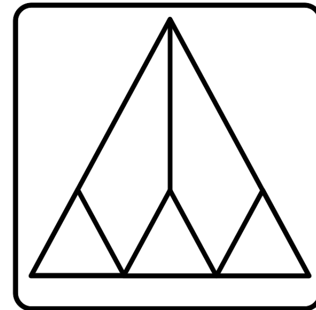
Proceedings of the Sixth Stratego User Days

Karl Trygve Kalleberg
Eelco Visser
(editors)



www.stratego-language.org

Institute of Information and Computing Sciences
Universiteit Utrecht
June 14th, 2005

Copyright © 2005 by the authors of the individual contributions.

Address:
Institute of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
email: karltk@cs.uu.nl
http://www.cs.uu.nl/~karltk/

Preface

These are the proceedings of the Sixth Stratego User Days, which were held on May 3-4, 2005 at Utrecht University. The User Days were preceded by a full day tutorial on May 2. The workshop and tutorial were supported by the Software Technology Group of the Institute for Information and Computing Sciences.

The workshop was attended by:

- Anya Helene Bagge (University of Bergen, Utrecht University)
- Alexandre Borghi (Epita)
- Martin Bravenboer (Utrecht University)
- Valentin David (LIP6, Epita)
- Akim Demaille (Epita)
- Renaud Durlin (Epita)
- Olivier Gournet (Epita)
- Rene de Groot (Utrecht University)
- Magne Haveraaen (University of Bergen)
- Karl Trygve Kalleberg (University of Bergen, Utrecht University)
- Mart Kolthof (Utrecht University)
- Thomas Largillier (Epita)
- Karina Olmos (Utrecht University)
- Nicolas Pouillard (Epita)
- Yaroslav Usenko (Technische Universteit Eindhoven)
- Huanwen Qu (Utrecht University)
- Rob Vermaas (Utrecht University)
- Eelco Visser (Utrecht University)
- Mikal Ziane (LIP6, Université Paris 5)

Utrecht, June 2005

Contents

1 C/C++ Disambiguation Using Attribute Grammars **4**
1.1 Introduction 4
1.1.1 The Transformers Project 4
1.1.2 C++ 4
1.1.3 Semantics Driven Disambiguation 5
1.2 Disambiguating with Attribute Grammars (AGs) 5
1.2.1 Ambiguous synthesis 5
1.2.2 Template instances 5
1.3 Discussion 6
1.3.1 Results 6
1.3.2 Others solutions 7
1.3.3 Further works 7
1.4 Conclusion 7

2 Implementing Attributes in SDF **9**
2.1 Introduction 9
2.2 Current implementation 9
2.2.1 Syntax 10
2.2.2 Evaluator Generation 10
2.2.3 Evaluation 11
2.3 Discussion 11
2.3.1 Related Work 11
2.3.2 Further Work 11
2.4 Conclusion 12

3 Spoofax: An Editor for Stratego **13**
3.1 Introduction 13
3.1.1 Requirements for an Editor 13
3.1.2 The Weave 13
3.2 The Spoofax Editor 14
3.2.1 Syntax Highlighting 14
3.2.2 Outliner 14
3.2.3 Code Completion 15
3.3 Discussion 15
3.4 Conclusion 15

4 ESDF: A Proposal for a More Flexible SDF Handling **16**
4.1 Introduction 16
4.2 Extended SDF (ESDF): An Syntax Definition Formalism (SDF) Chain Robust to New Annotations 17
4.2.1 Packing Modules: `pack-esdf` 17
4.2.2 Filtering Annotations Out: `sdf-strip` 17

4.2.3 Parsing Extended Grammars: `parse-esdf` 17
4.3 The lrde-syntax bundle 17
4.3.1 Pretty-Printing: `boxedsdf` 17
4.3.2 Disambiguation Tags: `sdf-detgen` 17
4.3.3 AG: `sdf-attribute` 18
4.3.4 Flexible Abstract Syntax Tree (AST) generation: `sdf-astgen` 18
4.4 Conclusion 19

Chapter 1

C/C++ Disambiguation Using Attribute Grammars

Valentin David
Akim Demaille
Renaud Durlin
Olivier Gournet

Abstract

We propose a novel approach to semantics driven disambiguation based on AGs. AGs share the same modularity model as its host grammar language, here SDF, what makes them particularly attractive for working on unstable grammars, or grammar extensions. The framework we propose is effective, since a full ISO-C99 disambigution chain already works, and the core of the hardest ambiguities of C++ is solved. This requires specific techniques, and some extensions to the stock AG model.

1.1 Introduction

1.1.1 The Transformers Project

In order to implement a fast and generic image processing library, the EPITA Research and Development Laboratory (LRDE) chose a language that supports several paradigms and never sacrifices speed for features. Back at the end of the 90's, C++ was the only reasonable answer, unchallenged as of today thanks to its incredibly powerful `template` mechanism allowing meta-programs (i.e., compile-time programs). Unfortunately, inheriting from C, a language designed in the early 70's, C++ features a poorly designed syntax and baroque semantics. This is troublesome both to compiler writers (most C++ crunching programs are not compliant and fail to capture the whole language) and to C++ writers. Indeed, `template` intensive programming is error-prone and often incomprehensible. Therefore we considered building a new language with better meta-programming support, a daunting task in itself, and rather focused on means to add syntactic sugar to C++. This prompted for the inception of the Transformers project, aiming at providing a C++ transformation framework, i.e., a modular C++ front-end (and "back-end": a pretty-printer), so as to enable its users to develop transformations.

1.1.2 C++

The C++ programming language is an object oriented extension to C. It is large and complex. It inherits from the old and well known ambiguities of C. For instance, parsing `a * b;` depends on the context: if `a` is a variable name, it denotes a product, and if `a` is a type name, it denotes the declaration of `b`. Many similar ambiguities exist, and their common root is the spirit in which the C++ grammar appears to be written: it is tailored for compilers which parser provide feed-back to the scanner to escape from context-free grammar by being able to distinguish different identifier types (variable name, type name, etc.). Clearly, the C++ grammar was written with implementations in mind, not with the language itself.

C++ also adds ambiguities of its own, even with known identifier kinds. For instance, let `T` be a type, depending on the context `T(a);` denotes either the declaration of the variable `a`, or a call to the constructor `T::T(a)`. According to the standard, the latter is the correct interpretation, unless it cannot be correct in the context.

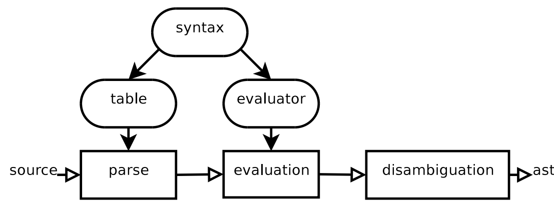


Figure 1.2: Disambiguation process

The `template` keyword comes with its set of specific issues, a single of which is be presented here, Fig. 1.1.

1.1.3 Semantics Driven Disambiguation

Because we aim at using Stratego/XT to write transformations (such as syntactic sugar) and in order to enjoy C++ concrete syntax in Stratego code, we chose to use regular "Stratego/XT parsing techniques" (2): SDF to define a context free super set of the C++ syntax, Scanner-less Generalized LR (SGLR) to parse possibly ambiguous C++ programs and to return an AsFix parse forest, and then a set of disambiguation filters. However our approach relies on AGs to disambiguate: computations similar to type checking are performed on the attributes, revealing the invalid alternatives of ambiguities. These branches are flagged, and pruned in a latter stage. We use a homegrown extension to SDF with attribute rules embedded (1). Fig. 1.2 presents the parsing chain, from source code to (unique) AST.

1.2 Disambiguating with AGs

The formalism of AGs is well defined, and well covered in the literature. Yet little attention was devoted to using *ambiguous* AGs: how can information flow when there is uncertainty. As usual, C++ added on top of this its own issues, requiring an extension to AGs.

1.2.1 Ambiguous synthesis

In attribute grammars, evaluation consists in information flowing across nodes of the Parse Tree (PT), either upwards (*synthesized* attributes), or downwards (*inherited* attributes). Extension of inherited attributes flow in ambiguity nodes is straightforward: the node

```
1 // Is T::t a type, or a value?
2 template <typename T>
3 void g() { int f(typename T::t); }
4
5 // Is T::t a type, or a value?
6 template <typename T>
7 void h() { int f(T::t); }
```

In the definition of `g`, using the keyword `typename`, the programmer declares any actual parameter will define `t` as a type. Conversely, for `h`, the absence of `typename` *requires* the user to provide actual parameters that define `t` as a value. In this specific case (extracting a symbol from a template parameter) the C++ standard mandates explicit disambiguation.

Figure 1.3: The mandatory `typename` disambiguation keyword

simply forwards the unique information flowing downwards to its children. Synthesized attributes are a problem: each children contributes a possibly different value. In our model, if the values are equal, this value is assigned to the ambiguity node, but if they are different, an error is raised and the evaluation stops. None of our current applications required a more subtle policy. In the future, we might consider ambiguity support for attributes, similar to the `amb` node in AsFix. A formal treatment of our extension of AGs to ambiguity remains to be done.

1.2.2 Template instances

C++ `templates` challenge the compiler writers: they are the number one reason for lack of compliance with the C++ standard. There is no exception for AGs: special mechanisms are required to cope with this part of the language.

One issue with parsing (and disambiguating) (class or function) templates is that their context is incomplete. Consider for instance the Fig. 1.3: what is the kind of the symbol `T::t`? It turns out that in this case the standard made provisions to turn this context sensitive problem into context-free thanks to the `typename` keyword.

Therefore template definitions are easily handled. However, template *uses* are much more delicate: they require that templates be (partially) instantiated. Consider for instance the Fig. 1.1: disambiguating lines 7 and 8 require the instantiation of `A` with its actual parameters.

```
1 template <int>
2 struct A { typedef int t; };    // A<i>::t is a type (by default).
3
4 template <>
5 struct A<0> { static int t; };  // A<0>::t is a value.
6
7 int v(A<0>::t);  // Defines a variable since A<0>::t is a value.
8 int f(A<1>::t);  // Declares a function since A<1>::t is a type.
```

The last two lines show the two possibilities for a single ambiguity: `int a(b);` may denote the definition of the variable `a` if `b` is a value (`int zero(0);`), or the declaration of a function if it's a type (`int abs(int);`). To disambiguate, the class template `A` must be instantiated with its actual parameter, since there is no requirement that a symbol (`t`) defined by a class template (`A`) has a constant kind (value or type).

Figure 1.1: Ambiguities on `A<?>::t` need `A` instantiations

```
1 // A<I>::t is A<I-1>::t.
2 template <int I>
3 struct A : public A <I-1> {};
4
5 // A<0>::t is a value.
6 template <>
7 struct A<0> { static int t; };
8
9 // A<14>::t is a type.
10 template <>
11 struct A<14> { typedef int t; };
```

Figure 1.4: Recursive template instantiations

To this end, AGs are troublesome. One simple solution involves two passes: the first pass gathers the actual template parameters (0 and 1 in our example), and the second pass instantiates the template definition with them, finally providing the dependent information (the kind of `A<?>::t`) to disambiguate (`v` is a variable, and `f` a function). Unfortunately because template instantiations can trigger arbitrarily many other template instantiations (Fig. 1.4) this is not possible.

Therefore *the template must be carried from its definition to its uses* together with its set of attribute rules, since they have to be evaluated "on site". This requires a dramatic extension to AGs: part of a PT is a possible attribute value, and a means to fire the evaluation of attributes is needed.

As a consequence, the template definitions are no longer evaluated where they are defined, but where they are used. This is insufficient though: errors in the template definitions must also be caught to comply with standard C++.

There are two options to evaluate template definitions. The simplest solution consists in instantiating the template with fake parameters. Alternatively, one could exploit information gathered from the uses of a template to disambiguate its body. Consider Fig. 1.3, line 5: any use of this template provides an actual value for `T`, hence an explicit definition for `T::t`. This framework would relieve the user from having to use the `typename` keyword to disambiguate by hand template uses.

1.3 Discussion

1.3.1 Results

We developed a complete AG for the ISO-C99 language. Our grammar strictly conforms to the standard in about 340 rules (half of which are lexical) and 90 attributes. Contrary to C++, it has few and easy-to-resolve ambiguities, making it a realistic test bed case. Its development represented about 1-2 month-man. The sheer size of this grammar prompted for the addition of AG syntax extension in order to minimize code duplication. In the long run, when a fully blown redesign of AG in SDF is completed, extensions such as featured by Utrecht University Attribute Grammar System (UU-AG) (3) will be implemented. Performances are, as expected from a naive implementation, poor: disambiguating `stdio.h` takes 75s on a 3GHz microprocessor.

The extension of this grammar to support GNU C is straightforward, and requires no modification to the baseline grammar. C++ is a much more challenging grammar to tackle (560

rules and more and much harder ambiguities). As a proof of concept, a mini C++ grammar was developed to include all the difficult aspects of C++ disambiguation (virtually everything related to templates). Its extension into a disambiguating AG is completed, which supports our claim that AG can disambiguate C++.

1.3.2 Others solutions

We propose the use of AGs to perform semantics driven disambiguation, but other techniques have been applied with success.

In (4), the authors use **ASF+SDF** to disambiguate ambiguous parse trees. Nevertheless, according to our initial experiments, this approach is delicate to extend to a fully blown language. It remains yet to be proved that templates can be properly handled.

Our first experiments using **Stratego** did not use the recently introduced dynamic rules, and therefore the code was entangled with scopes and tables processing. In addition, because a primary motivation for the inception of the Transformers project is to ease the implementation of C++ grammar extensions, we looked for seamless modularity. Embedding the disambiguation specifications in the grammar provides modularity for free.

1.3.3 Further works

ASF+SDF, Stratego, and AGs provide three different means to write elegantly and concisely disambiguation filters. Because of template instantiation, C++ challenges these techniques, and a thorough comparison between the three paradigms is underway (5). Once completed, this comparison will address a small subset of C++ containing the following features: modularity by modeling C++ as an extension of C, templates to mandate instantiations (comparable to extending the PT during its traversal), namespaces to introduce named scopes, and context sensitivity (by introducing two kinds in C, `union` and `typedef`, and a third for C++, `class`).

A formalization of our extensions to stock AGs remains to be done. Yet our model is still slightly changing, tailored to ease the implementation of disambiguation filters. For instance, it is considered to allow the evaluator to prune incorrect branches, sort of a `cut`, instead of merely flagging them as incorrect. Early experiment show a small speedup, but significant simplifications of attribute rules.

Finally, to implement actual transformations we wish to use C++ concrete syntax in Stratego, what prevents C++'s intrinsic ambiguity. We are toying with the idea of using AGs to disambiguate embedded languages. A successful early experiment allows the programmer to disambiguate C in Stratego by hand as follows.

```
mytest = ?|Expression[ (i0) (i1) ]|
            with i0 => "typedef",
                 i1 => "variable"
```

1.4 Conclusion

We demonstrated how (ambiguous) AGs can be used to perform semantics driven disambiguation. The disambiguation of difficult languages demonstrate the validity of the approach: C99 is fully covered, and the most delicate parts of C++ have been solved. It is ongoing work to address the full language. AGs are modular and extendable, which we shall use to implement grammar extensions, and explore embedded languages disambiguation.

References

[1] A. Borghi, V. David, A. Demaille, and O. Gournet. Implementing attributes in sdf, May 2005. Comm. to Stratego Users Day 2005.

[2] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

[3] S. D. Swierstra, A. Baars, and A. Löh. The UU-AG attribute grammar system. `http://www.cs.uu.nl/groups/ST`, 2003.

[4] M. van den Brand, S. Klusener, L. Moonen, and J. J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. volume 82 of *ENTCS*. Elsevier, 2003.

[5] C. Vasseur. Semantics driven disambiguation: a comparison of different approaches. Technical report, LRDE, 2004. `http://publis.lrde.epita.fr/20041201-Seminar-Vasseur-Disambiguation-Report`.

Chapter 2

Implementing Attributes in SDF

Alexandre Borghi
Valentin David
Akim Demaille
Olivier Gournet

Abstract

AGs provide a very convenient means to bind semantics to syntax. They enjoy an extensive bibliography and are used in several types of applications. Yet, to our knowledge, their use to disambiguate is novel. We present our implementation of an evaluator of attributes for *ambiguous* AGs, tailored to ambiguous parse trees disambiguation. This paper focuses on its implementation that heavily relies on Stratego/XT, which is also used as language to express the attribute rules. A companion paper presents the disambiguation process in details (2).

2.1 Introduction

In any typical compiler, or structured text crunching application, semantic passes follow the usual parser. Some of these passes are gatherers and merely compute additional information about the AST: binding, type-checking. Other passes involve modification of the AST, or even its full rewrite: desugaring, translation to another language etc. Stratego is a language of choice to express the latter kind of passes, where rewriting rules are the core of the process. ASF+SDF on the one hand, and Stratego with its dynamic rules on the other hand, will both happily help one to write annotating passes. Weirdly enough, the old and well-known AGs (5) do not seem to have make it into this world, although they are very well suited to write gather-and-annotate passes.

An AG is a context-free grammar enriched with attributes bound to its symbols, and rules attached to its production rules to express the relationship between the attributes of the symbols of the rule. A very specific feature of AGs is that these local relationships suffice: their analysis reveals their dependencies, from which the order of evaluation is computed. In other words the user focuses on local issues, and the system conducts the global evaluation. Evaluation strategies range from extremely naive (repeatedly traverse the tree and compute attributes which definition uses computed attributes), to extremely smart (“compile” the AG into an evaluator which makes the best use of statically computed dependencies).

This paper presents a functional (naive) prototype supporting attributes in SDF. As a novel feature, our proposal supports *ambiguous* AGs: attributes are computed on parse *forests*. This makes it possible to express disambiguation filters thanks to attributes: a special attribute is used to flag branches that are incorrect according to semantic rules; a latter filter prunes them.

2.2 Current implementation

The package `sdf-attribute` is a set of tools relying on Stratego/XT to provide attributes support within SDF. It includes an extended SDF grammar to specify the syntax of attribute rules (Sec. 2.2.1). An extended SDF grammar is processed by a chain of tools in order (i) to check that the attribute rules are complete, and (ii) to compile the attribute rules into an evaluator (Sec. 2.2.2). Finally, this evaluator in run on an actual input to evaluate the attributes (Sec. 2.2.3).

```
1 e1:Exp "+" e2:Exp → Exp
2   {attributes(eval:
3     root.value := <add> (e1.value, e2.value)
4   )}
```

Attribute rules are (currently) regular SDF annotations under the name `attributes`. To avoid attribute name clashes, named scopes are provided (`eval` in line 2). The SDF symbol labeling feature provides convenient shorthands for symbol names, or when symbols occur several times (two `Exps` in line 3). The special identifier `root` refers to the symbol defined, here `Exp`.

Figure 2.1: Example of attributes

2.2.1 Syntax

Attribute rules are embeded into SDF grammars via annotations (Fig. 2.1). In the future, a nicer syntax might be proposed. Each attribute has a name and a name space name, written `NodeName.namespace:name`. The optional name space name defaults to that of the rule list.

A rule specifies the value of an attribute via a Stratego strategy as follows: `Node.attr := strategy`. Within the strategies themselves, attributes are used like ordinary Stratego variables.

The node name is either `root` to refer to the produced non terminal, or the symbol name, or a label name. Label names are useful when a child is not a simple symbol (lists, options, etc.) or when symbol name is used several times.

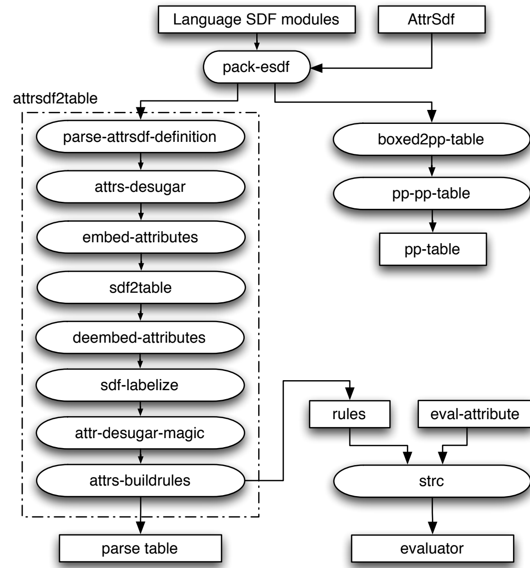


Figure 2.2: Parse table generation

2.2.2 Evaluator Generation

The SDF modules are packed according to `AttrSdf`, which extend SDF with our attribute syntax. This is performed by `esdf` described in another paper (3). Then, a parse table and an evaluator are genererated by the whole `attrsdf2table` chain, composed of several steps (Fig. 2.2).

Firstly, desugaring filters are run on the syntax definition to handle details such as adding implicit name spaces (`attrs-desugar`). In order to have `sdf2table` accept the Stratego code in the annotations, some transformations are applied (`embed-attributes`) and reversed afterwards in the parse table (`deembed-attributes`). Then, missing labels are inserted (`sdf-labelize`).

The next filter (`attr-desugar-magic`) addresses two important issues. First, at this stage, it is useful to check whether the attribute rules are well written or not. Since there is no debugging tool it is better to find and report errors instead of creating an invalid table. This definition checker traverses the graph of all possible trees beginning from the start symbols, carrying knowledge about attributes dependency. Second, taking advantage of this traversal, it also automatically propagates attributes which were declared inherited or synthesized, by adding the implicit rules. This is convenient symbol tables for instance.

Finally, attribute rule code is extracted from the parse table (`attr-buildrules`), and put in a rules section in a Stratego source file. It is compiled along with the tree traversal code (`eval-attribute`) to provide a filter to be used in the evaluator. This code is then erased from the parse table, to keep it as simple as possible. Only dependencies between synthesized and inherited attributes rules are kept for each

node, to help the evaluation.

2.2.3 Evaluation

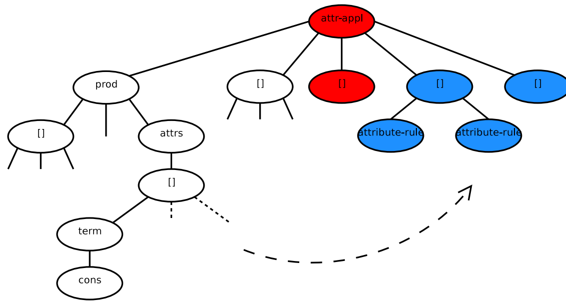


Figure 2.4: Making attribute more accessible

The evaluation of attributes is also performed by a chain of tools, presented in Fig. 2.3. SGLR is run using our tailored parse table to parse a source file. The resulting AsFix tree has attribute rules dependency and label tables as production annotations. To ease the evaluation, `prepare-attributes` moves them a more accessible place as shown on Fig. 2.4. In addition, an empty list is added to accept future attribute values.

Then `eval-attributes`, the evaluator compiled from the attribute rules, evaluates the tree using dynamic rules. The traversal depends on what is computed: when new attributes are evaluated, its dependencies can be visited. Afterward, `clean-attributes` can transform the tree back into regular AsFix as shown on the figure 2.5. The attribute *values* are put into production rule annotations. This tree can be imploded into an AST with attribute values as annotations (`attr-implode`).

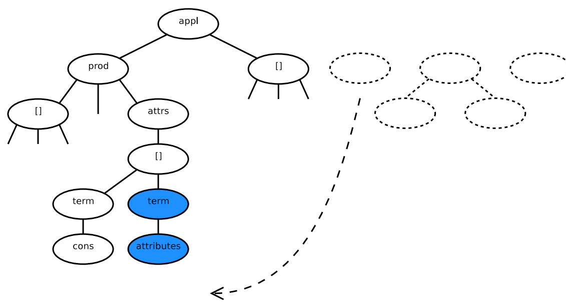


Figure 2.5: Back to regular AsFix

2.3 Discussion

This implementation of attribute grammars in SDF was developped over a few weeks, in order to provide a disambiguate-by-AG framework for the Transformers project (2). Many inspiring AG system exist; eventually our system will be completely rewritten to address its shortcomings.

2.3.1 Related Work

Of the many existing AG systems, a few caught our attention.

JastAdd II (4) is a language implementation tool supporting generation of compilers using extended AGs: Rewritable Reference Attribute Grammars (ReRAGs) and ordinary Java code. To test their work they implemented a Java 1.4 compiler which is only four times slower than the hand written javac compiler. They seem to be using their AG to specialize their ASTs, a (weak) form of disambiguation. Their AG system is powerful and offers interesting features to shorten the AGs.

Similarly, the UU-AG system (1), developped at the University of Utrecht, features nice concepts to factor rules. It also benefits from features of its target language, Haskell, to spare traversals.

2.3.2 Further Work

The current implementation of the attribute evaluator was quickly written and had for only goal to serve our needs. Hence, its implementation is naive and was designed to be as easy to implement in Stratego as it could be. As a consequence, the performance are poor, although a thorough comparison with other system was not done.

A new evaluator will be written to speed up the evaluation. The current evaluator is fully dynamic, although a lot of work can be done statically: to control the data flow between attributes an order of evaluation can be computed from specificities of the grammar and the attributes associated with.

The implementation language, and the language into which attribute rules are written is also subject to debate within our group. Some members believe that if Stratego is powerful in term rewriting, the evaluator does not need this

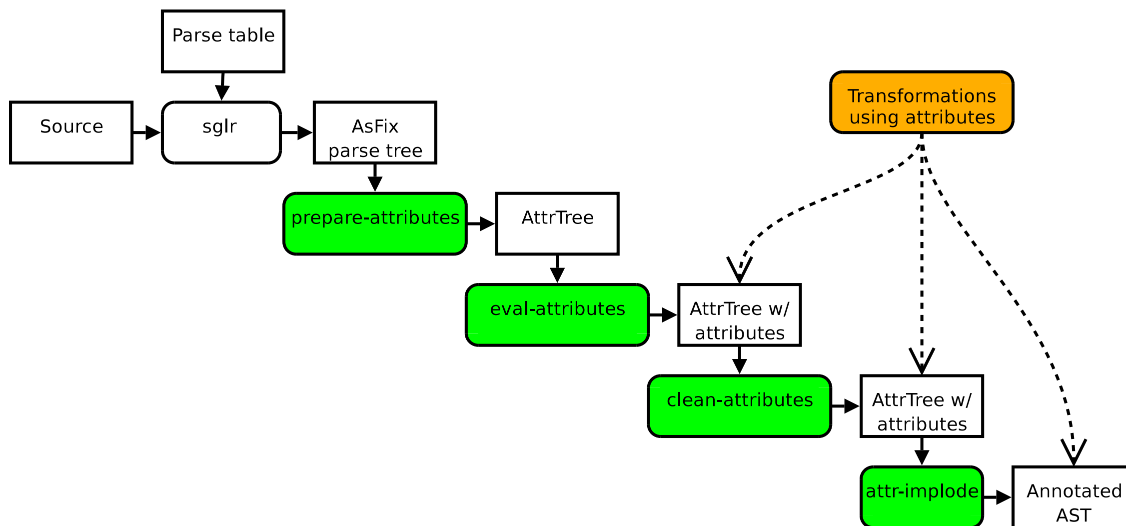


Figure 2.3: Evaluation process

specific feature and needs high speed in other ones.

2.4 Conclusion

In this paper we presented a simple but effective implementation of AGs for possibly ambiguous grammars in the world of SDF, using Stratego/XT as an implementation and execution framework. Lots of issues remain to be addressed: syntax improvement, additional features, formalization, and comparison with other information gathering schemes. This proposal nevertheless suffice to fully disambiguate ISO-C99, and even the most complex parts of C++. We are now looking forward meeting interesting in our system, and its development.

References

[1] A. Baars, D. Swierstra, and A. Lï£¡h. UU-AG System, 1999. `http://catamaran.labs.cs.uu.nl/twiki/st/bin/view/Center/AttributeGrammarSystem`.

[2] V. David, A. Demaille, R. Durlin, and O. Gournet. C/C++ disambiguation using attribute grammars, May 2005. Communication to Stratego Users Day 2005.

[3] A. Demaille, T. Largillier, and N. Pouillard. Esdf: A proposal for a more flexible sdf handling, May 2005. Communication to Stratego Users Day 2005.

[4] G. Hedin and E. Magnusson. JastAdd, 2001. `http://www.cs.lth.se/Research/ProgEnv/rags/`.

[5] D. E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145, 1968. Not read, but according to all other references, it is the first text on attribute grammars.

Chapter 3

Spoofax: An Editor for Stratego

Karl Trygve Kalleberg

Abstract Major challenges remain for Stratego to gain widespread adoption within the domain of program transformation. Many of these challenges are not of a research nature, and are therefore not prioritized. One example of this is the absence of a good editor. In this article, we introduce the Spoofax editor for Stratego which provides syntax highlighing, code completion and an outliner.

3.1 Introduction

The fundamental question Stratego was designed to answer has been resolved: It is now apparent that one can build a program transformation system which is both scalable and efficient around just two operations; 'match' and 'build'. Many of the challenges introduced by more widespread adoption of Stratego are not directly research. Consequently, they do not receive a high priority from the Stratego community, and this turns out to be a serious hindrance for wider adoption.

One of the most important factors in computer language adoption is the availability of good documentation, both tutorials, handbooks and API documentation. Another factor is tools, going from the traditional set of compilers, debuggers, profilers and editors to more domain-specific tools such as format checkers, parsers and parser generators. In the later years, the state of the basic Stratego toolchain has advanced rapidly, but there has been one group of tools which have received little or no attention, namely the interactive ones. In particular, there has not been a good, language-aware editor for Stratego. The fact that Stratego is a domain-specific language, is not a valid excuse for this omission.

In this paper, we present the Spoofax editor for Stratego, which provides syntax highlighing, code completion and an outliner. We will evaluate its feature set against the popular requirements for a modern editor, with an eye towards relevant research opportunities in this domain.

3.1.1 Requirements for an Editor

Arguably, the absolute minimum of functionality for a source code text editor is accurate syntax highlighting. Apart from being pleasing to the eyes, visual cues about many kinds of syntactical errors is helpful in catching many kinds syntax errors as early as possible. In present day software engineering, expectations for editors are higher. Various kinds of functionality requiring content awareness is necessary: outliner, code completion, structural navigations and searches, realtime error reporting, documentation popups, file and module management are all examples of this.

At the present time, we have only implemented outlining and code completion, so our discussion will mostly be restricted to these features.

3.1.2 The Weave

The immediate reason for writing the editor was scratching a personal itch. There is however, a more fundamental problem this work helps shed some light on, namely that of suitable program representations. As the the editor requires at least a rudimentary awareness of the program it edits, the structure of the program must be discovered and maintained incrementally by

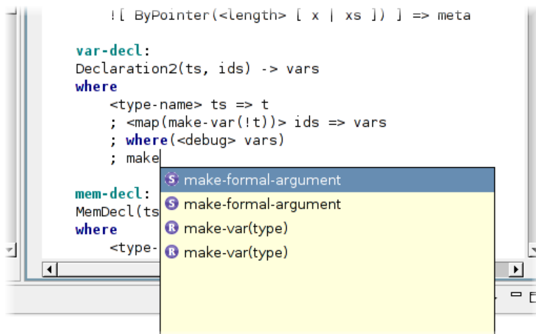


Figure 3.1: This figure shows the editor with a dropdown box listing possible completions for the prefix 'make'. The S and R icons denote strategis and rules, respectively.

the editor. We shall refer to this representation by the term *weave*, to signify that it is allowed to be an arbitrarily linked data structure, and not merely a traditional abstract syntax tree.

3.2 The Spoofax Editor

The current Spoofax editor provides three main features that we will discuss in turn: syntax highlighting, outlining and code completion.

3.2.1 Syntax Highlighting

The source code is coloured as you type. This made possible by a handcrafted rule-based, incremental parser. This parser detects definitions of rules, strategies and constructor; declarations of module name and module imports; comments, either one line (//), multiline (/*) or for documentation (/**).

Each kind of syntactical element the parser recognizes has formatting associated with it, that gives details about the presentation of this element; its color, whether it should be bold or not, and whether it should be italic or not. These settings are user-customizable. The editor window in Figure 3.1 shows an example of this.

The Spoofax editor also has a similar syntax highlighter for the SDF syntax definition language, and color settings for identical kinds of elements are shared between the languages. The intention is to provide the user with a consistently-looking environment for the most important languages in the Stratego/XT ecosphere.

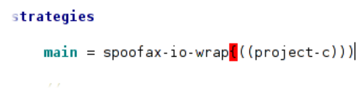


Figure 3.2: Parenthesis matching in Spoofax. The red box indicates that { does not match).

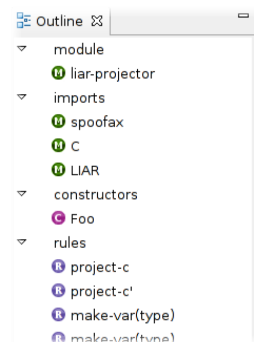


Figure 3.3: The outliner in Spoofax. It is divided into five sections, *module*, *imports*, *constructors*, *rules*, *strategies*, which may be collapsed independently. Withing each section, the entries appear in the order they are declared in the source file.

Parenthesis Highlighting

A minor but highly userful feature of the editor is a real-time parenthesis highlighter. As the user moves the cursor either directly with the arrow keys, or by typing characters, parentheses in the immediate vicinity the cursor are matched. If the cursor is after a closing parenthesis or in front of an opening parenthesis, its matching counterpart is found and marked visually with a box. If the source contains is a parenthesis mismatch, the cursor will turn red. See Figure 3.2 for examples.

3.2.2 Outliner

An outliner is a view (a window) which shows the main abstractions available inside a code file. In Stratego, the outliner view shows all constructor, rule and strategy declarations inside the file, in their declared order. Figure 3.3 provides an example.

The outline is retrieved directly from the weave. As long as the weave is up to date with respect to the textual source code, the outline is accurate. In the present version, the weave is updated every time the source code is saved to file. The updating of the weave is done using an

"outliner" parser. As with the highlighter, the outline parser is handwritten and rule-based. The outline parser is different from the highlighter in that it is not incremental – it reparses the entire document whenever the user saves, and also keeps track of the sections (signature, rules, strategies), to be able to tell constructor and rule declarations apart. After the outline parser has been run, its result is the new weave. In the present version, the weave is *not* identical to the AST, it only provides a rough outline of the source code. This is a clear candidate for future improvement. Whenever the weave is changed, it will inform all registered listeners about the change and request their update. The outline view is a client to the weave, and will therefore refresh itself upon save.

3.2.3 Code Completion

Code completion is an interactive aid which shows possible textual completions for a given prefix inside a given class of strings. In Stratego, we have one class for strategies and rules, and one for constructors. Figure 3.1 shows the code completer in action.

The two classes are populated from the weave. At the present time, they are only local to a Stratego source code file: when completing a strategy name, only strategy declarations occurring inside the same file will be suggested. This is another candidate for future improvement, and requires extending the weave.

3.3 Discussion

Constructing language-aware editors is an old craft, covered in detail by (1). In constructing the current version, we have employed tested and tried techniques by building the editor into a commercially supported, freely available software tool framework called Eclipse. Implementation was done entirely by one person in the course of just under two full weeks. The code size is about 4000 lines of Java code.

Experience from the implementation work, we can conclude that construction and maintenance of the code weave is by far the trickiest part of the editor, and that this area needs a lot more work. The major problems for future improvement here are all related to parsing. Functionality such as improved code completion, refactoring, incremental compilation, source code searches and similar, is something we want to write as Stratego programs, and invoke from inside the editor.

By looking at existing environments with these capabilities, we can see that for such interactive features to be practical, they must work in the presence of both (slightly) syntactically and (highly) semantically invalid source files. This is unfortunate, because it breaks squarely with the requirements placed by software engineering tools written in Stratego/XT, where a minimal requirement is that the program parses cleanly. Coming to grips with this will require some engineering ingenuity.

For the syntax highlighter, we also want it to behave sensibly in the presence of embedded concrete syntax. Apart from embedded conrete syntax (Stratego-in-Stratego in particular), the syntax highlighter works sufficiently well for everyday use.

3.4 Conclusion

We have presented Spoofax, an editor for Stratego. It provides syntax highlighting, an outliner and code completion. The two latter features are made possible by the presence of a code weave, which is in essence is a structural model of the source code text, kept up to date. The experience gained from the weave is that once constructed, retrieving and presenting the information, be it outlines, structural searches, code completion is surprisingly straightforward. The conclusion is therefore that future focus should be spent on the code weave, and how it can be shared between Stratego tools.

References

[1] T. Reps and T. Teitelbaum. The synthesizer generator. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 42–48, New York, NY, USA, 1984. ACM Press.

Chapter 4

ESDF: A Proposal for a More Flexible SDF Handling

Akim Demaille
Thomas Largillier
Nicolas Pouillard

Abstract

By the means on its annotations, SDF *seems* to be extensible: the user is tempted to tailor its grammar syntax by adding new annotation kinds. Unfortunately the standard SDF crunching tools from Stratego/XT do not support the extension of SDF, and the user has to develop the whole set of tools for her home grown extension(s). We present the ESDF tool set that provides "weak" genericity with respect to the grammar grammar: support for arbitrary SDF annotations. We would like to contribute it to Stratego/XT since its components subsume their stock peers. Finally, we present a set of four extensions we find useful.

4.1 Introduction

SDF (3) is modular, and extensible thanks to its support for annotations. The combination of these two features makes SDF a unique place where additional grammar features can be added, immediately taking advantage of all the other SDF features. Examples of embeddable data include pretty-print directives, extended AST generation directives, attribute rules, disambiguation tags, in addition to the "official" support for simple disambiguation filters, AST constructors, etc. With such self-contained ESDF files, mixing grammar modules is straightforward, and the user can focus on its extensions without having to bother with different concepts of modules.

Nevertheless aggregating several unrelated aspects of a grammar within a single file violates the principle of separation of concerns: a given facet of a grammar is surrounded with unrelated, distracting, material.

This contradiction is actually very comparable to a well-known object oriented design issue: given a family of related classes and a group of functions to implement on them, what is the best design? Classical Object Oriented Programming (OOP) recommends to implement the functions as methods. This eases the addition of new objects, but makes the addition of functions tedious: each class (read class implementation file) must be edited. The Design Pattern (DP) approach advocates the introduction of Visitors (objects that implement the functions) and the extension of the classes to cooperate with any kind of Visitor. The implementation of new functions is straightforward — implement new Visitors —, but the addition of new classes requires the edition of all the existing visitors. In fact there is not one right solution, both axis to read the matrix "classes × features" has its advantages: if the classes are stable and in fixed number, use Visitors (DP); conversely, if the functions are stable and in fixed number, sticks to methods[1] (OOP).

The same tension exists when working with grammars. If your grammar modules (classes) are versatile and numerous, then encapsulation (OOP) is more productive than separation of concerns: the grammar-centric vision is

[1] Actually this tension is at the origin of the inception of Aspect Oriented Programming (AOP), but it is unclear what the parallel would be in the context of SDF.

best suited. Conversely, stable grammars with well established features call for separation of concerns (DP): one feature should be isolated from unrelated issues (visitors), and the feature-centric approach is more suitable.

Because we work with several unstable grammar modules, because simple composition of modules eases our tasks, we wrote the ESDF set of tools to convert from the all-in-one paradigm to the separation-of-concern approach. ESDF should be enriched with reverse conversions, eventually providing the user with an easy means to zip and unzip grammar and grammar annotations.

We would like to contribute ESDF to Stratego/XT, since there is quite some code duplication between ESDF filters and their SDF peers, which they subsume. In the following, the components of ESDF are presented, and then a set of local SDF extensions we depend upon.

4.2 ESDF: An SDF Chain Robust to New Annotations

ESDF is a set of simple tools that provide generic support to SDF annotations.

4.2.1 Packing Modules: `pack-esdf`

The regular `pack-sdf` tool takes a grammar module as argument, gathers all its dependencies and produces a single big self-contained grammar file. If modularity was considered as syntactic sugar, then `pack-sdf` is its desugaring pass: none of the tools downstream need to support modularity. Unfortunately `pack-sdf` does not support annotation plug-ins: this is what `pack-esdf` addresses. It supports an additional option to be given the actual SDF grammar to use.

4.2.2 Filtering Annotations Out: `sdf-strip`

This simple tool strips (or preserves) selected annotations from a grammar. Of course, as `pack-esdf`, it needs to be given the SDF grammar used (unless it is the stock grammar).

4.2.3 Parsing Extended Grammars: `parse-esdf`

One would like to handle ESDF grammars seamlessly, like regular SDF grammars. Therefore it is the SDF parser that needs to be extended, or rather, extensible. Even the two aforementioned tools (`pack-esdf` and `sdf-strip`) need to parse ESDF grammars, and therefore demand a separated parser to avoid code duplication.

This tool is `parse-esdf`, an extension of `parse-sdf`. Based on the same ideas used to implement (foreign) concrete syntax within Stratego, `parse-esdf` looks for a `foo.meta` file for each `foo.sdf` file. This meta file describes the actual SDF grammar used.

This one tool factored several of the tools we had, since they all addressed the particular extension we were working on (BoxedSDF, DetGen etc.).

4.3 The lrde-syntax bundle

In addition to the general framework to extend SDF, we propose a set of specific extensions designed to support the grammar-centric vision.

4.3.1 Pretty-Printing: `boxedsdf`

Embedding the GPP pretty-printing tables in the grammar eases the maintenance, and provides a more comfortable environment to edit these tables: one can use names instead of numbers etc. See Fig. 4.1.

4.3.2 Disambiguation Tags: `sdf-detgen`

It is convenient, in particular to write disambiguation test cases or to check by human the result of a disambiguation pass, to enrich an ambiguous grammar with special comments to specify the correct alternative. For instance, in C++ `namespace A {}` is ambiguous: its actual nature depends whether the namespace name `A` was met for the first time (is "original") or not. Therefore parsing the following:

```
namespace A {}
namespace A {}
```

```
%% 7.3.1 [namespace.def]
"namespace" Identifier "{" NamespaceBody "}" → OriginalNamespace
Definition
    {pp (V[H[KW["namespace"] Identifier]
           V is=2[KW["{"]
                  NamespaceBody]
           KW["}"]])}
```

In BoxedSDF, one can use symbol names to denote nonterminals (e.g., `Identifier` and `NamespaceBody` in the first rule) or labels, instead of _1 as with GPP.

Figure 4.1: BoxedSDF sample

The following piece of C++ grammar extensions introduces special comments that can be used to disambiguate explicitly the "first occurrence of namespace name" issue.

```
"namespace" "/*<org>*/" Identifier "/*</org>*/" "{" NamespaceBody "}"
                                      → OriginalNamespaceDefinition
"/*<ns>*/" Identifier "/*</ns>*/"     → OriginalNamespaceName

"/*<org>*/" | "/*</org>*/"  → LAYOUT {reject}
"/*<ns>*/"  | "/*</ns>*/"   → LAYOUT {reject}
```

These rules were generated by `detgen` from the following annotated SDF rules. The first rules disambiguate the type names, and the last reject the parsing of the disambiguating tags as comments.

```
%% 7.3.1 [namespace.def]
Identifier → OriginalNamespaceName           {dettag("ns")}
"namespace" Identifier "{" NamespaceBody "}"
           → OriginalNamespaceDefinition     {dettag("org", 1)}
"namespace" OriginalNamespaceName "{" NamespaceBody "}"
           → ExtensionNamespaceDefinition
```

Figure 4.2: Disambiguation annotations.

results in the following disambiguated text, printed with disambiguation comments (`org` stands for original, and `ns` for namespace):

```
namespace /*<org>*/A/*</org>*/ {}
namespace /*<ns>*/A/*</ns>*/ {}
```

The generation of such comments *and* the rules that recognize them is straightforward. The most adequate place to specify these comments is the grammar, as additional `dettag` annotations (Fig. 4.2).

4.3.3 AG: sdf-attribute

A more ambitious extension of SDF consists in supporting AGs. Two companion papers present this topic in depth: **(author?)** (1) detail the evaluation mechanisms, and **(author?)** (2) demonstrate how AGs can be used to disambiguate C and C++. A simple sample follows.

```
e1:Exp "+" e2:Exp → Exp
 {attributes(eval:
   root.value := <add> (e1.value,
                        e2.value))}
```

4.3.4 Flexible AST generation: sdf-astgen

The cons annotations relieves the SDF user from having to implement an abstract syntax grammar: it is extracted from the concrete grammar. As long as the concrete syntax is "natural", the resulting ASTs are lightweight and pleasant to process. But if the grammar is entangled with Yacc idiosyncrasies, or disambiguates "by hand" instead of relying on precedence and as-

```
e:Expr "+" t:Term -> Expr  { ast(BinOp(Plus, e, t)) }
t:Term            -> Expr  { ast(t) }
t:Term "*" f:Fact -> Term  { ast(BinOp(Mult, t, f)) }
f:Fact            -> Term  { ast(f) }
n:NUM             -> Fact  { ast(Int(n)) }
"(" e:Expr ")"    -> Fact  { ast(e) }
```

Constructors such as `Infix`, `Mult` . . . are freely choosen by the user.

Figure 4.3: More flexible AST generation: the `ast` annotation

sociativity directives, then the ASTs look like PTs... Some advocate the de-Yaccification of the grammar, but when this is not possible or desirable, one would like a more powerful `cons` annotation.

`sdf-astgen` allows the user to specify her abstract syntax, using an extended `cons` annotation: `ast` (Fig. 4.3). This enables the exchange of ASTs between closely related, but different, grammars. For instance we plan to use `sdf-astgen` to bridge the gap between our (standard) C++ ASTs, and CodeBoost's tailored ASTs.

4.4 Conclusion

We emphasized that self-contained grammar modules can be the most productive paradigm depending on the actual constraints of the project at hand. Because regular Stratego/XT tools do not support SDF annotation variations, we propose to replace them with the ESDF set of tools that support genericity with respect to annotation kinds. We also submitted four such extensions that are useful in our framework. Interesting extensions to this work include the support of more ambitious changes in the grammar of the grammars, and the exploration of means to provide easy composition of several different aspects of grammar modules while keeping concerns separated.

References

[1] A. Borghi, V. David, A. Demaille, and O. Gournet. Implementing attributes in sdf, May 2005. Comm. to Stratego Users Day 2005.

[2] V. David, A. Demaille, R. Durlin, and O. Gournet. C/C++ disambiguation using attribute grammars, May 2005. Communication to Stratego Users Day 2005.

[3] E. Visser. A family of syntax definition formalisms. In M. G. J. van den Brand et al., editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.