Attribute Grammars for Modular Disambiguation

Valentin David† Akim Demaille‡ Olivier Gournet‡

† University of Bergen, Norway
‡ EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire - F-94276 Le Kremlin-Bicêtre Cedex - France
transformers@lrde.epita.fr, http://transformers.lrde.epita.fr.

Abstract

To face the challenges to tomorrow's software engineering tools, powerful language-generic program-transformation components are needed. We propose the use of Attribute Grammars (AGs) to generate language specific disambiguation filters. In this paper, a complete implementation of a language-independent AG system is presented. As a full scale experiment, we present an implementation of a flexible C front-end. Its specifications are concise, modular, and the result is efficient. On top of it, transformations such as software renovation, code metrics, domain specific language embedding can be implemented.

Introduction

Modern software engineering tools provide the programmer with a host of powerful features to manipulate source code. A trend to design such tools consists in building them from language-generic components: generic parsers, program transformation environments and pretty-printers, for instance, exist. It is then possible to provide a language with new tools (code metrics, refactoring environments, etc.) or new features (embedded SQL, design by contract, etc.).

In such a framework the parsing is truly context-free, but most programming languages are *not* context-free. To cope with this discrepancy, parsers actually accept an ambiguous superset of the language, leaving the context-sensitive disambiguation phase to a later filter. This paper presents a new approach based on an extension of AG [12] to cope with ambiguity. We report on an implementation of such an AG system, and demonstrate the soundness of the approach by describing a complete ISO C 99 polyvalent front-end. Thanks to the use of AGs, its implementation is both concise and modular. Both the AG engine and the C front-end are free-software covered by the GNU General Public License (GNU GPL), freely available on the Internet [13].

The paper is structured as follows. Section 1 presents the context of our work, and motivates the use of ambiguous AGs for the automatic generation of semantics driven disambiguation filters. Section 2 details the user side: how to run a disambiguation chain powered by an AG. Section 3 presents the implementer side: how to write disambiguating AG rules, and how the filter is generated. Section 4 reports about a full scale use of these tools to implement and use a C flexible front-end. Existing and future works are presented in Section 5. Finally Section 6 concludes.

1. Context

Generalized Parsing Even though LALR(1) parsing ruled the world thanks to Yacc, truly context-free languages are infrequent: most are context sensitive. This dependency is usually addressed with ad hoc actions in the parser such as symbol table maintenance, which prevents any form of modularity. Because in addition no interesting class of deterministic languages is stable under union, techniques supporting the full class of context-free languages are desirable. Generalized LR (GLR) parsing [15] meets these requirements.

GLR handles local ambiguities "for free" using unbounded look-ahead, and global ambiguities (requiring context sensitive information such as typing) typically by providing the user with a means to decide how to process alternatives. GLR parser generators such as Elkhound [14] or GNU Bison extend the Yacc model: user actions are executed. Using tailored actions during parsing enables excellent performances, comparable to usual parsers, but using similar tricks. Then again modularity is lost: user actions need to be modified when mixing several languages.

Alternatively some generated parsers directly build the Parse Tree (PT) / Abstract Syntax Tree (AST). Scannerless Generalized LR (SGLR) [18] is one such tool; if the input is ambiguous it yields a parse forest that a latter pass is expected to prune (Section 2).

Flexible Front-ends State of the art technologies maximize reuse, including of front-ends. This is in sharp contrast with the traditional front-ends that embed actions tailored to the job at hand: even if at some time the grammar was taken from a grammar base, hand editing made it diverge from its root. Using a formalism such as Syntax Definition Formalism (SDF) [17] together with a generic parser such as SGLR [18] makes it possible to design a library of program transformation components: parser, pretty-printer etc. The grammar behaves as a contract for the whole tool-chain [10]. These components can be used off-the-shelf, or modified in a modular way for a specific task.

Flexible front-ends unleash a host of new possibilities: Domain Specific Languages (DSLs) can be embedded in host languages, local idioms can be normalized (e.g., translation from GNU C to ISO C), etc. The METABORG method [8, 7] uses Dryad, a flexible Java front-end, together with a toolchain to implement *assimilation*, i.e., the compilation of extended Java down to regular Java. The authors demonstrate METABORG by hosting within Java a Domain Specific Embedded Language (DSEL) for Swing interface design, another for regular expressions, and another for Java Abstract Syntax Tree (AST) handling.

Unfortunately no such framework is available for C, let alone for C++. The Transformers project [13] aims at providing C (and eventually C++) flexible front-ends. Transformers is free-software covered by the GNU GPL, freely available on the Internet.

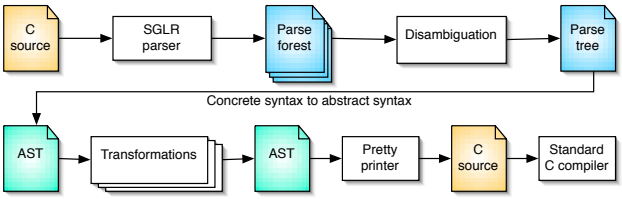
Tools for Program Transformation Flexibility results from tight common conventions between the tools, we chose SDF as the spine for the Transformers project and the Stratego/XT tool set [21]. This collection of tools provides language generic components for the whole processing. In particular the Stratego programming language [19] provides powerful term rewriting features, controlled and composed by rewriting strategies. A rich set of operators allows to build arbitrarily complex transformations (i.e., strategies) from simple atomic ones. Context sensitive transformations are easily coped with thanks to dynamic (rewriting) rules. Finally, thanks to a tight integration with SDF, Stratego features concrete syntax: although rewriting rules do transform *abstract* syntax trees, rules can be written in the target's language *concrete* syntax [20].

Although C and C++ front-ends already exist, two foundamental limitations prompted the design of an SDF/SGLR support for C and C++: firstly no other formalism features an equivalent level of modularity, and secondly this is mandated to enjoy the benefits of C or C++ concrete syntax in Stratego.

```
context-free syntax
  "true"          -> Bool
  "false"         -> Bool
   Bool "|" Bool  -> Bool
```

This grammar is taken from [5]. Contrary to the Backus Naur Form (BNF) tradition, in SDF arrows are oriented as reductions, not productions.

Figure 1. A simple ambiguous SDF grammar.



Given a C grammar written in SDF, the SGLR parser reads the text and yields a set of parse trees: a parse forest. A disambiguation step keeps a single parse tree (Section 2), transformed into an AST suitable for the transformation(s). Finally the AST is pretty-printed back into compiler-ready C source text.

Figure 2. A C program transformation chain

2. Disambiguation chain

The decomposition of the analysis of context-sensitive languages in two steps, (context-free ambiguous) parsing and then disambiguation, is well known and well described in the literature [11, 5, 8]. The main contribution of this paper is demonstrating that AGs, extended to support ambiguities, provide a nice way to implement such filters.

As a running example, consider the simple ambiguous SDF grammar from Figure 1. The associativity of `|` is unspecified, leaving two possible syntactic analysis of `true | true | true`. Such an ambiguity can be handled during the construction of the parser. Nevertheless we use it because it is extremely simple and already discussed by [5]. [16] discusses a more complex yet simple ambiguous language (Section 5.1).

The following sections follow the stream depicted in Figure 2 from the original source text, down to the possibly transformed final source text.

2.1. Upstream

Before attribute evaluation several phases may have been run. For instance, if we were to process C or C++, one would have to run the preprocessor to handle all the `#` directives

and macros. SGLR might be run by this phase, but whatever the architecture of the upstream phase, it must result in a (possibly ambiguous) PT.

In our running example, the analysis of a simple phrase such as `true | true | true` yields an ambiguous PT, which represents the two possible analysis[1]: `(true | true) | true` and `true | (true | true)`. In the following, we chose the former, corresponding to the left associativity.

2.2. Attributes

We propose the use of AGs to specify context-sensitive rules. Attribute Grammars (AGs) [12] is a formalism that supports syntax directed semantic analysis: (grammar) rules are decorated with a set of equations that relate a node's *attributes* with those of its parents and/or children. AGs allow to focus on local aspects, leaving the global evaluation order aside, under the responsibility of a generic engine. Although AGs cannot modify the trees, their use for disambiguation is straightforward. Attributes convey information, e.g., a symbol table. Conflicting branches of the parse forest are flagged, and a (language generic) filter is run afterward on the parse forest, pruning inconsistent alternatives.

Attribute Evaluation. In our implementation, rules relating attribute values are attached to the SDF grammar as *annotations* of type *attributes*. In the running example, left associativity expressed locally to a `|`-node stands as "no right-child of a `|`-node is a `|`-node". The following example presents a straightforward implementation of this idea.

```
context-free syntax
 "true" | "false"       -> Bool
    {attributes(assoc:
      root.is_atom := true
    )}

   b1:Bool "|" b2:Bool -> Bool
    {attributes(assoc:
      root.is_atom := false
      root.ok      := b2.is_atom
    )}
```

The symbol `root` always denotes the root of the node: the right-hand side of the (SDF-)rule. The user may use labels such as `b1`/`b2` to refer to symbols. A single attribute `is_atom` is computed for each node. The special attribute `ok` specifies whether the node is valid or not.

Technically the computation of the attributes is performed by the *evaluator* compiled from an AG. Its construction is presented in Section 3. It runs on a PT and yields an *attributed* PT, i.e., a PT whose nodes are decorated by attribute values. It is thanks to the extreme flexibility of the PT format, AsFix2, and the tools that process it that we can decorate it so easily.

The evaluation of an *ambiguous* AG is somewhat different from the usual case because a synthesized attribute could have different values on different branches of an ambiguity. To select the right one, the evaluator depends on the `ok` attribute. Indeed, directly or indirectly, every synthesized attribute should depend on an `ok`. If `ok` is `false`, the value of synthesized attribute is `fail`, which propagates to "infected" attribute values. In the end, the ambiguity node is presented only with a set of at most one possible value; if it turns out all the values `fail`, the ambiguity node's attribute itself is set to `fail`.

This design frees the user from having to explicitly carry the `ok` attribute everywhere. It is comparable to an exception system which destroys a branch of a computation until an exception handler is met: an `amb` node.

Pruning & Checking. The evaluation phase is pure: it does not modify the PT, it merely decorates its nodes. In particular it flags invalid alternatives of an ambiguous PT with a false `ok` attribute. A filter then prunes invalid branches, and afterwards another makes sure no ambiguity remains.

2.3. Downstream

Although they fall outside the scope of this paper, it is interesting to introduce passes that are typically run once an unambiguous PT is obtained. *Implosion* transforms the PT into an AST: basically nodes are no longer labeled by their corresponding production rule, but by simple labels, *constructors*, also specified as addition SDF rule annotations. For instance a PT node labeled by `Bool | Bool -> Bool` will implode into an AST node labeled by a simple constructor `Or`. This AST is the ideal format to run *program transformations* (1). Such transformations include software renovation, assimilation, and so on. Alternatively data can be extracted, such as code metrics. Finally, the AST is pretty-printed back into a source file.

3. Attribute Grammars in SDF

The generation and compilation of an AG evaluator for a specific grammar comprises several steps detailed below. A set of SDF modules with AG rules (Section 3.1) are packed together and preprocessed (Section 3.2), its AG rules are checked and completed (Section 3.3), and finally compiled into an evaluator and a parse table (Section 3.4).

[1] The parentheses are not part of the language under study, they are used as meta-notation to distinguish the alternatives.

3.1. Attribute Rules in SDF

From the point of view of syntax, there are many available options to specify attribute computation rules. Because developing an AG system for SDF was not our primary goal, we made a number of decisions to simplify the design. In the long run, several of these shortcuts should be reconsidered (Section 5.2).

We embed attribute rules in the SDF grammar. The rationale is that since we use AG to disambiguate an ambiguous grammar, the disambiguation information really belongs to the grammar itself. As a matter of fact, this is not different from embedding precedence and associativity information in the grammar, which is already the case in SDF. Actually SDF features several other disambiguation types of built-in filters [6].

Rules are embedded as regular SDF annotations, therefore we can use the usual SDF tool set: no additional development and maintenance is needed. As a natural consequence, our disambiguation filters benefit from the exact same concept of modularity as the SDF modules themselves. We can freely mix SDF modules: the disambiguation AG taking care of context-sensitivity is also intermixed. Of course, additional disambiguation rules might be needed.

3.2. Grammar Modules Preprocessing

Before generating the attribute evaluator several preprocessing phases take place. First, `pack-esdf`, an extended version of the `pack-sdf` tool from Stratego/XT, gathers and checks SDF grammar dependencies, and produces a self-contained unique grammar file. Some tweaks are also used to transform this unique file into a form suitable for `sdf2table`, the off-the-shelf parse-table generator. These parse tables are extremely rich and contain all the data needed to generate an AG evaluator: the production rules, the symbols labels, and all the annotations, including attribute rules.

3.3. Completion and Checking

The `attr-defs` tool processes these parse tables to implement a number of features such as automatic propagation. Indeed, some attributes such as symbol tables virtually traverse the whole PT; writing down their propagation is tedious and error prone. Support for synthesized (bottom up), inherited (top down), and chain (left to right) attributes is implemented. Without such a feature one could no longer benefit from operators such as $\star$, +, | and so forth: one would have to decompose into a set of plain syntax rules spelling out the detail of the propagation of attributes. In other words, without such a feature no AG system can pretend to be modular.

Cycles in attribute dependencies are looked using an algorithm from [12]. Because `attr-defs` is basically a graph cruncher, a data type with which Stratego is uneasy, it was rewritten in C++.

3.4. Evaluator Generation

The last module, `attrc`, handles two issues. First it removes all the AG related data to produce parse tables as expected by SGLR. Second and foremost, it generates the attribute evaluator.

This evaluator uses a strategy based on attribute dependencies to compute the order of evaluation. In this respect laziness is a nice feature that virtually determines the evaluation order by itself, a fact used in the implementation of Utrecht University Attribute Grammar System (UU-AG) in the Haskell lazy functional programming language [1]. To benefit from laziness, we generate a single (huge) Stratego program compiled by a Stratego compiler modified to support a weak form of laziness. This weak support consists in implementing all the attribute values as functions with memoization: the first time its value is requested, it is computed and cached for subsequent calls.

4. Case Study: C-Front

As a real world experiment of our AG system, we report its use to disambiguate ISO C 99. Although C is one of the most used languages, there are few flexible front-ends usable off-the-self. The front-end we describe aims at filling this gap for SDF users.

4.1. An SDF Attribute Grammar for C

In order to enable the experimentation of extensions to the standard and to minimize the risk of recognizing a language very close to, but different from the standard, we chose to remain close to the grammar as specified by the C standard [9]. As a consequence the style is sometimes a bit convoluted and unnatural: not all the SDF features are used; for instance precedence and associativity are encoded in the grammar via additional non-terminals and rules. This results in bigger ASTs.

The C grammar counts 126 symbols and 356 rules, split into 53 small and manageable sub-grammars. The boundaries of these sub-grammars were chosen to address coherent, atomic, related issues; they are finer than those of the standard that breaks the grammar in only 4 parts [9, Annex A]. The AG part of the grammar counts 10 different kinds

Filter	Duration (s)
`pack-esdf`	3.12
misc. processing	9.31
`sdf2table`	7.28
`attr-defs`	0.41
`attrc`	10.08
Stratego compilation	112.95
C compilation	32.35
Total	**175.54**

The experiment was run on a P4 3Ghz with 1Gb RAM. The memory footprint is below 120Mb.

Figure 3. Compile time performances

of attributes, and 190 attribute rules. The completion of attribute rules with automatic propagation raises that count up to 1183 (Section 3.3).

4.2. Performance

Efficiency is measured in two different contexts: the compilation of an AG, and then its execution.

Compile time.

The compilation of the evaluator is slow but bearable: three minutes (Figure 3). Interestingly a significant portion of the computation time is spent uselessly in pretty-printing (aka, unparsing) at the end of a tool, immediately followed by parsing by the following tool. This will be easily solved by the authors of the tools, since they all share the compact binary representation of trees, ATerms [3].

The resulting front-end passes all its test suite, composed of about 800 tests from the GCC test suite (the tests that were left out address either GNU extensions, or issues not related to parsing) plus 100 additional tests tailored to exercise the disambiguation.

Run time.

Finding large programs in *standard* C, to bench our disambiguating chain, turned out to be troublesome. In particular, GNU C extensions are fairly commonly used; its support in our C front-end is future work.

The figures are both reassuring, and disturbing. Indeed, the run time of the disambiguation filter is acceptable in most situations, and we are actually confident that several ad hoc optimizations could significantly cut it down. Unfortunately the performance of the rest of the tool-chain is harder to improve. As a matter of fact, the slowness of SGLR has already been reported [14]. It has been said that in the future SGLR might directly produce an AST instead of a PT, a much smaller data structure. More ideas for speed improvements are proposed in Section 5.2.

5. Discussion

5.1. Other works

Algebraic Specification Language & Syntax Definition Formalism (ASF+SDF) [4] is a complete environment that parallels SDF grammar modules with ASF equation modules. ASF is a declarative term rewriting language featuring concrete syntax, conditional rewrite rules, and traversal functions. Given an ambiguous grammar, ASF can be educated to process its extension with `amb`(iguity) nodes. Then algebraic specifications prune invalid alternatives. This setup is presented by [5]. Their approach shares several features with ours, most prominently declarativity. Indeed ASF equations are very comparable to attribute rules: one focuses only on the local computation, leaving the global evaluation order aside. Some modularity follows as a natural consequence, but with performance issues.

In **Stratego**, rewriting strategies allow to specify evaluation orders. While strategies are comparable to ASF+SDF's traversal functions, the concept is pushed much further. Strategies are explicit and programmable: starting with a set of primitive strategies and strategy combinators, the user can design higher level strategies. Early experiments of a C++ disambiguation filter in Stratego demonstrated that it is able to tackle its most difficult part, that related to `template`. Since then, benefiting from more recent features of Stratego (in particular concrete syntax and dynamic rewriting rules), Dryad, the Java front-end developed at the Utrecht University, demonstrated that the approach proposed by the METABORG method [8, 7] is very successful and efficient.

In both cases the user has to spell out the traversal order, either by choosing it, or programming it. AGs frees the developer from this task, which also enables an attractive form of modularity: composing two components will create a whole new traversal order, unrelated to the two "primitive" traversals. This order is also naturally efficient. A more extensive comparison similar to [16] is underway.

5.2. Future Work

Syntax. The current syntax can be improved in many ways. Probably foremost, the fact that it is physically bound to its production rule goes against the separation of concerns. This simplification is acceptable for our current application — disambiguation — but if new applications, such as typechecking, were to be developed, they would add clutter to the grammar. A better designed DSL should also include support to declare attributes, declare their special properties (such as default propagation rules) etc. much as is done in UU-AG [1].

	Queens		HelloW		Lemon		Eval	
Lines of code	56		448		4 135		28 392	
Ambiguities	78		103		6 410		68 195	
PT Sizes	Tree	Mem	Tree	Mem	Tree	Mem	Tree	Mem
Ambiguous	10	14	63	62	912	642	10 280	5 689
Attributed	9	28	109	403	813	2 884	7 625	19 661
Final		15		162		1 104		5 497
Duration	s	%	s	%	s	%	s	%
Preprocessing	0.09	29	0.2	2	0.2	1	0.2	1
Parsing	0.04	12	1.3	33	4.8	17	33.6	10
Concatenation	0.01	3	0.7	18	1.5	5	3.2	1
Evaluation	0.07	22	0.8	20	11.4	40	177.6	55
Pruning	0.05	16	0.5	13	5.6	20	60.4	18
Cleaning	0.01	3	0.2	4	1.5	5	12.2	3
Checking	0.01	3	0.1	1	0.6	2	4.6	2
Conversion	0.03	9	0.2	3	2.1	7	29.3	9
Total	0.31	100	3.8	100	28.0	100	322.5	100

Queens is an extremely short program that includes no header. *HelloW* is the simple "hello, world" program with an `#include <stdio.h>`. *Lemon* is a parser generator that fits in a single C file. *Eval* is the AG evaluator for C generated in C by the Stratego compiler. The number of lines was computed by `sloccount`. The sizes of the Parse Tree count the total number of nodes and its memory footprint in Kb; the measurements were performed before the disambiguation (*Ambiguous*), after the disambiguation and pruning (*Attributed*), and after the removal of the attributes (*Final*). The timings were performed on a P4 3Ghz with 3Gb RAM.

Figure 4. Running time of the C-Transformers chain

In parallel some idioms could be isolated for frequent types of disambiguation schemes, for instance the dependency on the kind of an identifier (a type name? a value name?), and a dedicated syntax could be submitted.

Performance. The current run time enables to experiment C program transformations, explored in [2]. Several minor optimizations can probably cut down the execution time, nevertheless we believe that a better set up would be to evaluate the attributes during parse time. While it is clear that parsers that automatically build an PT or AST are an improvement over hand-written user actions à la Yacc, it is also a significant loss in performance: none of the authors believes this technique will ever be able to compete with industrial strength compilers such as the GNU C++ Compiler that includes parse-time disambiguation actions. The true added value is in *modularity* and *separation of concerns*, not the *physical* separation of phases. Provided modularity and separation of concerns are available there is no reason to banish the possibility to generate a tailored parser. The benefits are immediate: because most real-world ambiguities are solved by a simple left-to-right reading, most of the ambiguous nodes would not even be built. The peak of memory consumption, the size of the graph the evaluator would have to compute, would both become much smaller, henceforth, much faster to process. As an example out of 68 195 ambiguities in *Eval* in Figure 4 no less than 58 460 are due to value identifiers that can either denote a variable name, or a enum value. This possibility to disambiguate at parse time is specific to AGs. This requires a complete rewrite of SDF, a whole topic in itself.

6. Conclusion

We have proposed a new declarative approach to the specification of context-sensitivity using ambiguous attribute grammars. We report about the implementation of a such tool well integrated in SDF frameworks such as ASF+SDF or Stratego/XT. This tool was used to implement a fully C standard compliant flexible front-end. These experiments demonstrated that Attribute Grammars are very well suited to the generation of disambiguation filters. In particular, their specifications are very concise, and modular. The run time performance of the whole system are very satisfying, enabling the handling of C in any SDF powered transformation framework.

Acknowledgments The authors thank Karl Trygve Kalleberg for his comments on earlier drafts.

References

[1] A. Baars, D. Swierstra, and A. Lh. Utrecht University Attribute Grammar System, 1999. http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem.

[2] A. Borghi, V. David, and A. Demaille. C-transformers - a framework to write C program transformations. *ACM Crossroads*, ???(??):???, ??? 2006. Accepted.

[3] M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[4] M. van den Brand, A. v. Deursen, J. Heering, H. d. Jonge, M. d. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC'2001)*, volume 2027 of *LNCS*, pages 365 – 370. Springer-Verlag, 2001.

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

[6] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.

[7] M. Bravenboer, R. de Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal, July 2005.

[8] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[9] ISO/IEC. ISO/IEC 9899:1999 (E). *Programming languages - C*, 1999.

[10] M. de Jonge and J. Visser. Grammars as contracts. In G. Butler and S. Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposion, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, pages 85–99, Erfurt, Germany, Oct. 2001. Springer.

[11] 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.

[12] D. E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145, 1968.

[13] LRDE — EPITA Research and Developpement Laboratory. Transformers home page, 2005. http://transformers.lrde.epita.fr.

[14] S. McPeak. *Elkhound: A Fast, Practical GLR Parser Generator*, Dec 2002. http://www.cs.berkeley.edu/~smcpeak/elkhound/.

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

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

[17] E. Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997.

[18] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[19] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[20] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[21] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.