Context in Parsing: Techniques and Applications

Eric Van Wyk
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

Abstract

This paper discusses two approaches to parsing: Eelco Visser's scannerless generalized LR parsing and our context-aware scanning paired with deterministic LR parsing. We compare the underlying techniques, specifically how parser context is used to disambiguate lexical syntax, and their use in the context of language evolution and composition applications. We also reflect on the many discussions shared with Eelco on these topics, and on our shared realization that our different assumptions about the contexts in which our approaches were used drove and justified the technical decisions made in each.

2012 ACM Subject Classification Software and its engineering → Syntax; Software and its engineering → Parsers; Software and its engineering → Domain specific languages

Keywords and phrases Parsing, Generalized LR Parsing, Context-aware Scanning

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.30

1 Introduction

One thing we have all missed due to the Covid pandemic, and fear missing in the future due to reduced travel for climate change reasons, is the *hallway track* at conferences. These opportunities to talk with fellow computer scientists are an important part of research and helpful in driving it forward. They are also a lot of fun. These informal interactions are, at least so far, difficult to recreate online at virtual events. Other losses include the highly-interactive discussions and presentations at workshops and smaller conferences like the discontinued, but much loved, Workshop on Language Descriptions, Tools, and Applications (LDTA), its vibrant replacement Software Language Engineering (SLE), and the occasional satellite workshops such as Parsing@SLE and OOPSLE.

The discussions at these events – in meeting rooms, in the hallway, or at dinner in the evening – are almost always lively ones. Over the years, the conversations I had with Eelco Visser stand out as some of the most enjoyable and thought provoking. Eelco and I were friendly intellectual sparring partners, each of us advocating for our own approach to scanning and parsing. Eelco had developed scannerless generalized-LR parsing [25, 26] (SGLR) while August Schwerdfeger and I had developed context-aware scanning used with deterministic LR parsers [24, 18] (CAS+LR).

Both of our approaches treat the processes of recognizing lexical syntax (scanning) in the text and determining its phrase structure (parsing) as interdependent ones. This was in contrast to the traditional approach that sees these two tasks as separate; in this view a scanner can first consume the input text and generate a complete sequence of lexical tokens. These tokens are then provided to the parser. This separation of lexical syntax and phrase (context-free) syntax is a natural one. In a program text file the sequence of characters is naturally seen as a sequence of words (identifiers, keywords, punctuation, numeric literals, etc.) interspersed with whitespace and comments. It is the scanner's job to recognize these words in the sequence of characters, discarding whitespace (when it is not relevant) and comments. These lexical symbols are commonly specified using regular expressions. Note that we will use the terms *scanner* and *scanning* as shorthand for the component for, and process of, recognizing lexical syntax even in SGLR despite there being no separate scanner. The parser consumes these lexical tokens and attempts to recognize the underlying

phrase structure, for example, that a *while-loop* begins with a `while` keyword, followed by an expression acting as the loop condition, and then a statement wrapped in curly braces. Context-free grammars are typically used to define the phrase structure of the language.

Context in lexical syntax – a shared divergence

In both SGLR and CAS+LR, the state of the parser provides contextual information to the process for recognizing lexical syntax. Both use LR parsing [14, 1] and thus the current state of the LR parser automatically provides this notion of context. This is so that in different parsing contexts the same sequence of characters can be recognized in different ways, as different lexical tokens.

In his paper "Pure and Declarative Syntax Definition: Paradise Lost and Regained" from Onward! 2010 [10], Eelco and his coauthors explain this need for context with a few examples. One is the challenge of recognizing the lexical syntax in the text "`array [1..10] of integer`". A fragment of a specification using a context-free grammar and regular expressions to define this can be seen in Figure 1. Nonterminals `Type` and `Expr` derive type expressions and value expressions as expected. Lexical syntax would be defined as expected as well. Keywords and punctuation (written between single quotes) are recognized as such; the only non-constant lexical syntax (of interest here) is for integer and floating point literals. The first being a non-empty sequence of digits, the second being a sequence of digits with a trailing, leading, or contained period ("`.`").

In the string "`array [ 1..10 ]`" we need to recognize "`1`" as an integer literal and not recognize "`1.`" as a floating point literal. A traditional scanner would prefer the longer, yet incorrect, match. If the parser context can be used, then the fact that floating-point literals are not allowed in between the square brackets of an array type declaration can disallow the longer match of "`1.`" and instead correctly recognize the shorter integer literal of "`1`".

```
1  Type ::= 'integer'
2         | 'float'
3         | 'array' '[' IntLiteral '..' IntLiteral ']' 'of' Type
4
5  Expr ::= IntLiteral
6         | FloatLiteral
7         | ...
8
9  IntLiteral /[0-9]+/
10 FloatLiteral /([0-9]+\.)|(\.[0-9]+)|([0-9]+\.[0-9]+)/
```

Figure 1 A partial specification for recognizing `array [1..10] of integer`.

Eelco's paper on parsing AspectJ [2], a language that introduced the notion of aspects and aspect-oriented programming [12, 11] to Java, provides the first declarative specification of the language's concrete syntax. As an example, consider the admittedly contrived and abbreviated AspectJ fragment shown in Figure 2. It contains a standard Java class `Sample` with 4 integer fields, "getter" methods for two of them, and a method named `after`. The aspect `Example` will increment the `count` field on an object `o` of type `Sample` *after* a call to either of the getter methods. The pattern "`get*`" on line 9 matches any calls to methods beginning with the letters "`get`" and the increment action takes place after the call to any matched methods.

```
1  class Sample {
2    int get, count, size, shape;
3    int getSize () { ... }
4    int getShape () { ... }
5    boolean after() { return (get*3 > 15) ; }
6  }
7  aspect Example {
8    ...
9    after(): o.get*()  { o.count ++; }
```

Figure 2 An abbreviated example of AspectJ.

We are not so interested in the semantics of AspectJ and more concerned with how its lexical and context-free syntax can be recognized. The first point to note is that "`after`" is used as an identifier on line 5 and as a keyword on line 9. AspectJ introduces new keywords inside an "`aspect`" block, but these are still legal as identifiers elsewhere. On line 5 we see the text "`get*`" which should be recognized as two lexical symbols: the identifier "`get`" and a multiplication sign "`*`". But in the context of the AspectJ pattern on line 9 this same text should instead be seen as pattern matching any identifier beginning with "`get`". Again, we see that in different parts of the program – that is, in different contexts – the same text is recognized as different lexical symbols.

Both SGLR and CAS+LR use parser context to drive the recognition of lexical syntax and thus both directly and easily handle the examples above. The CAS+LR specification of AspectJ [16, 17] was developed in response to Eelco's paper on this topic as we wondered if a deterministic specification was possible in our approach. The techniques these two approaches use are similar in some regards, but different in others. SGLR uses generalized LR parsing and can thus parse the entire class of context-free grammars, while our implementation of CAS+LR uses an LALR(1) parser. In SGLR there is no scanner. Parsing is done down to the character level, thus the term "scanner" is a misnomer since it is all just parsing. Thus parser context is directly used in recognizing lexical syntax. The scanner in CAS+LR is called by the parser each time a new token is needed and it tells the scanner which lexical symbols are currently valid. The scanner uses this to return only lexical symbols that are valid for the current parsing context.

Context in applications

Despite the similarities in recognizing lexical syntax, Eelco and I frequently debated the relative merits of our respective approaches. The answer to the (usually) unspoken question of which one is "better" also relies on a notion of context. The short answer is that neither is generally better; there are contexts in which each might be the wiser choice and these different contexts justify the technical decisions made in each approach.

The different assumptions that underpin our different approaches can be most directly seen in a sentence from the Eelco's "Paradise Regained" paper [10] mentioned above. This paper embraces the freedom of expression afforded papers at the Onward! conference and describes the use of parsing that requires a deterministic subclass of context-free grammars such as LALR(1) as a Biblical "fall from grace." The paper is a joy to read. Eelco lays out the case that language engineers (those who write grammars and build parsers) should be *oblivious* to the parsing algorithms used in parser generators and be *free* to use any

context-free grammar. This is his "Garden of Eden." The serpent that leads to the fall from grace and exiting of the garden is the pursuit of speed and efficiency that forces one into limited deterministic subclasses such as LALR(1).

The linchpin in understanding the different paths that SGLR and CAS+LR have taken can be seen in Eelco's proclamation in his paper [10, page 918] that in the Garden of Eden

> "The language engineers were software engineers and
> the software engineers were language engineers."

In this paper we consider the ramifications of this proclamation for SGLR and how CAS+LR takes a different path by not adhering to it. In Section 2 we revisit some of the technical aspects of SGLR and CAS+LR and discuss the similarities and differences between the two approaches. The focus here is on their use of parsing context in recognizing lexical syntax. In Section 3 we explore the different scenarios, that is application contexts, in which the two approaches best serve their users and discuss how these contexts and assumptions inherent in them led to the different techniques used in each. Section 4 concludes.

2 Context in scanning and parsing techniques

In this section we briefly describe the fundamental techniques in SGLR and CAS+LR, discussing a few relative merits. The description is brief; the cited papers provide the details.

Scannerless Generalized LR Parsing: SGLR

The main points of interest in scannerless generalized LR parsing (SGLR) are right there in the name: the use of generalized parsing techniques and parsing down to the character level so that a separate scanner is not needed.

Generalized parser generators can build parsers for any context-free grammar, thus freeing their users from the restrictions of writing only grammars within a deterministic subclass such as LR(k) or LALR(k). Since all grammars are allowed, even ambiguous ones, users must be alert to this possibility. Ambiguity can often be resolved by refactoring the grammar but this can lead to unnatural and convoluted grammars. For example, ambiguities in expressions with infix operators can be handled by breaking a single *expression* nonterminal into layers (with names such as *term* and *factor*) corresponding to operator precedence. The famous dangling-else ambiguity splits a *statement* nonterminal into so-called *open* and *closed* varieties that is even more convoluted than the expression refactoring.

The "pure" solution to this problem, as advocated by Eelco [10], is to use disambiguation filters. As the name suggests, when an ambiguous parse produces two (or more trees) for the input text, the filters remove the undesired trees, keeping the desired one. These filters can take the form of stating a preference for one grammar production over another, or higher level filters for specifying infix operator precedence and associativity. In SDF, the grammar formalism originally used in SGLR, one can write the specification on the left of Figure 3 (as shown in [10, page 925]) instead of the more verbose one on the right.

A sequence of Eelco's papers provides a compelling case for this approach [13, 21, 4, 5]. It allows one to write more natural grammars, free of the convolutions required to fit the concrete grammar into a deterministic subclass. The resulting concrete syntax trees generated by the parser are simple and thus there is often no need for a second simplified "abstract" tree representation for semantic analysis.

The "scannerless" in SGLR means what it says; there is no scanner. Instead it is parsing all the way down to the character level. As Eelco says in the "Paradise Regained" paper, "and the words were trees." This results in a uniform specification for recognizing both lexical

```
E "*" E -> E {left} >
E "+" E -> E {left}
Literal -> E
```

```
E "+" T -> E
T -> E
T "*" F -> T
F -> T
Literal -> F
```

Figure 3 Grammar disambiguation by filters (left) and nonterminal refactoring (right).

and context-free syntax. Whitespace symbols are ignored, and parsing of comments allows for nested comments – something that is not possible when using only regular expressions to specify lexical syntax. Here disambiguation filters are used to ensure that the traditional longest match ("maximal munch") behavior of matching lexical syntax is enforced.

Returning to the examples from Section 1 we can now see how SGLR uses the context of the parser to properly recognize lexical syntax. In "`array [ 1..10 ]`" the bottom-up behavior of SGLR may recognize "`1`" as an integer literal and also recognize "`1.`" as a floating point literal, but this temporary ambiguity is resolved in the context of "`array [`" since only the former can be used to form a syntactically valid tree. The latter leads to a syntax error and thus that parsing thread is abandoned. Here the parser context recognizing lexical syntax is implicit in that it is all part of a single parser. In the AspectJ example [2] the same techniques are applied. For example, the text "`after`" on line 9 may be recognized as both a Java identifier and an AspectJ keyword, but only the keyword is valid at the beginning of this line, and thus it is used and the identifier tree is discarded.

Context-aware Scanning and LR Parsing: CAS+LR

The primary insight of context-aware scanning [17, 24] is (again) that the parser context can be used by the scanner to be more discriminating in how it recognizes lexical syntax. Both a scanner and parser are used in this approach, but there is a modification in how these two components interact. Lexical syntax is still defined by regular expressions, and a scanner is generated by converting these to a deterministic finite automata (DFA). The difference is that a context-aware scanner has two types of input when called by the parser to produce the next token. The first is the text being scanned. The second is the set of terminal symbols that are *valid* in the current parser context. When context-aware scanning is used with an LR parser, this set of valid terminal symbols is the set of terminals in the current LR parse table state that have entries of *shift*, *reduce*, or *accept*, but not *error*. While the details of how the context-aware scanner is constructed and makes use of this input is to be found in the papers cited above, the idea is a simple one. Where a traditional scanner follows the longest-match rule to prefer a longer token over a shorter one, a context-aware scanner will prefer shorter valid tokens over longer invalid ones.

In the "`array [1..10] of integer`" example, after the parser has shifted the "`array`" and the "`[`" tokens it is a state in which `IntLiteral` has an action of *shift* but `FloatLiteral` has an *error* action. Thus the context-aware scanner only returns the former, and does not consume the "`.`" character and thus does not recognize the latter. The same process plays out in the AspectJ example. In Figure 2, the "`after`" on line 9 is in a parser context in which the AspectJ keyword `after` is valid but the Java identifier is not. The scanner DFA reaches a final state labeled by both terminal symbols and returns the one that is in the valid terminal set for that call to the scanner.

3 Application Contexts: Language Evolution and Composition

The various merits, advantages, and disadvantages of SGLR and CAS+LR were the fodder for many discussions that Eelco and I enjoyed. While we occasionally talked about performance, our main topic was the issue of ambiguity in grammars. It was at one Parsing@SLE, in Indianapolis I believe, that I asked Eelco if the benefit of using an LALR(1) grammar and thus knowing, statically, that the grammar was unambiguous was like static typing in programming languages. Is it not better to know that an ambiguous parse is impossible in the same way that static type systems ensure that run-time type errors are impossible? Eelco agreed that it was better to know desirable properties statically, but that this was more like statically checking for program termination than statically checking for type errors. Many programmers use statically typed languages because the effort to show a program is well-typed is not an onerous one. However, showing that a program terminates is onerous and there is little tool support for doing so. I agreed, but felt that ambiguities can be more difficult to identify than non-termination. He felt the cost of contorting a grammar into a deterministic class like LALR(1) was too high a price to pay for that guarantee. It seemed that perhaps it was simply a matter of personal preference – balancing static guarantees and performance against flexibility in expression and ease of use.

We were both interested in domain-specific languages (DSLs), and extensible languages in which programmers add domain-specific language features to a host language such as Java or C. In this context some clarity about the technical decisions made by the two approaches can be found. Consider a language component introducing syntax for SQL database queries to Java. It may allow for syntax like the following for establishing a connection named `mydb` to a database server and ensuring that a `person` table there has the expected columns:

```
connection mydb "jdbc:derby:./person_db"
   with table person [ person_id INTEGER, first_name VARCHAR ];
```

Another language extension may introduce condition tables, a construct useful for understanding complex boolean expressions. A simple example can be seen below:

```
boolean b = table ( c1 :  T F
                    c2 :  F * );
```

This sets a boolean identifier `b` to the value of the condition table. The table will evaluate to true if condition `c1` is true (`T`) and `c2` is false (`F`) or if `c1` is false and `c2` has either value (`*`). This condition is equivalent to `(c1 && !c2) || (!c1)` and may be translated down to this expression or an equivalent one that avoids evaluating `c1` twice. Tables like these are used in modeling language such as RSML^{-e} [19] and SCR* [6] where complex Boolean conditions need to be understood by software stakeholders. This is another example in which parser context disambiguates lexical syntax, in this case to recognize "`table`" as a keyword token from the correct extension specification. Eelco's MetaBorg approach using SGLR allows language users to extend Java with new syntax similar to the examples above [3] while we did it using CAS+LR in for Java in our ableJ [23] system and later for C in ableC [8].

Eelco's research, embodied in the Spoofax language workbench [9] and its collection of language processing tools, allows software engineers (the language users) to have a hand in prototyping, designing, and implementing the languages they wanted to use to solve their problems. This can be done using extensible languages, as described above, in developing new domain-specific language features and composing them with others to create an extended language tailored for a particular task. Eelco's work also enables and encourages engineers to design their own stand-alone DSLs. In creating a new language it is not uncommon for

the (composed) context-free grammar to be unambiguous even if it is not in a deterministic subclass like LALR(1). If it is ambiguous, the ambiguities are seen as bugs. One can still use the generated parser and simply treat an ambiguous parse as a syntax error. If this kind of error occurs too frequently, then one may work to resolve the ambiguity in the specification by adding or modifying a disambiguation filter, such as those shown on the left of Figure 3. This approach allows an engineer to easily prototype a new language, a central focus of Eelco's work. The effort, sometimes a considerable one, to transform an initial grammar to be a member of a deterministic subclass can be avoided. This view rests on his proclamation that the roles of language engineer and software engineer should be interchangeable, and is realized by the technical choices made in SGLR and its supporting tools.

My students and I were primarily interested in language evolution and feature composition in a different context – one in which the roles of language engineer and software engineer are kept as distinct. That is, we start with a set of assumptions different from those encapsulated in the proclamation from Eelco's "Paradise Regained" paper. In this view, language extensions are independently developed by authors (language engineers) familiar with context-free grammars and parser-generator tools (and the attribute grammar formalisms used to specify the semantics of the language features [22]). The users of language extensions however – the software engineers composing a language from some set of extensions – need *not* be familiar with the parsing techniques. This view acknowledges that some software engineers may simply not want to know about grammars and parsing (as bizarre as it may seem to many readers of this paper.) Software engineers are not only oblivious to any underlying parsing algorithms, they are also oblivious to parsing and context-free grammars. Our interest was in seeing how far we could go with this different set of assumptions. This view was realized in the ABLEC extensible specification for C and a collection of modular and reliably composable language extensions [8, 7].

For this approach to be feasible, language extensions must compose with the host language automatically, and the resulting composition must be well-formed. For concrete syntax specifications, this requires the composed context-free grammar to be unambiguous and for there to be no lexical ambiguities in the lexical syntax. Thus we developed a *modular determinism analysis* [18, 17] for context-free and lexical syntax provides a guarantee for composed specifications: if the extensions independently pass the analysis, then the specification automatically composed from the user-selected extensions would be deterministic, specifically, LALR(1) and the lexical specification would have no lexical ambiguities.

Composing context-free grammars and regular expressions is simply a matter of taking the union of the sets of productions, nonterminals, and terminals (with associated regular expressions) in the separate specifications, denoted as $\cup^*$ below. We are concerned with some specification C that is the composition of a host language H and set of independently-developed extensions $\{E_1, ..., E_n\}$, that is $C = H \cup^* E_1 \cup^* ... \cup^* E_n$. The property provided by the modular determinism analysis can be expressed as

$$(\forall i \in [1, n] \ .det_m(H, E_i)) \ \Rightarrow \ det(H \cup^* E_1 \cup^* ... \cup^* E_n)$$

This property states that if each extension E_i satisfies the modular composability criteria (det_m) with respect to the host language H, then the lexical and context-free syntax of the composition of all components ($H \cup^* E_1 \cup^* ... \cup^* E_n$) is also deterministic (det). For context-free syntax, the modular analysis det_m checks certain characteristics of the grammar $H \cup^* E_i$ to ensure that the composition of all the grammars ($H \cup^* E_1 \cup^* ... \cup^* E_n$) will be LALR(1) and thus non-ambiguous. First it requires $H \cup^* E_i$ to be LALR(1). Additionally det_m requires any production in E_i with a host language nonterminal on the left-hand side

to have, as the first symbol on its the right-hand side, a new so-called *marking* terminal [18] introduced by that extension. The `table` keyword from the condition table example above is a marking terminal. It also checks that the follow sets of host nonterminals in $H \cup^* E_i$ do not exceed those from H alone. Additional checks [18] on the LR-automata generated from $H \cup^* E_i$ ensure that each extension creates an effectively isolated part of the eventually composed LR-automata that corresponds to having no conflicts in the parse table.

For lexical syntax, determinism requires that in any parsing context only one terminal symbol be matched. The primary requirement is that for any two different terminals t_1 and t_2, if (i) they are both valid for some parser state p of the LR-parse table and (ii) t_1 and t_2 both label a final state in the scanner DFA (meaning both could be returned by the scanner), then one has lexical precedence over another. An example of this is keyword terminals being preferred over identifier terminals. This ensures that SQL `table` keyword will never conflict with the condition-table `table` keyword from above. The isolation of LR-automata for different extensions, as checked by det_m, ensures that this determinism condition is met in the final composition for all LR parse states except for those states that are considered to be "host" states. For example, if a third extension introduces a marking terminal matching "`table`" then there is a lexical ambiguity between two extensions that cannot be detected by the analysis. Fortunately, these ambiguities can be easily resolved by the software engineer composing the extensions (with a bit of tool support) to require a short disambiguating prefix be written before each use of "`table`". This is similar to the Java requirement that if two imported packages define the same type, then some form of qualified name be used to distinguish them. The full description of the analysis can be found in earlier works [18, 17].

The point of this modular analysis is that the cost of being constrained to deterministic grammars is paid by language engineers. This has not, in Eelco's words, "delivered language engineers out of the slavery to parser generators" as they may still resort to the "process of trial-and-error" to "simply torture the parser definition until it confesses" and satisfies the analysis requirements. A significant benefit, however, is reaped by the software engineers *choosing* a set of extensions. They need not know anything about grammars or parsing but are assured that the syntax specifications of the extensions they have chosen will be automatically composed and guaranteed to form an unambiguous specification. This provides the freedom to freely pick the independently-developed language extensions they desire.

It is clear then that these different contexts – in which software engineers *are* or *are not* language engineers – drove the technical decisions made in both SGLR and CAS+LR.

4 Conclusion

We have compared the SGLR and CAS+LR approaches to parsing: the underlying techniques used by both, and the application contexts in which they each shine. Eelco and I eventually realized that we each had a different view of language evolution and composition and that each had developed tools and techniques that supported our individual views. After this, our conversations shifted to focus on other problems in the field of software language engineering, including some on Eelco's influential scope graphs [15, 20].

That said, I do miss our "pugnacious" debates about parsing. Eelco was a wonderful sparring partner in these: committed, insightful, helpful, and caring. It is a great joy to have someone like him who pushes you to think more carefully and clearly about your work and helps you better understand the context in which it resides. He will be sorely missed.

References

1 A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

2 Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for aspectJ. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 209–228. ACM, 2006. doi:10.1145/1167473.1167491.

3 Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, pages 365–383. ACM, 2004. doi:10.1145/1028976.1029007.

4 Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Deep priority conflicts in the wild: a pilot study. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE*, pages 55–66. ACM, 2017. doi:10.1145/3136014.3136020.

5 Luis Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. Towards zero-overhead disambiguation of deep priority conflicts. *Programming Journal*, 2(3):13, 2018. doi:10.22152/programming-journal.org/2018/2/13.

6 C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS)*, 1995.

7 Ted Kaminski. *Reliably Composable Language Extensions*. PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2017. URL: http://hdl.handle.net/11299/188954.

8 Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98:1–98:29, October 2017. doi:10.1145/3138224.

9 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*, OOPSLA. ACM, 2010. doi:10.1145/1869459.1869497.

10 Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Onward!'10, pages 918–932. ACM, 2010. doi:10.1145/1869459.1869535.

11 G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 Object–Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001. doi:10.1007/3-540-45337-7_18.

12 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object–Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997. doi:10.1007/BFb0053381.

13 Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

14 D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. doi:10.1016/S0019-9958(65)90426-2.

15 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

16 August Schwerdfeger. A declarative specification of a deterministic parser and scanner for AspectJ. Technical Report 09-007, University of Minnesota, March 2009. URL: https://hdl.handle.net/11299/215794.

17 August Schwerdfeger. *Context-Aware Scanning and Determinism-Preserving Grammar Composition, in Theory and Practice*. PhD thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, 2010. URL: http://purl.umn.edu/95605.

18 August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210, New York, NY, USA, June 2009. ACM. doi:10.1145/1542476.1542499.

19 Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, September 1999. doi:10.1145/318774.318940.

20 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

21 Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002. doi:10.1007/3-540-45937-5_12.

22 Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010. doi:10.1016/j.scico.2009.07.004.

23 Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599. Springer, 2007. doi:10.1007/978-3-540-73589-2_27.

24 Eric Van Wyk and August Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE)*, pages 63–72, New York, NY, USA, 2007. ACM. doi:10.1145/1289971.1289983.

25 Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, August 1997. URL: https://eelcovisser.org/publications/1997/Visser97-SGLR.pdf.

26 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997. URL: https://eelcovisser.org/publications/1997/Visser97.pdf.