Formal Foundations for Semi-parsing

Zaytsev, V.

DOI
10.1109/CSMR-WCRE.2014.6747184
Publication date
2014
Document Version
Author accepted manuscript
Published in
2014 Software Evolution Week

Formal Foundations for Semi-parsing

Vadim Zaytsev, vadim@grammarware.net, http://grammarware.net,
Universiteit van Amsterdam, The Netherlands

***Abstract*—There exist many techniques for imprecise manipulation of source code (robust parsing, error repair, lexical analysis, etc), mostly relying on heuristic-based tolerance. Such techniques are rarely fully formalised and quite often idiosyncratic, which makes them very hard to compare with respect to their applicability, tolerance level and general usefulness. With a combination of recently developed formal methods such as Boolean grammars and parsing schemata, we can model different tolerant methods of modelling software and formally argue about relationships between them.**

I. Motivation

In traditional software language processing, we expect an input program to conform to syntactic constraints of the language, typically expressed by a grammar (hence the term "grammarware" for software which input respects grammatical structure). However, in practice programs may contain errors (e.g., the compiler can be more permissive than the documentation; or a refactoring tool may transform source code into something unacceptable for execution), fragments written in embedded language extensions, dialect-specific code, etc. These pose significant challenges for grammar engineers and for source code analysts in general. Since it has been noted that many program analyses do not need the entire parse tree to generate useful results, many less precise methods are employed instead. They can be referred to as "agile", "tolerant", "robust", "lightweight", "permissive", "recovering" etc; we use the term "semi-parsing" coined by Dean et al. [9] to denote that we intend to parse fragments of input while leaving others unparsed.

A program must be precisely parsed, if it needs to be executed or undergo any other process involving all or almost all the nodes or its syntax tree. On the other side of the tolerance spectrum, pure lexical analysis can be used for low precision tasks and for token-level queries like name mining or keyword frequency analysis. Between these two extremes, there are many techniques like fuzzy parsing, island parsing, bridge parsing, error recovery, etc. Such approaches typically use techniques specific to particular toolkits and workbenches, and are not directly comparable and hard to reuse.

We start the paper with listing 22 existing methods of semi-parsing in subsection II-A. Then, we address the above-mentioned problem of method comparison by proposing two uniform formalisations: one for the syntactic specification (subsection II-B) and one for the process of parsing with it (subsection II-C). The research plan is proposed in section III. To argue in support for its feasibility, we also briefly describe several cases of semi-parsing and sketch their possible formal descriptions (section IV) before arriving at a preliminary conclusion (section V).

II. Background

A. Semi-parsing

In [40, §3.8], different methods of semi-parsing were identified and cited. The list here is updated, extended and reordered:

- *ad hoc lexical analysis* [16]: usually with readily available yet unreliable tools like `grep` or `sed`;
- *hierarchical lexical analysis* [25]: essentially a character-level regular (Type 3) grammar;
- *lexical conceptual structure* [13]: relying on categorisation of tokens instead of tokens themselves;
- *iterative lexical analysis* [7]: persistent bottom-up matching with regular expressions that delivers a parse tree;
- *fuzzy parsing* [17]: parsing triggered by anchor terminals;
- *parsing incomplete sentences* [18]: input may contain unknown parts of unknown length;
- *island grammars* [10]: formally defined in a CFG with water defined lexically;
- *lake grammars* [23]: islands are allowed to have lakes or be explicitly sunk;
- *robust multilingual parsing* [39]: islands can interact;
- *gap parsing* [4]: automated recognition of gaps with water;
- *noise skipping* [19]: a minimum number of unrecognised tokens is skipped;
- *bridge grammars* [27]: semi-automated construction of bridges between islands with the help of reefs such as indentation information;
- *skeleton grammars* [15]: introduction of default rules for water in a baseline complete grammar;
- *breadth-first parsing* [20]: parsing of well-defined islands is triggered by feature tokens;
- *iterative syntactic analysis*: an approximate structure is obtained first, then refined;
- *grammar relaxation* [1]: the grammar is extended with permissive definitions;
- *agile parsing* [9]: additional rules are included to connect several grammars together;
- *permissive grammars* [8]: explicit recovery rules present in a grammar;
- *hierarchical error repair* [3]: the parsing automaton is adjusted;
- *panic mode* [1]: all input is skipped until a synchronisation token;
- *noncorrecting error recovery* [33]: backend has error tolerance;
- *practical precise parsing* [1]: only whitespace information and comments are ignored.

What puts these methods aside from the classic perception on parsing with analytical semantics for the textual input and generative semantics for corresponding trees, is their highly versatile ways of introducing tolerance to the process at different stages in varying quantities. It even remains to be seen whether there is a straight tolerance spectrum from lexical analysis to strict syntactic analysis, as claimed by Klusener and Lämmel [15].

B. Boolean grammars

Boolean grammars [31] extend context-free grammars by augmenting the juxtaposition (sequentiality) and language union (disjunction) operators with two additional ones: language intersection (conjunction) and language complement (negation). They can be seen as a formalisation and generalisation of syntactic predicates [32] and reject productions [5] commonly found in practical parser generation frameworks and systems (ANTLR, TXL, ASF+SDF, PEG, etc).

Technical support for Boolean grammars includes parsing algorithms in the styles of recursive descent [29], LL(k) [30], LR [28], SGLR/RNGLR [22] and TXL parse views [38].

Boolean grammars give us a formal instrument to model constructions such as ordered disjunction [11], forcefully rejected clauses [5], lookahead negation [6], quasi-synchronisation [37], parallelism [2], etc. Such extensions on context-free grammars are common in practical software language processing and also used beyond the context of semi-parsing. However, most of the time their implementations are idiosyncratic and thus hardly comparable to similar metaconstuctions proposed in other frameworks.

C. Parsing schemata

Parsing schemata are high-level abstract declarative descriptions of parsing algorithms [35]. They are an extension of the deductive view on parsing [34]. A parsing process is represented by a set of initial items (partial parse trees), a set of deduction steps (based on grammar production rules) and a set of final items (full parse trees or other artefacts that can be used to obtain full parse trees) — the triplet is usually called a "parsing system". A "parsing schema" is a mapping between any grammar and a corresponding parsing system.

An extension of parsing schemata that deals specifically with error-repair, has been introduced by Gómez-Rodríguez [12] and is based on allowing a distance of several transformation steps between the input program and the one recognised as grammatically correct. By itself this extension is not powerful enough to express any given semi-parsing method, but the result demonstrates the feasibility of the approach proposed by our paper.

Parsing schemata represent (semi-)parsing algorithms as formal entities that can be not only executed, but also analysed, compared, optimised, checked for conformance to certain properties such as "is tolerant with respect to" [15] or proven to belong to the same class of equivalence. A parsing system describes rules for deriving a data structure (such as a tree) from a stream of input symbols, so it can potentially be used to define any conceivable (semi-)parsing method.

III. Proposed method

Extending the parsing schemata formalisation to work with Boolean grammars can be done in a similar style to generalising any other particular particular parsing technique to work with Boolean grammars [22], [28], [29], [30], [38]. Armed with such a combined methodology, we plan to construct a parsing schema for each of the semi-parsing methods listed before in subsection II-A. Preferably, the implementation details are to be pushed to the Boolean part while keeping the schema as uniform as possible for the sake of later comparison. When such formalisations are completed, it should become apparent how semi-parsing techniques relate to one another, and become possible to draw conclusions on the spectrum of existing methods and position them all along it.

Another result we are aiming at, is to provide automated and semi-automated means of converting any baseline grammar into a Boolean grammar with explicitly encoded level of tolerance. These means can then be tested on a repository of grammars of wildly varying size and nature, such as Grammar Zoo, http://slps.github.io/zoo [41].

IV. Case studies

A. Island grammars

In island grammars, we distinguish between islands of interest and the sea of water: usually the former are specified in detail while the latter are left with a less precise and more robust definition. This raises a problem of preferring the islands over water during parsing process, which cannot be expressed directly in a context-free grammar. Within the islands-and-lakes paradigm, there are different solutions: van Deursen and Kuipers [10] assign a higher priority to the production rules corresponding to the islands; Moonen [24] annotates the production rule expressing water, with an "avoid" property; he also needs [23] a "reject" property to explicitly sink some islands — obviously, in case of using both, rejection should be stronger than avoidance, and more nesting levels are impossible to model; Kats et al. [14] use a "recover" property in their permissive grammars in order to separate concerns of disambiguation and error recovery; Dean et al. [9] in similar circumstances use a (negative) lookahead check (on top of the ordered choice already present in TXL). All these approaches are hard to compare because of the technology-specific features.

However, the "reject" annotation is relatively easy to model with Boolean grammars: for nonterminals A, B and C, if $A ::= B$ is a normal production rule and $A ::= C$ is a rejected clause, then $A ::= B\&\neg C$. The ordered choice present in TXL and PEG, is modelled in a similar fashion: if $A|B$ is classic (unordered) disjunction and A/B is the ordered one, then $A/B = A|(B\&\neg A)$. For the case of a lookahead check, $B\&\neg A$ is a stronger formalisation because it automatically assumes that serialisations of A and B are of the same length.

The "avoid" and "recover" clauses are semantically equivalent yet kept as separate features to separate their intentions [8]. Their semantics is impossible to encode with just

Boolean grammars, so we use parsing schemata. For each nonterminal, its normal production rules are mapped to deduction steps as usual [35], but for each "avoid"/"recover" production rule, its set of antecedents (hypotheses) is extended with negations of antecedents of all normal production rules. Under such conditions, the parsing system will not make a step with an avoided rule unless all other steps are disabled.

Another possible use for Boolean grammars is merging the more permissive definitions with the more precise ones, within one grammar. If a nonterminal N is defined as $N \to A\&B$, where A is its precise syntactical specification and B is its tolerant one, then we can either formally reason about the inclusion $L(A) \subset L(B)$, or test the parser of N on a corpus of programs with an assertion that the list of leaves of the parse tree of A must always be identical to the list of leaves of the parse tree of B. This kind of claims has never been entirely possible with semi-parsing methods, but the concern for its necessity has been raised [15].

B. Error recovery

The Dragon Book [1] proposes several simple strategies of error recovery and error repair: for instance, panic mode is one of the simplest methods to detect multiple syntax errors in one pass of the parser. It uses a list of synchronising tokens: in case of a parse error, all input content is skipped until the next synchronising token is encountered, and then precise parsing is resumed [1]. The task of determining and specifying the synchronising tokens is left for the grammar engineer, and the correctness claim about this error recovery method is never backed up.

With a Boolean grammar, we can rewrite any production rule of the form $N \to \alpha t \beta$, where t is a synchronising token, as $N \to (\alpha \& (\neg t)^*) t \beta$, to formally specify the assertion that the string that we expect to parse with α, should be composed of non-t characters.

A more sophisticated improvement of a panic mode based error recovery is hierarchic error repair [3], where synchronising tokens are not provided statically a priori, but dynamically inferred from the synchronisation stack. In that technique, a parser is interpreted as an automaton with states corresponding to correct states of the parsing process and transitions are labelled with consumed tokens. A synchronisation stack is kept during the parsing process in order to help choosing a transition when a parse error occurs (i.e., when there is no transition corresponding to the actual input token). The relation between Boolean grammars and parsing automata is not as strong and well-researched as the relation between more mature grammar classes, so it is at least not trivial to model hierarchic error repair with Boolean grammars. However, it can be easily specified as a parsing schema, since intermediate items in the deduction system can be automata or states in an automaton.

C. Bridge grammars

Bridge grammars [27] are a relatively recent extension of the island grammar paradigm. The method of bridge parsing relies on "reefs" such as indentation or bracketing; bridges between islands are constructed based on such reefs, and artificial islands are created where missing ones should be. As we can see, this is a method of introducing imprecision to the process of parsing, not to the specification per se — hence, parsing schemata should be a feasible way to encode "bridge repair" actions as deduction steps.

V. Conclusion

Preliminary attempts show it to be possible to gain some benefits by formalising semi-parsing. For instance, we can engineer relaxed grammars for semi-parsing systematically instead of reimplementing them within a desired target framework — such an experiment was conducted on a C# grammar in Rascal, which can be inspected at the Software Language Processing Suite repository [42] as the demo::IslandBoolean module, some fragments also shown on Figure 1 and Figure 2. One of the ultimate objectives of laying out formal foundations for semi-parsing is automated derivation of such semi-parsing grammars in a broad sense from baseline grammars, which would have a list of advantages by itself (easy choice among semi-parsing methods, reusing defective grammars, etc).

One of the possibly hardest parts of the future work is addressing a family of search-based error recovery methods, typically relying on some variation of Levenshtein distance [21] sometimes incorrectly referred to as Hamming distance in parsing literature (cf. [36, p.290]). It also remains to be seen whether negation in Boolean grammars and correctness validation function in parsing systems can always be transformed into each other, since both serve as additional limitation mechanisms for researchers and engineers committed to either of formalisations.

We do not claim to be the first in our attempt to formalise semi-parsing approaches: some classification schemes have been proposed at least twice before, by Klusener and Lämmel [15] and by Nierstrasz and Kurš [26]. Our proposed method of comparison is more refined (generalises 22 approaches instead of 3–5) and relies on recent advancements in theoretical computer science.

Acknowledgement

The author is grateful for discussions of his ideas on this topic with Oscar Nierstrasz, Jim Cordy and Zinovy Diskin, which resulted in many insights concerning the use of Boolean grammars and parsing schemata, as well as for opportunities to present and discuss some of this work at CWI PEM 2012 in Amsterdam, SATToSE 2013 in Bern and Parsing@SLE in Indianapolis.

```
include |project://grammarlab/zoo/csharp/ecma-334-1.glue|.
DeYaccifyAll.
UnchainAll   .
InlinePlus   .
inline using-alias-directive.
inline using-namespace-directive.
factor ("using" identifier "=" namespace-or-type-name ";" | "using" namespace-name ";")
    to ("using" (namespace-name | identifier "=" namespace-or-type-name) ";")
    in using-directive.
extract
    using-directive-insides ::= namespace-name | (identifier "=" namespace-or-type-name);
    globally.
inline using-directive.
splitT ",]" into "," "]" in global-attribute-section.
factor
    ( "[" global-attribute-target-specifier attribute-list "]"
    | "[" global-attribute-target-specifier attribute-list "," "]")
  to ("[" global-attribute-target-specifier (attribute-list | attribute-list ",") "]")
  in global-attribute-section.
inline global-attribute-target-specifier.
inline global-attribute-target.
extract global-attribute-section-insides ::= attribute-list | attribute-list ","; globally.
inline class-declaration.
inline struct-declaration.
inline interface-declaration.
inline enum-declaration.
inline delegate-declaration.
rename class-modifier to modifier globally.
unite struct-modifier with modifier.
```

Fig. 1. A fragment of a grammar manipulation script taking a C# grammar extracted from the corresponding ECMA standard (available at Grammar Zoo [41]), in the GrammarLab[2] syntax, which we expect to be intuitively understood. The syntax highlighting scheme shows terminals in green, nonterminals in blue, transformation operators and other keywords in red, and grammar mutations with a green background.

```
importG
    compilation-unit ::=
        ("using" using-directive-insides ";")*
        ("[" "assembly" ":" global-attribute-section-insides "]")*
        namespace-member-declaration* ;
    namespace-member-declaration ::=
        ("[" attribute-section-insides "]")*
        modifier*
        namespace-member-main;
    using-directive-insides ::=
        not-semicolon & identifier "=" namespace-or-type-name;
    global-attribute-section-insides ::=
        not-right-square-bracket & {attribute-list ","}+;
    attribute-section-insides ::=
        not-right-square-bracket & {(attribute-target-specifier? attribute-list) ","}+;
```

Fig. 2. A fragment of a Boolean grammar for C# modelling the skeleton grammar approach by Klusener and Lämmel [15] while preserving both the baseline production rules for precise parsing, and the relaxed production rules for semi-parsing within the same grammar by using conjunctive clauses (shown with infix ampersands).

[2] *GrammarLab: Foundations for a Grammar Laboratory*, a Rascal language workbench library for manipulating grammars in a broad sense, http://grammarware.github.io/lab.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2006.

[2] H. Alblas, R. op den Akker, P. O. Luttighuis, and K. Sikkel, "A Bibliography on Parallel Parsing," *SIGPLAN Notices*, vol. 29, no. 1, pp. 54–65, Jan. 1994.

[3] D. T. Barnard and R. C. Holt, "Hierarchic Syntax Error Repair for LR Grammars," *International Journal of Computer and Information Sciences*, vol. 11, no. 4, pp. 231–258, 1982.

[4] E. Bertsch and M.-J. Nederhof, "Gap Parsing with LL(1) Grammars," *A Journal of Mathematical Research on Formal and Natural Languages*, vol. 8, pp. 1–16, 2005.

[5] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser, "The ASF+SDF Meta-Environment: a Component-Based Language Development Environment," *Electronic Notes in Theoretical Computer Science*, vol. 44, no. 2, pp. 3–8, 2001.

[6] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[7] A. Cox, "Syntactic Approximation Using Iterative Lexical Analysis," in *Proceedings of the International Workshop on Program Comprehension*, 2003, pp. 154–163.

[8] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg, "Natural and Flexible Error Recovery for Generated Modular Language Environments," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 34, no. 4, pp. 15:1–15:50, Dec. 2012.

[9] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider, "Agile Parsing in TXL," *Journal of Automated Software Engineering*, vol. 10, no. 4, pp. 311–336, 2003.

[10] A. van Deursen and T. Kuipers, "Building Documentation Generators," in *Proceedings of International Conference on Software Maintenance (ICSM 1999)*, 1999, pp. 40–49.

[11] B. Ford, "Parsing Expression Grammars: a Recognition-Based Syntactic Foundation," in *Proceedings of the Symposium on Principles of Programming Languages*, January 2004.

[12] C. Gómez-Rodríguez, M. A. Alonso, and M. Vilares, "Error-Repair Parsing Schemata," *Theoretical Computer Science*, vol. 411, no. 7–9, pp. 1121–1139, 2010.

[13] R. Jackendoff, "Semantic Structures," *Current Studies in Linguistics*, vol. 18, 1990.

[14] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser, "Providing Rapid Feedback in Generated Modular Language Environments. Adding Error Recovery to SGLR Parsing," in *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2009)*, G. T. Leavens, Ed. ACM Press, Oct. 2009.

[15] S. Klusener and R. Lämmel, "Deriving Tolerant Grammars from a Base-line Grammar," in *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society, Sep. 2003, pp. 179–188.

[16] S. Klusener, R. Lämmel, and C. Verhoef, "Architectural Modifications to Deployed Software," *Science of Computer Programming*, vol. 54, pp. 143–211, 2005.

[17] R. Koppler, "A Systematic Approach to Fuzzy Parsing," *Software—Practice & Experience*, vol. 27, no. 6, pp. 637–649, 1997.

[18] B. Lang, "Parsing Incomplete Sentences," in *Proceedings of the 12th Conference on Computational linguistics, Volume 1*, ser. COLING '88. Association for Computational Linguistics, 1988, pp. 365–371.

[19] A. Lavie and M. Tomita, "GLR* — An Efficient Noise-Skipping Parsing Algorithm for Context-Free Grammars," in *Recent Advances in Parsing Technology*, ser. Text Speech and Language Technology, H. Bunt and M. Tomita, Eds. Kluwer Academic Press, Aug. 1996, vol. 1, pp. 183–200.

[20] J. A. N. Lee, *The Anatomy of a Compiler*. Van Nostrand, 1967.

[21] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[22] A. Megacz, "Scannerless Boolean Parsing," in *Sixth Workshop on Language Descriptions, Tools and Applications*, J. Boyland and A. Sloane, Eds., 2006, pp. 106–111.

[23] L. Moonen, "Generating Robust Parsers using Island Grammars," in *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society Press, Oct. 2001, pp. 13–22.

[24] ——, "Lightweight Impact Analysis using Island Grammars," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*. IEEE Computer Society Press, Jun. 2002.

[25] G. C. Murphy and D. Notkin, "Lightweight Source Model Extraction," in *Proceedings of the Third ACM SIGSOFT symposium on Foundations of Software Engineering*, ser. SIGSOFT '95. ACM, 1995, pp. 116–127.

[26] O. Nierstrasz and J. Kurš, "Parsing for Agile Modeling," *Science of Computer Programming*, vol. LAFOUS, 2013.

[27] E. Nilsson-Nyman, T. Ekman, and G. Hedin, "Practical Scope Recovery Using Bridge Parsing," in *Post-proceedings of the Second International Conference on Software Language Engineering*, D. Gašević, R. Lämmel, and E. Van Wyk, Eds. Springer-Verlag, 2009, pp. 95–113.

[28] A. Okhotin, "LR Parsing for Boolean Grammars," in *Developments in Language Theory*, ser. Lecture Notes in Computer Science, C. Felice and A. Restivo, Eds. Springer Berlin Heidelberg, 2005, vol. 3572, pp. 362–373.

[29] ——, "Recursive Descent Parsing for Boolean Grammars," *Acta Informatica*, vol. 44, no. 3-4, pp. 167–189, 2007.

[30] ——, "Expressive Power of LL(k) Boolean Grammars," *Theoretical Computer Science*, vol. 412, no. 39, pp. 5132–5155, 2011.

[31] ——, "Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars," *Computer Science Review*, vol. 9, pp. 27–59, 2013.

[32] T. J. Parr and R. W. Quong, "Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k)," in *Compiler Construction*, ser. Lecture Notes in Computer Science, P. A. Fritzson, Ed. Springer Berlin Heidelberg, 1994, vol. 786, pp. 263–277.

[33] H. Richter, "Noncorrecting Syntax Error Recovery," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 478–489, Jul. 1985.

[34] S. M. Shieber, Y. Schabes, and F. C. N. Pereira, "Principles and Implementation of Deductive Parsing," *Journal of Logic Programming*, vol. 24, no. 1&2, pp. 3–36, 1995.

[35] K. Sikkel, *Parsing Schemata — a Framework for Specification and Analysis of Parsing Algorithms*. Springer, 1997.

[36] S. Sippu and E. Soisalon-Soininen, *Parsing Theory. Vol. II: LR(k) and LL(k) Parsing*, W. Brauer, G. Rozenberg, and A. Salomaa, Eds. Springer-Verlag, 1990.

[37] D. A. Smith and J. Eisner, "Quasi-Synchronous Grammars: Alignment by Soft Projection of Syntactic Dependencies," in *Proceedings of the Workshop on Statistical Machine Translation*, ser. StatMT'06. Association for Computational Linguistics, 2006, pp. 23–30.

[38] A. Stevenson and J. R. Cordy, "Parse Views with Boolean Grammars," *Science of Computer Programming*, vol. LAFOUS, 2013.

[39] N. Synytskyy, J. Cordy, and T. Dean, "Robust Multilingual Parsing using Island Grammars," in *Proceedings CASCON'03, 13th IBM Centres for Advanced Studies on Collaborative Research*. IBM Press, 2003, pp. 149–161.

[40] V. Zaytsev, "The Grammar Hammer of 2012," *Computing Research Repository (CoRR)*, vol. 1212.4446, pp. 1–32, Dec. 2012.

[41] ——, "Grammar Zoo: A Repository of Experimental Grammarware," 2013, submitted to the Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5). Under review after major revision.

[42] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, and G. Wachsmuth, "*Software Language Processing Suite*[3]" 2008–2014, http://slps.github.io.

[3]The authors are given according to the list of contributors at http://github.com/grammarware/slps/graphs/contributors.