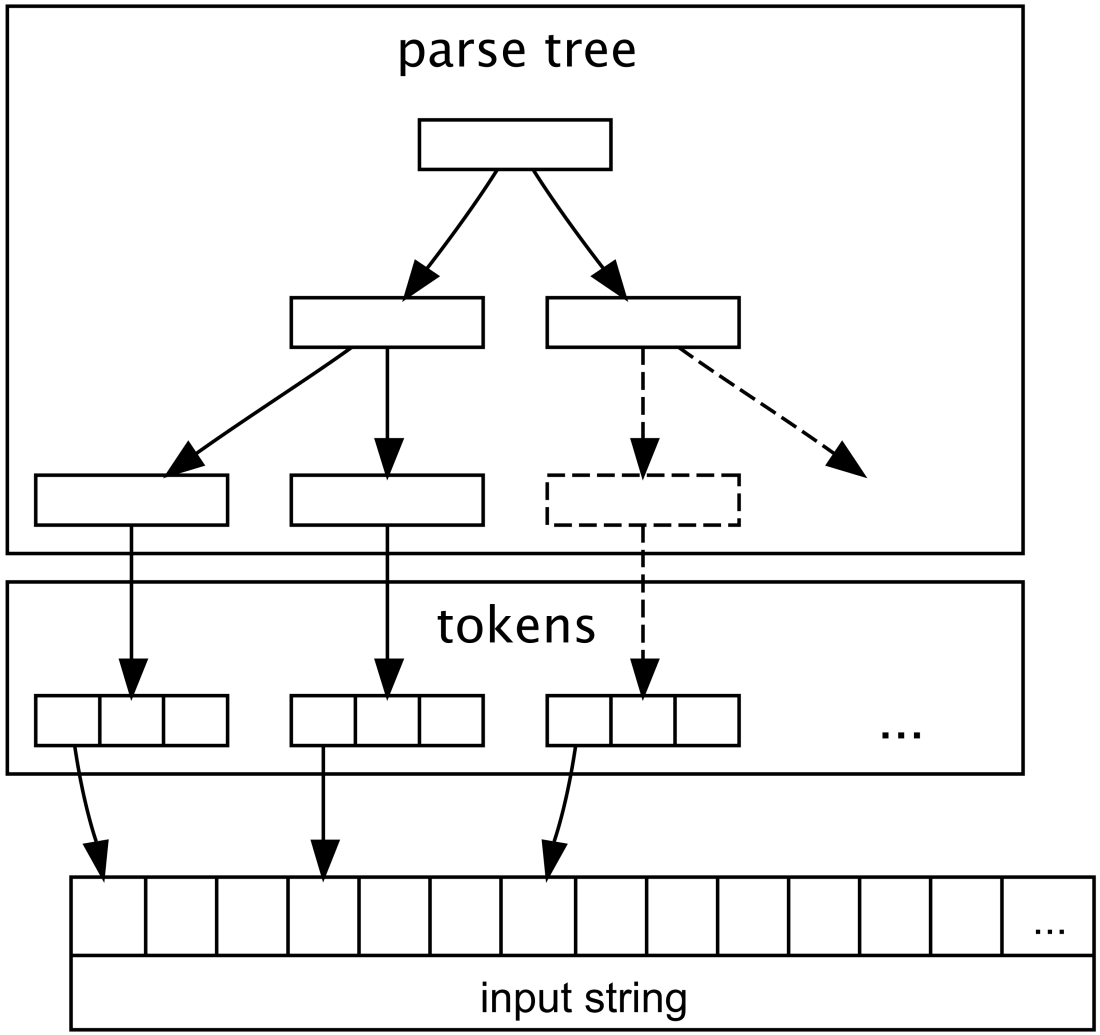
```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

What is the problem?


```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```

```
public aspect Caching {
  private Map<Integer, Integer> cache =
      new HashMap<Integer, Integer>();

  pointcut cached(int value):
      execution(* Fib.calc(int)) && args(value);

  int around(int value): cached(value) {
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value);
      cache.put(value, result);
      return result;
    }
  }
}
```


parse tree
tokens
...
...
input string

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

Keywords cannot be reserved globally

- `get(* Foo.*)`, `target(e)`, `args(event)` ...

Lexical syntax is context-sensitive

- Java context:

```
int x = get* y + z;
```

- Pointcut context:

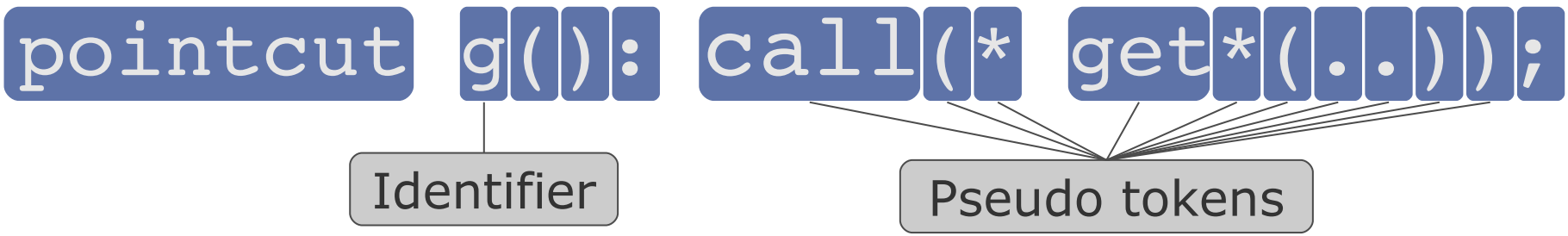
```
pointcut g(): call(* get*(..));
pointcut c(): call(Foo+.new());
```

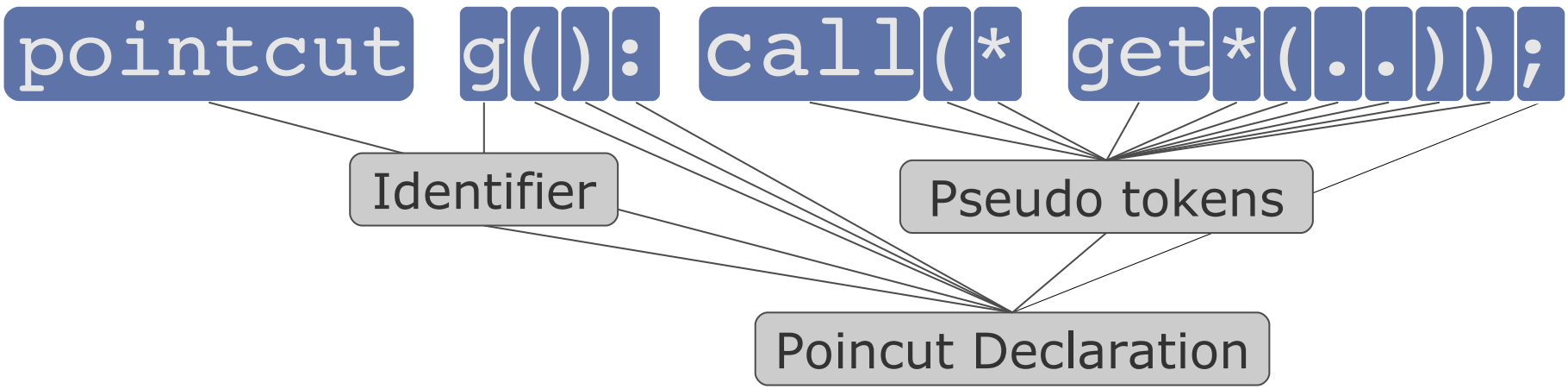
Keywords cannot be reserved globally

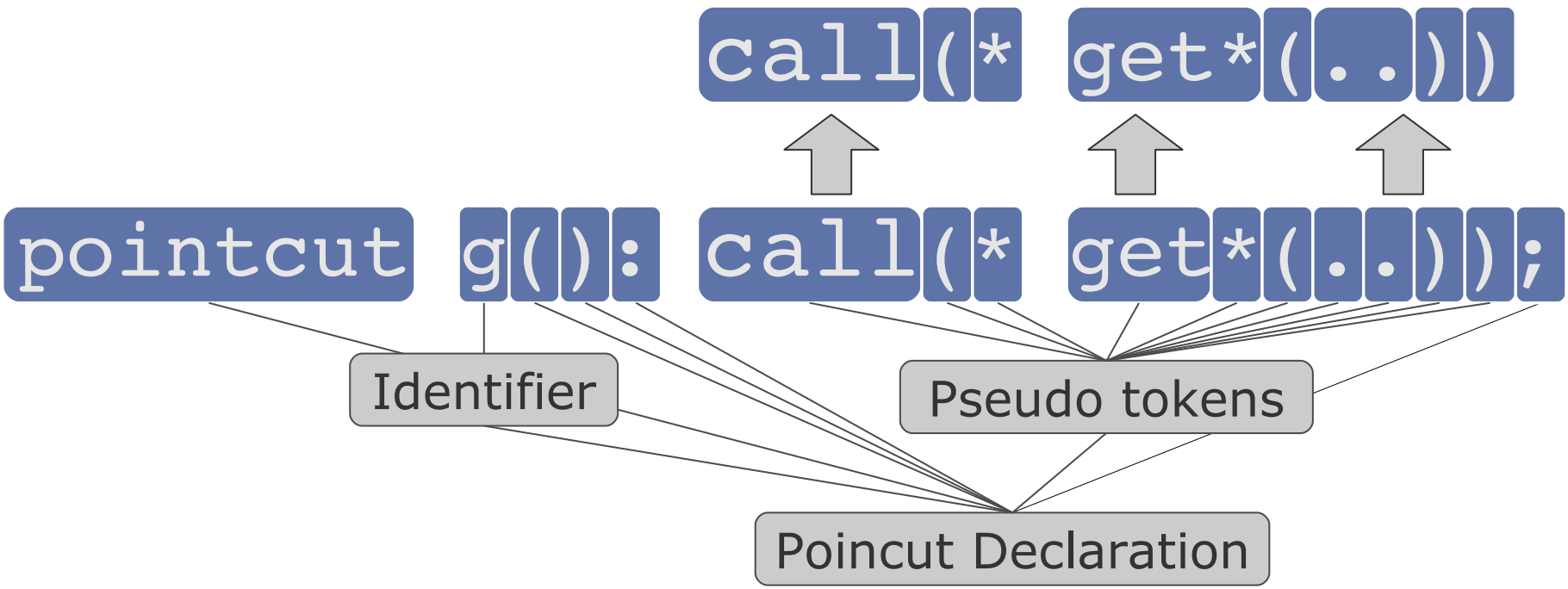
- `get(* Foo.*)`, `target(e)`, `args(event)` ...

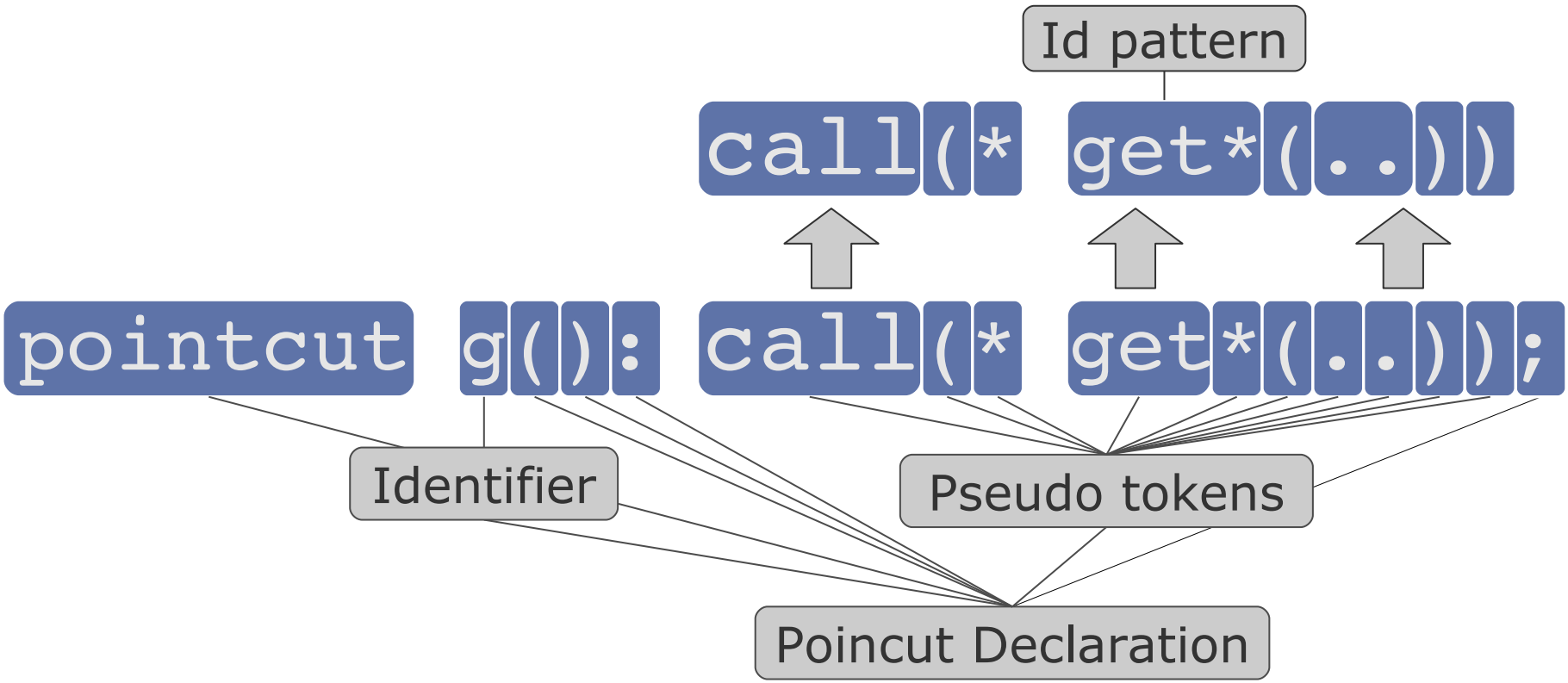
```
pointcut g(): call(* get*(..));
```

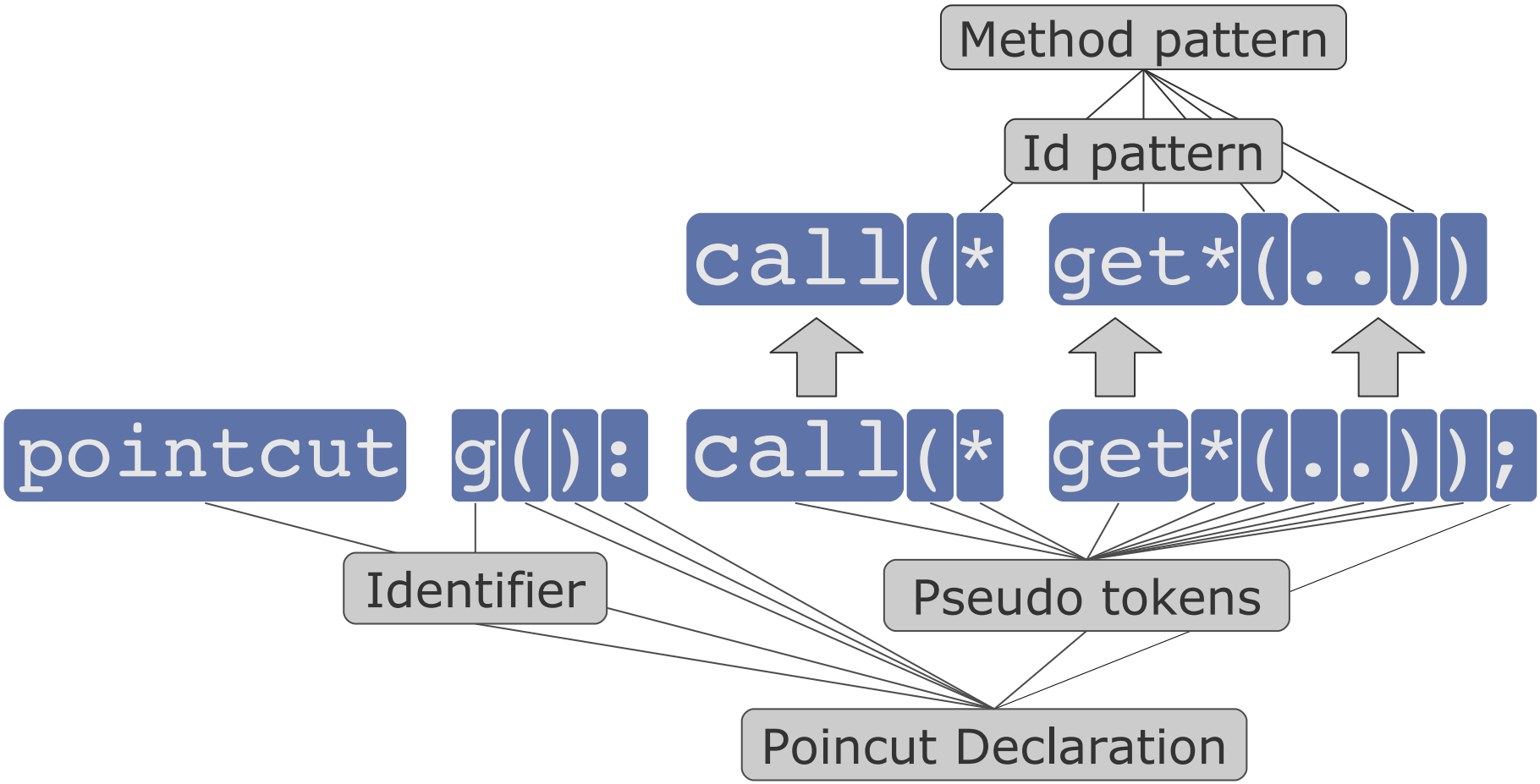
```
pointcut g(): call(* get*(..));
```

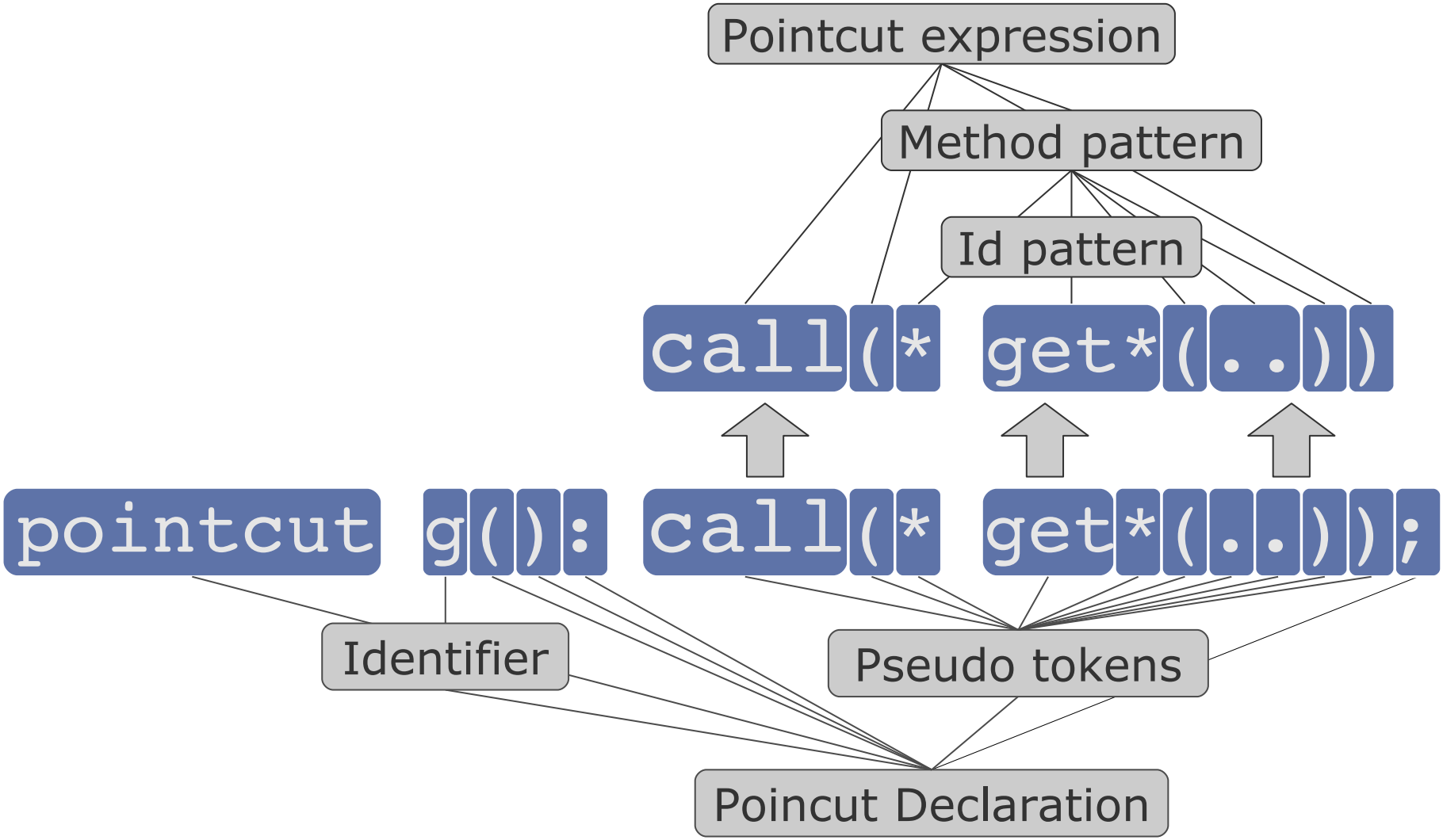

pointcut g(): call(* get*(..));
Identifier
Pseudo tokens

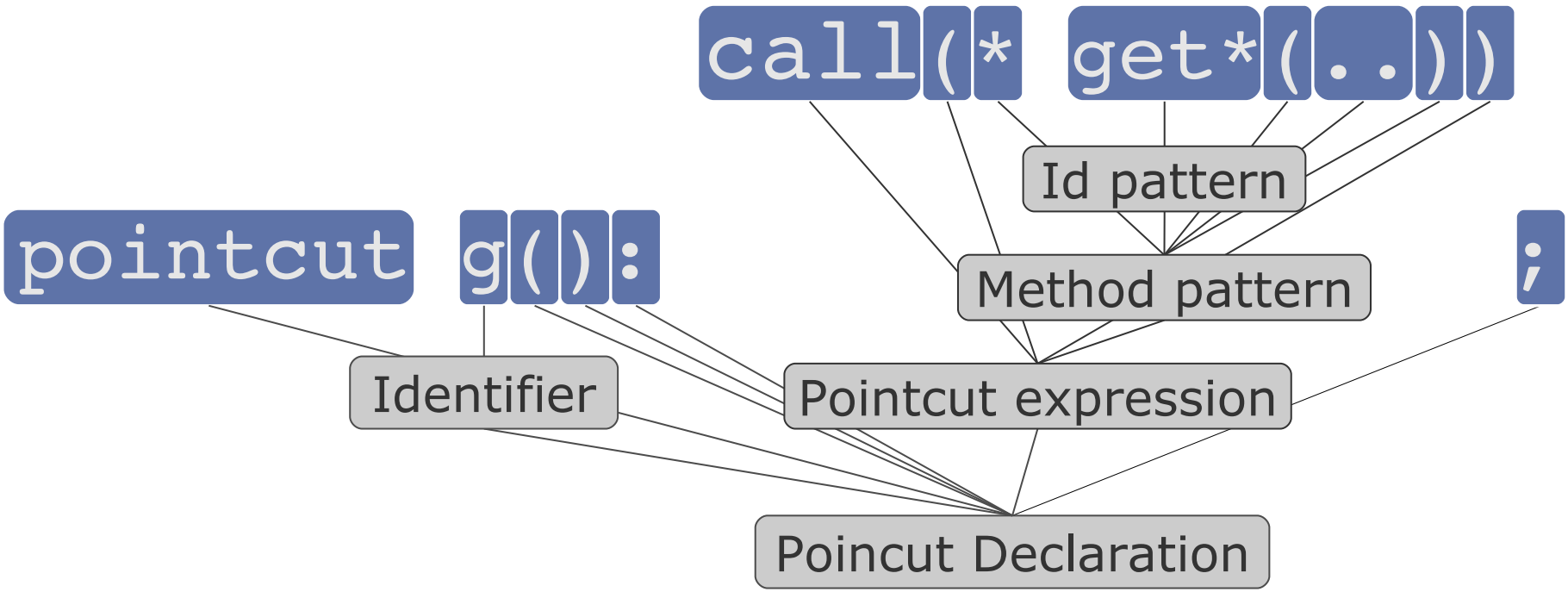

pointcut g(): call(* get*(..));
Identifier
Pseudo tokens
Poincut Declaration

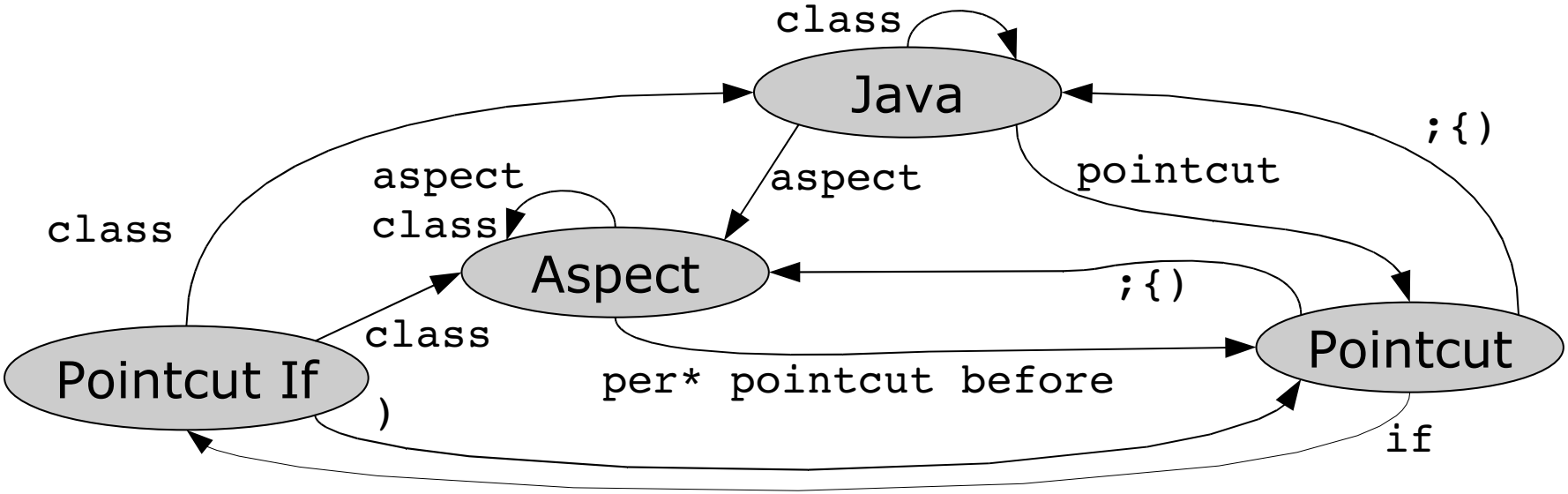

call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration

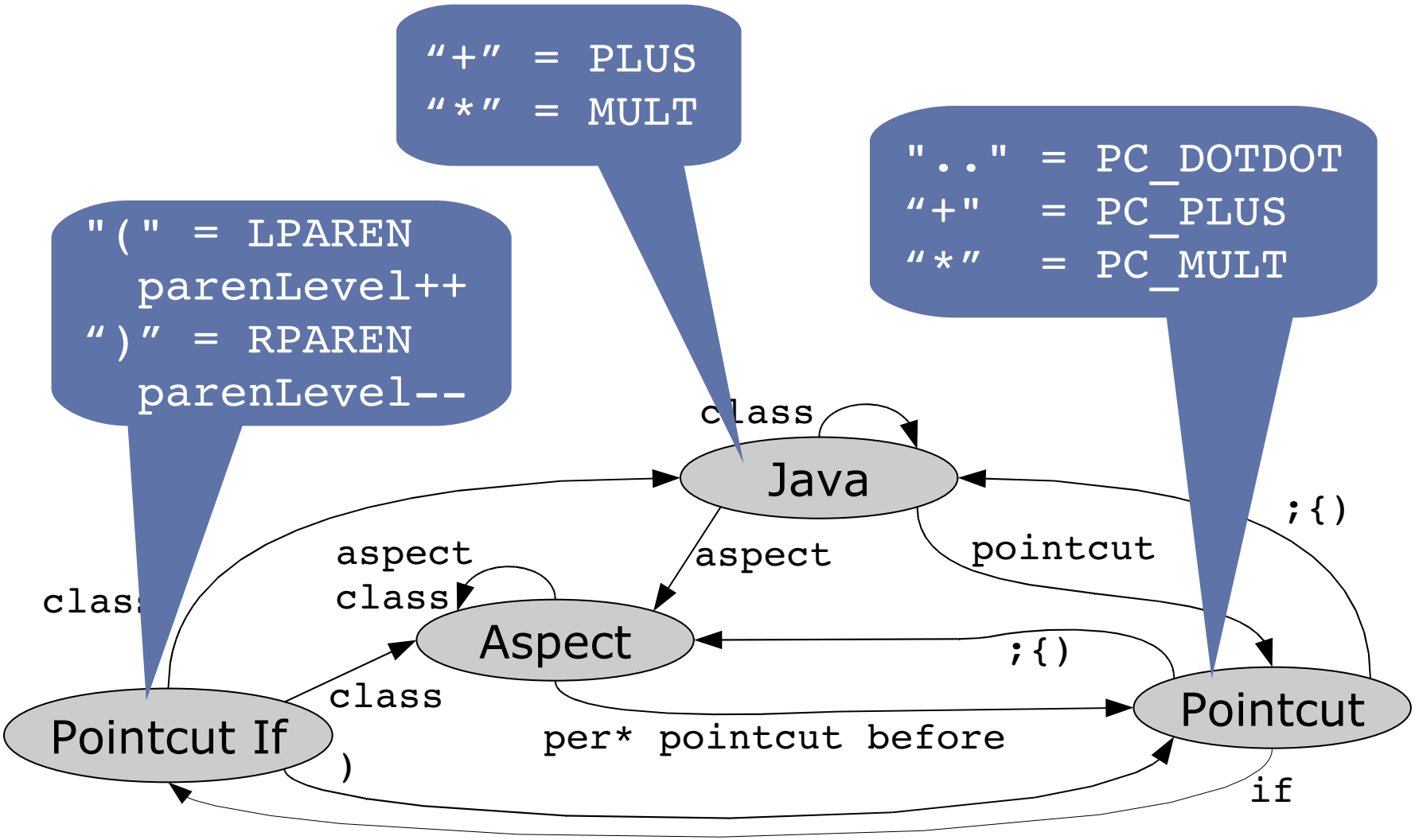

Id pattern
call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration


Method pattern
Id pattern
call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration

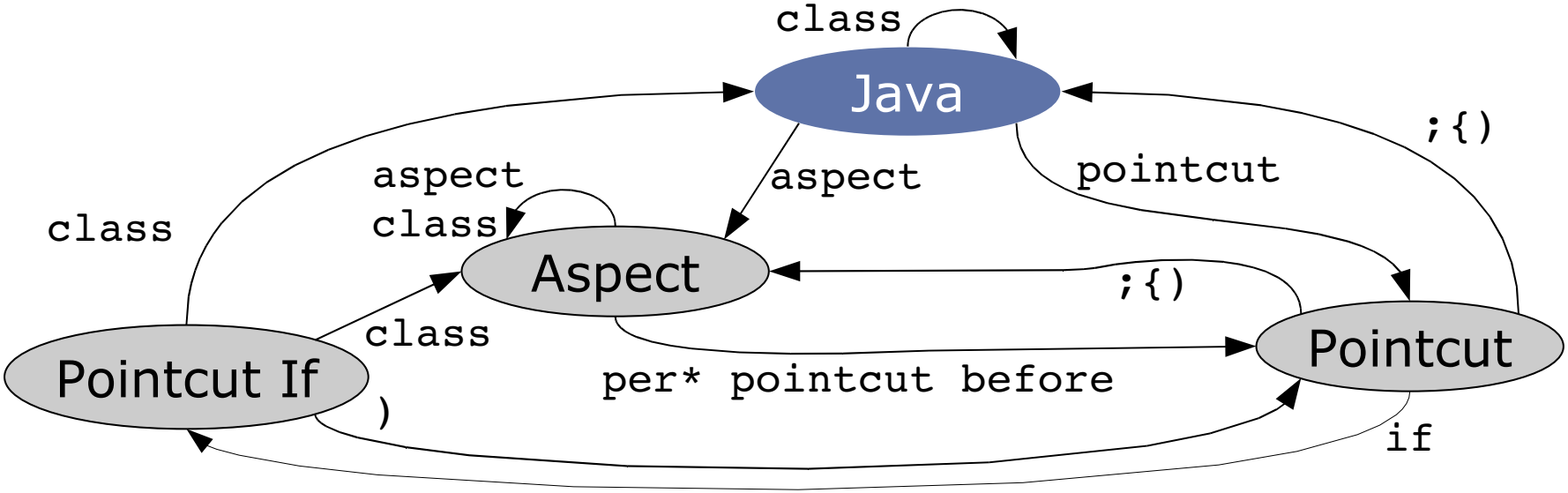

Pointcut expression
Method pattern
Id pattern
call(*
get*(..))
pointcut
g():
call(*
get*(..));
Identifier
Pseudo tokens
Poincut Declaration


call(* get*(..))
Id pattern
pointcut g():
Method pattern
;
Identifier
Pointcut expression
Poincut Declaration

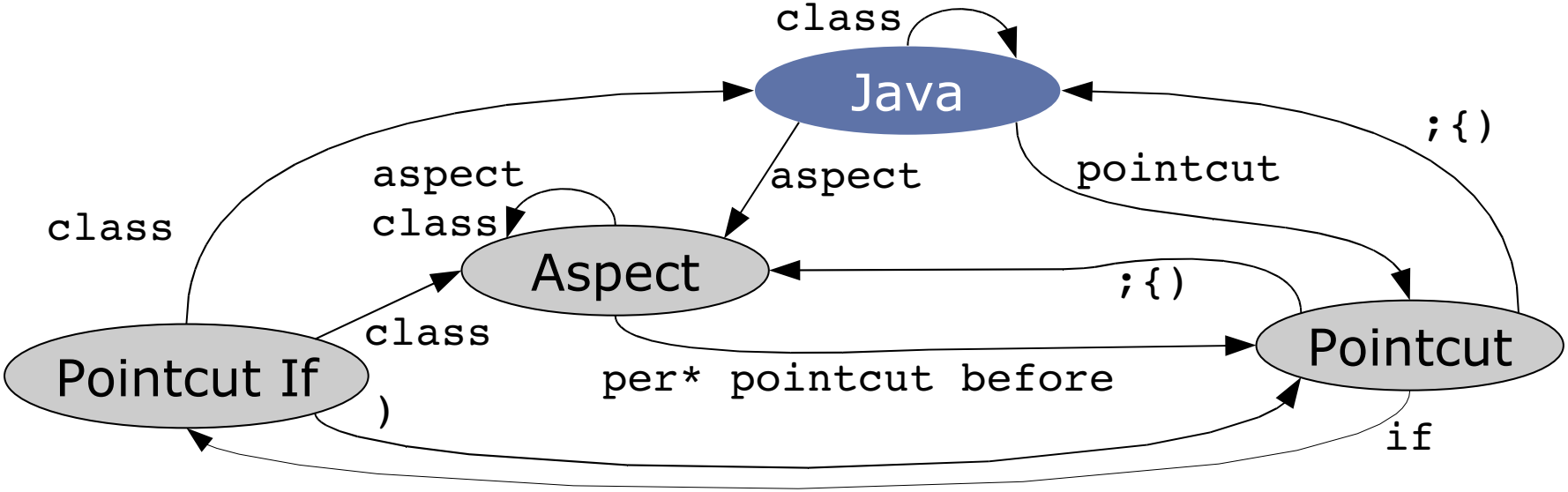

class
Java
;{)
aspect
class
aspect
pointcut
class
Aspect
;{)
class
Pointcut If
per* pointcut before
Pointcut
)
if


"+" = PLUS
"*" = MULT
".." = PC_DOTDOT
"+" = PC_PLUS
"*" = PC_MULT
"(" = LPAREN
parenLevel++
")" = RPAREN
parenLevel--
class
Java
aspect
class
aspect
pointcut
;{}
clas
Aspect
;{}
class
Pointcut If
per* pointcut before
Pointcut
)
if

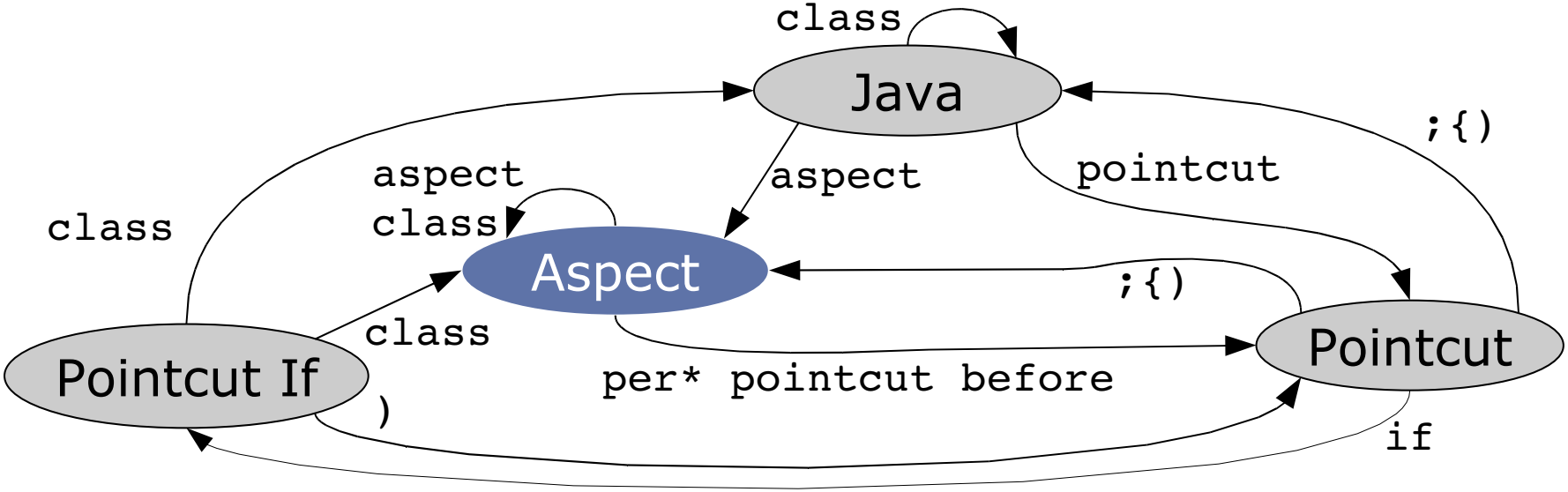
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



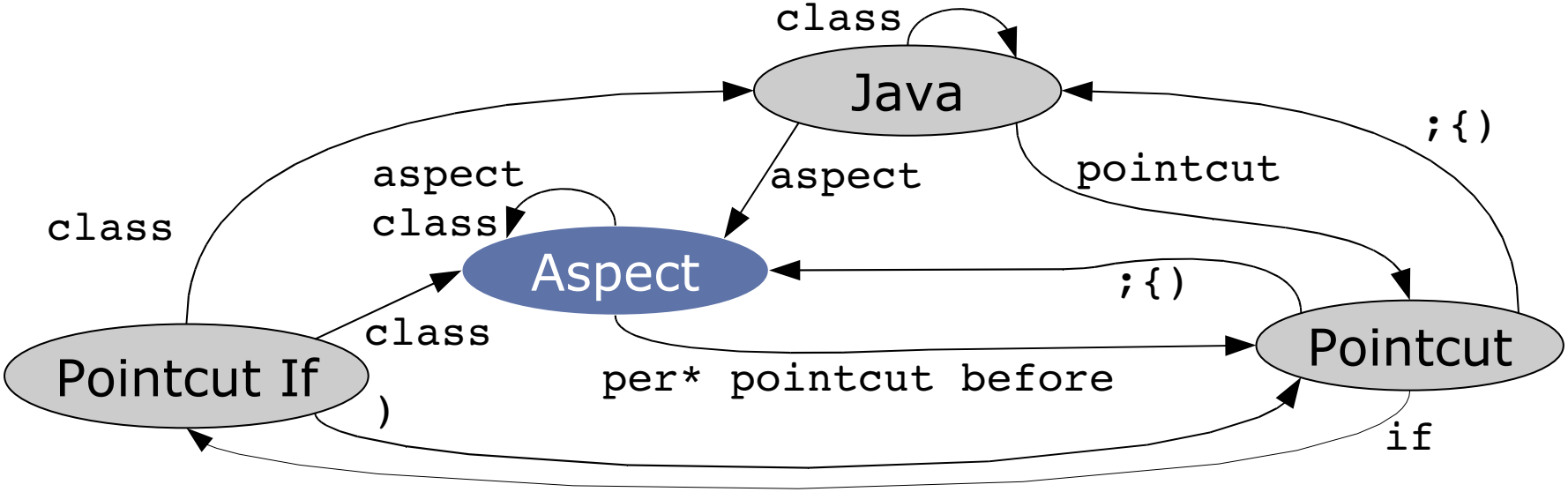
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



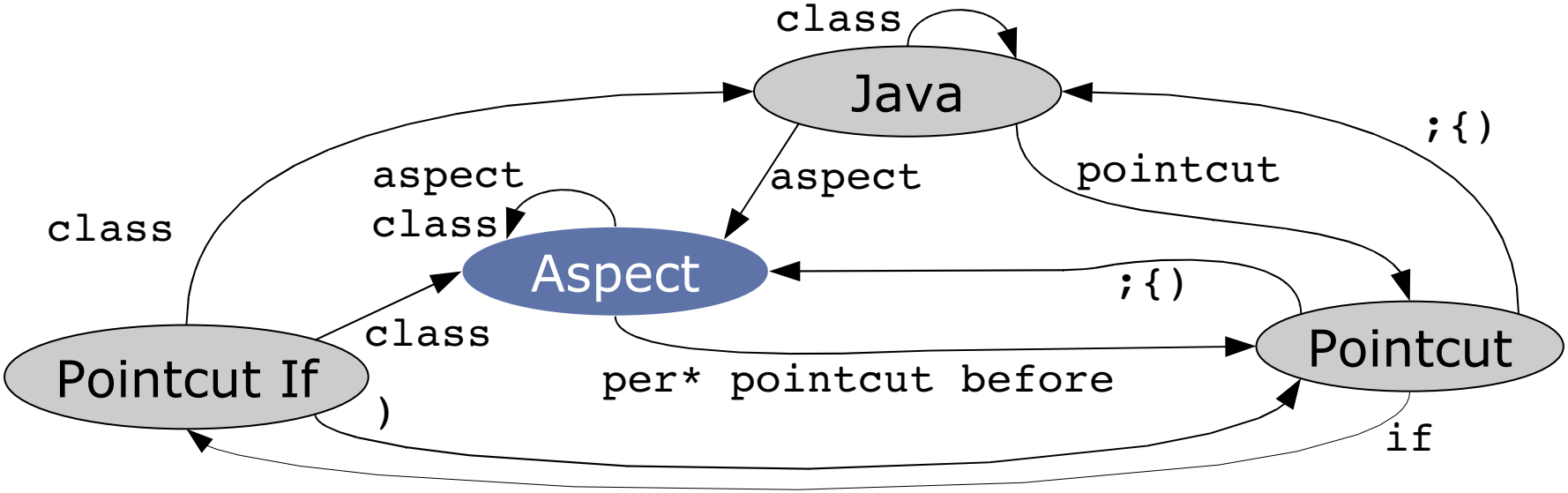
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



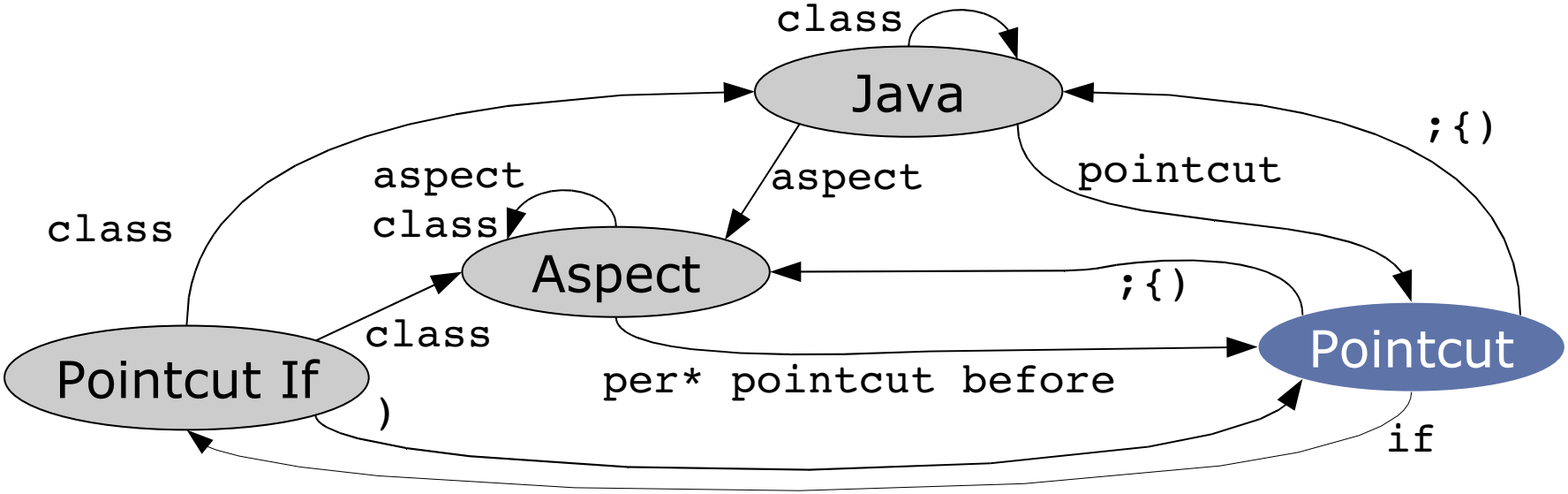
```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



```
public aspect LogFoo {
  pointcut logCalls() : call(* Foo.*(..));
  before() : logCalls() && if(isEnabled()) {
    System.out.println("Enter " + thisJoinPoint);
  } ...
```



Scannerless parsing elegantly deals with these issues

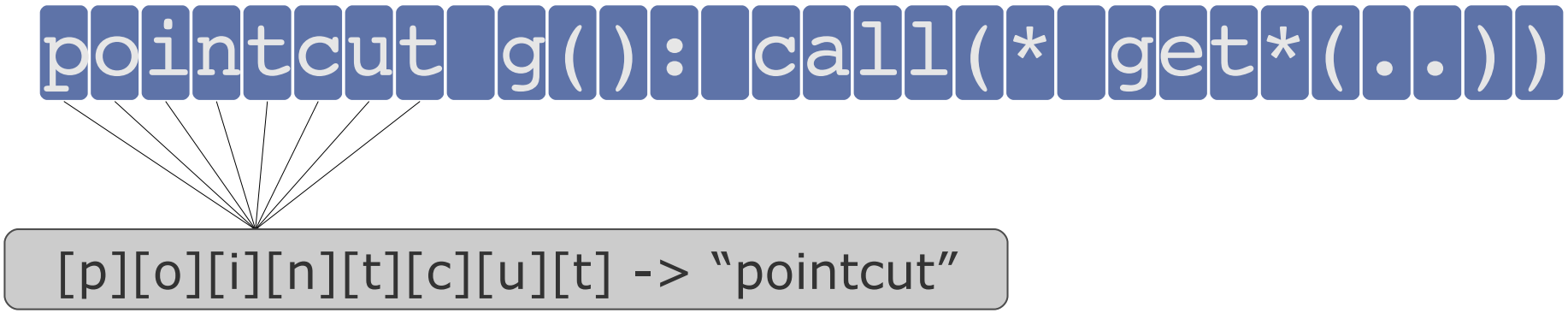
- no separate lexical analysis
- parser operates on individual characters
- scannerless parsing needs generalized LR parsing

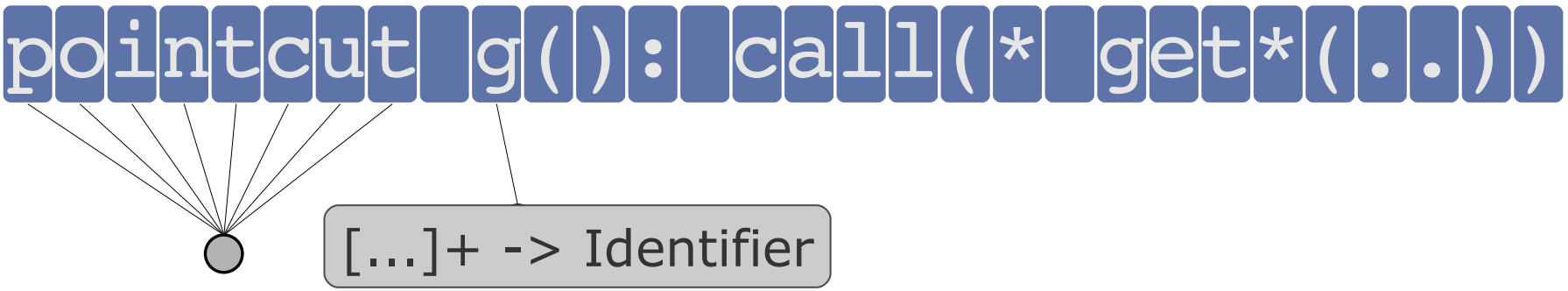
SDF – Syntax Definition Formalism

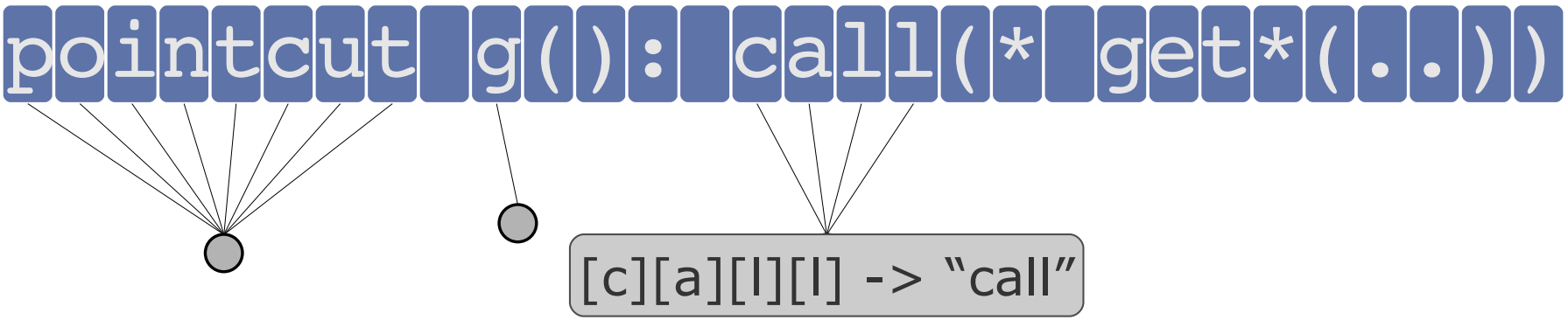
- declarative
- modular
- context-free and lexical syntax
- declarative disambiguation
- all context-free grammars

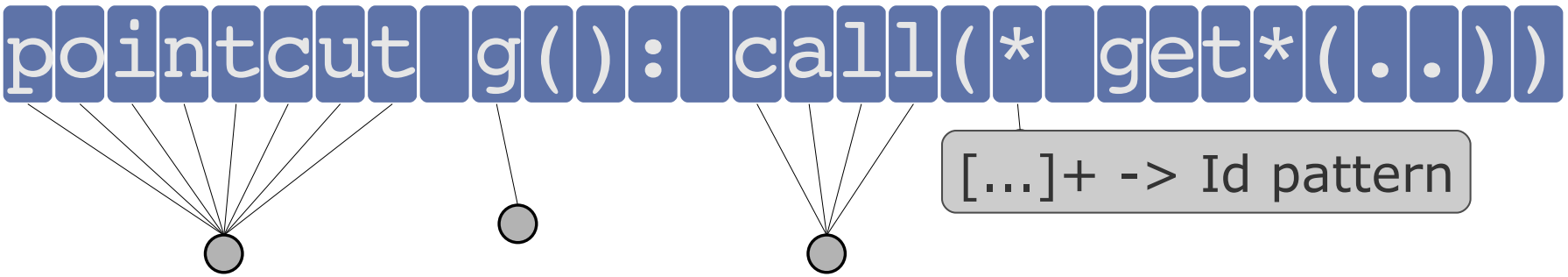
```
pointcut g(): call(* get*(..))
```

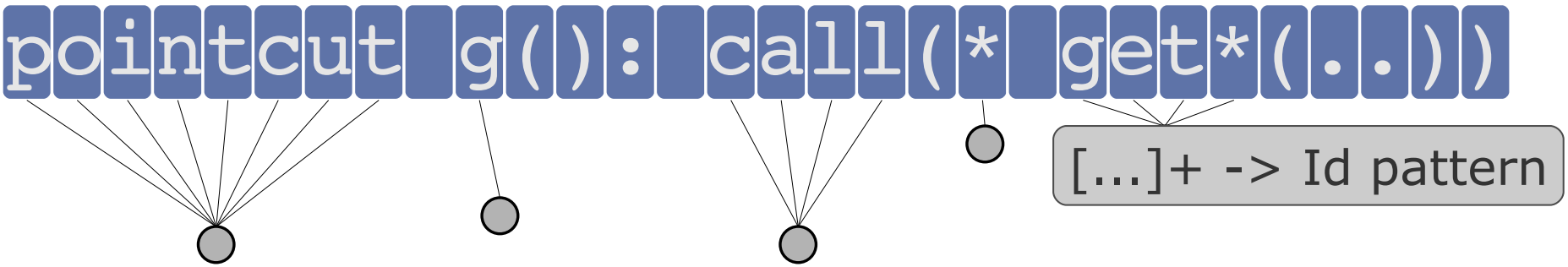
```
pointcut g(): call(* get*(..))
```

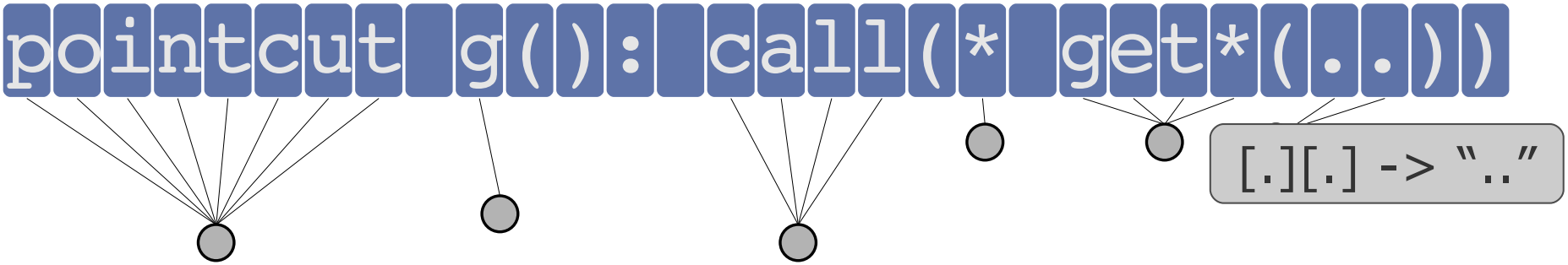

pointcut g(): call(* get*(..))
[p][o][i][n][t][c][u][t] -> “pointcut”

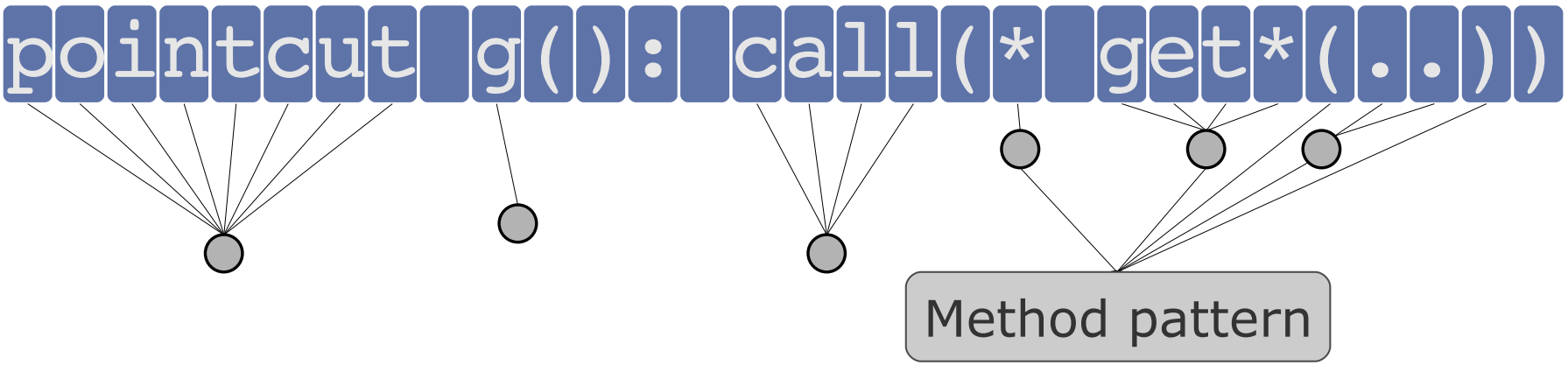

pointcut g(): call(* get*(..))
[...]+ -> Identifier

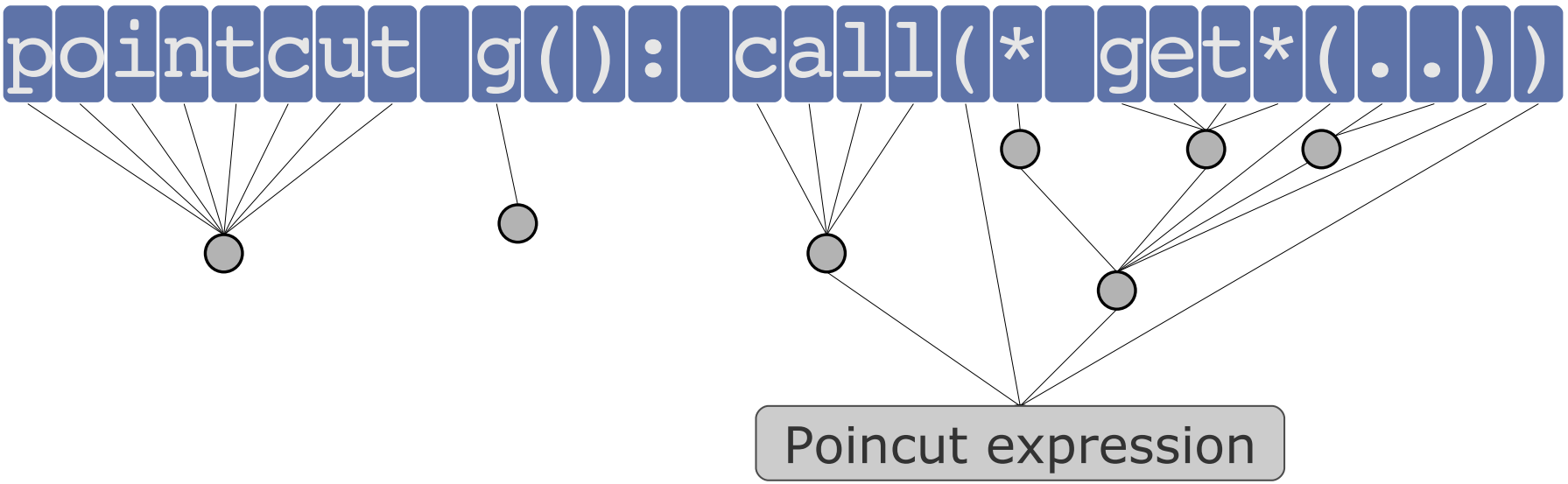

pointcut g(): call(* get*(..))
[c][a][l][l] -> “call”

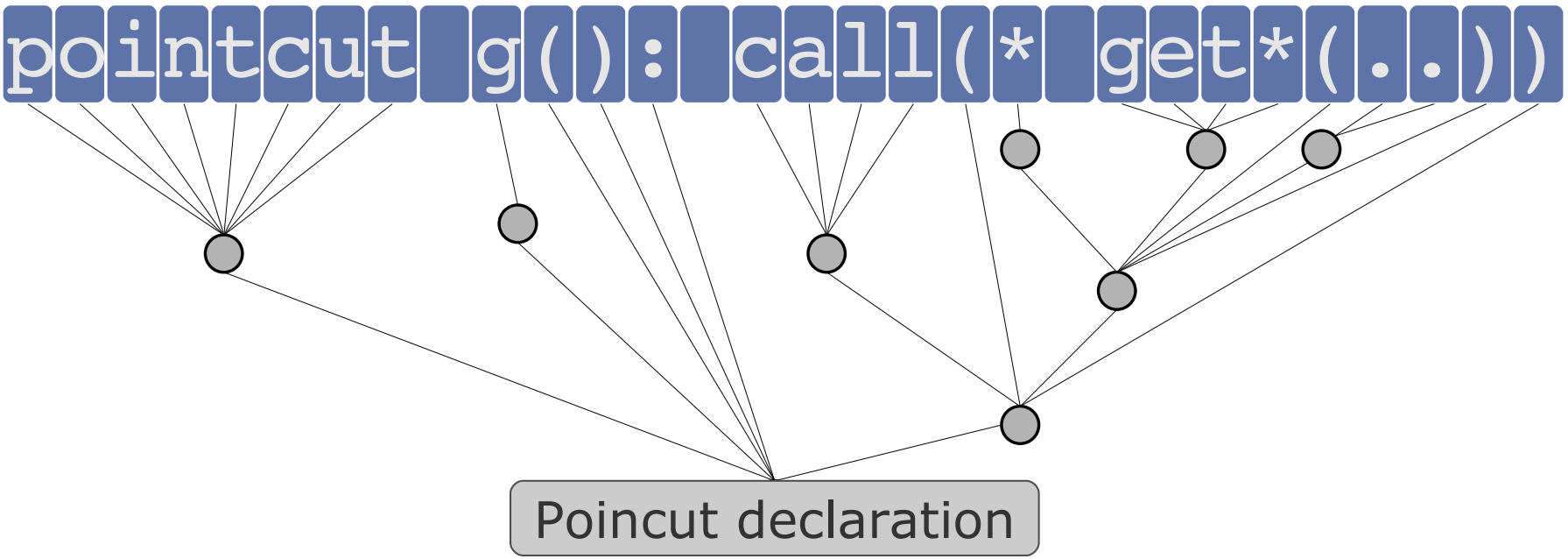

pointcut g(): call(* get*(..))
[...]+ -> Id pattern


pointcut g(): call(* get*(..))
[...]+ -> Id pattern


pointcut g(): call(* get*(..))
[.][.] -> “..”


pointcut g(): call(* get*(..))
Method pattern


pointcut g(): call(* get*(..))
Poincut expression


pointcut g(): call(* get*(..))
Poincut declaration

Part VI

Applications

```
entity-to-java-Entity :
  Entity(x_Class, x_super, props, functions) -> JavaFile(["src"],
  compilation-unit|[
    package pkgname;

    import java.util.*;
    import org.jboss.seam.persistence.*;
    import javax.persistence.*;

    @Entity
    public class x_Class extends x_super implements Serializable {
      public x_Class () {}

      public boolean equals(Object o) {
        return o != null && o instanceof x_Class
           && ((x_Class)o).getId().equals(getId());
      }
    }
  ]|)
```

```
email-to-xml(|call_from_page) :
  Define([Email()], x, fargs, elems1) ->
    XmlFile(["view"], <concat-strings>[template_name, ".xhtml"],
      %>
        <?xml version="1.0" encoding="UTF-8"?>

        <m:message xmlns="http://www.w3.org/1999/xhtml"
                   xmlns:m="http://jboss.com/products/seam/mail"
                   xmlns:h="http://java.sun.com/jsf/html">
          <%= elems3 ::*%>
        </m:message>
      <%)
```

```
php-prod2literal-method-dec(|grammar, cfg) :
  sym@lex(_) -> php:class-member |[
    public static function x($s) {
      if(!is_string($s)) {
        throw new Exception("Only strings ...");
      }

      $result = new y($s);
      $result->symbols[e] = true;
      return $result;
    }
  ]|
  where
    mod := <conc-strings> ("//", <write-to-string> sym)
    ; x := <conc-strings> ("literal_", <key-of-sym-sdf> sym)
    ; y := <php-class-name-string>
    ; e:= <write-to-string; string-to-php-asfix-string> sym
```

Stratego in Stratego

```
Desugar :
  |[ s => t ]| -> |[ s ; ?t ]|

Desugar :
  |[ <s> t :S]| -> |[ !t ; s ]|

Desugar :
  |[ f(as) : t1 -> t2 where s ]|
    ->
  |[ f(as) = ?t1 ; where(s); !t2 ]|

Desugar :
  |[ if s1 then s2 else s3 end ]| -> |[ where(s1) < s2 + s3 ]|
```

Transformation of Stratego programs in Stratego

```
ra2mathml(s) =
  !%>
    <?xml version="1.0"?>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <% s %>
    </math>
  <%

ra-to-presentation-mathml(x) :
  Project(ps, r) ->
    %><mrow>
        <msub>
          <mo>&pi;</mo>
          <mrow>
            <mfenced open="" close="">
              <% <map(x)> ps :: * %>
            </mfenced>
          </mrow>
        </msub>
        <mfenced><% <x> r %></mfenced>
      </mrow><%
```

```
context-free syntax
  "type" "|[" Type "]|" -> MetaExpr {cons("ToMetaExpr")}

variables
  "e" [0-9]*     -> Expr          {prefer}
  "e" [0-9]* "*" -> {Expr ","}*   {prefer}
```

Assimilation rules for Eclipse JDT Core API

```
  Assimilate(r) :
    type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

  Assimilate(r) :
    |[ y(e*) ]| -> |[
      {| MethodInvocation x = ast.newMethodInvocation();
         x.setName(ast.newSimpleName("~y"));
         bstm* | x |}
    ]|
    where <newname> "inv" => x
        ; <ExplodeArgs(r | x)> e* => bstm*
```

```
expr-to-box :
  Plus(b1, b2) -> H hs=1 [ b1 "+"  b2]

UglyPrint :
  If(b1, b2, b3) ->
    V vs=0 [
      H hs=0 [KW["if"] "(" b1 ")"]
      b2
      KW["else"] b3]

PrettyPrint :
  If(b1, b2, If(b3, b4, b5)) ->
    V vs=0 [
      H hs=0 [KW["if"] "(" b1 ")"]
      b2
      H hs=1 [KW["else"] H hs=0 [KW["if"] "(" b3 ")"]]
      b4
      KW["else"] b5]
```

```
function webpage(title : elt, body : elt) : elt =
  <html>
    <head>
      <title> $title </title>
    </head>
    <body>
       $body
    </body>
  </html>
```

Embed XML in a programming language

Part VII

Conclusion

What stops us from applying this immediately?

- limitations of parsing techniques
  - not interactive
  - not Java
  - no error recovery
  - poor error reporting
  - performance could be better
  - no unicode support
  - context-sensitive languages
- poor grammar engineering practices
  - need high-quality grammars
  - reliable migration of grammars
  - testing

Extend meta language with object language syntax

- combine arbitrary meta and object language
- meta-language does not need to be designed for this purpose
- limitation: language should have context-free syntax

Modular syntax definition is essential

- reuse existing syntax definitions – no changed needed

Scannerless Generalized LR parsing

- Only context-free grammars closed under composition
- No compositionality: regular grammars, LR, LL