Parsing for Agile Modeling

Oscar Nierstrasz, Jan Kurš

Software Composition Group, University of Bern, Switzerland

Abstract

In order to analyze software systems, it is necessary to model them. Static software models are commonly imported by parsing source code and related data. Unfortunately, building custom parsers for most programming languages is a non-trivial endeavour. This poses a major bottleneck for analyzing software systems programmed in languages for which importers do not already exist. Luckily, initial software models do not require detailed parsers, so it is possible to start analysis with a coarse-grained importer, which is then gradually refined. In this paper we propose an approach to "agile modeling" that exploits island grammars to extract initial coarse-grained models, parser combinators to enable gradual refinement of model importers, and various heuristics to recognize language structure, keywords and other language artifacts.

Keywords: Parsing, Software Modeling, Software Analysis

Preprint. Please cite this article as: Oscar Nierstrasz and Jan Kurš. Parsing for agile modeling. In Science of Computer Programming (2013) doi:10.1016/j.scico.2013.11.011

Personal message to Paul Klint from Oscar Nierstrasz. I first met Paul at a CERN School of Computing in 1986 where I was evangelising on Object-Oriented Programming. After the event Paul kindly sent me a copy of an LNCS volume based on his dissertation entitled "A Study in String Processing Languages". Over the years our paths crossed many times, as our research fields morphed and evolved. His work on language engineering, reverse engineering, metaprogramming and domain-specific languages has been a great inspiration. Now indeed it seems we are both interested in the analysis of evolving software systems. I am looking forward to learning more about Paul's current work while I am visiting CWI during my sabbatical semester. Happy birthday, Paul!

1. Introduction

It is widely accepted that software developers spend more time reading code than writing it [1]. Reading code not only promotes program comprehension, but helps developers understand the impact of their changes on the existing system. Nevertheless there are numerous questions developers ask that cannot simply be answered by reading code, such as *"which code implements this feature?"*, or *"what is the impact of this change?"*, and for which dedicated analyses are needed [2, 3].

Dedicated platforms exist to model and analyze software systems, such as Moose [4] and Rascal [5]. A prerequisite for using such tools, however, is that a *model importer* exist for the programming language (or languages) in which the system is developed.

Our goal is to be able to *build the model of a software system in the morning, and analyze it in the afternoon*. Unfortunately implementing an importer for a new language is a non-trivial task that may entail several person-months of effort. Adapting an existing parser for the host language is often not an option, especially for legacy languages.

By "agile modeling" we refer to the ability to quickly build a model of a software system suitable for both initial analysis and incremental refinement. We believe this is feasible for the following reasons:

— Many analyses do not require an accurate, fine-grained model. An approximate, coarse-grained parser may suffice initially, and can be incrementally refined as needed.

— Most programming languages share features with other languages, such as control structures, programming abstractions, commenting conventions, indentation and so on. Such features could be semi-automatically recognized and used to compose parsers from reusable parts.

— In any given project, a considerable quantity of existing code is available as "training data" for iteratively and incrementally constructing a suitable parser. Examples can be used to infer grammar rules, and further examples can be used to test and refine these rules.

— Knowledgeable developers are available to guide the training process.

The fact that models do not need to be precise for many analyses suggests that it should be possible to develop importers more quickly than precise parsers. The fact that programming languages have many similarities suggests that heuristics can help in composing parsers. The fact that considerable code and expertise are available suggests that these can be exploited to iteratively and incrementally refine model importers.

Agile modeling trades time for precision. Parsers constructed by our methods are not complete and they can misinterpret a source code. Although we believe that incremental refinement can lead to complete parsers, we have no proof for this so far. Moreover, models are constructed based on static analysis so that they can miss information available only at run time.

In section 2 we introduce selected related work. In section 3 we summarize our previous efforts to enable agile modeling for Moose. In section 4 we present some early results of our research into exploiting island grammars to incrementally refine model importers, semi-automatically recognizing keywords in unknown languages, and exploiting indentation to recognize structure. We conclude in section 5 with some remarks on future work.

2. Related Work

The problem of developing a parser for an existing programming language is related to the field of "grammar induction" (or "grammar inference"). Most of this work, however, has focused on applying machine learning techniques to regular grammars and deterministic finite automata, not software modeling [6].

Lämmel and Verhoef have reported considerable success in scavenging grammars for existing languages from documentation and other sources [7]. Nevertheless, with their approach it can still take several weeks to construct a full grammar for a language as complex as COBOL. Kraft *et al.* have successfully used a similar approach to recovering grammars by analyzing the parse trees generated by the parser itself [8].

Language workbenches consist of sets of tools for developing compilers, translators, interpreters and analyzers for various languages. *Stratego*[1] and Xtext[2] are well-known examples. Some of these workbenches are based on *scannerless parsers* [9], which eliminate the need for a separate lexical pass, thus making it easier to cope with multi-language code without a consistent set of tokens.

Parser combinators [10] and parsing expression grammars (PEGs) [11] offer means to compose parsers from reusable parts. Island grammars [12] consist of a technique to develop robust parsers that distinguish parts of interest (*i.e.*, the "islands") from the rest (*i.e.*, the "water"). This technique is especially relevant to software modeling, where a fine-grained parser may not be needed for many analyses.

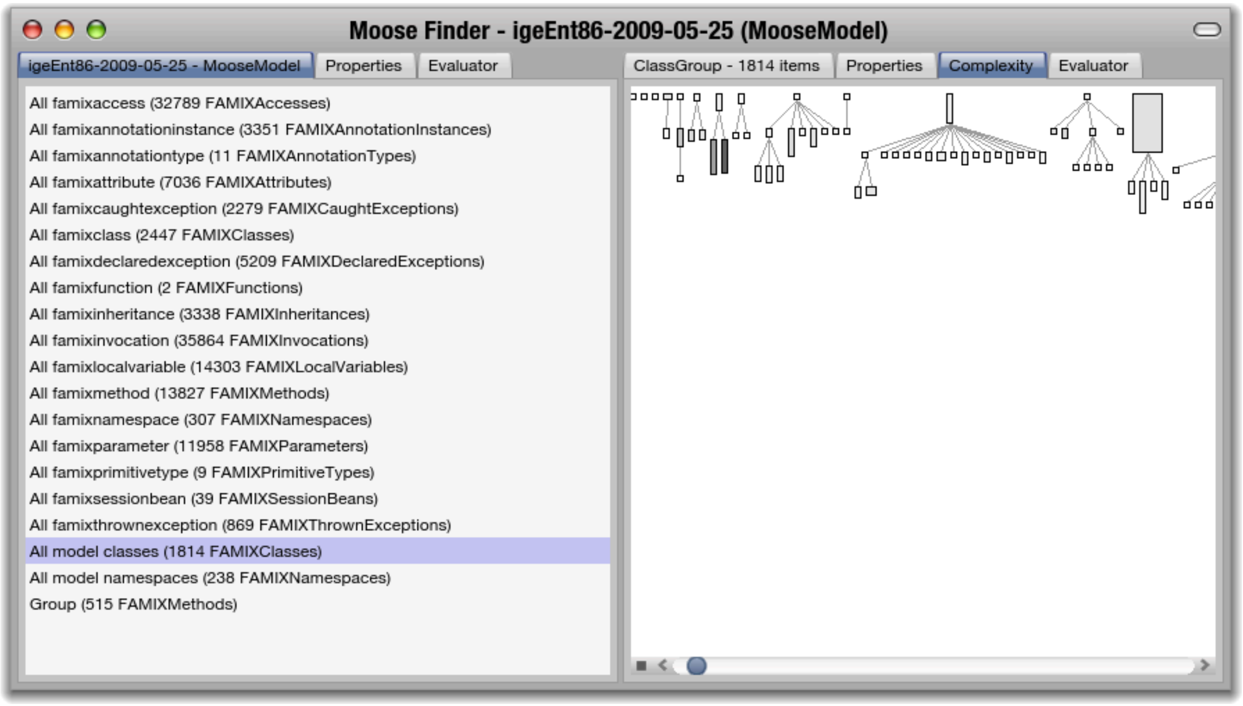
Finally, it may be possible to extract useful information from source files without building a parser at all. Hindle *et al.*, for example, have shown that indentation can be used as a reliable proxy for structure and thus for complexity in source code [13].

3. Experience with Moose

Moose[3] [4] is a platform for data and software analysis that provides numerous services for querying, navigating, summarizing and visualizing models of software systems. Figure 1 illustrates Moose being used to browse the model

[1]http://strategoxt.org/
[2]http://xtext.itemis.com/
[3]http://moosetechnology.org

of a Java Enterprise application. A system complexity view is automatically generated, showing the class hierarchy with selected metrics being directly mapped to the visualization. The visualization is also interactive, enabling further navigation to individual entities and their properties.


Moose Finder - igeEnt86-2009-05-25 (MooseModel)
igeEnt86-2009-05-25 - MooseModel
Properties
Evaluator
ClassGroup - 1814 items
Properties
Complexity
Evaluator

All famixaccess (32789 FAMIXAccesses)
All famixannotationinstance (3351 FAMIXAnnotationInstances)
All famixannotationtype (11 FAMIXAnnotationTypes)
All famixattribute (7036 FAMIXAttributes)
All famixcaughtexception (2279 FAMIXCaughtExceptions)
All famixclass (2447 FAMIXClasses)
All famixdeclaredexception (5209 FAMIXDeclaredExceptions)
All famixfunction (2 FAMIXFunctions)
All famixinheritance (3338 FAMIXInheritances)
All famixinvocation (35864 FAMIXInvocations)
All famixlocalvariable (14303 FAMIXLocalVariables)
All famixmethod (13827 FAMIXMethods)
All famixnamespace (307 FAMIXNamespaces)
All famixparameter (11958 FAMIXParameters)
All famixprimitivetype (9 FAMIXPrimitiveTypes)
All famixsessionbean (39 FAMIXSessionBeans)
All famixthrownexception (869 FAMIXThrownExceptions)
All model classes (1814 FAMIXClasses)
All model namespaces (238 FAMIXNamespaces)
Group (515 FAMIXMethods)

Figure 1: The Moose Finder

Models are imported into Moose with the help of importers that parse source code and generate MSE (Moose Exchange format) files. Numerous importers exist for established languages like Java and C++, but developing a new importer can be a major undertaking. Development of importers for languages as diverse as PHP and PL/1 have consumed many person-months of effort.

We have explored various approaches to speed up this process, which we will briefly summarize here.

3.1. *Parsing by Example*

CodeSnooper [14] is a tool that attempts to generate a parser automatically from example mappings from source code fragments to model elements. The developer is presented with a source code browser, and is asked to indicate where in the source model elements such as class names and methods names occur. The tool then infers an island grammar from these examples, and attempts to parse as much code as possible. Where the tool fails to parse code, it prompts the user for further examples, and the process repeats.

Although the results were promising, there were numerous technical obstacles that prevented the approach from being practical. CodeSnooper was built on top of an LALR parser generator. We believe that parsing-by-example approach should produce better results when generating parsers from PEGs, since PEGs are both unambiguous and composable.

3.2. *Recycling Trees*

A very different approach is to avoid generating a parser altogether. One approach we have previously explored was the possibility to recycle and transform parse trees produced by other tools directly into models [15].

The key difficulty encountered was that the parse tree implementations in different modern IDEs (like Eclipse, NetBeans and Selenium) varied considerably, making it impractical to automatically recognize and process them. It is an open question whether a more clever technique would improve the situation (for example, searching for source code strings in memory to identify the AST nodes).

3.3. Evolutionary Grammar Generation

A third approach of evolutionary grammar generation is reminiscent of the machine learning approaches used in the grammar induction community. Here we randomly generate populations of PEGs, select the fittest individuals, generate a new population using mutation and crossover operators, and we iterate [16].

Unfortunately this approach is extremely slow and costly, and yields only modest results for even simple languages. We expect that the approach could be considerably improved by exploiting the fact that real programming languages are not random but have similar features.

3.4. Helvetia and PetitParser

We have also worked on the development of language workbenches. PetitParser is a framework for developing PEGs in Smalltalk. Helvetia [17] is a language workbench built on top of PetitParser, and designed to extend a host language with new, embedded DSLs in such a way that the development tools of the host language can easily be adapted to the new embedded languages.

We are currently working on extending PetitParser to better support the process of incrementally refining a parser for a given language with the help of island grammars.

4. Agile Modeling

Agile modeling is one of the tracks in a research project on "Agile Software Assessment"[4]. The project aims to offer software developers effective tools and techniques to model, analyze and assess complex software systems within the broader context of an evolving software ecosystem.

The agile modeling track focuses on the problem of quickly constructing a software model for a previously unknown software system for which a model importer does not already exist. Particular challenges include the fact the software may be written in a dialect of an existing language, may consist of heterogeneous code written in multiple languages, may include home-grown DSLs or scripting languages, may include other relevant data sources or configuration files, and parsers for the various languages may not be readily available.

Generating models from source code is, however, fundamentally different from constructing a conventional parser for a compiler or interpreter. These differences lead us to believe that "agile modeling" is feasible for the following reasons:

— *Precise parsing is not needed:* initial analyses typically don't require precise information. For example, the class and method hierarchy is often used to obtain an initial understanding of a system. *Island grammars* can be used to obtain coarse models.

— *More details may be needed over time:* models can be incrementally and iteratively refined. *Composable grammars* built using PEGs can enable incremental refinement of model importers.

— *Most programming languages belong to a group of similar languages:* most languages reuse syntactic and semantic notions from other languages. *Heuristics* (based on *e.g.*, indentation, word frequency, common programming language vocabulary) should help to identify common language constructs in an "unknown" language.

— *A substantial code base already exists:* the code to be analyzed offers a rich set of data. *Examples* can be exploited to infer grammar rules using heuristics already mentioned.

— *Experts are readily available:* developers and domain experts can help in the process of refining model importers. *Semi-automated refinement* of model importers can exploit expert knowledge by asking them to intervene just when importers fail, as outlined earlier in the "parsing by example" approach.

In this section we outline some of our initial results in pursuing these ideas.

[4]http://scg.unibe.ch/asa

4.1. Island parsing with robust water

An island grammar focuses on the parts of interest (islands) and skips parsing of unimportant parts (water). We propose to exploit island grammars to build coarse-grained parsers that can extract high-level information from source code (*e.g.*, class names, methods and instance variables) without needing to understand the source code in detail (*e.g.*, we might not care about the details of method bodies). This coarse-grained grammar can later be incrementally refined and extended to capture more detail. Unfortunately we encountered some difficulties in using island grammars due to conflicts between the rules for water and islands. We now outline these problems and the solutions we have developed.

The simplest implementation of an island grammar could be described as *skip everything until you find what you are looking for*. This is analogous to a regular expression such as "`.*xxx`" that eagerly gobbles up everything until it finds `xxx`. To illustrate the problem, consider the source code in Listing 1 from which we would like to extract all the methods of a class. A parser using the *skip everything* approach does not consider the borders between classes and therefore it incorrectly identifies getColor, and getRadius methods as members of the Shape class.

```
class Shape
    Color color
    method getColor
        return color
    end
endclass

class Circle extends Shape
    int radius
    method getRadius
        return radius
    end
endclass
```

Listing 1: A naive island grammar incorrectly identifies getRadius as a method of Shape

A solution for this problem is to extend islands with a new parameter, a *terminator*, that specifies when to stop parsing. In our example (Listing 1), the terminator would be endclass. This ensures that the island will not cross the border of a class. Unfortunately, this approach does not work well with *parser combinators*, well-structured, modular and easily maintainable building blocks for creating parsers as a graph of composable objects.

When we extended our parsers with the terminator parameter, we bound our parsers to a particular grammar. To illustrate, consider the parser in Listing 2 where we defined a *methodIsland* with the terminator parameter bound to endclass. But the *methodIsland* is no longer a parser combinator, because it cannot be reused in other grammars, for example in grammars where a class body is bound between { and } — this parser can be used only in the grammars with the endclass terminator.

```
<methodIsland> → method <identifier>
                       <body>
                     end
<methodIsland> terminator → endclass
```

Listing 2: Definition of method island

Therefore we focused on an automatic computation of the terminator parameter by analyzing the context in which an island parser is used. A standard parser combinator graph has links from parents to its children. We have extended the parser combinators with the information about the root parser. Knowing about the root parser, the island parser can access its surrounding via child links. When the island parser is supposed to search for an island, it analyzes in which context the island is used, it finds the parsers that are supposed to follow the island and it sets the terminator to these parsers. When the island is later re-used in a different graph of objects, the root parser is different and therefore the terminator is updated.

In the end, we are able to define an island parser for a method as follows:

```
<methodIsland> → method <identifier>
                      <body>
                    end
```

Listing 3: Method island without the terminator parameter specified

When the method island is used in both classDef island grammar rules from Listing 4, the terminator will be automatically computed for each of the use cases of methodIsland. This way, we can reuse the same method island in different grammars and the properties of parser combinators are preserved.

```
<classDef> → class <identifier> {
                 <methodIsland>⋆
             }

<classDef> → class <identifier>
                 <methodIsland>⋆
             endclass
```

Listing 4: Method island used in two different grammars

4.2. Indentation parsing with PetitParser

PEGs are not able to parse layout-sensitive languages such as Python or Haskell [11]. A Python scanner uses leading whitespaces at the beginning of a logical line to compute an indentation level. Depending on indentation level and additional stack structure, the INDENT and DEDENT tokens are generated. Haskell uses a special stage between the scanning and parsing phases to implement language's layout-sensitive features. The problem of parsing layout-sensitive languages was addressed by Erdweg who proposed declarative specifications of layout sensitive languages, integrated the necessary rules into the SDF (Syntax Definition Formalism) and implemented a generalized LR parser [18].

In our research, we focused on extending the PetitParser framework with support for indentation. We have implemented *indent* and *dedent* parser combinators that can signalize the INDENT and DEDENT tokens in the same way as a Python scanner. To allow for a Haskell-like grammar, we implemented a parser combinator that signalizes if the new line has the same indentation level as the previous one. To achieve this, we had to extend the PetitParser with a stack structure that remembers indentation levels.

We are now able to parse Python-like and Haskell-like layout-sensitive grammars. The performance of layout-insensitive grammars implemented in our extended version of PetitParser framework is comparable to the performance of the original version of PetitParser.

4.3. Automatic keyword recognition

We are exploring the possibility of automatically inferring keywords in the code base of an unknown programming language by applying various heuristics. Initially we carried out a study to see how well keywords correlate with basic metrics such as the number of occurrences of a token and the distribution of tokens in source code.

We measured the precision (P) and recall (R) for Java and C languages with these results: $P_{Java} = 0.55$, $R_{Java} = 0.22$, $P_C = 0.35$ and $R_C = 0.22$. We tried to find the top 20 keywords out of 50 (in case of Java) or out of 32 (in case of C). The precision is defined as number of real keywords found divided by twenty. The recall is defined as number of real keywords found divided by number of keywords in the language. So far the project is a preliminary phase and the metrics we used were rather simple. Moreover, the results are negatively affected by the fact that we included strings and comments in our analysis. We plan to further pursue this line of research by experimenting with other metrics and heuristics, for example taking into account domain knowledge of programming languages.

5. Conclusion

We are exploring a number of techniques to enable what we called "agile modeling" — the ability to rapidly construct a model from "unknown" source code to enable various analyses. The central idea is to start with an imprecise, coarse parser to enable simple analyses, and then iteratively and incrementally refine this parser to obtain more finely-grained models.

So far we have focused on setting up the infrastructure for our agile modeling research. We have added support for island parsing and layout-sensitive grammars into the PetitParser framework.

We plan to explore the construction of island grammars from "reusable" and configurable building blocks for common languages constructs (such as method declarations, class declarations *etc.*). Last but not least, the basic island building blocks could be used in a genetic algorithm that tries to infer a grammar for an unknown language.

In the keyword recognition effort, we have also started to explore approaches not based on parsing technologies. We plan to use more advanced heuristics in the keyword recognition project to achieve better precision and recall. Furthermore, we plan to exploit indentation to recognize constructs such as methods, classes and blocks.

As the project advances, we also intend to explore ways of exploiting expert knowledge. We envisage an environment in which model importers are semi-automatically constructed using a cocktail of different techniques (composable island parsers, heuristics, data mining, genetic algorithms), and applied to a large code base. The environment then asks the developer to intervene only to provide missing information (*e.g.*, how do language constructs map to model elements), or to check whether inferred rules are correct (*e.g.*, identification of keywords, annotations, and other language constructs). Although our initial experiments have been encouraging, we are still a very long way from reaching this goal.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also thank Niko Schwarz and Erwann Wernli for their helpful comments.

References

[1] T. D. LaToza, G. Venolia, R. DeLine, Maintaining mental models: a study of developer work habits, in: ICSE '06: Proceedings of the 28th international conference on Software engineering, ACM, New York, NY, USA, 2006, pp. 492–501. doi:10.1145/1134285.1134355.

[2] J. Sillito, G. C. Murphy, K. De Volder, Questions programmers ask during software evolution tasks, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 23–34. doi:10.1145/1181775.1181779.

[3] O. Nierstrasz, M. Lungu, Agile software assessment, in: Proceedings of International Conference on Program Comprehension (ICPC 2012), 2012, pp. 3–10. doi:10.1109/ICPC.2012.6240507.

[4] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), ACM Press, New York, NY, USA, 2005, pp. 1–10, invited paper. doi:10.1145/1095430.1081707.

[5] P. Klint, T. van der Storm, J. Vinju, RASCAL: A domain specific language for source code analysis and manipulation, in: Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on, 2009, pp. 168–177. doi:10.1109/SCAM.2009.28.

[6] C. de la Higuera, A bibliographical study of grammatical inference., Pattern Recognition 38 (9) (2005) 1332–1348. doi:10.1016/j.patcog.2005.01.003.

[7] R. Lämmel, C. Verhoef, Semi-automatic grammar recovery, Software—Practice & Experience 31 (15) (2001) 1395–1438. doi:10.1002/spe.423.abs.

[8] N. Kraft, E. Duffy, B. Malloy, Grammar recovery from parse trees and metrics-guided grammar refactoring, Software Engineering, IEEE Transactions on 35 (6) (2009) 780 –794. doi:10.1109/TSE.2009.65.

[9] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam (Jul. 1997).

[10] R. Frost, J. Launchbury, Constructing natural language interpreters in a lazy functional language, Comput. J. 32 (2) (1989) 108–121. doi:10.1093/comjnl/32.2.108.

[11] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2004, pp. 111–122. doi:10.1145/964001.964011.

[12] L. Moonen, Generating robust parsers using island grammars, in: E. Burd, P. Aiken, R. Koschke (Eds.), Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, 2001, pp. 13–22. doi:10.1109/WCRE.2001.957806.

[13] A. Hindle, M. W. Godfrey, R. C. Holt, Reading beside the lines: Indentation as a proxy for complexity metrics, in: ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2008, pp. 133–142. doi:10.1109/ICPC.2008.13.

[14] O. Nierstrasz, M. Kobel, T. Gîrba, M. Lanza, H. Bunke, Example-driven reconstruction of software models, in: Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2007), IEEE Computer Society Press, Los Alamitos CA, 2007, pp. 275–286. doi:10.1109/CSMR.2007.23.

[15] D. Langone, T. Verwaest, Extracting models from IDEs, in: 2nd Workshop on FAMIX and Moose in Software Reengineering (FAMOOSr 2008), 2008, pp. 32–35.

[16] S. D. Zanet, Grammar generation with genetic programming — evolutionary grammar generation, Master's thesis, University of Bern (Jul. 2009).

[17] L. Renggli, T. Gîrba, O. Nierstrasz, Embedding languages without breaking tools, in: T. D'Hondt (Ed.), ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming, Vol. 6183 of LNCS, Springer-Verlag, Maribor, Slovenia, 2010, pp. 380–404. doi:10.1007/978-3-642-14107-2_19.

[18] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, Layout-sensitive generalized parsing, in: SLE, 2012, pp. 244–263. doi:10.1007/978-3-642-36089-3_14.