# Semantics Driven Disambiguation: A comparison of different approaches

Akim Demaille, Renaud Durlin, Nicolas Pierron, Benoît Sigoure

*EPITA Research and Development Laboratory (LRDE)*
*14-16, rue Voltaire - FR-94276 Le Kremlin-Bicêtre Cedex - France*

**Abstract**

Context-sensitive languages such as C or C++ can be parsed using a context-free but ambiguous grammar, which requires another stage, disambiguation, in order to select the single parse tree that complies with the language's semantical rules. Naturally, large and complex languages induce large and complex disambiguation stages. If, in addition, the parser should be extensible, for instance to enable the embedding of domain specific languages, the disambiguation techniques should feature traditional software-engineering qualities: modularity, extensibility, scalability and expressiveness.

We evaluate three approaches to write disambiguation filters for SDF grammars: algebraic equations with ASF, rewrite-rules with programmable traversals for Stratego, and attribute grammars with TAG (Transformers Attribute Grammar), our system. To this end we introduce Phenix, a highly ambiguous language. Its "standard" grammar exhibits ambiguities inspired by those found in the C and C++ standard grammars. To evaluate modularity, the grammar is layered: it starts with a small core language, and several layers add new features, new production rules, and new ambiguities.

*Key words:* Transformers, context-free grammar, attribute grammar, Stratego, ASF, SDF, disambiguation, parsing, program transformation, term rewriting

## 1 Introduction

Of course C and C++ are not ambiguous languages: there exist nonambiguous grammars to analyze them — compilers do that every day world wide. Unfortunately these grammars are not context-free in many ways. Some context dependencies can easily be solved in a deterministic way; for instance, the question whether '`a * b`' is a product or a declaration of a variable `b` of type

pointer-to-`a` can be answered by having the parser maintain a symbol-table that is consulted by the scanner. C++ includes many other forms of ambiguities that require much deeper analysis of the context, and compilers have no choice but to accept temporarily some ambiguities, deferring disambiguation to a later step.

The TRANSFORMERS [2] project aims at producing a high quality flexible C/C++ front-end for either the Stratego/XT toolset, or the ASF+SDF meta-environment. Both require that we use SDF. Since the SDF parser, SGLR, is *scannerless*, the traditional hacks (maintaining a symbol table) do not apply. Contrary to tools such as Yacc, SGLR (Scanner-less Generalized LR) rather than supporting user-actions, it directly builds the parse tree, and therefore, no tricks to implement context-sensitivity apply. Therefore parsing C or C++ typically yields a *parse forest*, several parse trees, and a later stage, *disambiguation*, performs semantics driven context-sensitive analysis to guide the removal of invalid parse trees.

Because of their size and complexity, C and especially C++ are extremely delicate languages to disambiguate. To this end, the TRANSFORMERS project initially used Algebraic Specification Formalism (ASF), then Stratego, and we finally decided to implement TRANSFORMERS Attribute Grammar (TAG), an Attribute Grammars (AGs) engine for Syntax Definition Formalism (SDF) grammars [7]. In this paper, we compare these three approaches.

Small and moderately ambiguous grammars do not stress the implementation enough, so we looked for more realistic test cases. C and C++ are so large and so complex that the authors decided not to consider starting three implementations. Nevertheless, since we are especially interested in C and C++, we reduced their ISO grammars to a more reasonable size. Finally, because we want modular C and C++ front-ends, for instance, in order to be able to add domain specific extensions, modularity is of first importance. Therefore, we designed PHENIX, a family of languages, largely inspired by the features (read "deficiencies") of the C and C++ standard grammars. We used it for the comparison and wrote an ambiguous SDF grammar for this language.

**Contributions** The contributions of this paper are: (1) PHENIX, a family of ambiguous grammars that can be used to compare disambiguation methods. (2) Implementations of its disambiguation in ASF, Stratego, and TAG. (3) A comparison between these methods.

**Outline** The remainder of this paper is organized as follows: In Sect. 2, we present SDF and SGLR and the three environments we used to implement disambiguation filters: ASF, Stratego, and TAG. They are evaluated in Sect. 3 on the various layers of PHENIX. Sect. 4 summarizes the results, and Sect. 5 concludes.

## 2 Semantics Driven Disambiguation for SDF

### *2.1 SDF/SGLR*

**SDF (Syntax Definition Formalism)** SDF [11] is the syntax used by the generic parser SGLR. SDF is intended for the high-level description of grammars for programming languages, application languages, domain-specific languages, data formats and other computer-based formal languages. It features: Modularity (parametrized modules, symbol renaming), Scannerlessness (integrated lexical and context-free syntax), Declarative disambiguation constructs (priorities, associativity, and more), Regular expression shorthands.

**SGLR (Scanner-less Generalized LR)** SGLR [12] supports any context-free grammar. It implements generalized LR parsing [10], a parsing method supporting nondeterminism and ambiguity, local or global. For an ambiguous input it yields a parse forest, i.e., all the possible derivation trees.

### *2.2 Disambiguation methods*

In this paper, we compare three systems to implement disambiguation: ASF, Stratego and TAG. Because of space constraints, we can only describe them superficially, see the references for more detailled presentations.

**ASF (Algebraic Specification Formalism)** ASF [4] is intended for the high-level, modular, description of the syntax and semantics of computer-based formal languages. It supports conditional rewrite rules and traversal functions using a user-defined (concrete) syntax. It allows the concise specification of program transformation, therefore it is suitable for semantics driven disambiguation, as described by [3].

**Stratego** The Stratego language [5] provides rewrite rules in concrete syntax to express basic program transformations using of the syntax of the object language, programmable rewriting strategies to control the application of rules, and dynamic rewrite rules to capture context-sensitivity, thus supporting the development of transformation components at a high level of abstraction.

**TAG (Transformers Attribute Grammar)** AGs [8] express syntax-driven specifications of the semantics of context-free languages. *Attributes* hold values that are attached to symbols of the grammar. Semantic rules are bound to production rules, and they express only *local* computations: they relate attributes of the symbols of the production only. Attributes computed using values from the parent are *inherited*, whereas those using values from the children are *synthesized*. At run-time, an evaluator computes the (global) order in which attributes can be computed for the whole parse tree.

TAG, the implementation used in TRANSFORMERS [7,9], embeds Stratego code in the SDF grammar in order to compute the values. Attributes are

arbitrary terms (integers, strings, trees, tables...). TAG features automatic attribute propagation that frees the programmer from providing semantic rules to forward some values from their definition site to their possibly very distant uses.

## 3 Disambiguation of the Phenixes

This section introduces the layers of PHENIX and their disambiguation by the three methods. The first layer is especially detailed to present how disambiguation (i.e., the removal of invalid derivation trees) is performed in the three frameworks. But because of space constraints only the most significant results are presented. Knowing the three tool-sets is probably needed to fully understand the code.

### *3.1* PHENIX *1 — Declaration and use*

This module lets us declare *int* and *float* variables *and gives them different lexical categories*. This ambiguity does not exist as such in C or C++, but it is a stripped down version of the kind-of-the-identifier problem: in '`a * b`', is `a` a type, or a variable? Besides, disambiguating C++ templates requires type-checking, which is exactly what disambiguating PHENIX 1 requires:

```
int foo;  foo;
```

The ambiguity is when `foo` is used; its declaration must be remembered.

**ASF** The original SDF grammar must be extended with rules to be able to define the symbol table, and to enable concrete syntax in ASF equations (Fig. 1.a). The disambiguation module consists in equations that perform our own traversal (default traversals do not fit to our needs) and equations that perform disambiguation (Fig. 1.b).

The first ASF equation of Fig. 1.b is used when an `int` is declared. It stores the information that the identifier `Id` is an `int` in the symbol table `env`. The second equation is called when an identifier is used. If an `int` is found in the symbol table, the ambiguous node is rewritten into an `int` node. Similar equations are defined for `float` declaration and use.

**Stratego** A set of filters (Fig. 1.c) rewrite the parse forest into a parse tree, resolving the ambiguities using specified traversals and transformations. Dynamic rules are used to keep the necessary knowledge about the declared symbols. The main strategy, `core-disamb`, is a combination of other strategies:

- `decl` to create a dynamic rule when an identifier is declared. The first part, '`?VarDecl(IntType(), x)`', tries to match an `int` declaration. If it succeeds, a dynamic rule is created.

```
imports
  containers/Table[Id Kind]

exports
  context-free syntax
    Table[[Id, Kind]] → Env
    ("int" | "float") → Kind
  variables
    "S"[0-9\']*       → Stm
    "S*"[0-9\']*      → Stm*
    "TU"[0-9\']*      → TypeUse
    "TU*"[0-9\']*     → {TypeUse ","}*
    "env"[0-9\']*     → Env
```

(a) ASF variables

```
equations
  [env-int]
    store-env(int Id;, env) = store(env, Id, int)

  [use-int]
    Id := Int,
    lookup(env, Id) == int
    ========================================
    disamb(amb(TU*1, Int, TU*2), env) = Int
```

(b) ASF equations

```
module core-disamb
strategies
 decl = ?VarDecl(IntType(), x)
      ; rules(use:+ amb(as) → t where <getfirst(?IntUse(x))> as
          ⇒ t)

 core-disamb = decl <+ use <+ all(core-disamb)
```

(c) Stratego

```
TypeDecl Id  → Decl
  {attributes(disamb:
    root.lr_table_syn :=
      ![(Id.string, TypeDecl.type) | root.lr_table_inh]          )}

Id           → Int
  {attributes(disamb:
    root.ok := <lookup(Id.string)> root.lr_table_inh ⇒ Int()    )}
```

(d) TAG

Fig. 1. Disambiguation of Phenix 1 in ASF, Stratego, and TAG.

- `use` the dynamic rule. Rewrite an ambiguous node, '`amb(as)`', to an `IntUse` if one of its children is an `int`.
- `all` to apply a strategy to all the children of the current term.

Even though Stratego supports specification of patterns in the concrete syntax of the object language, this feature is not employed because Phenix is too ambiguous. Each construction should be prefixed to make it clear to which node it refers. In fact, the matching with the abstract syntax tree is simpler.

**TAG** Attributes are embedded in SDF using the `attributes` annotation. In Fig. 1.d, a single value is computed in the `lr_table_syn` attribute of the root node (`Decl`). It adds '`(Id.string, TypeDecl.type)`' to the symbol table. `Id.string` is the identifier name and `TypeDecl.type` is the type of the variable. For attributes automatically propagated from left to right (`lr_`), the suffixes (`_inh`/`_syn`) distinguish incoming/outgoing values. This rule adds the identifier's name to the *inherited* symbol table, mapped to a `Type`, and puts it in the *synthesized* table.

To filter ambiguous nodes using AGs, an attribute, `ok`, is used to store the validity of a node with a Boolean value. The nodes which are not `ok` are removed from the tree after the attribute processing is done. Thus, the disambiguation step needs to adjust the `ok` attributes for some nodes, depending on a symbol table, in order to prune the invalid derivations. The semantic rules of Phenix 1 require that identifiers used as integers have been declared with `int`; this is what asserts the second attribute rule in Fig. 1.d. If the `Id` was declared as an `Int`, the lookup succeeds and the attribute `ok` is true otherwise it is false.

### *3.2* Phenix *2 — Scope*

This first extension adds scoping: a variable name may denote different bindings. Declarations made in a scope must be discarded when leaving it. In the following example, the innermost use of *foo* is bound to the `float` declaration, whereas the outermost *foo* is an `int`.

```
int foo;   { float foo; foo; }   foo;
```

**ASF** To restore the previous symbol table when leaving the scope, first the `disamb` equation is called on the scope to get the corresponding disambiguated tree. Then the equation is called on the following nodes with the same symbol table as before the scope (Fig. 2.a).

**Stratego** Stratego's *scoped dynamic rules* [6] are used to handle scopes (Fig. 2.b). If the current node is a scope, '`?Scope(_)`', '`all(disamb)`' applies `disamb` to all its children. The rule is *scoped* (with '`{|`' and '`|}`'): all dynamic rules named `use` created inside this strategy will be discarded at the end of the strategy.

```
equations
  [scope]
    S*1' := disamb(S*1, env)
    ========================================
    disamb({ S*1 } S*2, env) = { S*1' } disamb(S*2, env)
```

(a) ASF

```
module scope-disamb
strategies      enter-scope = ?Scope(_) ; {| use: all(disamb) |}
```

(b) Stratego

```
"{" Stm+ "}"   → Stm
  {attributes(disamb:   root.lr_table_syn := !root.lr_table_inh  )
      }
```

(c) TAG

Fig. 2. Disambiguation of Phenix 2 in ASF, Stratego, and TAG.

**TAG** When leaving a scope the symbol table must be restored: the synthesized table is equal to the inherited table (Fig. 2.c).

### *3.3* Phenix *3 and 4 — Namespace and Structure*

The second extension adds namespaces (named scopes). The third extension, which adds structures, introduces the same ambiguities because all members inside a structure are static so the notation '::' can be used.

```
namespace A
{
  int foo;
  foo;
}
A::foo;
```

These extensions combine the two previous problems:

- When using the notation with '::', the grammar cannot make the difference between an `int` and a `float`.
- A declaration can hide a previous one. Outside of the namespace, the prior type must be retrieved.

It also adds another ambiguity: when the parser sees something like the following example, it must not only check whether *foo* is an `int` or a `float` but also whether `S` is a `namespace` or a `structure`.

```
struct S { int foo; };
S::foo;
```

```
equations
  [namespace]
    <S*1', ns' env'> := disamb(S*1, Id::ns env),
    ========================================
    disamb(namespace Id { S*1 } S*2, ns env) =
      namespace Id { S*1' } disamb(S*2, ns env')

  [struct]
    <S*1', ns' env'> := disamb(S*1, Id::ns env),
    ========================================
    disamb(struct Id { S*1 }; S*2, ns env) =
      struct Id { S*1' }; disamb(S*2, ns env')
```

(a) ASF

```
module namespace-disamb

strategies
  enter-namespace(|ns) = ?Namespace(n, _) ; all(disamb(| [ns | n])
      )
  enter-struct(|ns)    = ?Struct(n, _)    ; all(disamb(| [ns | n])
      )
```

(b) Stratego

```
"namespace" Id "{" stm:Stm+ "}"  → Stm
  {attributes(disamb:
    stm.lr_ns_inh   := ![Id.string | root.lr_ns_inh]          )}

"struct" Id "{" stm:DeclStm* "}"  → Decl
  {attributes(disamb:
    stm.lr_ns_inh   := ![Id.string | root.lr_ns_inh]          )}
```

(c) TAG

Fig. 3. Disambiguation of Phenix 3&4.

**ASF** The current namespace name must be kept in addition to the symbol table. Unfortunately this requires to update all previous equations to reflect this change. The first equation in Fig. 3.a appends the namespace name (`Id`) to the current namespace name (`ns`). The second equation addresses the `structs`.

**Stratego** As for ASF, this extension is very intrusive because all the previous strategies must be updated to take a term as parameter, the namespace name for disambiguation purpose. This term is updated when entering in a namespace or in a structure (Fig. 3.b). At the beginning of a namespace or a structure, the name (`n`) is concatenated with the current namespace name (`ns`) to obtain the new namespace name.

**TAG** Another attribute (called `ns`) is added to store the current namespace name. All the previous code must be updated to use this new attribute. When entering in a namespace, the new namespace name becomes the con-

```
module typedef-disamb

strategies
  typedef(|ns) = ?Typedef(type, x)
               ; rules(get-type:+ amb(as) → (type, t)
                       where <getfirst(?TypedefUse(x))> as ⇒ t)
               ; rules(use:+ amb(as) → t
                       where <getfirst(?TypedefUse(x))> as ⇒ t)
```

(a) Stratego

```
"typedef" td:TypeDecl Id → Decl
  {attributes(disamb:
    root.lr_table_syn :=
      ![Typedef(td.lr_ns_syn, td.type) | td.lr_table_syn]     )}
```

(b) TAG

Fig. 4. Disambiguation of PHENIX 5 in Stratego, and TAG.

catenation of the name of the new namespace (`Id.string`) and the current namespace name (`root.lr_ns_inh`) (Fig. 3.c). This attribute is used similarly when entering a structure.

### *3.4* PHENIX *5 — Typedef*

A `Typedef` declaration introduces a name that, within its scope, becomes a synonym for the given type. In order to be able to find the type of *x*, the disambiguation must check which type *t* denotes:

```
typedef int t;          t x;          x;
```

**ASF** From this extension, the construction requires complex traversals and complex manipulations of the structure used by the disambiguation process. The disambiguation of remaining extensions is made only with Stratego and TAG.

**Stratego** When a typedef is seen, two dynamic rules are created. The first one holds the type of a variable. This rule returns a tuple, the type of the variable and its name. The second rule is used when an ambiguous node is rewritten into the good sub-tree (Fig. 4.a).

**TAG** With AGs, the disambiguation process uses the symbol table to store all the information needed to disambiguate `Typedef` (Fig. 4.b). The constructor '`Typedef()`' is used to remember the data type. Two parameters are used to store the current namespace name (this attribute also contains the name of the variable) and the type associated to the typedef.

```
module using-disamb

strategies
  using(|ns) = ?UsingNs(n)
  ; {|get-access:
      all(disamb(|ns))
    |}
  ; rules(get-access := <conc> ([ns | n], <get-access>))

  using(|ns) = ?Using(n)
  ; {|get-access:
      all(disamb(|ns))
    ; get-type ⇒ (type, n)
    ; rules(use:+ amb(as) → t where <getfirst(?type#([n]))> as ⇒
        t)
    |}
```

(a) Stratego

```
"using" "namespace" NamespaceName ";"    → Stm
 {attributes(disamb:
   local.access          :=
     <lookup> (root.lr_ns_inh, root.lr_table_inh) ⇒ Namespace(<id
         >)
   ; <conc> (NamespaceName.lr_access_ns_syn, <id>)

   root.lr_access_ns_syn :=
     <conc> (local.access, root.lr_access_ns_inh)

   root.lr_table_syn     :=
     ![(root.lr_ns_inh, Namespace(local.access) | root.
         lr_table_inh])}

"using" TypeUse ";"                      → Stm
 {attributes(disamb:
   local.type :=
     <lookup> (TypeUse.lr_ns_syn, root.lr_table_inh)

   root.lr_table_syn     :=
     ![(root.lr_ns_inh, local.type) | root.lr_table_inh]
                   )}
```

(b) TAG

Fig. 5. Disambiguation of Phenix 6 in Stratego, and TAG.

### *3.5* Phenix *6 — Using and Using namespace*

This extension allows importing (parts of) a namespace into another one. In the example below, after '`using namespace A`', *`foo`* and *`bar`* are reachable. After '`using B::baz`', only *`baz`* is reachable (not *`qux`*).

```
namespace A { int foo; int bar; }
namespace B { int baz; int qux; }
using namespace A;
using B::baz;
```

**Stratego** With this extension, another dynamic rule (*get-access*) must be added to keep a list of accessible namespaces. This rule needs to be used everywhere the Stratego code uses the other dynamic rules to find the type of a variable. This extension leads to an update of all the previous code.

Fig. 5.a (first strategy) illustrates how the *get-access* dynamic rule is created. First, the node inside the using namespace is disambiguated, then the dynamic rule is spawned with a new value for *get-access*. The current namespace name is added to the list of accessible namespaces.

Fig. 5.b (second strategy) describes what the disambiguation does when a *using* is seen. The type of the variable inside the using is extracted thanks to the dynamic rule *get-type* that was previously defined.

**TAG** This extension is very intrusive: many changes are necessary. The attribute `ns` does not suffice to know whether a variable is reachable. A new attribute, `access_ns`, stores a list of reachable namespaces. This attribute must be added and kept up-to-date in all the rules.

The first production rule in Fig. 5.b outlines how the attribute `access_ns` is updated and stored in the symbol table. The local attribute `access` extracts the current `access_ns` from the symbol table and concatenates it with the `access_ns` of the namespace declared in the '`using namespace`'. Then the result is added to `root.lr_access_ns_syn`, the list of reachable namespaces, and put in `root.lr_table_syn`, the symbol table.

A '`using namespace`' extends the attribute adding a new reachable namespace in the list. When the disambiguation process checks if a variable is reachable, it no longer uses the `ns` attribute. the variable is looked up in all the namespaces in `access_ns`.

The second production rule in Fig. 5.b presents the disambiguation of `using`. The local attribute `type` extracts the type of the variable used in the `using` from the symbol table. It is then stored in the symbol table.

### *3.6* Phenix *7* — *Template*

This extension adds a simplified version of C++ templates with only one parameter. Template specialization is also added:

```
// S<T> is a container of elements of type T.
template <T> struct S { T x; };
S<int>::x;

// S<float> has a special definition.
template <> struct S<float> { float x; };
S<float>::x;
```

**Stratego** The first *template* strategy in Fig. 6.a is used when a template declaration is seen. The rule *add-var* creates a dynamic rule in order to be able to retrieve the tree associated to the declaration. This is done because

```
  instantiate-template =
    ?spe@Instance(tn, ptd)
  ; get-var(|tn) ⇒ (ns, TemplateDecl(tree))
  ; !tree ⇒ Template(param, _, _)
  ; <add-var> (param, <data-var>)
  ; <add-nsn> (param, TypedefType(<data-nsn>))
  ; <specialized-tree> (tree, ptd)
  ; register-specialization
  ; <remove-var> param
  ; <remove-nsn> param

  template(|ns) = ?tree@Template(param, tn, decls)
  ; where(<add-var> (tn, TemplateDecl(tree)))

  template(|ns) = {|get-access:
    instantiate-template ⇒ (ns, TemplateInst(access))
  |})
```

(a) Stratego

```
"template" "<" tn:Id ">" "struct" sn:Id "{" stm:DeclStm* "}"  →
    Decl
 {attributes(disamb:
  local.object_ns   := ![sn.string | root.lr_ns_inh]
  local.tree        := id
  root.lr_table_syn := ![(NotSpecialized(local.object_ns), local.
      tree)
                        | root.lr_table_inh]
  root.ext_table    := extern
                                          )}

Id "<" TypeDecl ">"   → Struct
 {attributes(disamb:
  local.object_ns := ![Id.bu_string | root.lr_ns_inh]
  root.lr_table_syn :=
    <lookup> (NotSpecialized(local.object_ns), root.lr_table_inh)
  ; <set-attribute>(<id>, "disamb", "ext_table", root.lr_table_inh
      )
  ; attr-eval
                                                          )}
```

(b) TAG

Fig. 6. Disambiguation of Phenix 7 in Stratego, and TAG.

the disambiguation process must be delayed until all the needed information (the type of the parameter) is known.

The second *template* strategy is used when a template is instantiated. This strategy calls *instantiate-template* that retrieves the tree previously stored. The tree is then used to instantiate the template and creates a specialized version of this template.

**TAG** As the type is not fully known at the declaration site, the evaluation must be delayed until the template is instantiated, that is to say, until all the needed information (the type of the parameter) is known. The sub-tree corresponding to the template is kept in the symbol table (Fig. 6.b).

When the template is instantiated, the system has enough information to disambiguate the template. The type of the parameter is added to the symbol table and then the attribute evaluator is called with the template sub-tree. The attributes are computed and the symbol table is filled with the variables corresponding to the instantiation.

## 4 Discussion

ASF provides tight coupling with concrete syntax, supports functional and algebraic specification, and a fast and scalable underlying term rewriting engine. Using equations to describe term rewriting leads to a clean formalism. Unfortunately, many equations have to be written with ASF, since the traversals must be explicitly described, intertwined with the computation itself. Every rewrite rule is potentially executable on any node, there can be hidden interactions between some equations, leading to cycles or unwanted effects. Our initial attempt to disambiguate C++ was with ASF, but it turned out to be exceedingly complex, and the approach was dropped. The CodeBoost project, which is interested in C++ program transformation, faced the same issues and moved from ASF to Stratego [1, Section 8].

Stratego is very expressive. Complex traversals can be expressed easily using the various combination operators and user-defined strategies. Additional flexibility comes with the concept of scoped dynamic rules, which allow to have a global context without explicitly carrying it everywhere. Scoped dynamic rules provide an excellent support for modularity: they traverse transparently unrelated nodes, and directly apply where needed. Of course, when the grammar is extended, one must be careful to make sure that the code can handle it because Stratego modules are not directly linked to the grammar.

AGs are conceptually elegant, being both declarative at the rule level and imperative at the attribute level. The rules are fairly independent from each other. AGs are naturally extended, just like the SDF grammar. Since the disambiguation code is broken down at the rule level, adding new rules tends to fit well into the existing code. Nevertheless AGs by-the-book do not support modularity: if an extension introduces new attributes, then forwarding these attributes in the rest of the grammar may require additional rules. Our TAG system solves this problem, since attributes may be propagated automatically, much like with Stratego's dynamic rules. Using Stratego in order to compute the attributes brings several good features, such as the flexibility of the ATerm format, the Stratego standard library, and more generally the expressiveness of the language.

But like the other methods, TAGs are not perfect. Separation of concerns is hard to achieve. Since the semantical rules are embedded in the grammar, the various processing (e.g., disambiguation, type-checking, etc.) are intertwined

in the same file. TAG provides namespaces, but this is not a full scale solution. TAGs tend to clutter the grammar. The code for the attribute computation is somewhat mixed with the rule declarations and the other annotations. When the code gets long and complicated, the grammar becomes less readable.

Yet we recommand the use of AGs to disambiguate and to type languages. The main reason people use Context-Free Grammar (CFG) grammars is because most context-sensitive grammar formalisms are complex and hard to understand, but most languages are definitely *context-sensitive* (think of type-checking). Disambiguation filters and type checkers are not just "some" program transformation filters: they belong to the very definition of the language itself. As a matter of fact, AGs, a combination of a CFG and some semantic rules, are precisely a means to define *context-sensitive* grammars.

## 5 Conclusion

We have presented the concept of Scanner-less Generalized LR parsing and shown why this kind of parsers requires a disambiguation step. We have introduced Phenix, our own toy language that replicates ambiguities from both C and C++ ISO standards. An ambiguous grammar that generates Phenix language has been written in SDF. We have seen how to perform semantic disambiguation on this language with three different approaches:

- Term rewriting using the ASF which is purely declarative.
- Term rewriting using the paradigm of strategies (Stratego), which is mostly imperative (with a functional flavor).
- TAG, using Stratego code to compute the attributes, are both declarative and imperative.

The modularity of these three methods has been evaluated.

**Acknowledgments** We thank the anonymous reviewers for their comments, Daniela Becker and Steve Frank for their proofreading. The ASF and Stratego/XT communities are very reactive and helped us whenever we needed it. Martin Bravenboer also brought invaluable help during the development of the Transformers project.

## References

[1] Bagge, O. S., K. T. Kalleberg, M. Haveraaen and E. Visser, *Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs*, in: D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)* (2003), pp. 65–74.

URL http://www.stratego-language.org/Stratego/DesignOfTheCodeBoostTransformationSystem

[2] Borghi, A., V. David and A. Demaille, *C-Transformers — A framework to write C program transformations*, ACM Crossroads **12** (2006), http://www.acm.org/crossroads/xrds12-3/contractc.html.

[3] van den Brand, M., S. Klusener, L. Moonen and J. J. Vinju, *Generalized parsing and term rewriting: Semantics driven disambiguation*, Electronic Notes in Theoretical Computer Science **82** (2003).

[4] van den Brand, M. G. J., A. van Deursen, T. B. Dinesh, J. F. T. Kamperman and E. Visser, editors, "Proceedings of the Workshop on Generating Tools from Algebraic Specifications (ASF+SDF'95)," Technical Report P9504, Programming Research Group, University of Amsterdam, 1995.
URL http://ftp.wins.uva.nl/pub/programming-research/reports/1995/P9504/

[5] Bravenboer, M., K. T. Kalleberg, R. Vermaas and E. Visser, *Stratego/XT 0.16. Components for transformation systems*, in: *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)* (2006).

[6] Bravenboer, M., A. van Dam, K. Olmos and E. Visser, *Program transformation with scoped dynamic rewrite rules*, Fundamenta Informaticae **69** (2006), pp. 123–178.

[7] David, V., A. Demaille and O. Gournet, *Attribute grammars for modular disambiguation*, in: *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06)*, Technical University of Cluj-Napoca, Romania, 2006.

[8] Knuth, D. E., *Semantics of context-free languages*, Journal of Mathematical System Theory (1968), pp. 127–145.

[9] Pierron, N., *Formal Definition of the Disambiguation with Attribute Grammars*, Technical report, EPITA Research and Development Laboratory (LRDE) (2007).
URL http://publications.lrde.epita.fr/200706-Seminar-Pierron

[10] Tomita, M., "Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems," Kluwer Academic Publishers, 1985.

[11] Visser, E., *A family of syntax definition formalisms*, Technical Report P9706, Programming Research Group, University of Amsterdam (1997).
URL http://www.wins.uva.nl/pub/programming-research/reports/1997/P9706.ps.gz

[12] Visser, E., *Scannerless generalized-LR parsing*, Technical Report P9707, Programming Research Group, University of Amsterdam (1997).
URL http://www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps.gz